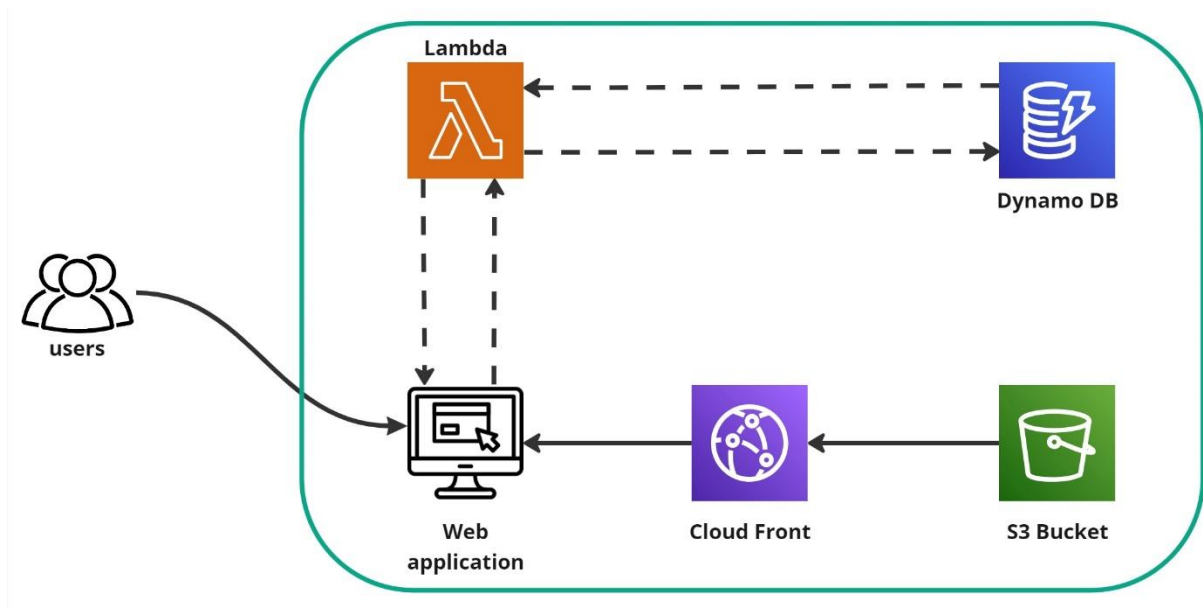


Serverless web application

Project Description:

In this project, you will build a serverless web application using AWS Lambda, CloudFront, DynamoDB, and S3. The application will allow users to update the view count from a DynamoDB table.

Project Architecture:



Steps to build the project:

- Create a DynamoDB table to store the items as view count.
- Build a Lambda function to handle the view count operations on the DynamoDB table.
- Use S3 to store and host the web application's static files (HTML, CSS, and JavaScript).
- Create a CloudFront distribution to serve the S3-hosted static files with low latency.

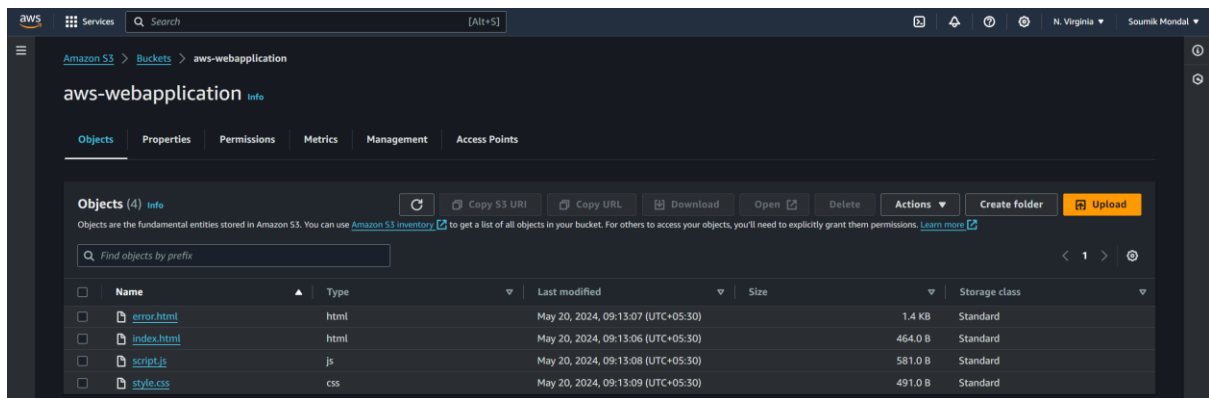
Expected outcome:

Upon completing the project, you will have a working serverless web application hosted on AWS. Every time you refresh the webpage, the view count will increase.

This project will help you improve your skills in cloud computing, serverless architecture, and AWS services.

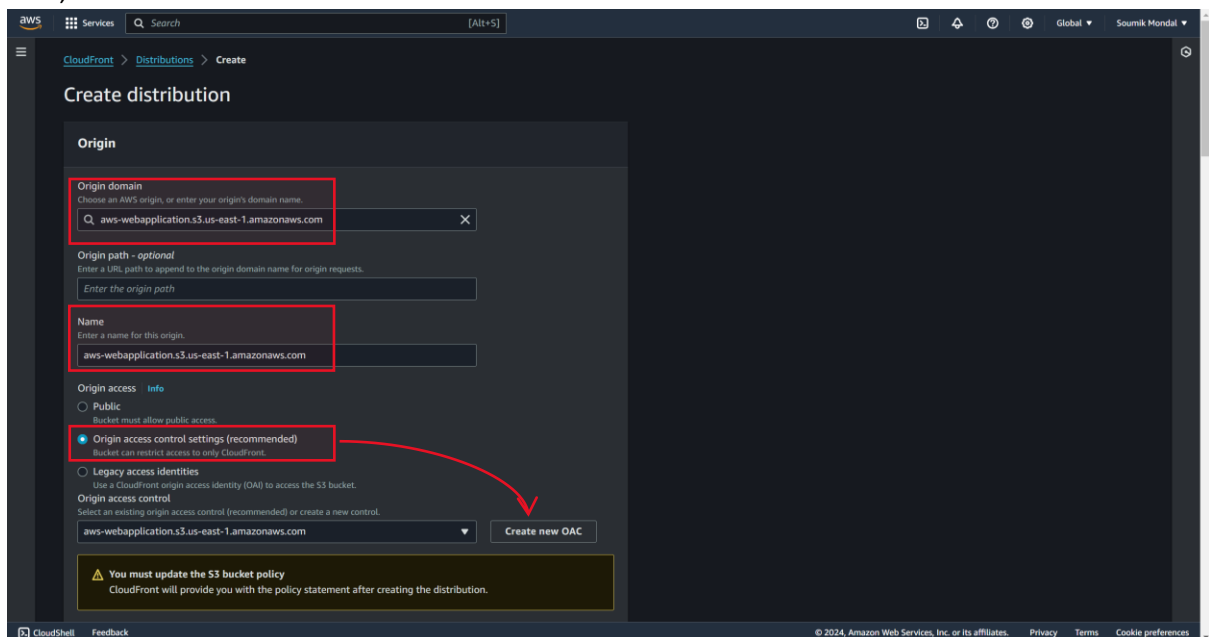
Code repo: <https://github.com/MiksVeg/aws-webapplication.git>

