**Internet of Everything**

**Continuous Assessment - 2**

**Name: Shashwat Tripathi Division: D20A Roll no: 64**

**CA2:** Demonstrate the Implementation of Edge to cloud Protocols (Minimum 3) using a dummy data set.

1. Explain in brief each of the 3 protocols

2. Code

3. Output

Edge-to-cloud protocols are communication protocols designed to enable the transmission of data between edge devices and cloud servers. These protocols facilitate the movement of data from where it is generated or collected to centralized cloud platforms for processing, storage, and further analysis.

Edge-to-cloud protocols bridge the gap between these two environments by ensuring data can be reliably, securely, and efficiently transmitted from edge devices to cloud platforms. These protocols often need to account for bandwidth constraints, intermittent network connectivity, and resource-limited devices.

Few of the Edge-to-cloud protocols implemented are as follows:

1. **MQTT(Message Queuing Telemetry Transport):**

MQTT is a lightweight, publish/subscribe messaging protocol designed for resource-constrained devices and unreliable networks. It’s ideal for IoT devices where bandwidth is limited, and reliable, low-latency communication is required.

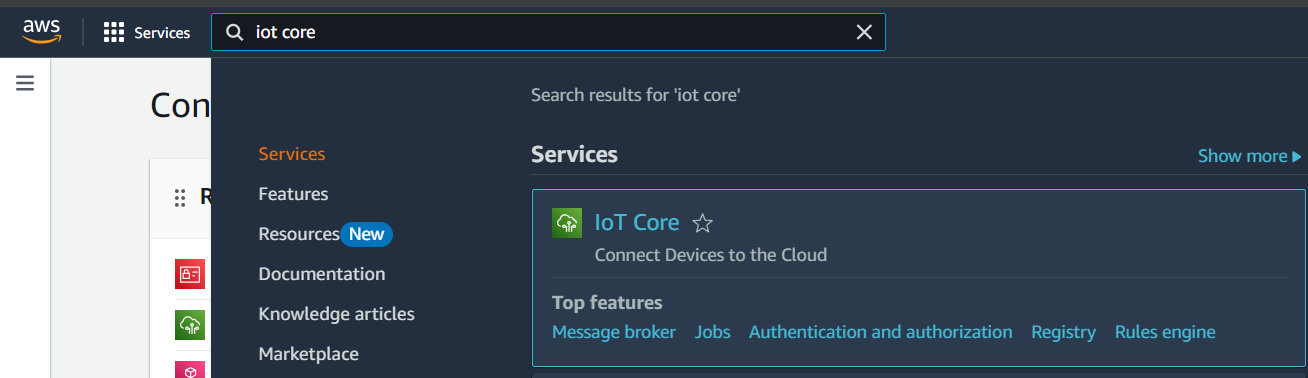
#### **Key Features**

1. Low bandwidth usage
2. Supports real-time data transmission
3. Quality of Service (QoS) levels to ensure message delivery
4. Pub/Sub architecture

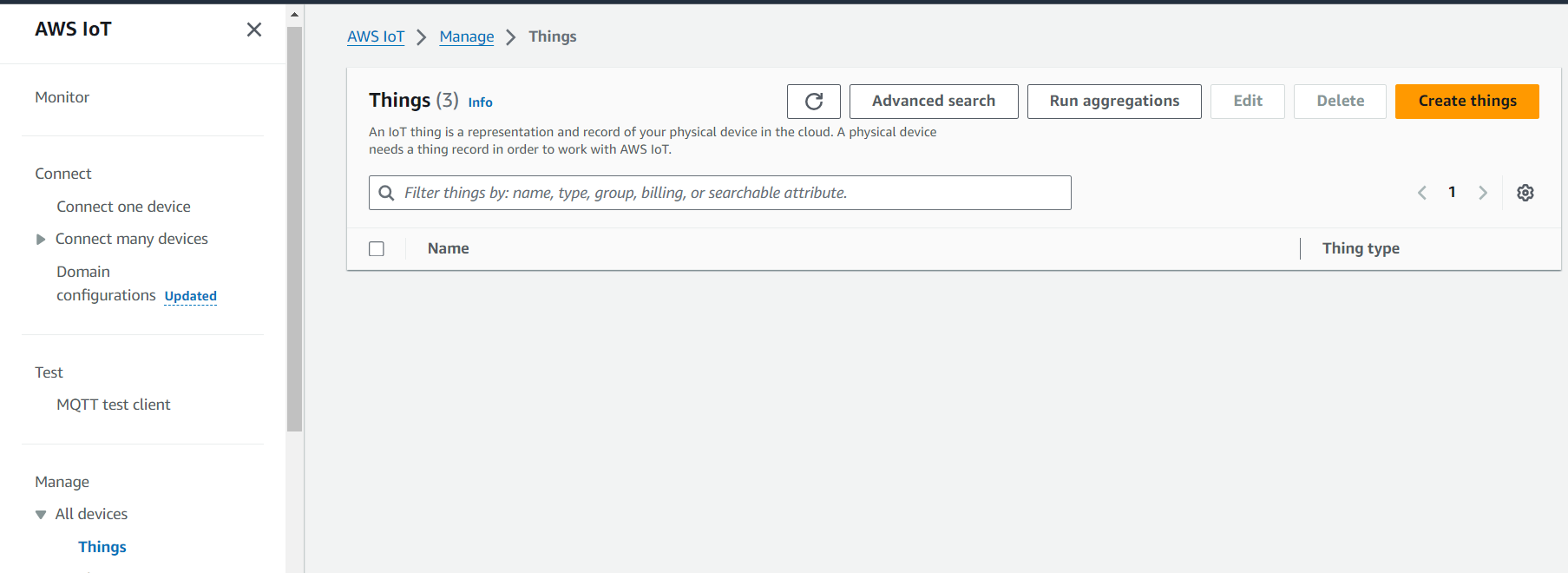
**Implementation:**

Step 1: Create a thing in AWS IoT Core

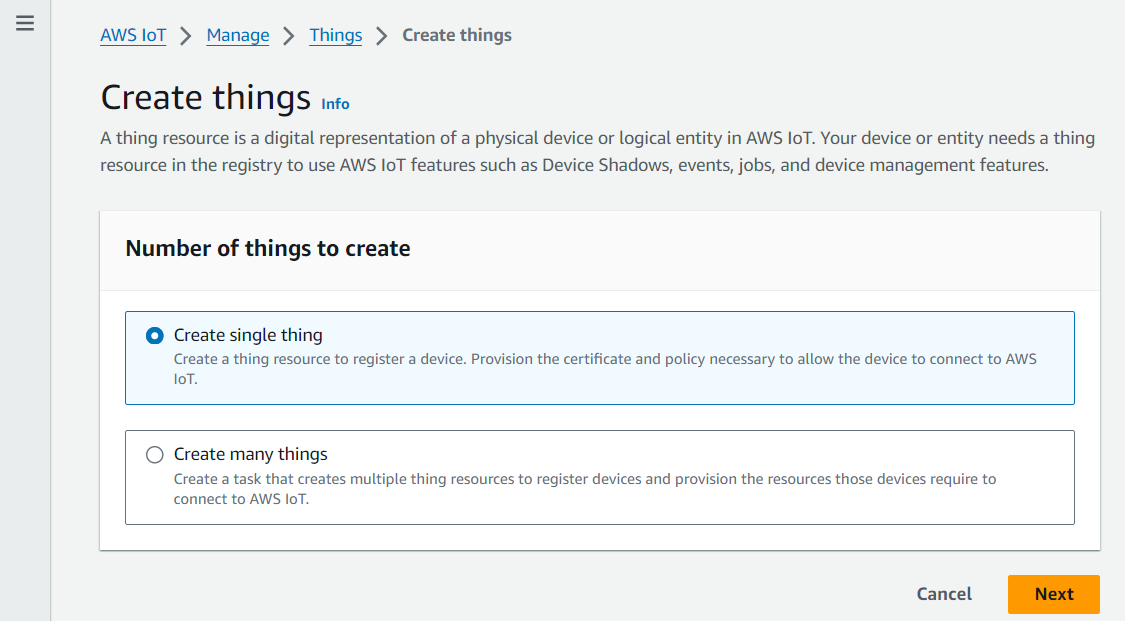
* Search for IoT Core on your AWS account



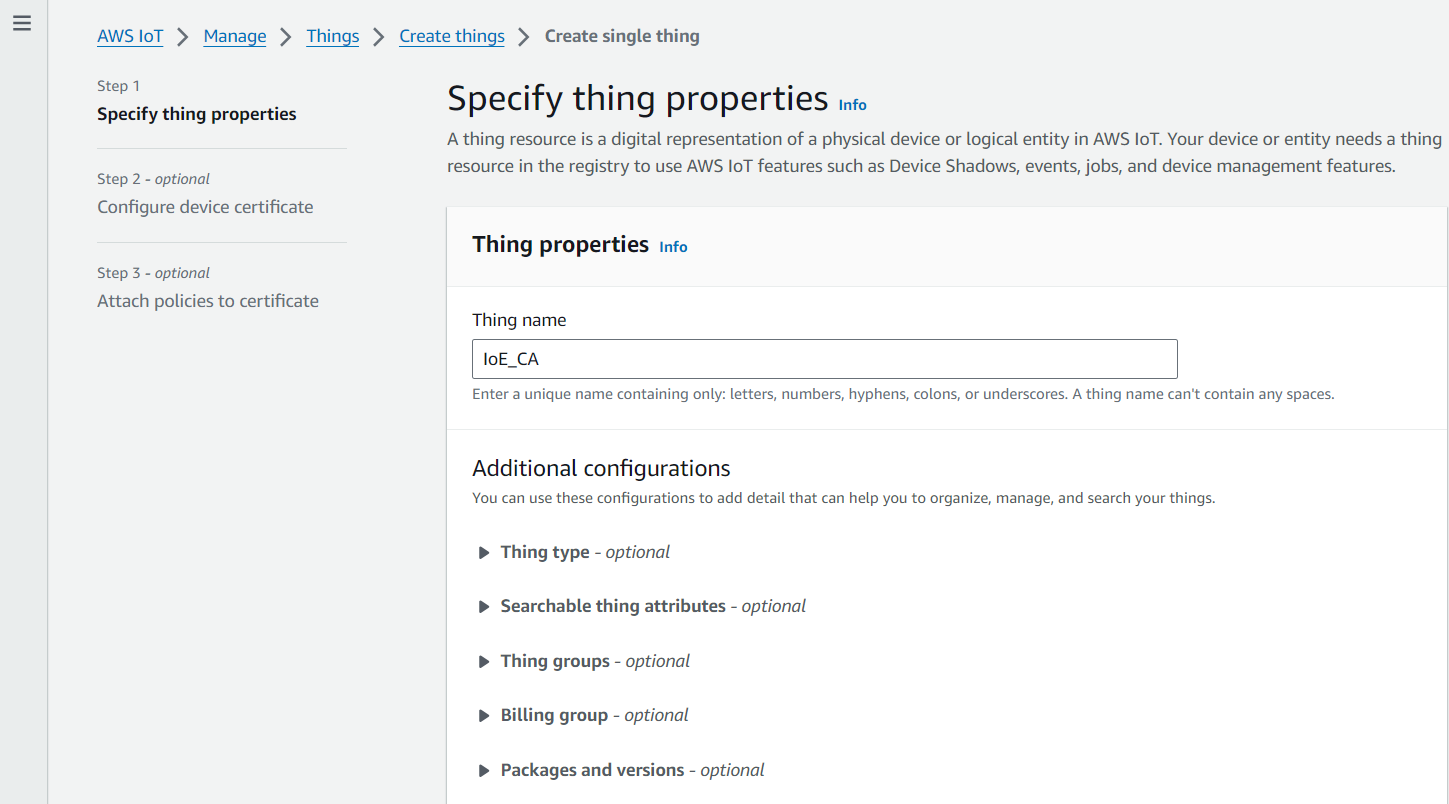
* Manage > All devices > Things > Create things



