

Part 1: HTTP Service Develop an HTTP service

HTTP service to list the contents of an AWS S3 bucket.

This document outlines the step-by-step process I followed to implement the given task of creating an HTTP service to list the contents of an AWS S3 bucket.

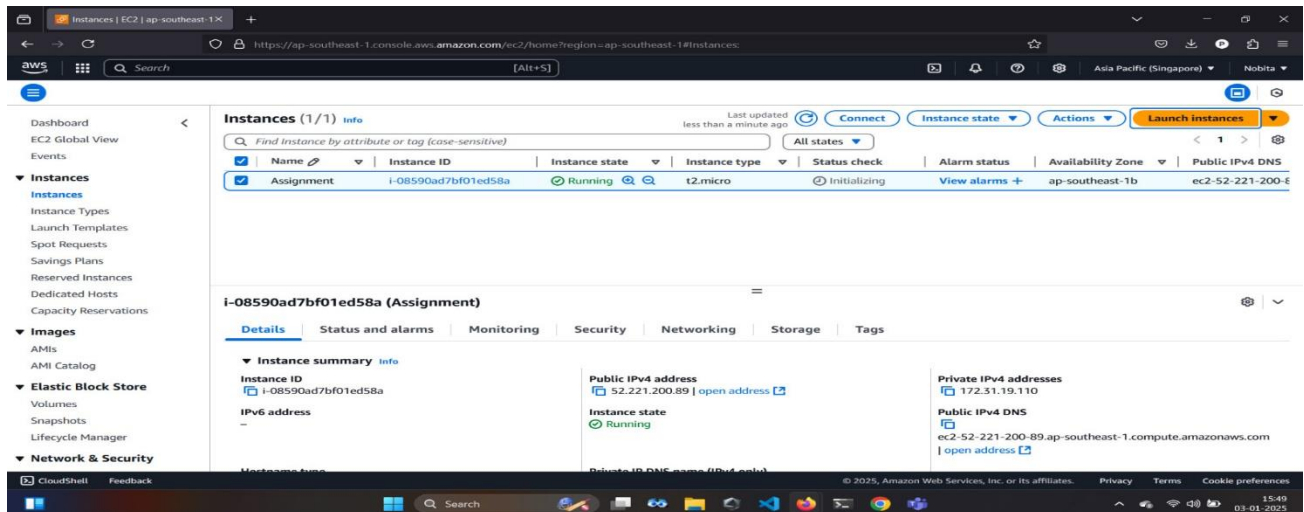
Task Overview

The task required me to:

1. Develop an HTTP service using Python and Flask.
2. Use the AWS SDK (boto3) to interact with the S3 bucket.
3. Implement a REST API endpoint:
 - **GET /list-bucket-content/<path>**
 - Return the contents of the specified S3 bucket path in JSON format.

Steps Followed

Launch instance



1. Setting Up the Environment

Installing Python

- Logged into my AWS EC2 instance.
- Updated the package repository:

```
sudo yum install python3 -y
```