Let's create a S3 bucket-

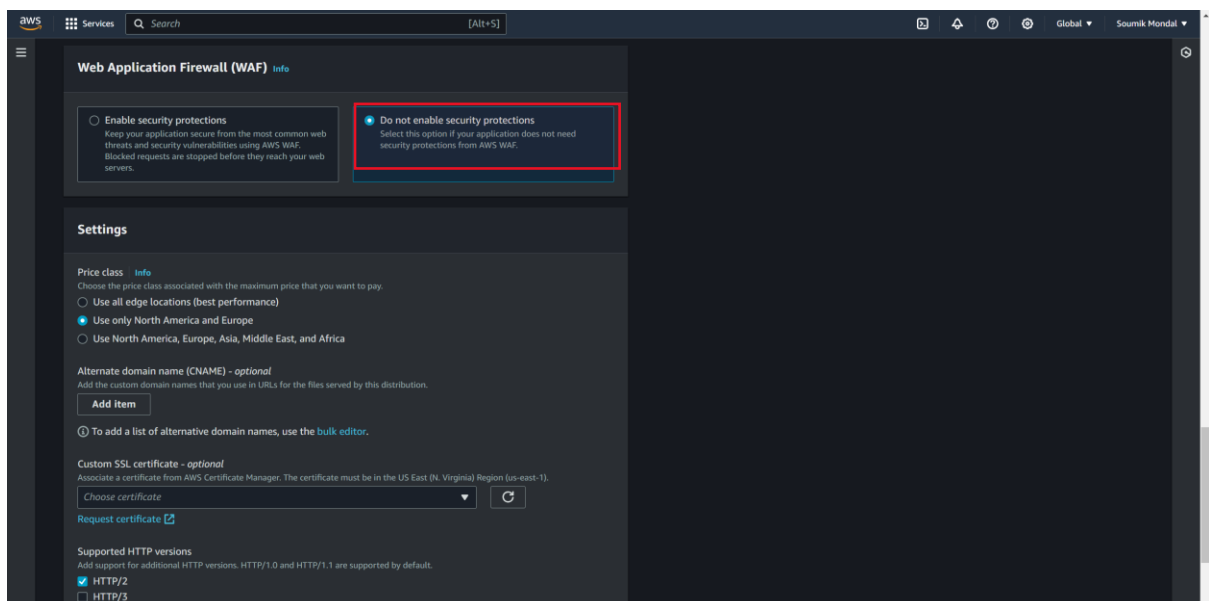


Created an S3 bucket and uploaded index.html, script.js, and style.css files

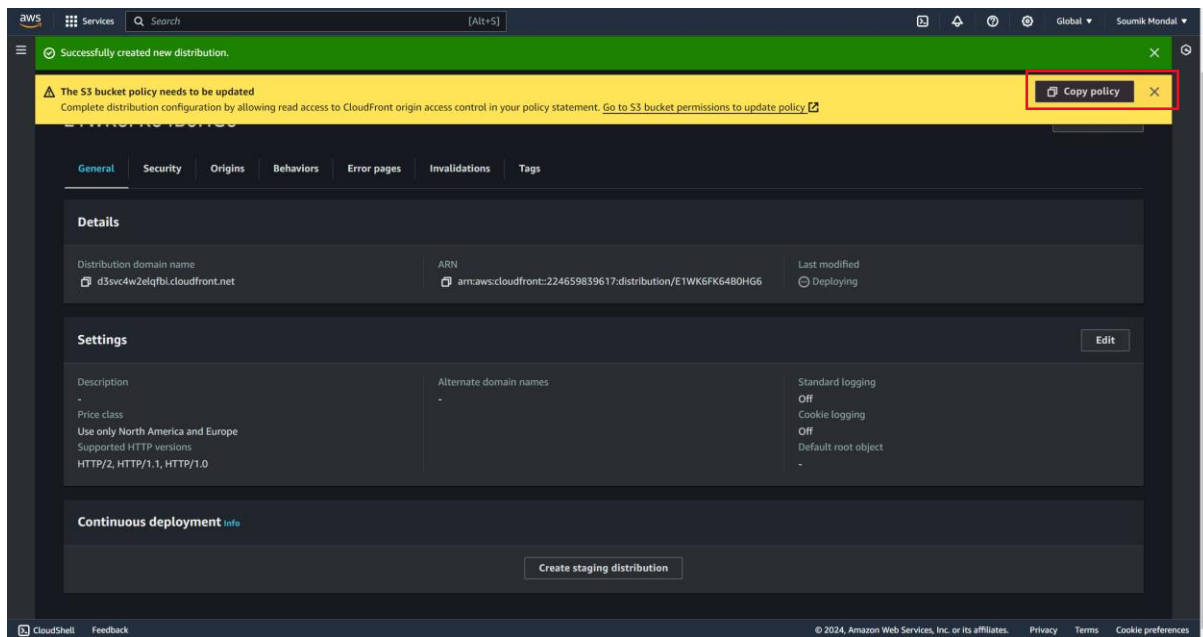
Now, create cloud front distribution -



Create a new OAC (Origin Access Identity) so that only CloudFront can access S3 bucket