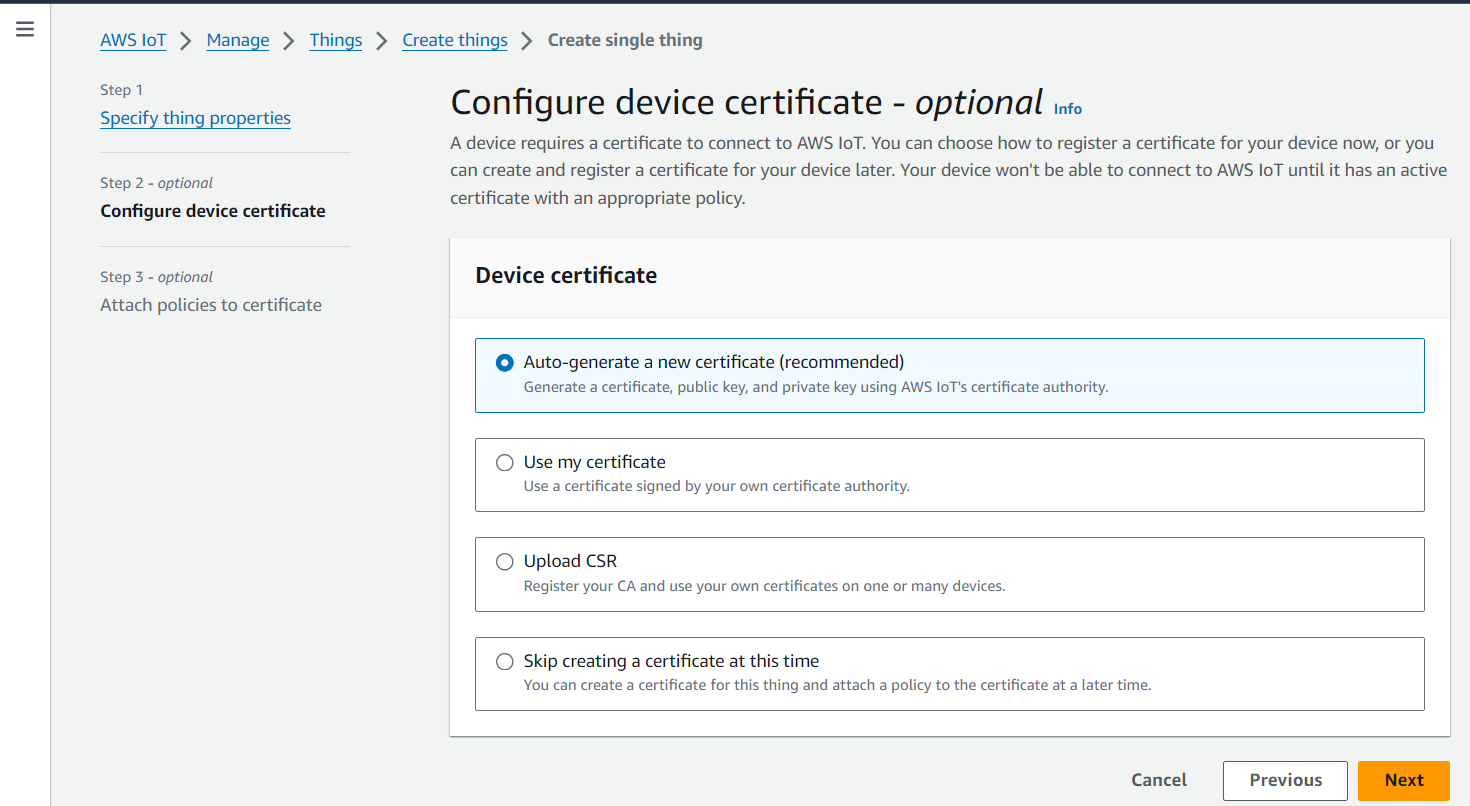
* Create a single thing



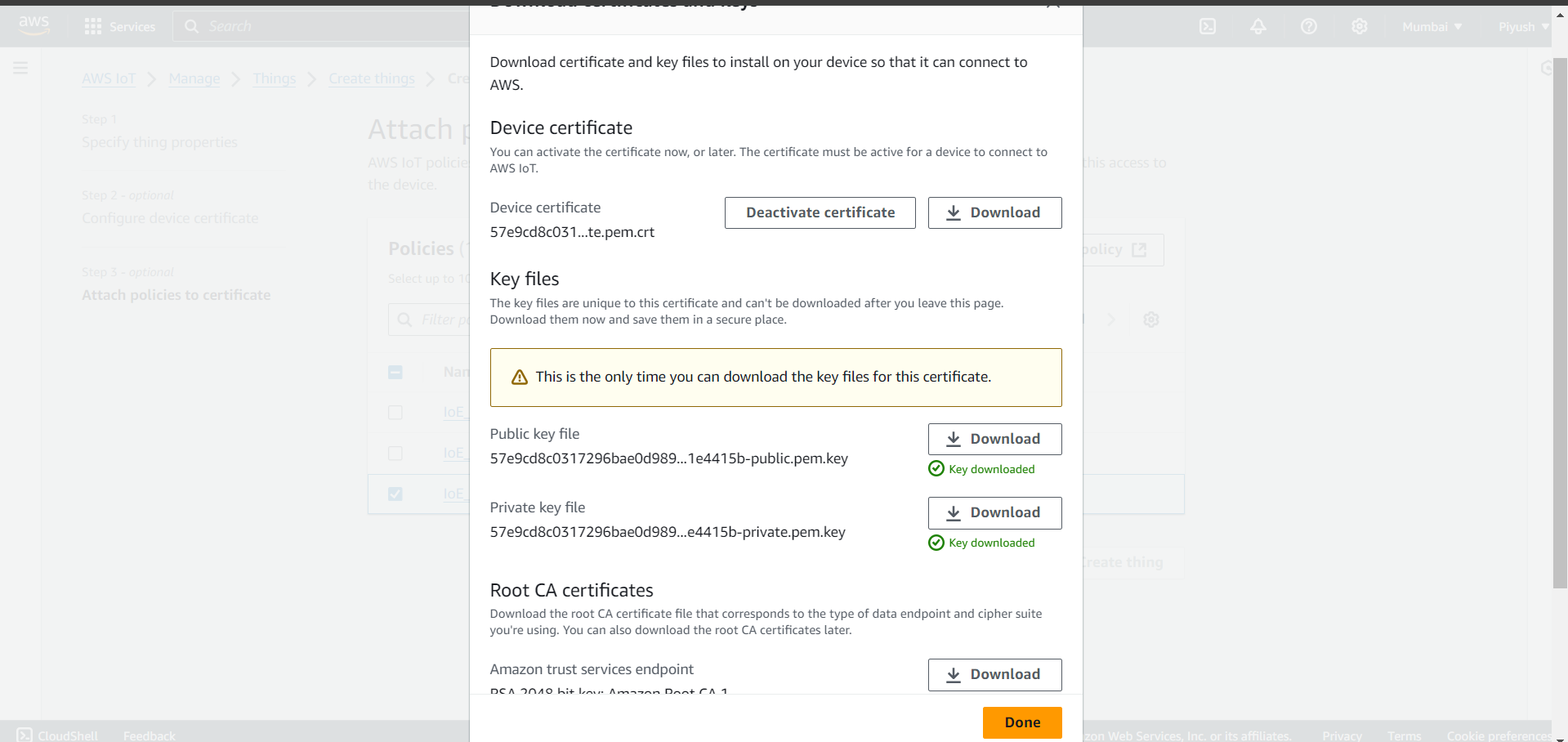
* Give a name to your thing



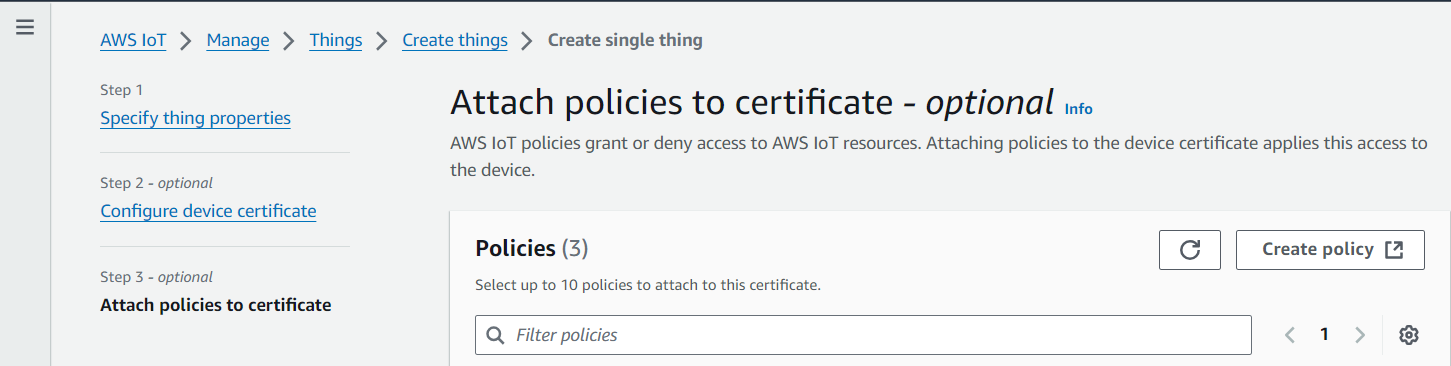
* Auto generate certificates



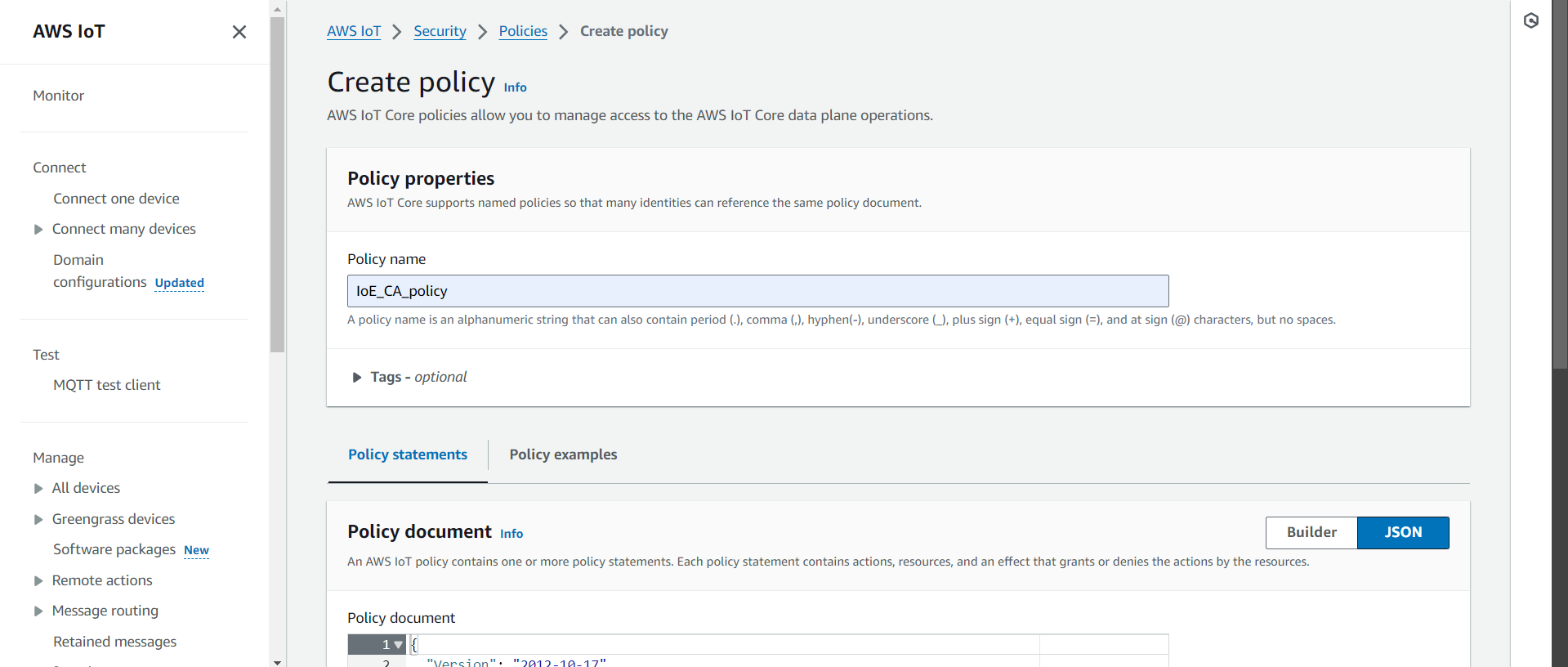
* Download your certificates



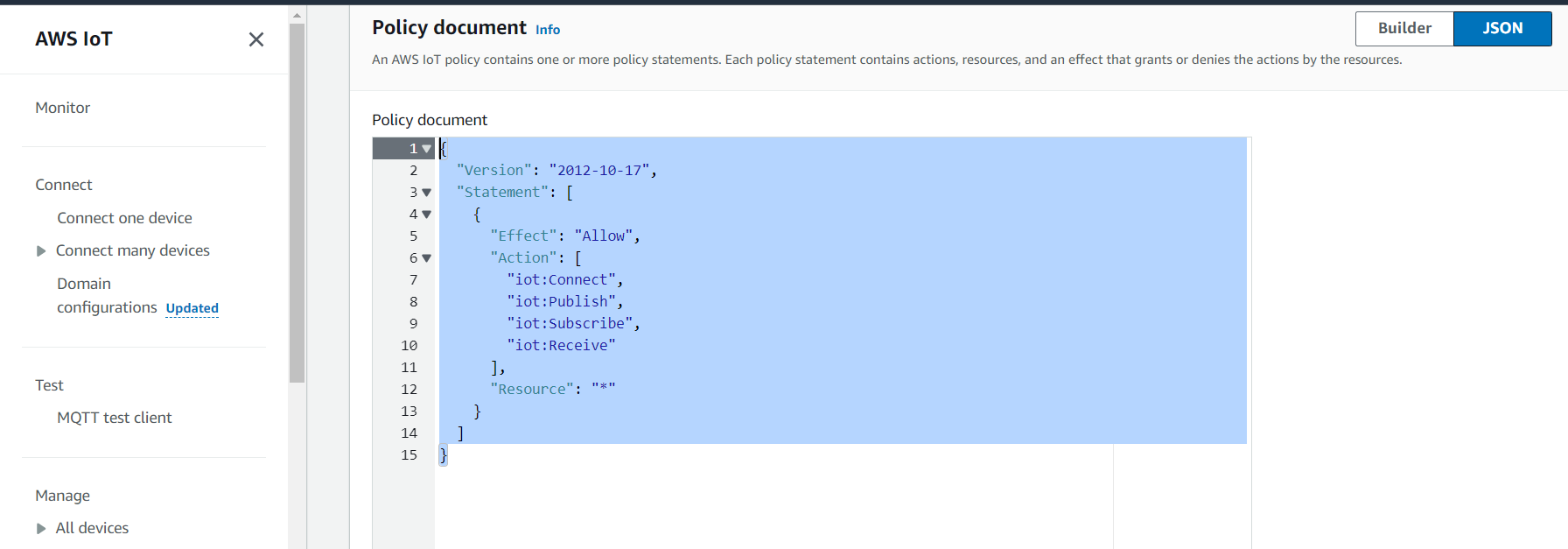
* Create a new policy for your thing



* Give a name to your policy



* Add the following policy to allow Publish, Subscribe and Receive from this IoT thing