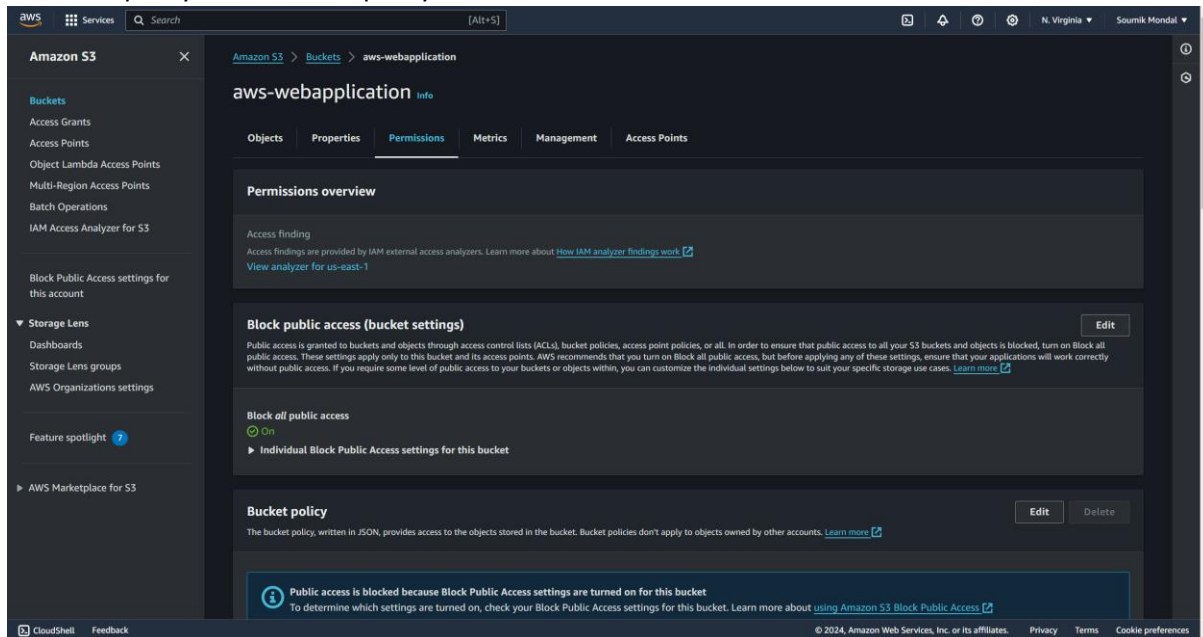


Now, we have to give permission to CloudFront to access the files of the S3 bucket -

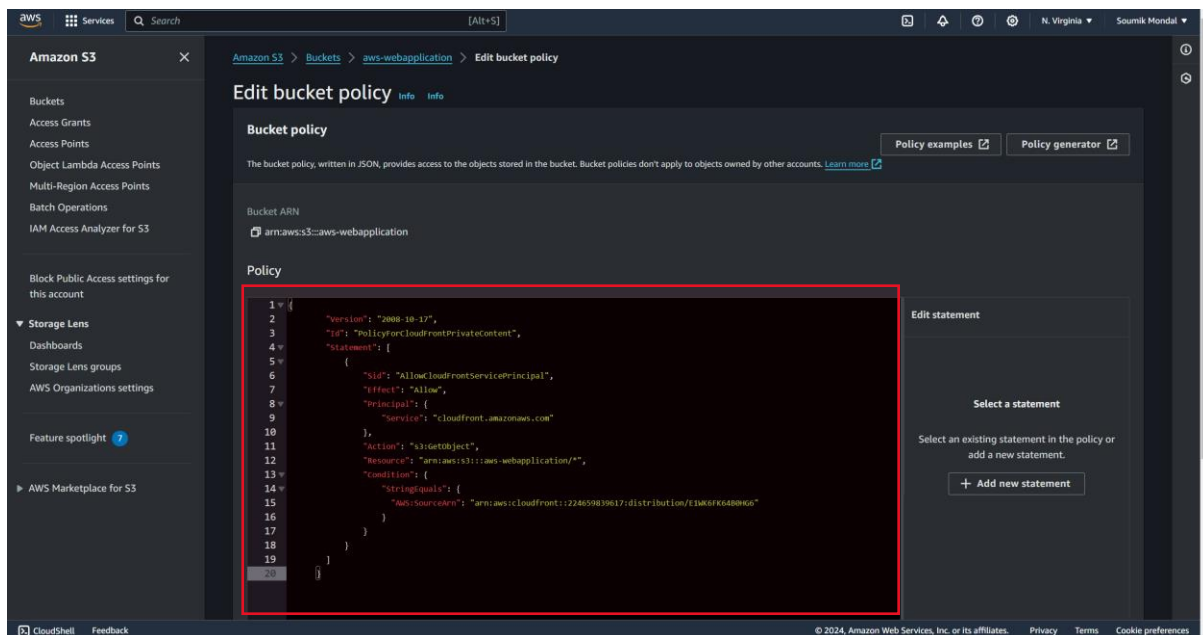


Copy the policy

Paste the policy in the Bucket policy of S3 -

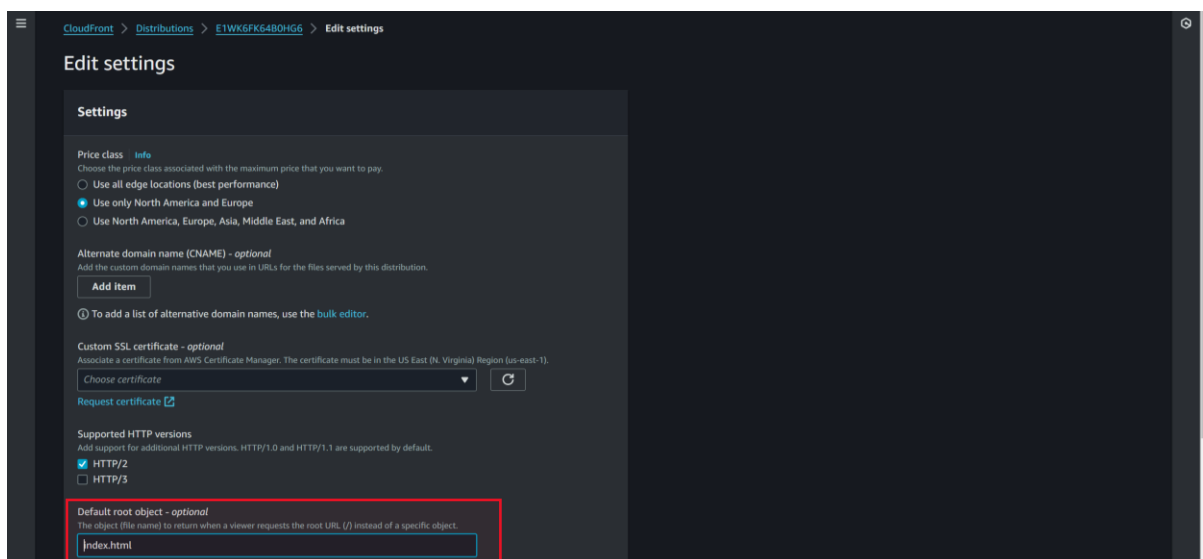
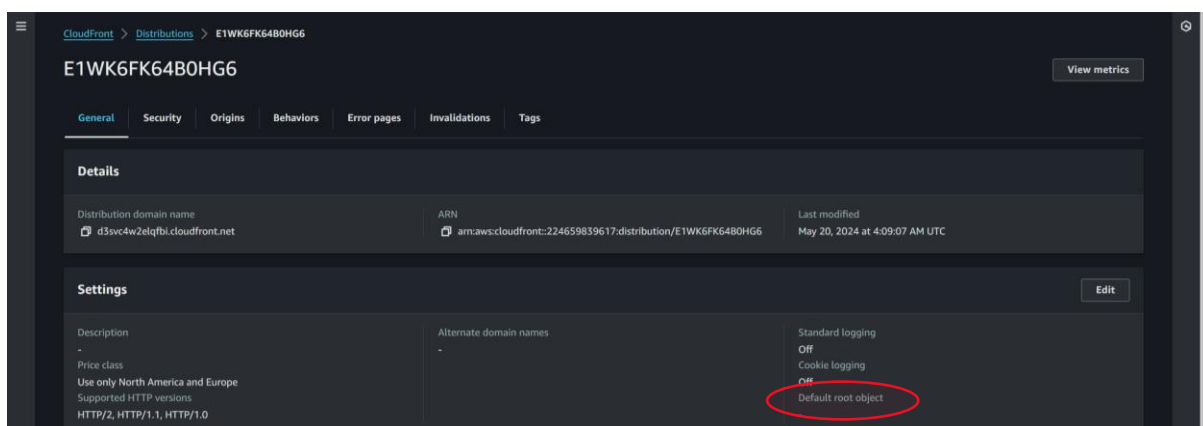


Go to the S3 Bucket and find the permission tab.

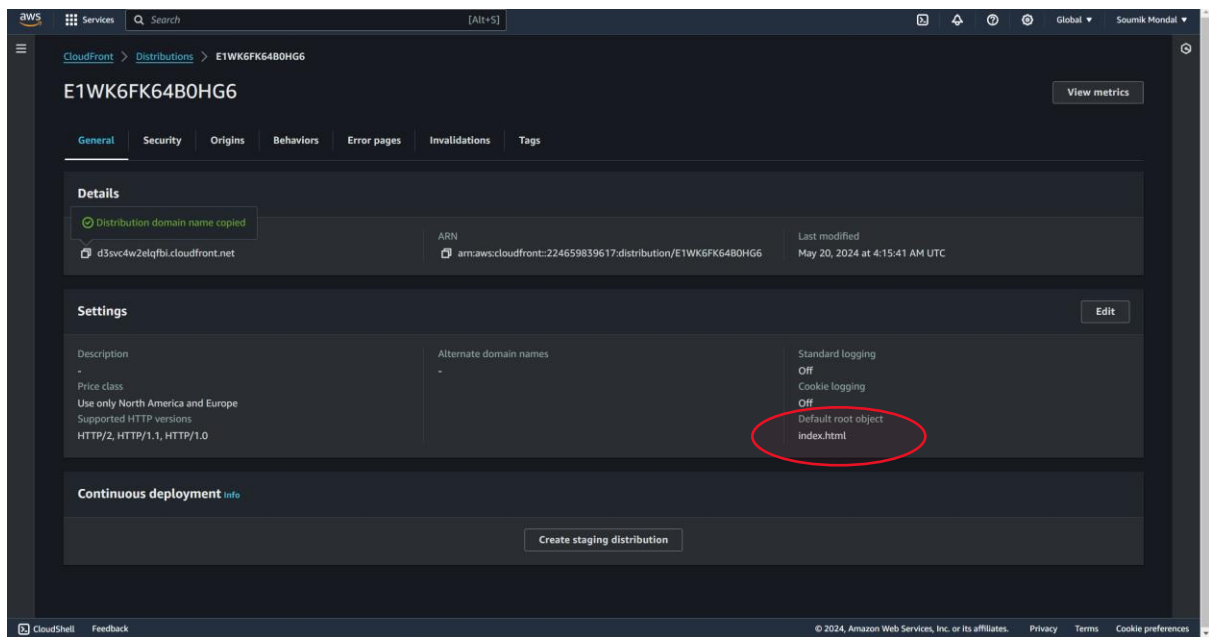


Paste and save the bucket policy

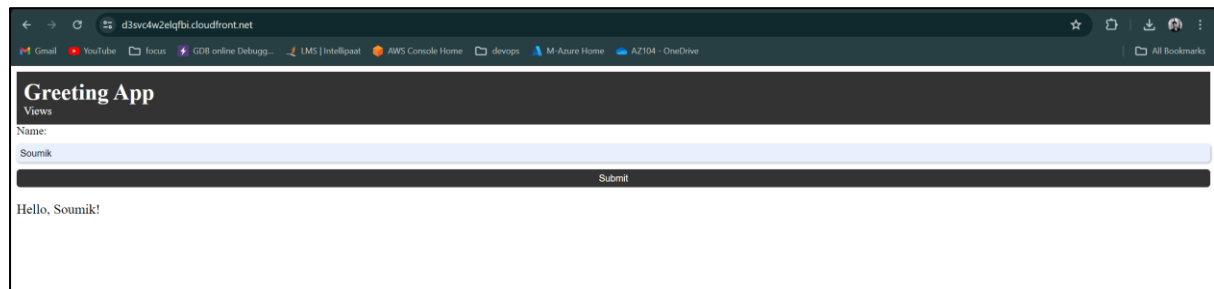
Now, configure the cloud front to face the home page to index.html -