Policy:

{

"Version": "2012-10-17",

"Statement": [

{

"Effect": "Allow",

"Action": [

"iot:Connect",

"iot:Publish",

"iot:Subscribe",

"iot:Receive"

],

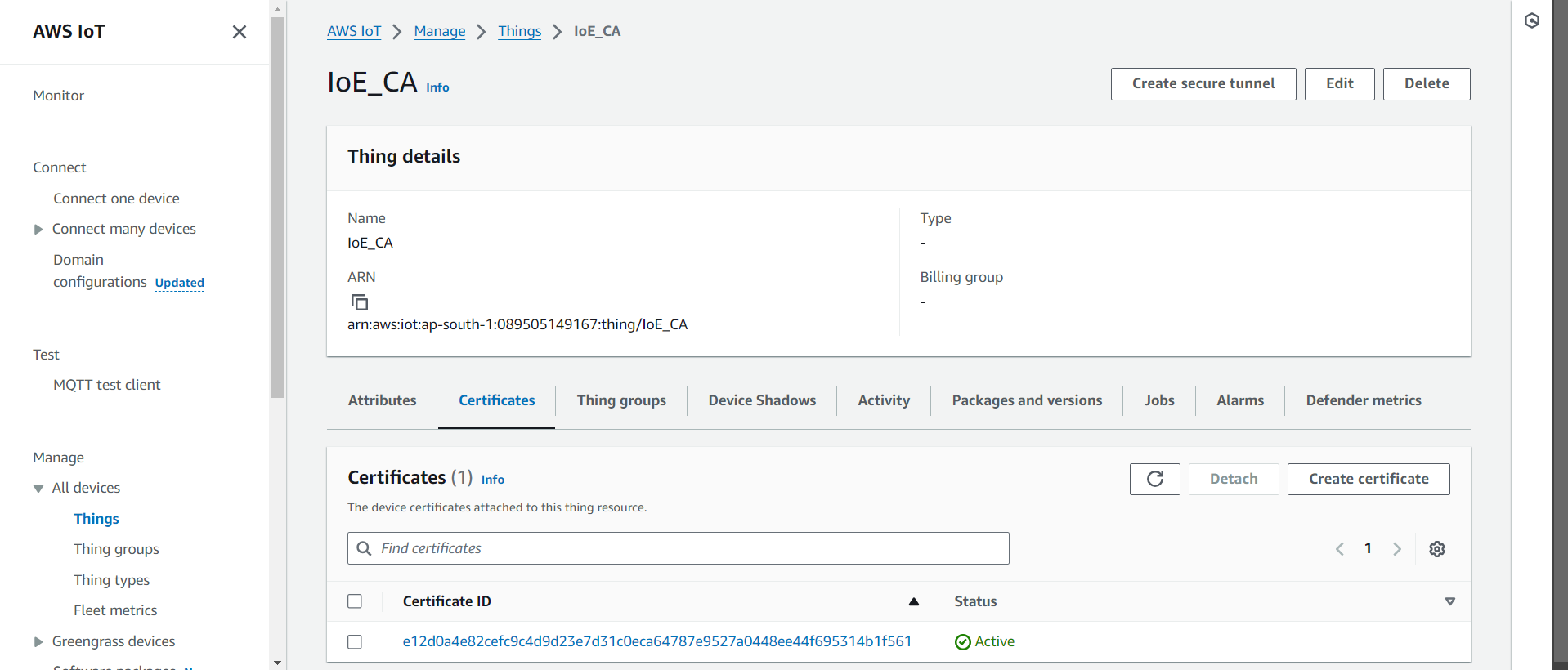
"Resource": "\*"

}

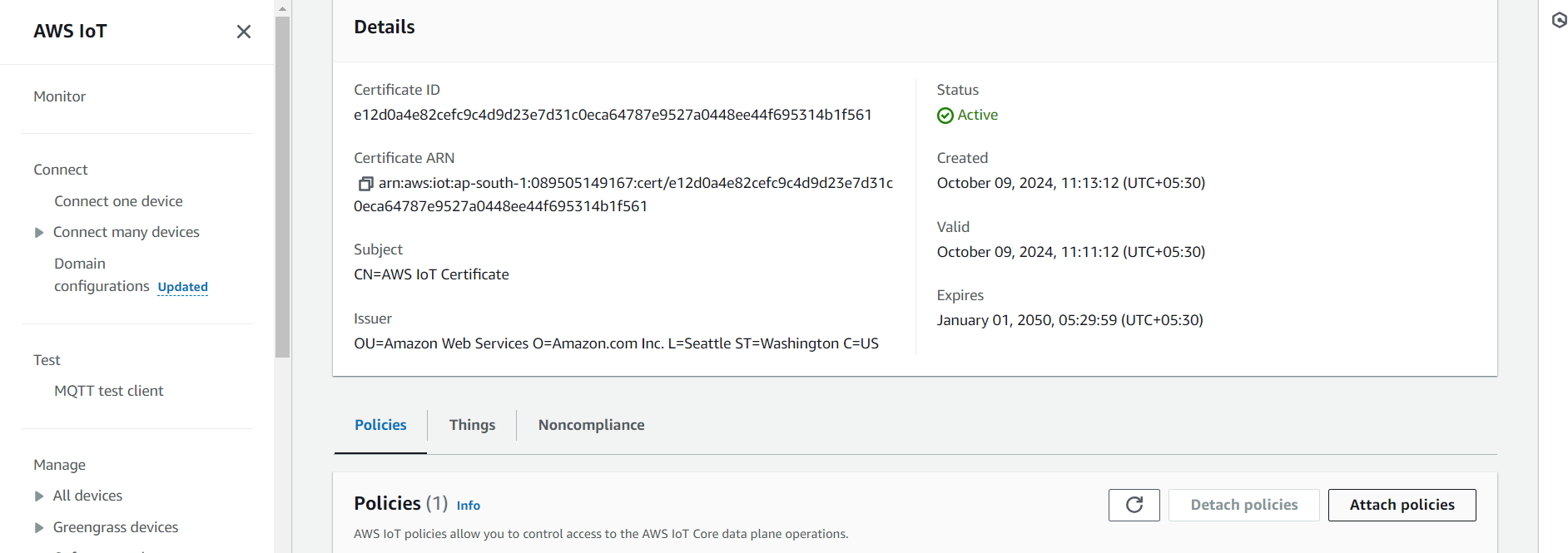
]

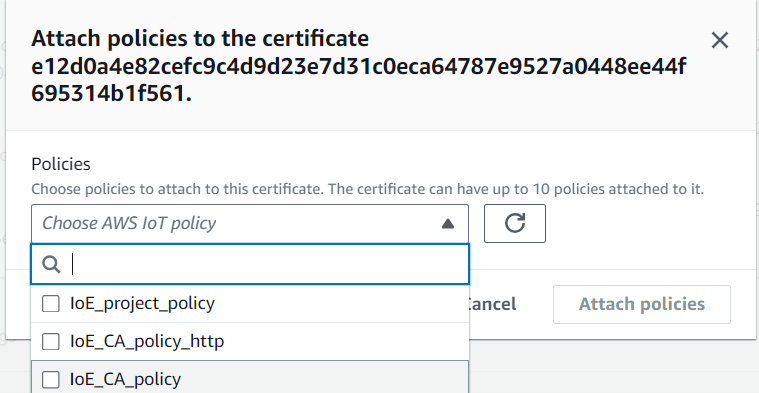
}

* Now we need to attach the policy that we created to our IoT thing



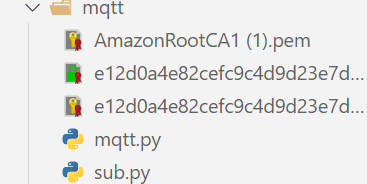
* Attach the policy that we just created



****

Now we have the AWS IoT thing created and rightly configured

**Step 2:** Create python file to generate dummy data values and send it to AWS



In our folder, we have the following files

* mqtt.pt: To publish data to AWS
* sub.py: To subscribe to a publisher
* 3 key files that have our private key, device certificate and AWS root CA to send data from python file to AWS

**Code:**

mqtt.py

import paho.mqtt.client as mqtt

import ssl

import random

import time

import json

# AWS IoT MQTT broker details

mqtt\_broker = "a1v291oedb7que-ats.iot.ap-south-1.amazonaws.com"

mqtt\_port = 8883 # Secure port for TLS communication

mqtt\_topic = "sensors/data" # The topic to publish sensor data

# Paths to certificate and key files

ca\_cert = "./AmazonRootCA1 (1).pem"

certfile = "./e12d0a4e82cefc9c4d9d23e7d31c0eca64787e9527a0448ee44f695314b1f561-certificate.pem.crt"

keyfile = "./e12d0a4e82cefc9c4d9d23e7d31c0eca64787e9527a0448ee44f695314b1f561-private.pem.key"

# Callback when the client connects to the broker

def on\_connect(client, userdata, flags, rc):

print(f"Connected to AWS IoT broker with result code {rc}")

# Function to publish sensor data (simulated)

def publish\_data(client):

while True:

gas = random.uniform(100, 500) # Simulated gas sensor value

flame = random.uniform(0, 100) # Simulated flame sensor value

light = random.uniform(0, 1023) # Simulated LDR sensor value

# Create JSON payload

data = {

"gas": gas,

"flame": flame,

"light": light

}

data\_json = json.dumps(data)

# Publish data to AWS IoT Core

client.publish(mqtt\_topic, data\_json)

print(f"Published data: {data\_json}")

# Wait before sending the next data point

time.sleep(5)

# Create an MQTT client

client = mqtt.Client()

# Assign the on\_connect callback function

client.on\_connect = on\_connect

# Configure TLS/SSL settings using the certificates

client.tls\_set(ca\_certs=ca\_cert, certfile=certfile, keyfile=keyfile, tls\_version=ssl.PROTOCOL\_TLSv1\_2)

# Connect to AWS IoT Core

client.connect(mqtt\_broker, mqtt\_port, 60)

# Start the MQTT loop in a separate thread

client.loop\_start()

# Start publishing data

publish\_data(client)

**sub.py**

import paho.mqtt.client as mqtt

import ssl

import json

# AWS IoT MQTT broker details

mqtt\_broker = "a1v291oedb7que-ats.iot.ap-south-1.amazonaws.com"

mqtt\_port = 8883 # Secure port for TLS communication

mqtt\_topic = "sensors/data" # The topic to subscribe to

# Paths to certificate and key files

ca\_cert = "./AmazonRootCA1 (1).pem"

certfile = "./e12d0a4e82cefc9c4d9d23e7d31c0eca64787e9527a0448ee44f695314b1f561-certificate.pem.crt"

keyfile = "./e12d0a4e82cefc9c4d9d23e7d31c0eca64787e9527a0448ee44f695314b1f561-private.pem.key"

# Callback when the client connects to the broker

def on\_connect(client, userdata, flags, rc):

if rc == 0:

print("Connected to AWS IoT broker successfully.")