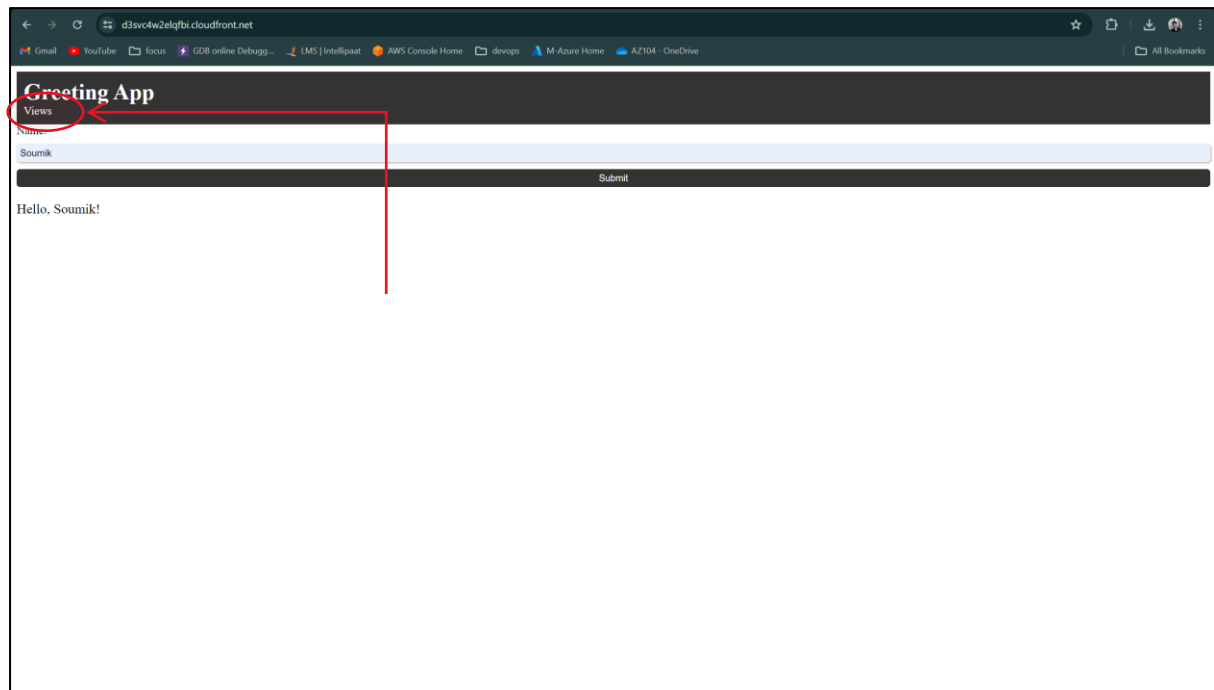
Edit the default root object to index.html



Successfully configured CloudFront



Successfully deployed and accessed the webpage.



Now, we will work on view count, which will increase when we refresh the page.

Let's create a DynamoDB table -

The screenshot shows the 'Create table' page in the AWS Management Console. The table name is 'aws-webapplication'. The partition key is 'id' with a data type of 'String'. The sort key is optional and is not set. The table settings are set to 'Default settings'.

Table details [Info](#)
DynamoDB is a schemaless database that requires only a table name and a primary key when you create the table.

Table name
This will be used to identify your table.

Between 3 and 255 characters, containing only letters, numbers, underscores (_), hyphens (-), and periods (.).

Partition key
The partition key is part of the table's primary key. It is a hash value that is used to retrieve items from your table and allocate data across hosts for scalability and availability.

1 to 255 characters and case sensitive.

Sort key - optional
You can use a sort key as the second part of a table's primary key. The sort key allows you to sort or search among all items sharing the same partition key.

1 to 255 characters and case sensitive.

Table settings

☒ **Default settings**
The fastest way to create your table. You can modify these settings now or after your table has been created.

☐ **Customize settings**
Use these advanced features to make DynamoDB work better for your needs.

Create partition key as id, and value will be as string

The screenshot shows the 'Create item' page in the AWS Management Console. The table is 'aws-webapplication'. The 'Attributes' section shows two attributes: 'id - Partition key' with a value of '0' and a type of 'String', and 'views' with a value of '0' and a type of 'Number'. The 'views' attribute is highlighted with a red box.

Create item

You can add, remove, or edit the attributes of an item. You can nest attributes inside other attributes up to 32 levels deep. [Learn more](#)

Attributes

Attribute name	Value	Type
id - Partition key	0	String
views	0	Number

Create an item as Views and give the value as 0

Next, create a role in IAM to give access of DynamoDB to Lambda -

The screenshot shows the 'Create role' page in the AWS Management Console. The 'Trusted entity type' is 'AWS service'. The 'Service or use case' is 'Lambda'. The 'Use case' is 'Lambda'.

Step 1: Select trusted entity

Select trusted entity [Info](#)

Trusted entity type

☒ **AWS service**
Allow AWS services like EC2, Lambda, or others to perform actions in this account.

☐ **AWS account**
Allow entities in other AWS accounts belonging to you or a 3rd party to perform actions in this account.

☐ **Web identity**
Allows users federated by the specified external web identity provider to assume this role to perform actions in this account.

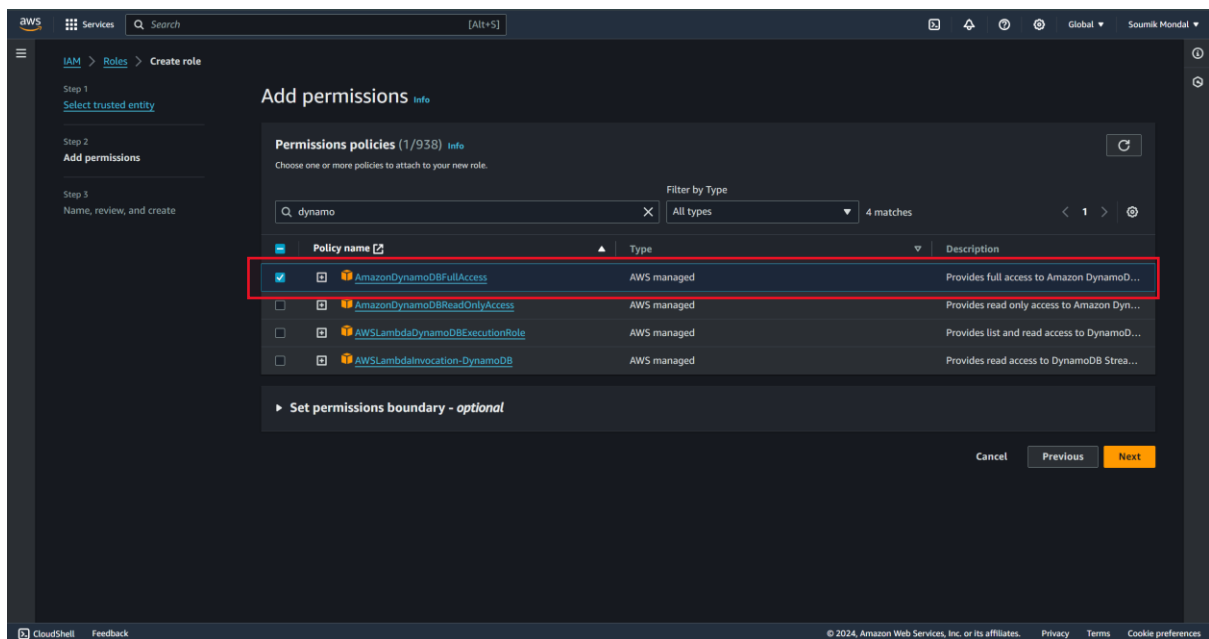
☐ **SAML 2.0 federation**
Allow users federated with SAML 2.0 from a corporate directory to perform actions in this account.

☐ **Custom trust policy**
Create a custom trust policy to enable others to perform actions in this account.

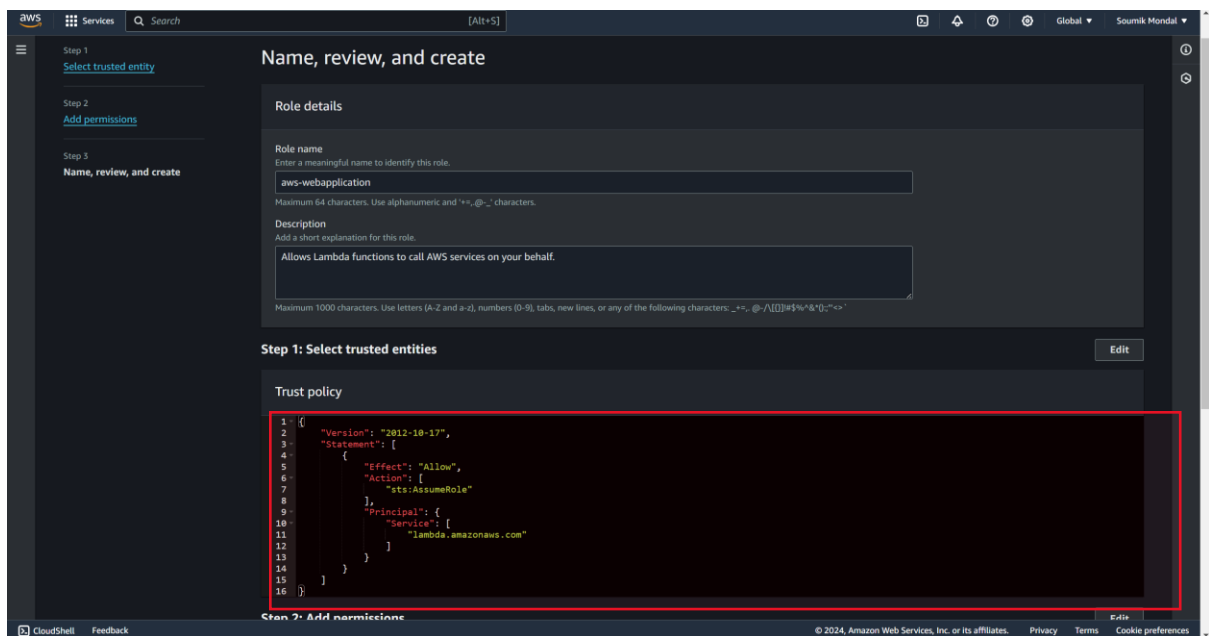
Use case
Allow an AWS service like EC2, Lambda, or others to perform actions in this account.

Service or use case

Choose a use case for the specified service.
Use case
☒ **Lambda**
Allows Lambda functions to call AWS services on your behalf.



Give full access to DynamoDB



Check that the policy gives full access of DynamoDB to Lambda

Now, create a Lambda function

Create function [Info](#)

Choose one of the following options to create your function.

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

Basic information

Function name
Enter a name that describes the purpose of your function.

Use only letters, numbers, hyphens, or underscores with no space.

Runtime [Info](#)
Choose the language to use to write your function. Note that the console code editor supports only Node.js, Python, and Ruby.

Python 3.8

⌵

↻

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](#)

[▼ Advanced settings](#)

As My code is written on Python code so, we have to take Python as runtime.

☐ **Enable code signing** [Info](#)
Use code signing configurations to ensure that the code has been signed by an approved source and has not been altered since signing.

☒ **Enable function URL** [Info](#)
Use function URLs to assign HTTP(S) endpoints to your Lambda function.

Auth type
Choose the auth type for your function URL. [Learn more](#)

- ☐ **AWS_IAM**
Only authenticated IAM users and roles can make requests to your function URL.
- ☒ **NONE**
Lambda won't perform IAM authentication on requests to your function URL. The URL endpoint will be public unless you implement your own authorization logic in your function.

Function URL permissions

When you choose auth type **NONE**, Lambda automatically creates the following resource-based policy and attaches it to your function. This policy makes your function public to anyone with the function URL. You can edit the policy later. To limit access to authenticated IAM users and roles, choose auth type **AWS_IAM**.

[▼ View policy statement](#)

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "StatementId": "FunctionURLAllowPublicAccess",
      "Effect": "Allow",
      "Principal": "*",
      "Action": "lambda:InvokeFunctionUrl",
      "Resource": "arn:aws:lambda:us-east-1:224659839617:function:aws-webapplication",
      "Condition": {
        "StringEquals": {
          "lambda:FunctionUrlAuthType": "NONE"
        }
      }
    }
  ]
}
```

Invoke mode
Choose how your function returns responses. [Learn more](#)

- ☒ **BUFFERED (default)**
The invocation results are available when the payload is complete. Response payload max size: 6 MB.
- ☐ **RESPONSE_STREAM**
Stream the invocation results. Streaming responses incurs additional costs. Refer to the documentation for payload size limitations. [Learn more](#)

☒ **Configure cross-origin resource sharing (CORS)**
Use CORS to allow access to your function URL from any origin. You can also use CORS to control access for specific HTTP headers and methods in requests to your function URL. By default, all origins are allowed. You can edit this after creating the function. [Learn more](#)

☐ **Enable tags** [Info](#)
A tag is a label that you assign to an AWS resource. Each tag consists of a key and an optional value. You can use tags to search and filter your resources, track your AWS costs, and enforce attribute-based access control.

☐ **Enable VPC** [Info](#)
Connect your function to a VPC to access private resources during invocation.

[Cancel](#) [Create function](#)

Now paste this code to code editor of lambda -

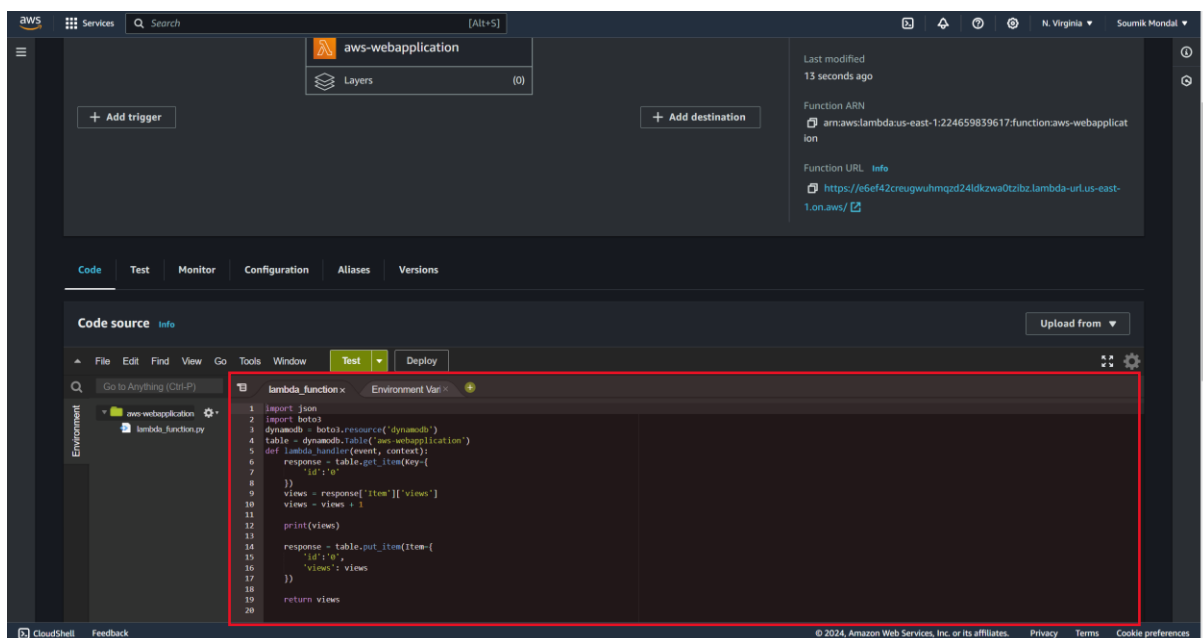
```
import json
import boto3
dynamodb = boto3.resource('dynamodb')
table = dynamodb.Table('aws-webapplication')
def lambda_handler(event, context):
    response = table.get_item(Key={
        'id':'0'
    })
    views = response['Item']['views']
    views = views + 1