# Subscribe to the MQTT topic

client.subscribe(mqtt\_topic)

else:

print(f"Failed to connect, return code {rc}")

# Callback when a message is received

def on\_message(client, userdata, msg):

try:

# Decode the message payload and load as JSON

payload = msg.payload.decode('utf-8')

data = json.loads(payload)

print(f"Received data: {data}")

except json.JSONDecodeError as e:

print(f"Error decoding JSON: {e}")

# Create an MQTT client

client = mqtt.Client()

# Assign the on\_connect and on\_message callback functions

client.on\_connect = on\_connect

client.on\_message = on\_message

# Configure TLS/SSL settings using the certificates

client.tls\_set(ca\_certs=ca\_cert, certfile=certfile, keyfile=keyfile, tls\_version=ssl.PROTOCOL\_TLSv1\_2)

# Connect to AWS IoT Core

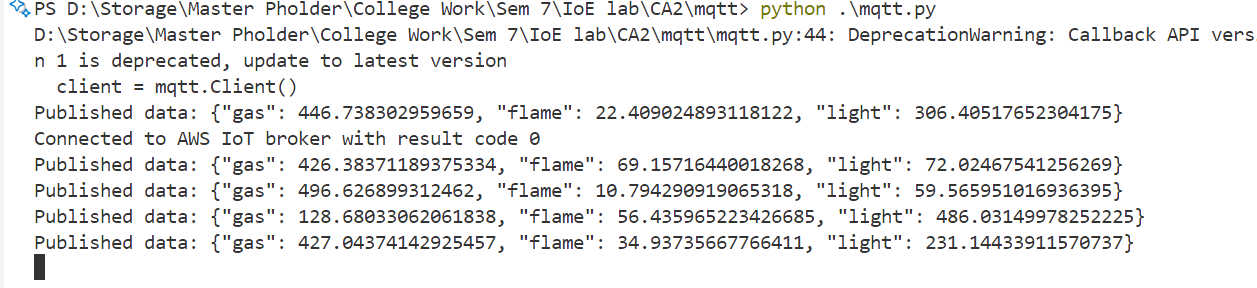
client.connect(mqtt\_broker, mqtt\_port, 60)

# Start the MQTT loop to handle incoming and outgoing messages

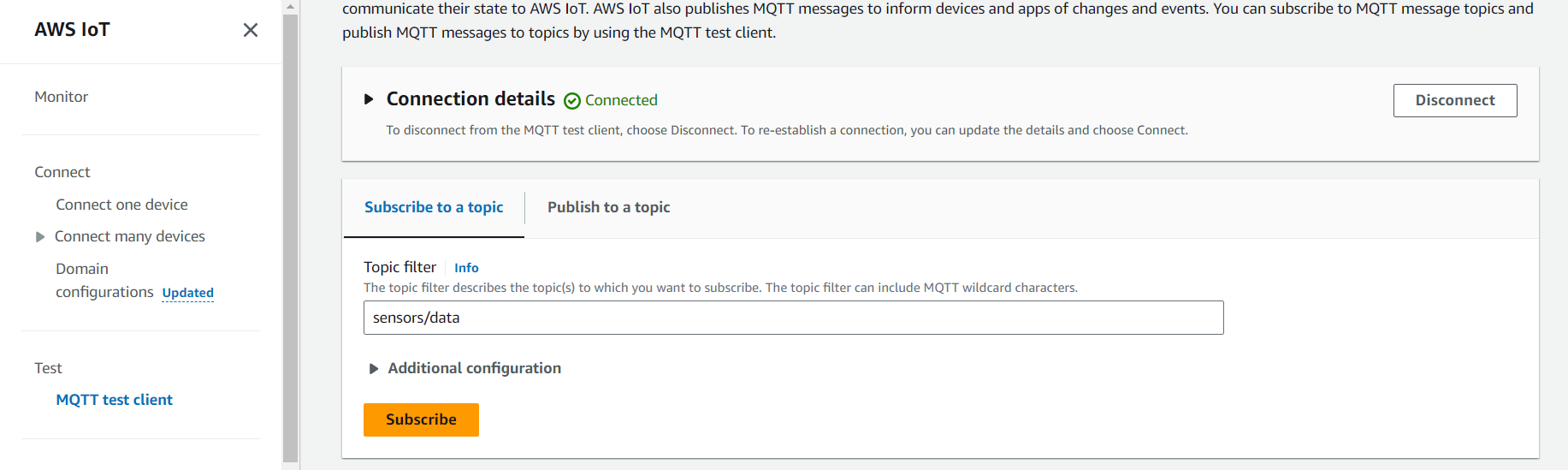
client.loop\_forever()

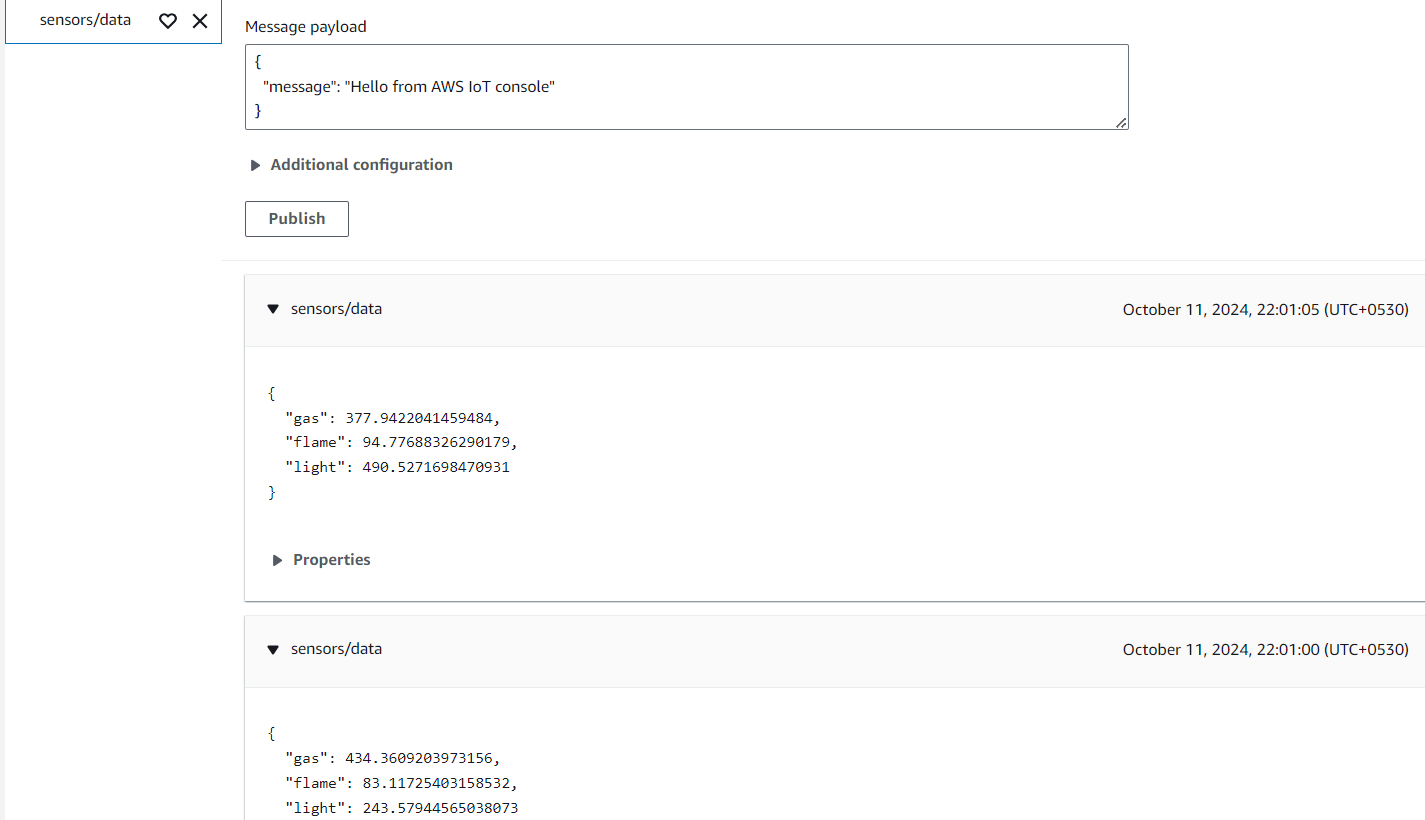
**Step 3: Execution:**

* Run the mqtt.py file



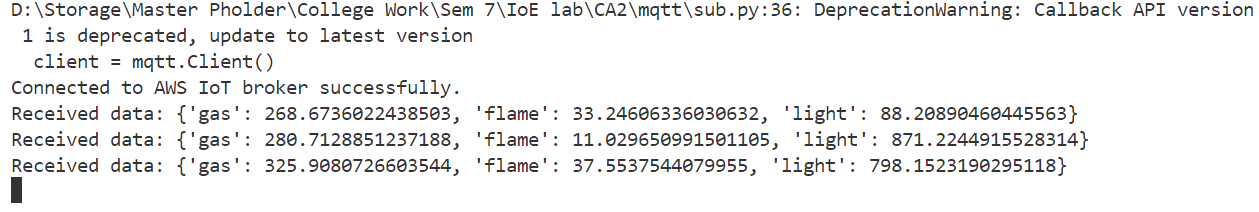
* On AWS, subscribe to the topic name that is in mqtt.py file (sensors/data)

****

****

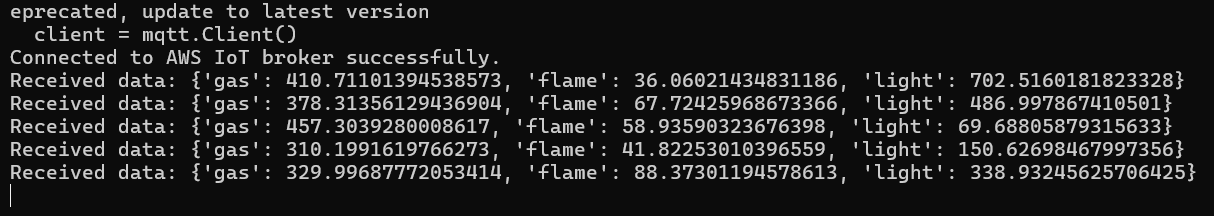
MQTT data POST

* To execute GET on MQTT execute the sub.py file



MQTT data GET

* For ,multiple subscribers, execute the same sub.py file on another device or another terminal



MQTT multiple subscribers

**Thus, MQTT protocol has been implemented**

1. **HTTP(Hypertext Transfer Protocol)**

HTTPis a widely used stateless protocol for transmitting hypermedia documents and data over the internet. While it’s not as lightweight as MQTT, it’s commonly used in IoT applications for sending data from devices to cloud services in a RESTful manner. It uses the request-response architecture for exchanging the data.

#### **Key Features**

* Well-established and easy to implement
* Works well for transferring larger datasets
* Secure (HTTPS) for encrypted communication
* Used in REST APIs

**Implementation:**

* **Step 1: Create and setup an IoT thing on AWS**

We will use the same thing we created for MQTT as the policies required are the same

* **Step 2: Create an HTML file for interacting with AWS IoT thing**

**Code:**

index.html

<!DOCTYPE html>

<html>

<head>

<title>AWS IoT Publish Example with Dummy GET</title>

<script src="https://sdk.amazonaws.com/js/aws-sdk-2.1037.0.min.js"></script>

</head>

<body>

<h1>Publish to AWS IoT and Simulate GET Request</h1>

<!-- Button to publish data -->

<button onclick="publishData()">POST and GET</button>

<!-- Div to display the data -->

<div id="iotDataDisplay"></div>

<script>

// Configure AWS

AWS.config.region = 'ap-south-1'; // Your AWS region

AWS.config.credentials = new AWS.CognitoIdentityCredentials({

IdentityPoolId: 'ap-south-1:aa8d847e-ffab-434c-9d33-f4c66e26794c' // Your Identity Pool ID

});

const iotData = new AWS.IotData({ endpoint: 'a1v291oedb7que-ats.iot.ap-south-1.amazonaws.com' }); // Your IoT endpoint

// Function to simulate the GET request after publishing data

function simulateGetRequest(flame, gas, timestamp) {

// Simulate a GET request (in reality, you're just displaying the same data you sent)

setTimeout(() => {

console.log('GET request');

// Update the DOM with the retrieved data

document.getElementById('iotDataDisplay').innerHTML =

`GET Data:<br>

Flame: ${flame}<br>

Gas: ${gas}<br>

Timestamp: ${timestamp}`;

}, 1000); // Simulate some delay for the GET request

}

// Function to publish data (POST)

function publishData() {

const flame = Math.floor(Math.random() \* 51); // Random value between 0 and 50

const gas = Math.round(Math.random());

const timestamp = new Date().toISOString();

const params = {

topic: 'iot/dummy/data',

payload: JSON.stringify({

flame: flame,

gas: gas,

timestamp: timestamp

}),

qos: 0

};

iotData.publish(params, function (err, data) {

if (err) {

console.log('Error publishing:', err);

} else {

console.log('Message published:', data);

// Immediately simulate a GET request after publishing

simulateGetRequest(flame, gas, timestamp);

}

});

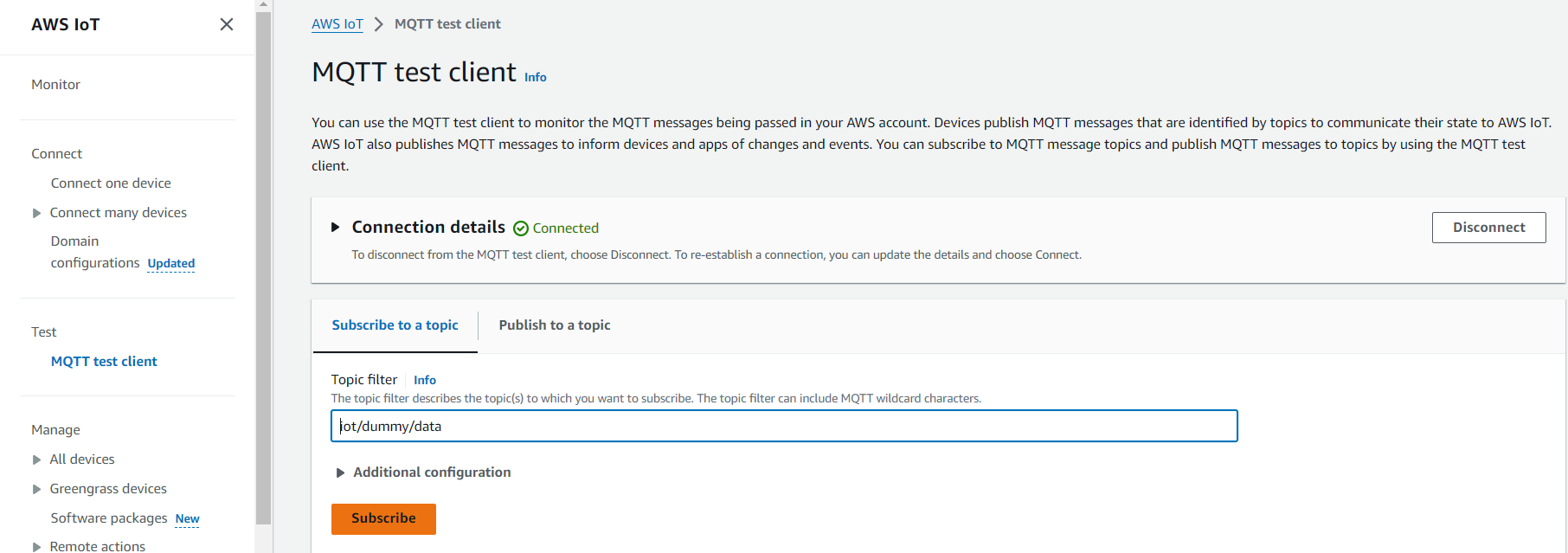
}

</script>

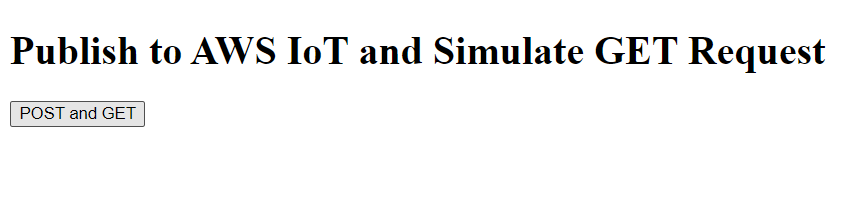
</body>

</html>

* In the index.html file, the highlighted part should contain our AWS credentials (includes: IoT endpoint, IndentityPoolId, region, topic)
* On AWS, subscribe to the topic that is highlighted in the index.html file(iot/dummy/data)