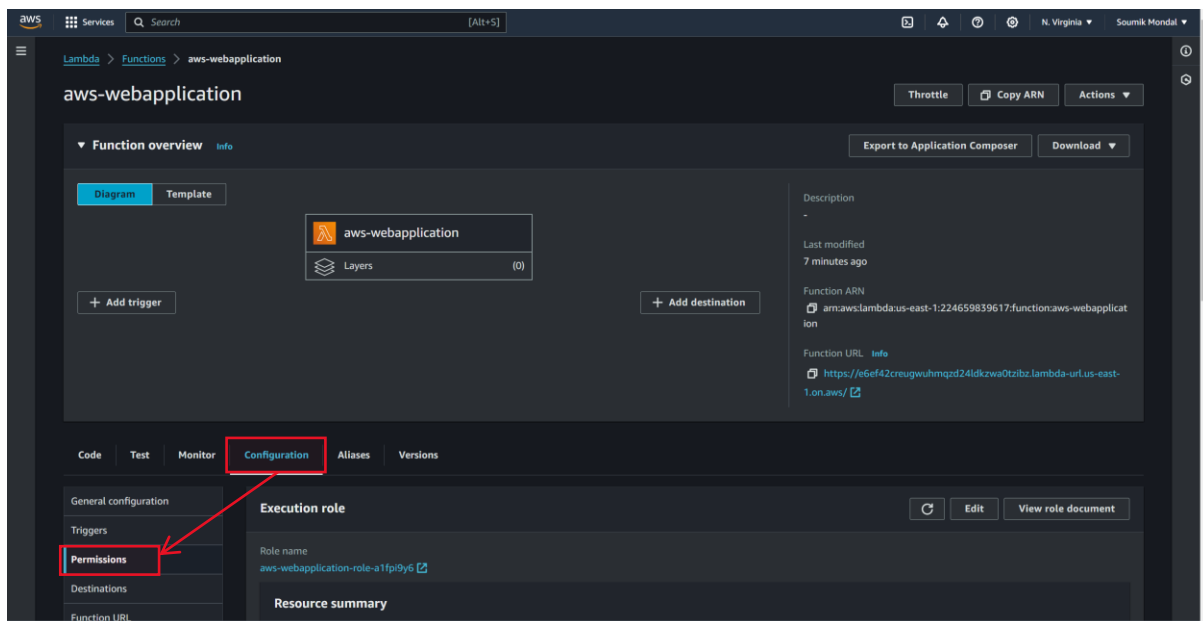
    print(views)

    response = table.put_item(Item={
        'id':'0',
        'views': views
    })

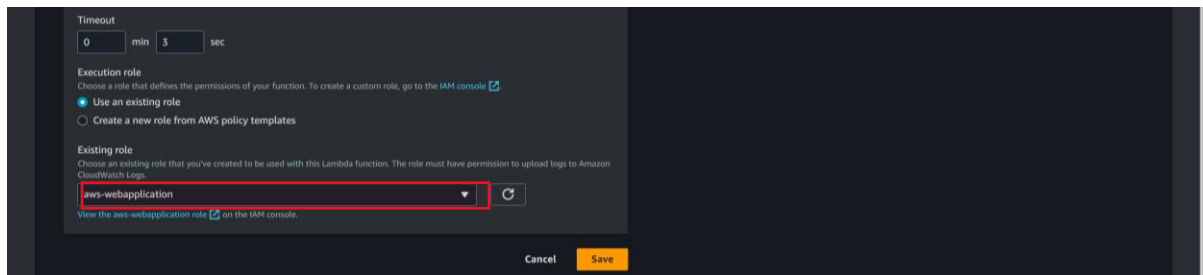
    return views
```

Make sure to replace the table name with your DynamoDB table name



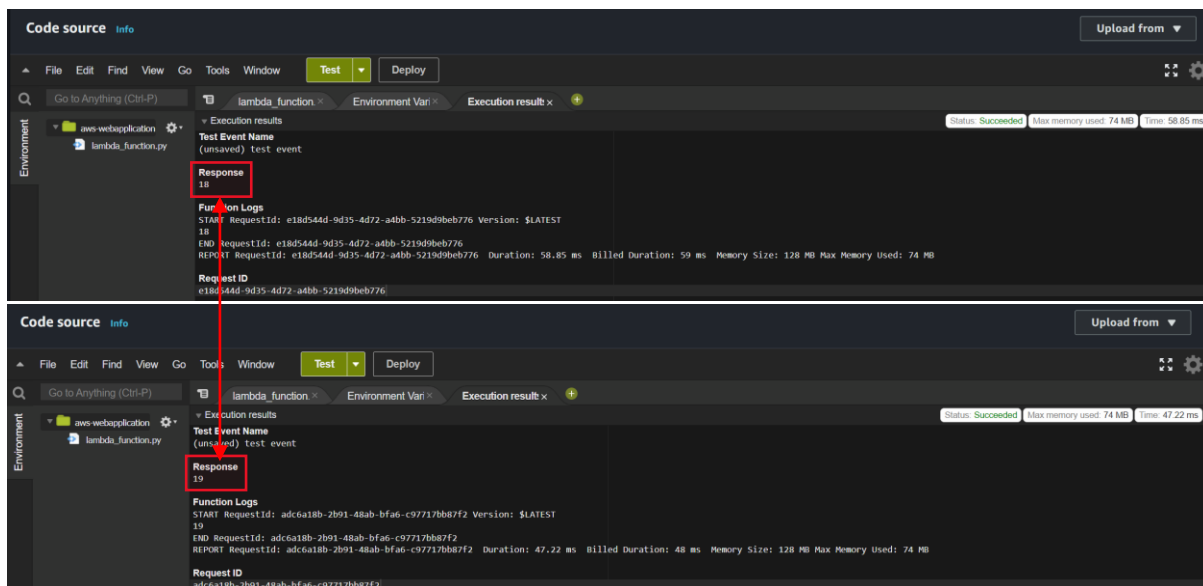


Go to Configuration and then go to the Permission tab



Select the role that you created earlier in IAM and save

Now deploy and test the code -



Took multiple tests and successfully got a response

Now edit the script.js file and re-upload to the S3 bucket -