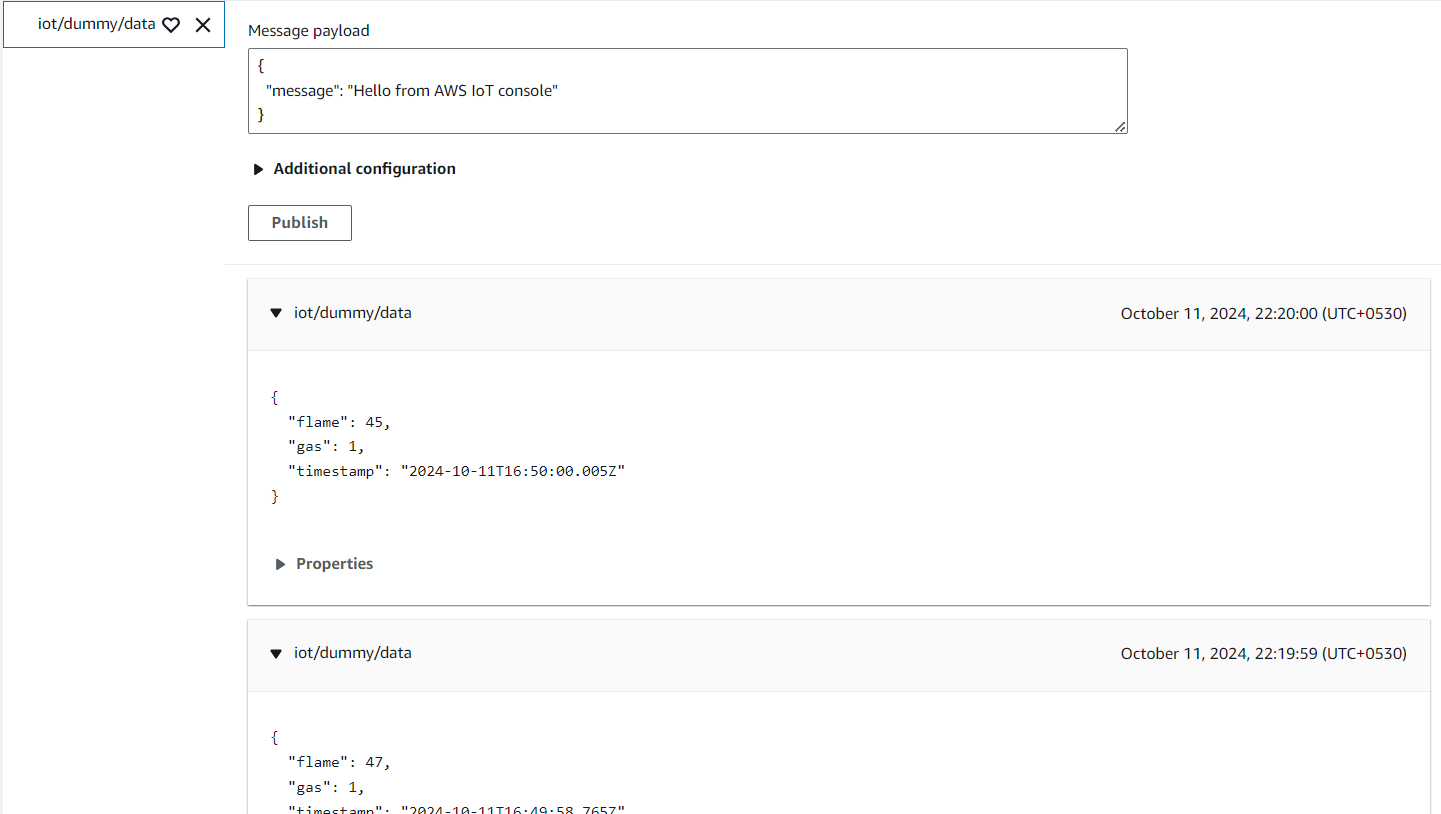


* Open the index.html file on live server

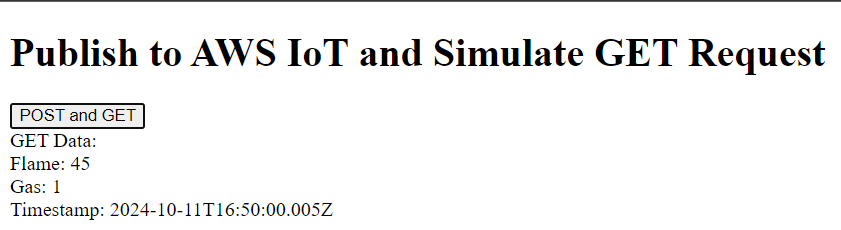


* Click on the POST and GET button to send data to the AWS IoT thing and receive





http POST



http GET

**Thus, HTTP protocol has been implemented**

1. **WebSocket**

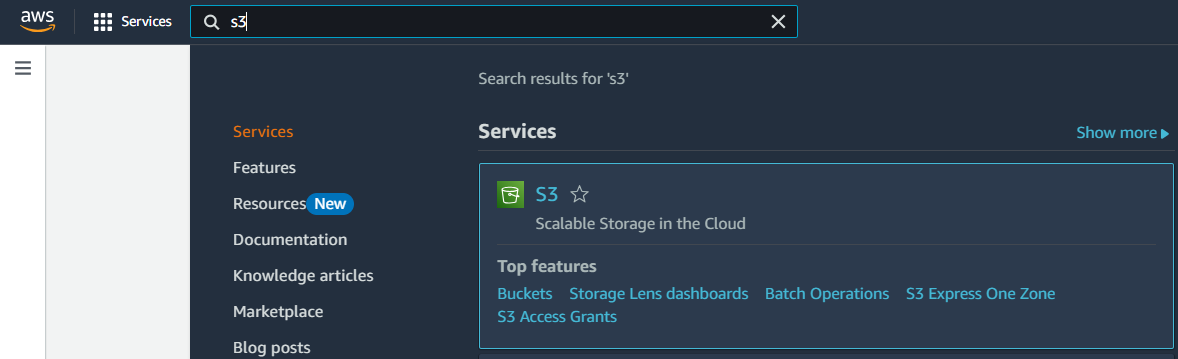
WebSocket is a full-duplex communication protocol that operates over a single, long-lived TCP connection. It allows real-time, bidirectional data exchange between edge devices and cloud servers, making it ideal for scenarios where continuous communication and low latency are required. Unlike HTTP, WebSocket enables both the client and server to send messages independently once the connection is established.

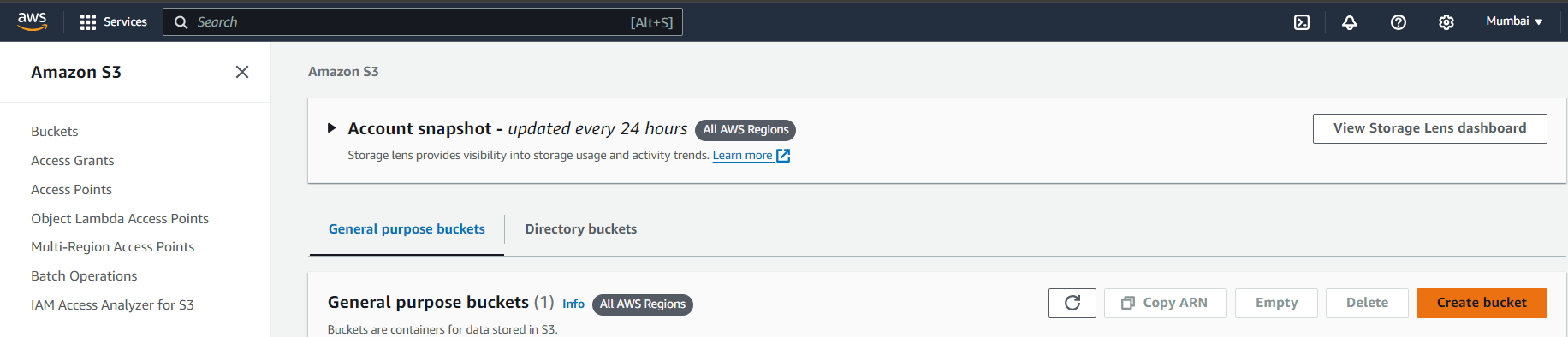
#### **Key Features**

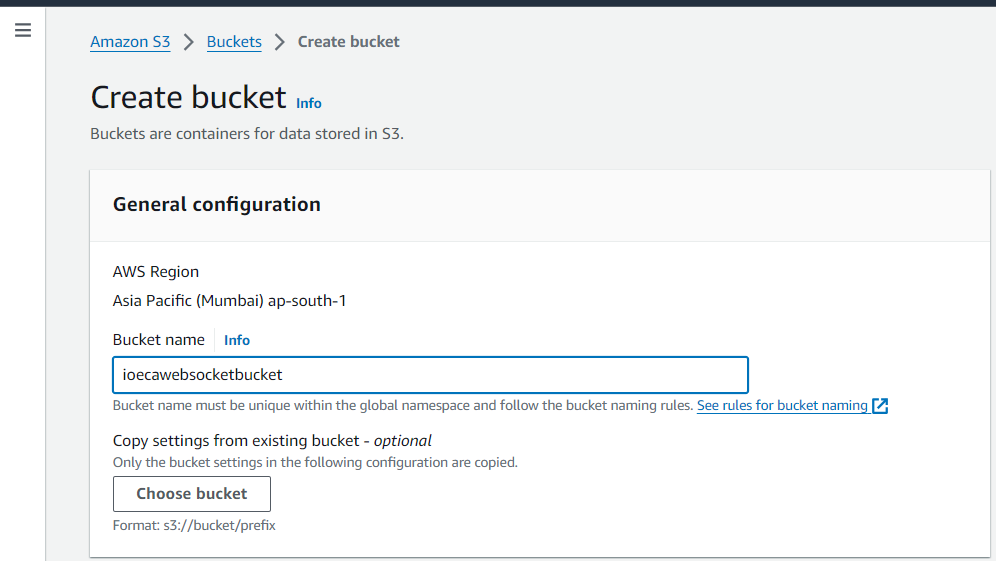
* Full-duplex communication over a single TCP connection
* Low latency and minimal overhead compared to HTTP/HTTPS
* Persistent connection, reducing the need for frequent handshakes
* Suitable for real-time applications such as live updates and streaming

**Implementation:**

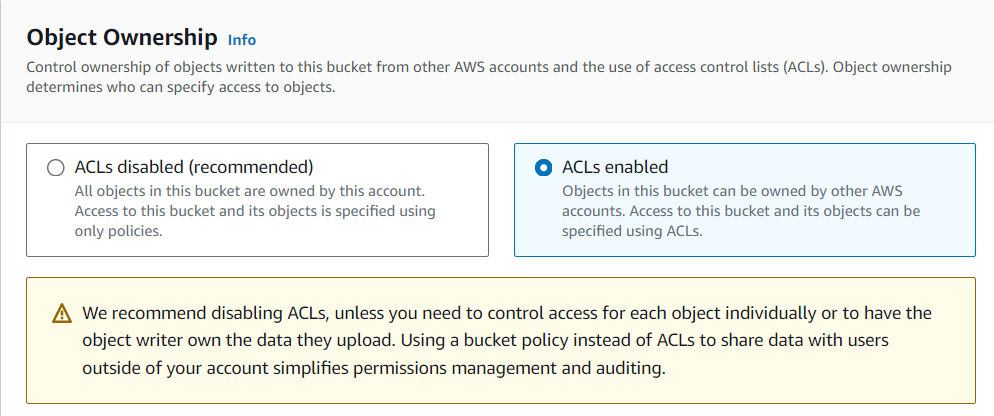
**Step 1: Create a public S3 bucket on AWS**



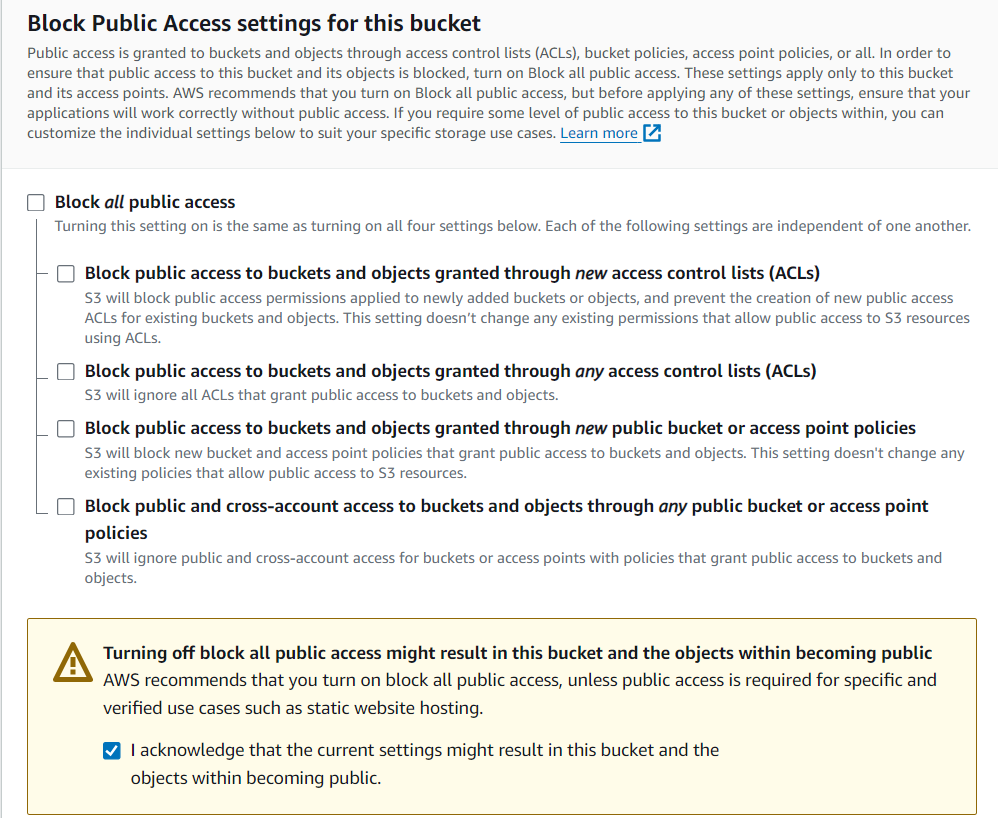




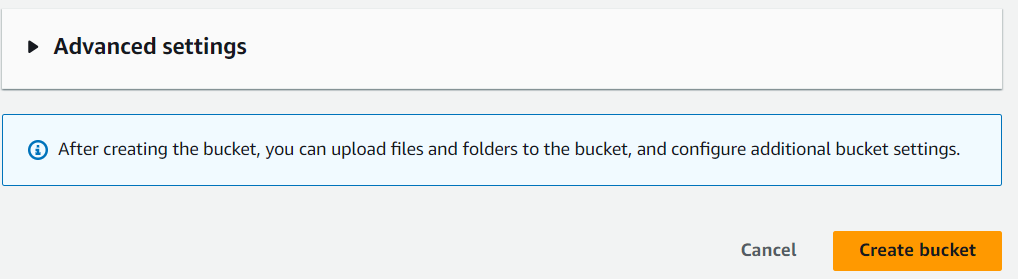
* Set Object ownership



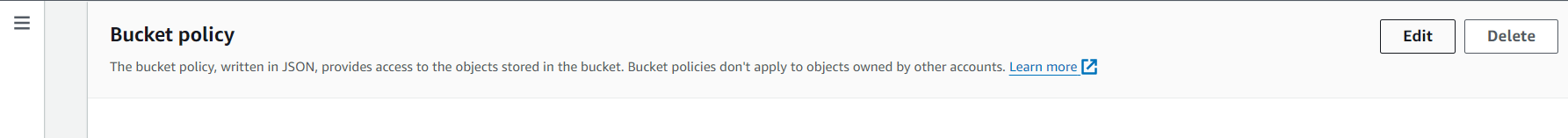
* Allow public access



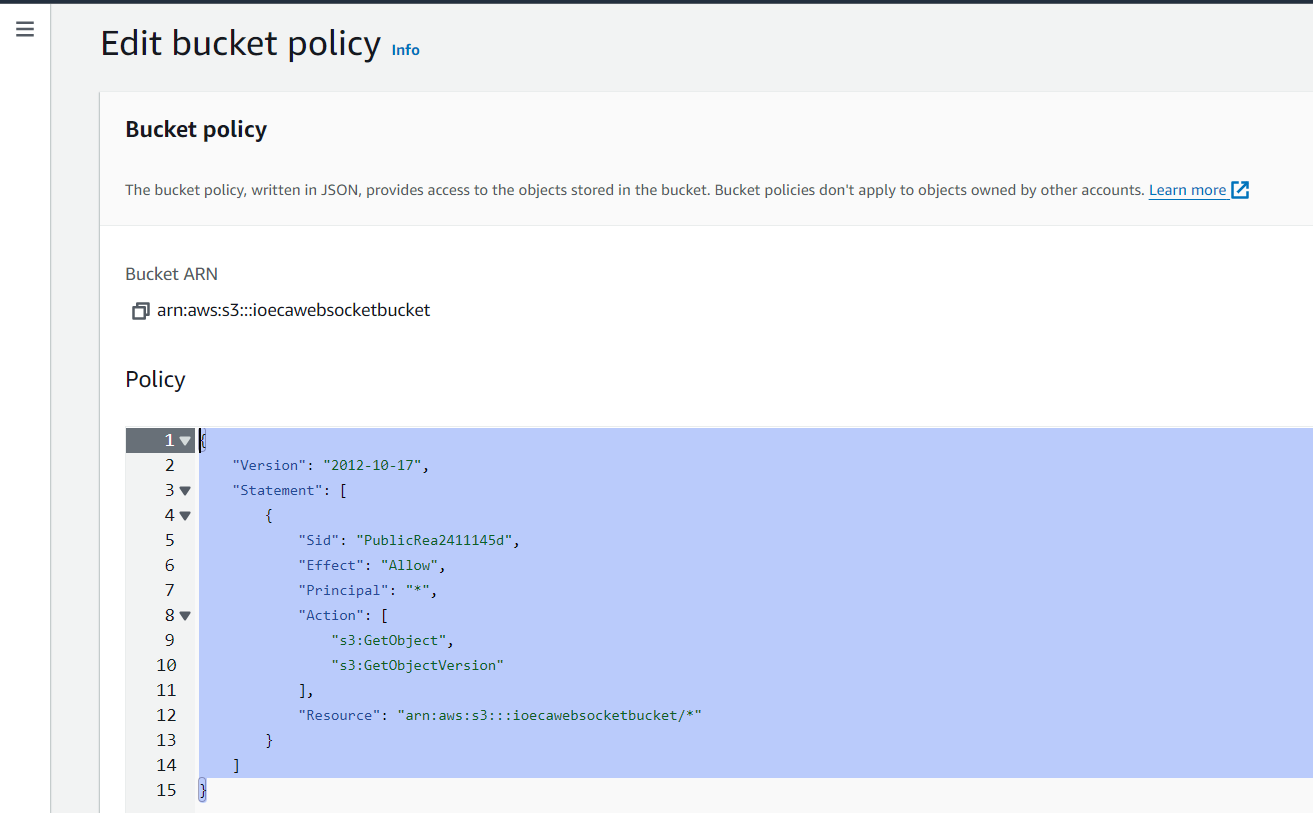
* Create bucket



* Now go to Amazon S3 > Buckets > ioecawebsocketbucket (the bucket that I created)
* Look for the permissions tab, and in Bucket policy, choose edit option



* Edit the bucket policy to include GetObject and GetObjectVersion actions



policy:

{

"Version": "2012-10-17",

"Statement": [

{

"Sid": "PublicRea2411145d",

"Effect": "Allow",

"Principal": "\*",

"Action": [

"s3:GetObject",

"s3:GetObjectVersion"

],

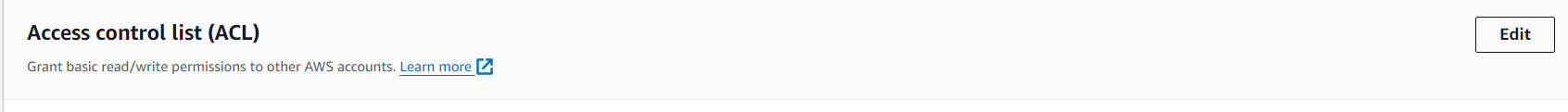
"Resource": "arn:aws:s3:::ioecawebsocketbucket/\*"

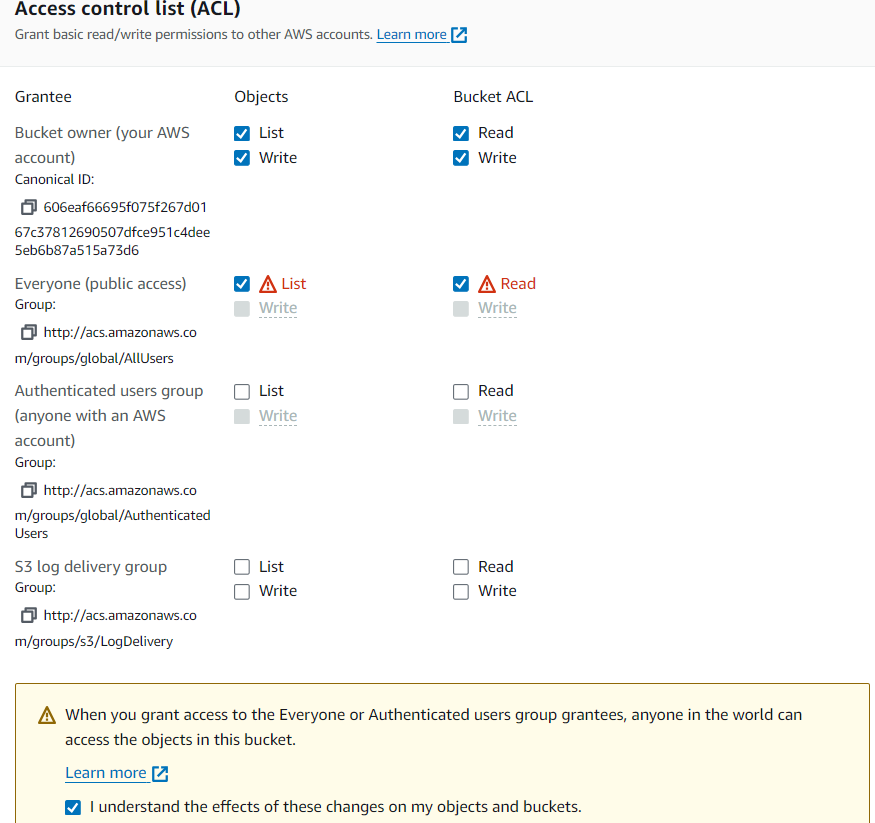
}

]

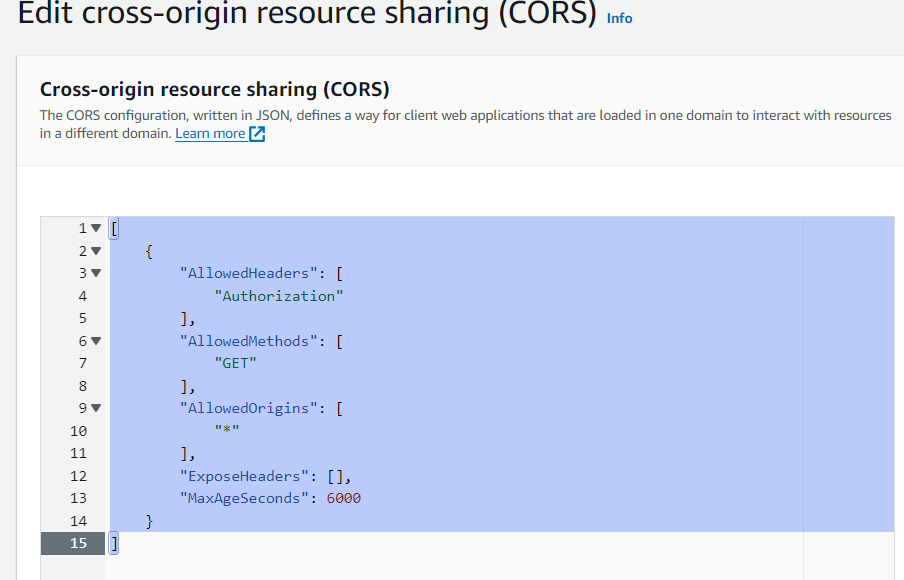
}

* Now under the same permissions tab, scroll down to find Access control list (ACL)
* Edit Access control list





* Now under the same permissions tab, scroll down to find Cross-origin resource sharing (CORS)
* Edit Cross-origin resource sharing



CORS policy:

[

{

"AllowedHeaders": [

"Authorization"

],

"AllowedMethods": [

"GET"

],

"AllowedOrigins": [

"\*"