```
const form = document.querySelector('form');
const greeting = document.querySelector('#greeting');

form.addEventListener('submit', (event) => {
  event.preventDefault();
  const name = document.querySelector('#name').value;
  greeting.textContent = `Hello, ${name}!`;
});

const counter = document.querySelector(".counter-number");
async function updateCounter() {
  let response = await fetch(
    "https://e6ef42creugwuhmqzd24ldkzwa0tzibz.lambda-url.us-east-1.on.aws/"
  );
  let data = await response.json();
  counter.innerHTML = `Views: ${data}`;
}
updateCounter();
```

Update the link with function URL, which can be found on Lambda main page

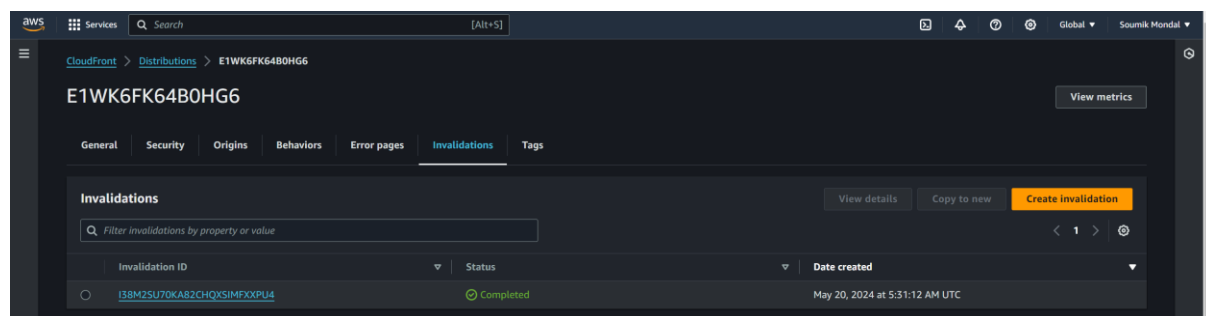
Greeting App
Views: 9
Name:

Submit

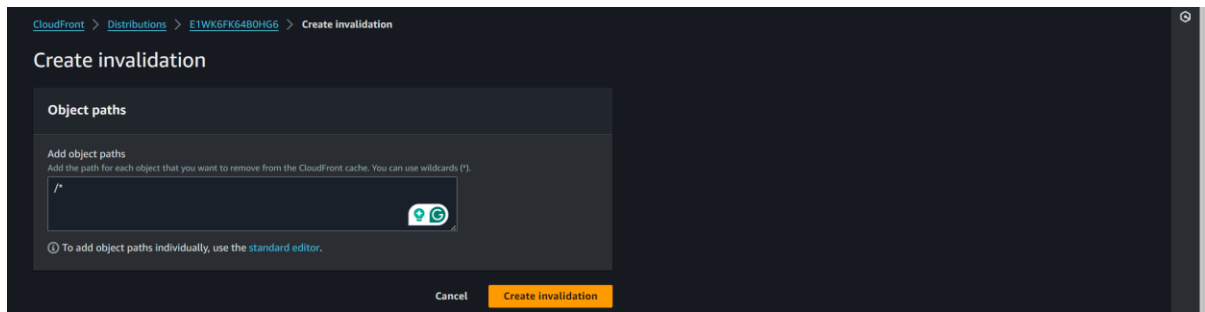
Successfully Fetched the view count, Now every time we refresh the page, the view count will gradually increase

Troubleshoot –

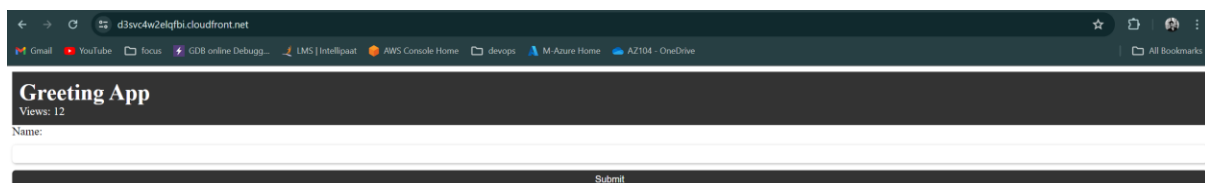
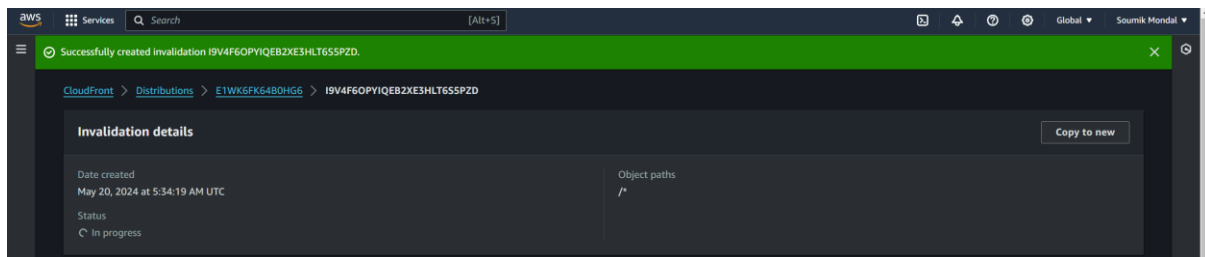
If the view count is not working, we have to create an invalidation and delete the cache of all edge locations.



Go to the invalidations tab.



Give “/*” delete all cache in all edge locations



Now, we can access the webpage and view count seamlessly.

N.B. - After this demo, make sure to delete all resources

-----completed-----