],

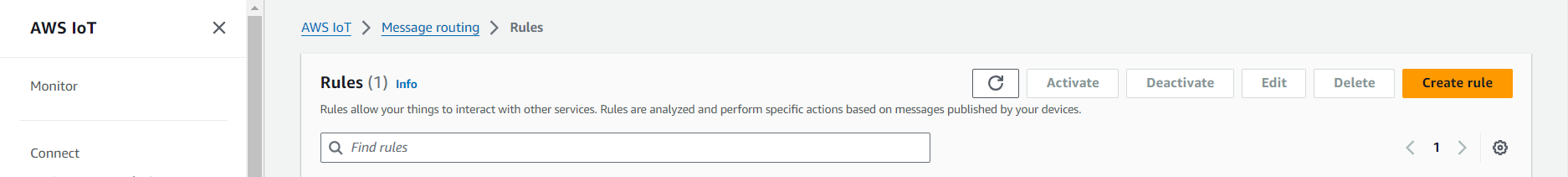
"ExposeHeaders": [],

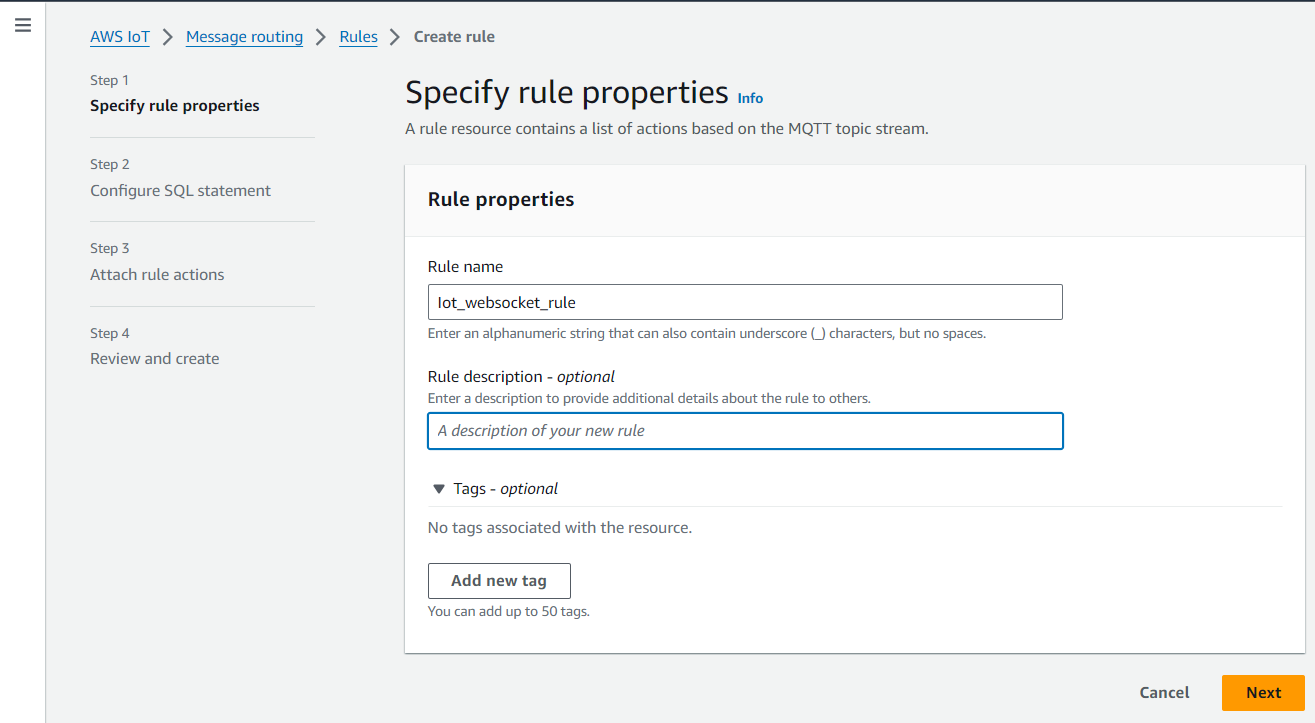
"MaxAgeSeconds": 6000

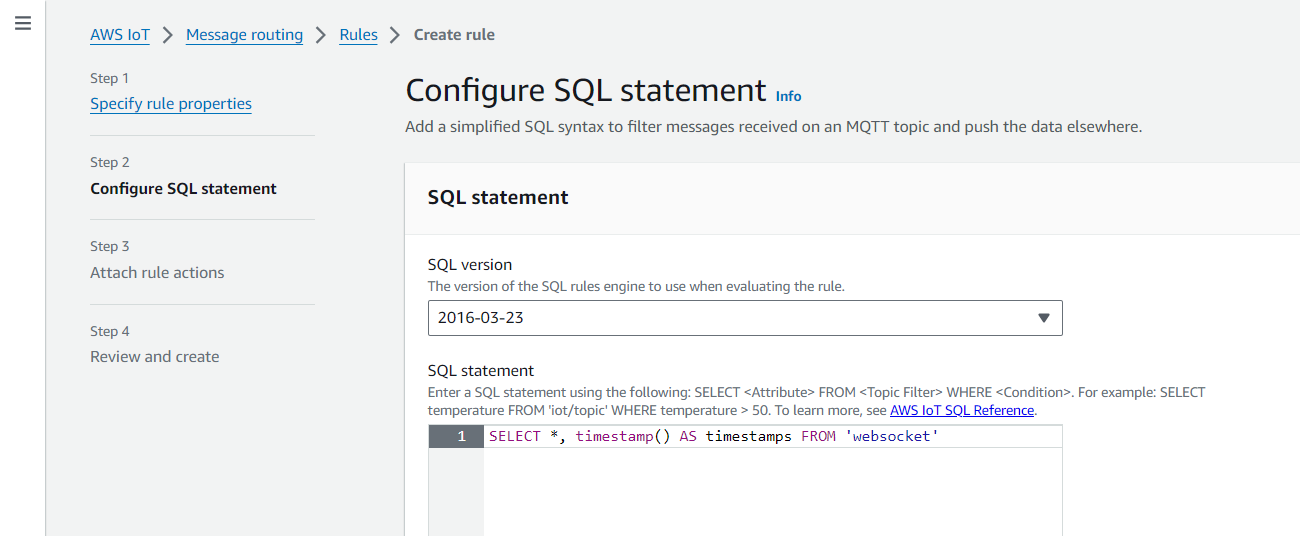
}

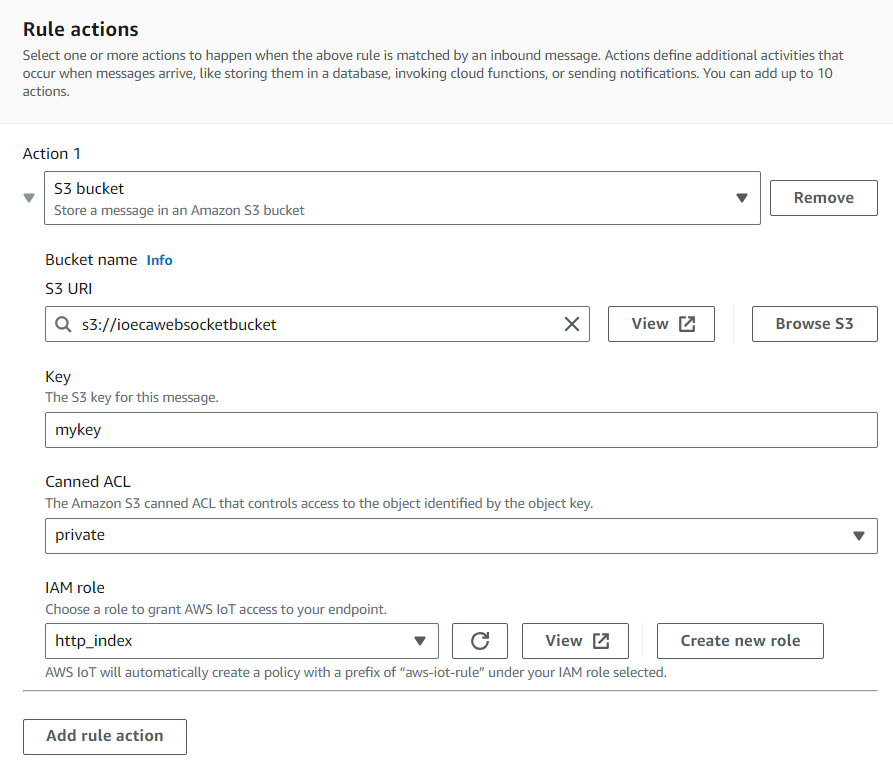
]

**Step 2: Create an Action and Rule in AWS IoT Core**

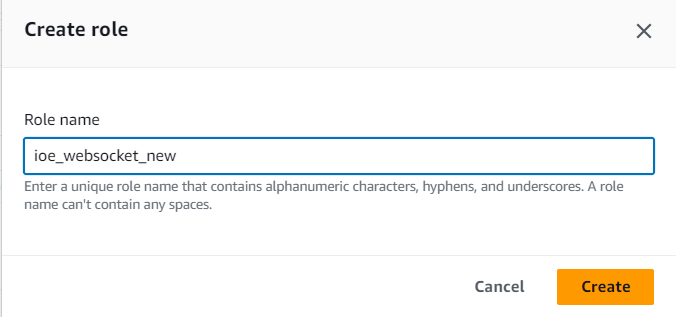
****

****

****

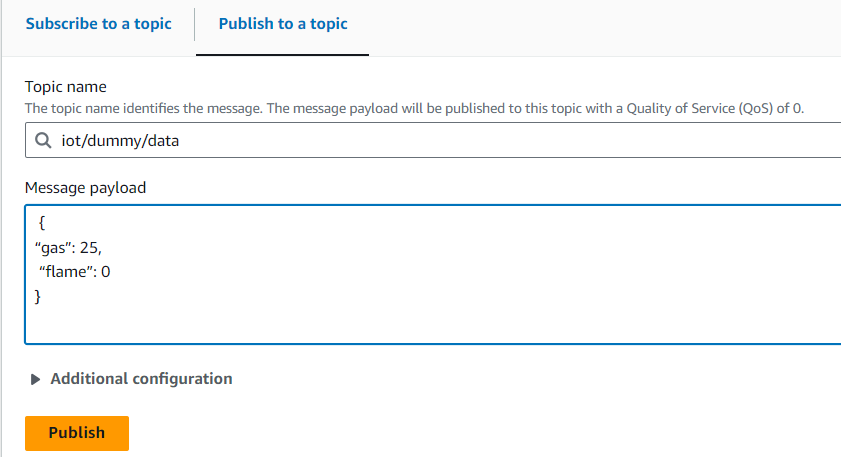
****

* Create new role

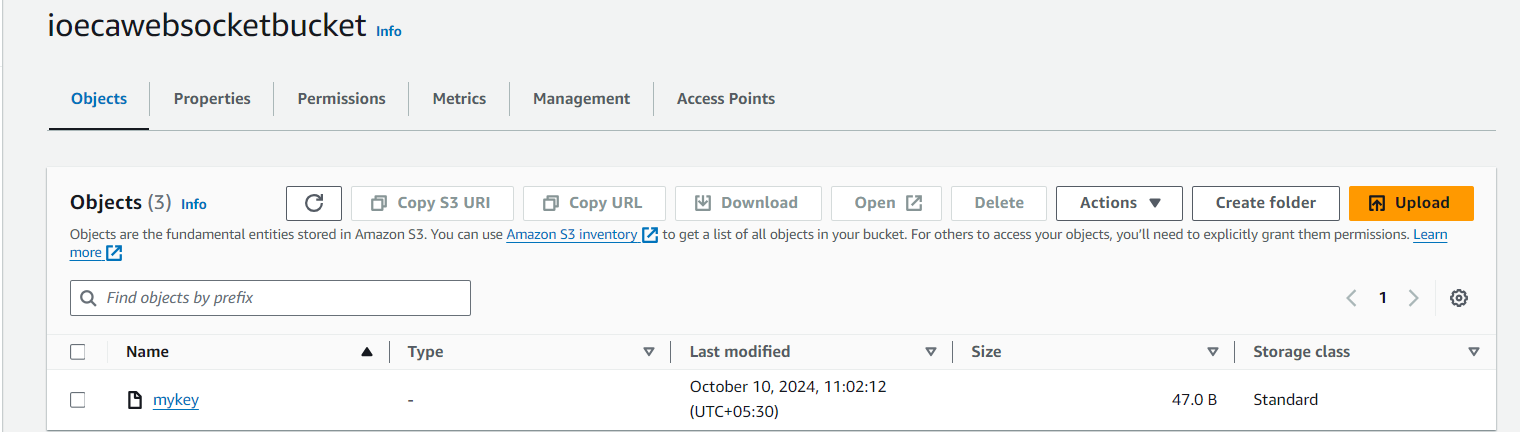


**Step 3: Test your Serverless IoT design flow**

* Select MQTT test client.
* Select the Publish tab.
* Type a test payload in proper JSON format like: { “gas”: 25, “flame”: 0}
* Press the “Publish” button.

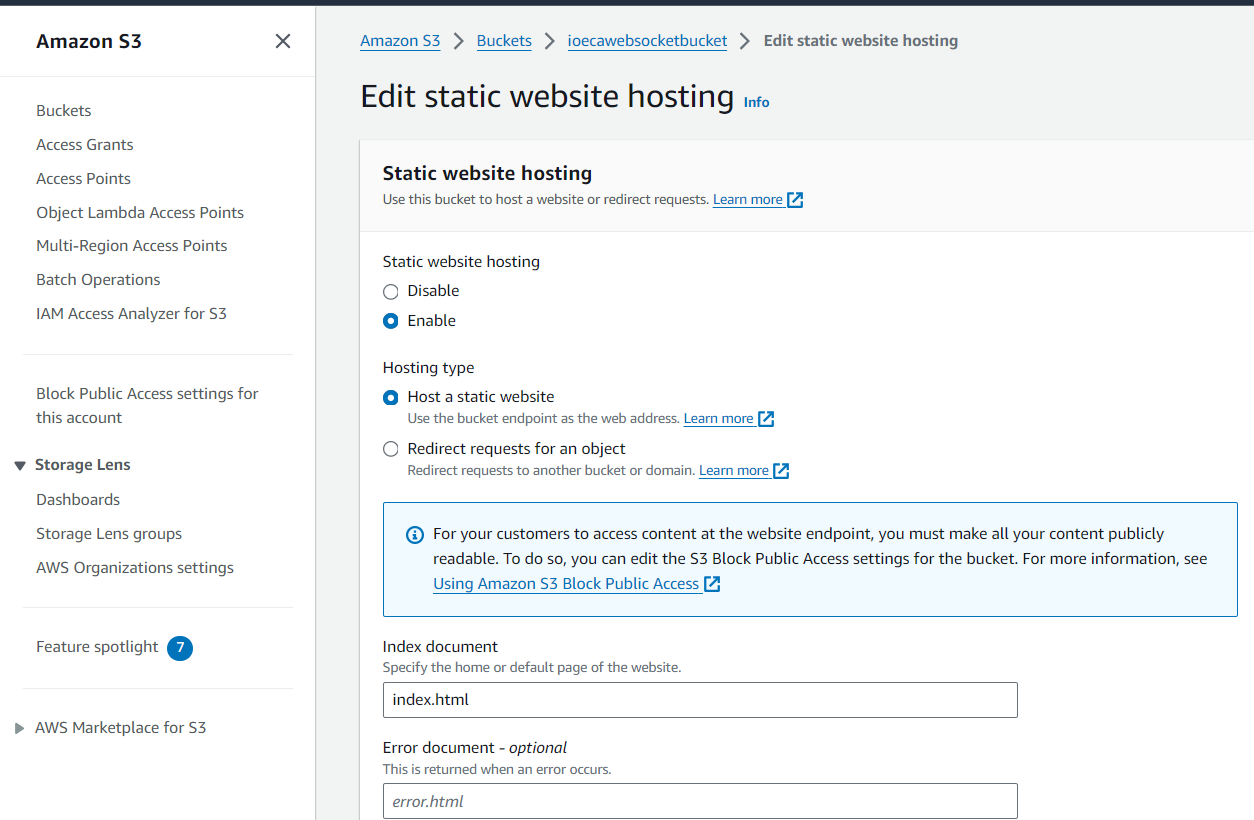


* Now go to your S3 bucket and download the key file



**Step 4 - Convert your S3 bucket into a static Webhost**

* Specify index.html as you index document

****

**Step 5: Create you HTML and JavaScript files and upload them to the same S3 bucket**

**code:**

index.html

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<meta http-equiv="X-UA-Compatible" content="ie=edge">

<title>Dashboard</title>

</head>

<body>

<div class="container">

<h1>Synchronous flame sensor Data on Interval</h1>

<div class="panel panel-info">

<div class="panel-heading">

<h3 class="panel-title"><strong>Line Chart</strong></h3>

</div>

<div class="panel-body">

<div id="container1"></div>

</div>

</div>

</div>

<script src="https://code.jquery.com/jquery-3.1.1.min.js"></script>

<script src="https://code.highcharts.com/highcharts.js"></script>

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

</body>

</html>

main.js

let humArr = [], tempArr = [], upArr = [];

let myChart = Highcharts.chart('container1', {

title: {

text: 'Line chart'

},

subtitle: {

text: 'subtitle'

},

yAxis: {

title: {

text: 'Value'

}

},

xAxis: {

categories: upArr

},

legend: {

layout: 'vertical',

align: 'right',

verticalAlign: 'middle'

},

plotOptions: {

series: {

label: {

connectorAllowed: false

}

}

},

series: [{

name: 'Gas',

data: []

}, {

name: 'Flame',

data: []

}],

responsive: {

rules: [{

condition: {

maxWidth: 500

},

chartOptions: {

legend: {

layout: 'horizontal',

align: 'center',

verticalAlign: 'bottom'

}

}

}]

}

});

let getWheatherData = function () {

// Simulate random values for gas (0-50) and flame (0 or 1)

let data = {

gas: Math.floor(Math.random() \* 51), // Random value between 0 and 50

flame: Math.round(Math.random()), // Random binary value 0 or 1

timestamps: Date.now() // Current timestamp

};

console.log('data', data);

drawChart(data);

}

let drawChart = function (data) {

let { gas, flame, timestamps } = data;

humArr.push(Number(gas));

tempArr.push(Number(flame));

upArr.push(Number(timestamps));

myChart.series[0].setData(humArr , true);

myChart.series[1].setData(tempArr , true);

}

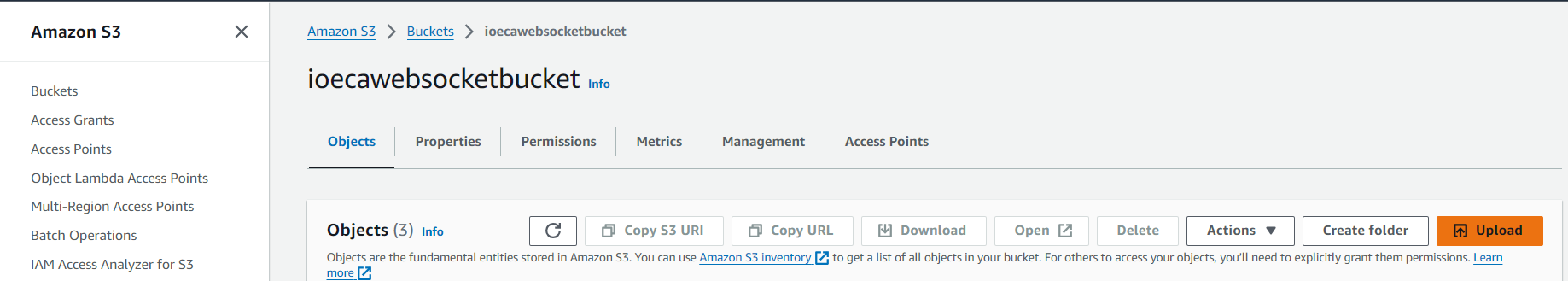
let intervalTime = 3 \* 1000; // 3 second interval polling

setInterval(() => {

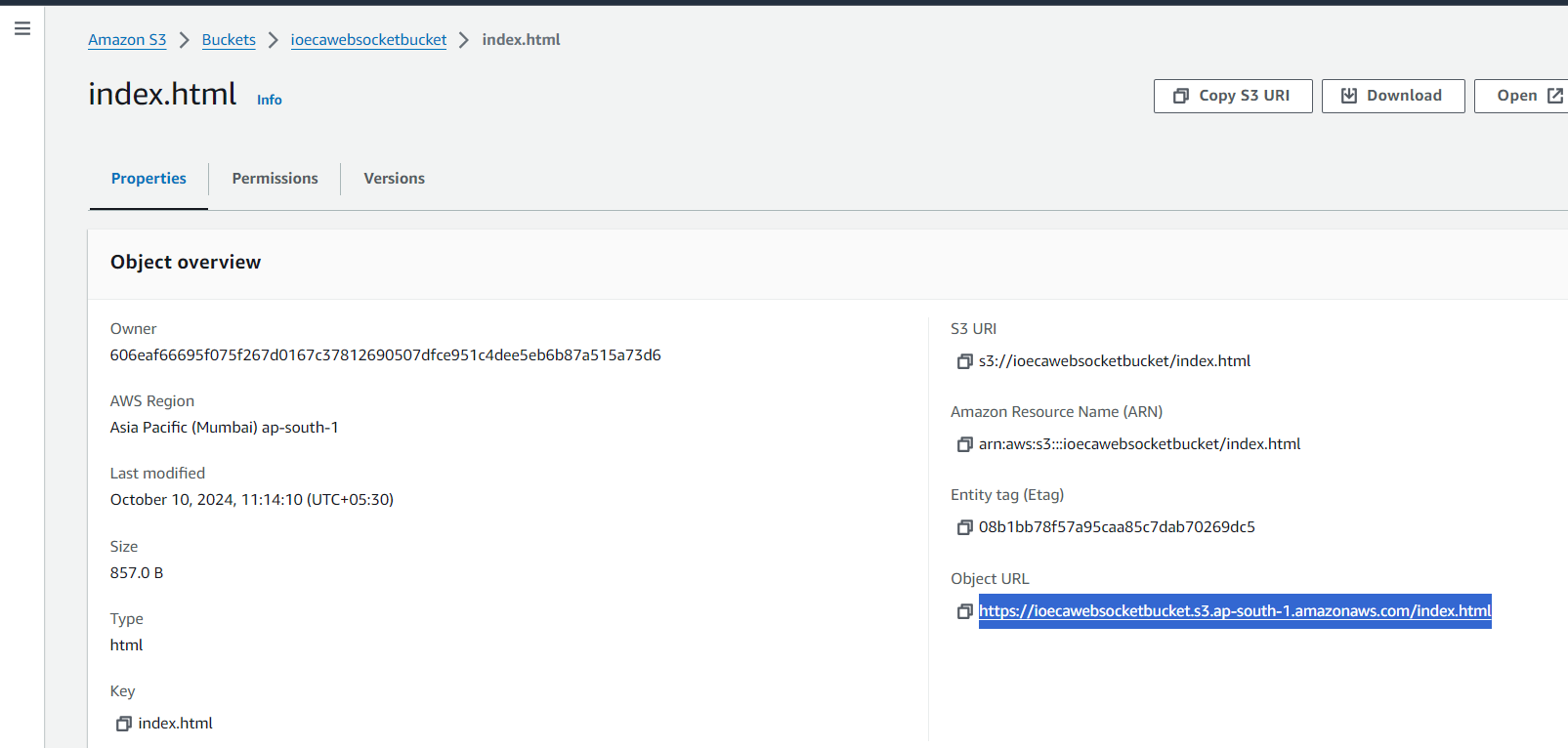
getWheatherData();

}, intervalTime);

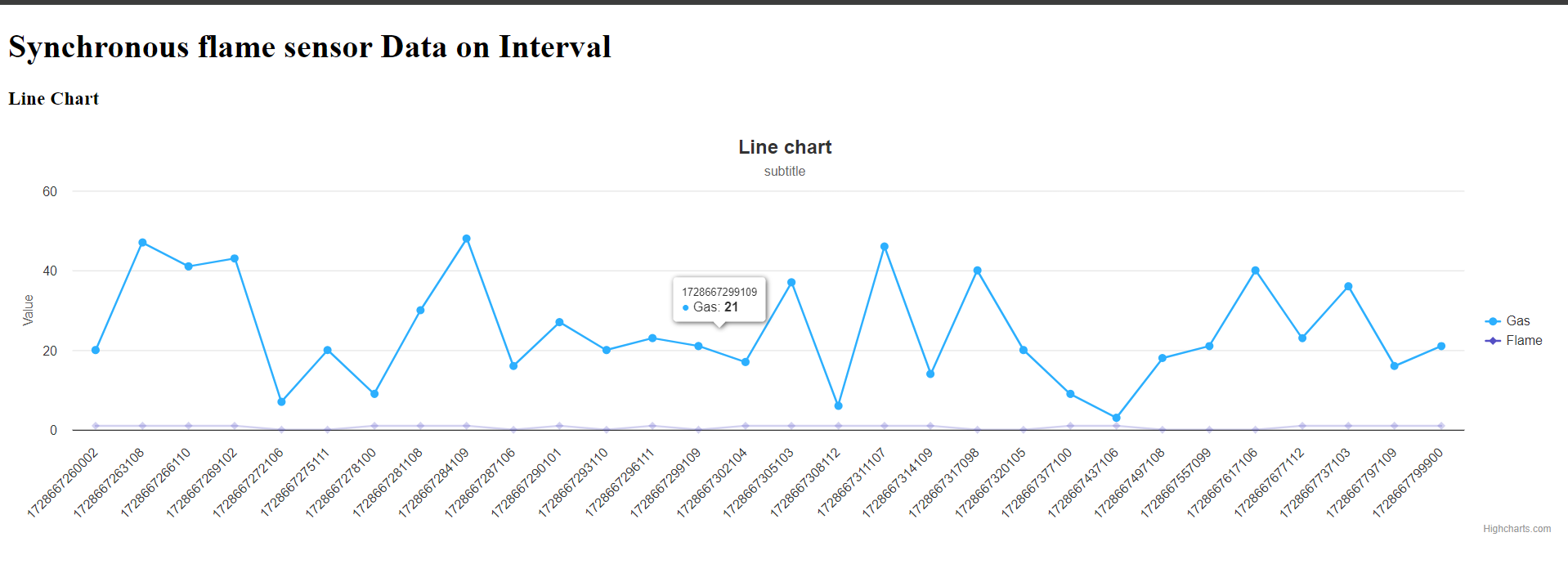
* Upload the index.html and main.js files



* Open the index. html file in the S3 bucket and visit the Object’s URL



* The main.js file generates dummy values and sends it to the hosted index.html file in real-time using WebSocket



**Thus, WebSocket protocol has been implemented**