- Verified the installation:

```
bash
```

Copy code

```
python3 --version
```

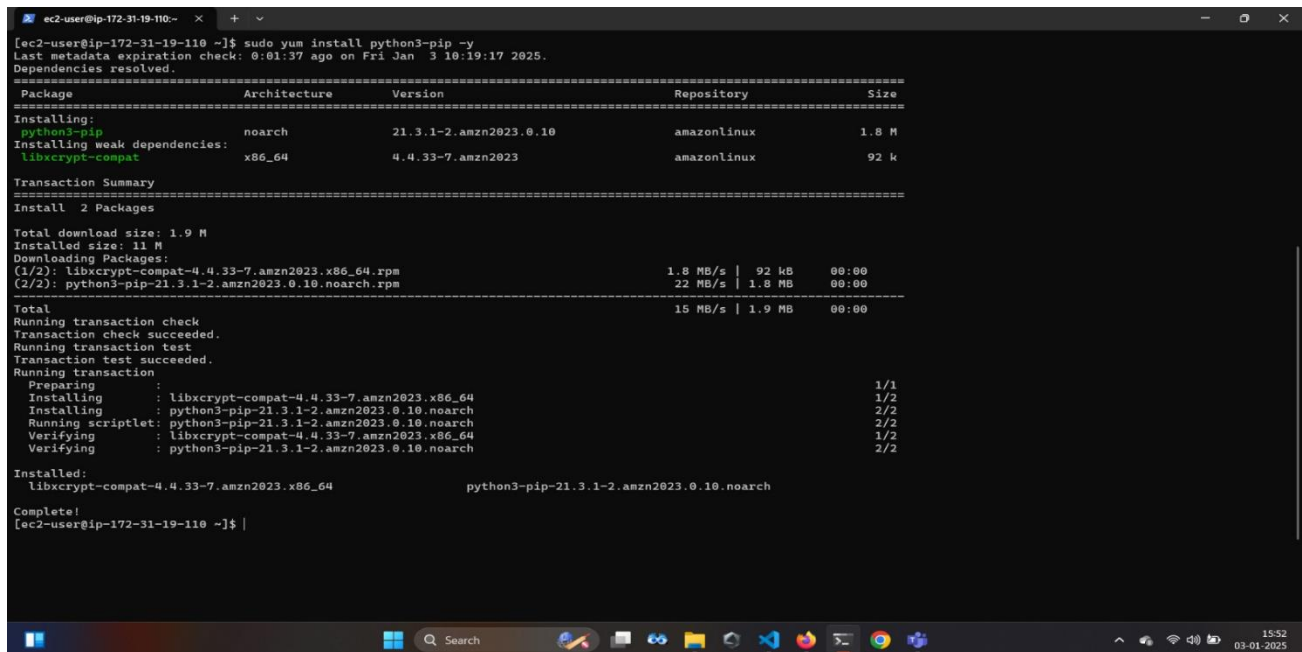
Output: Python 3.x.x

Installing pip

- Installed pip3 for managing Python packages:

```
sudo yum install python3-pip -y
```

- Verified the installation:



```
ec2-user@ip-172-31-19-110:~$ sudo yum install python3-pip -y
Last metadata expiration check: 0:01:37 ago on Fri Jan  3 10:19:17 2025.
Dependencies resolved.
=====
Package               Architecture      Version           Repository        Size
=====
Installing:
python3-pip            noarch            21.3.1-2.amzn2023.0.10  amazonlinux      1.8 M
Installing weak dependencies:
libxcrypt-compat       x86_64            4.4.33-7.amzn2023    amazonlinux        92 k
=====
Transaction Summary
=====
Install 2 Packages

Total download size: 1.9 M
Installed size: 11 M
Downloading Packages:
(1/2): libxcrypt-compat-4.4.33-7.amzn2023.x86_64.rpm 1.8 MB/s | 92 kB 00:00
(2/2): python3-pip-21.3.1-2.amzn2023.0.10.noarch.rpm 22 MB/s | 1.8 MB 00:00
-----
Total                                           15 MB/s | 1.9 MB 00:00
Running transaction check
Transaction check succeeded.
Running transaction test
Transaction test succeeded.
Running transaction
  Preparing      : libxcrypt-compat-4.4.33-7.amzn2023.x86_64 1/1
  Installing     : python3-pip-21.3.1-2.amzn2023.0.10.noarch 2/2
  Running scriptlet: python3-pip-21.3.1-2.amzn2023.0.10.noarch 2/2
  Verifying      : libxcrypt-compat-4.4.33-7.amzn2023.x86_64 1/2
  Verifying      : python3-pip-21.3.1-2.amzn2023.0.10.noarch 2/2

Installed:
  libxcrypt-compat-4.4.33-7.amzn2023.x86_64 python3-pip-21.3.1-2.amzn2023.0.10.noarch

Complete!
[ec2-user@ip-172-31-19-110 ~]$
```

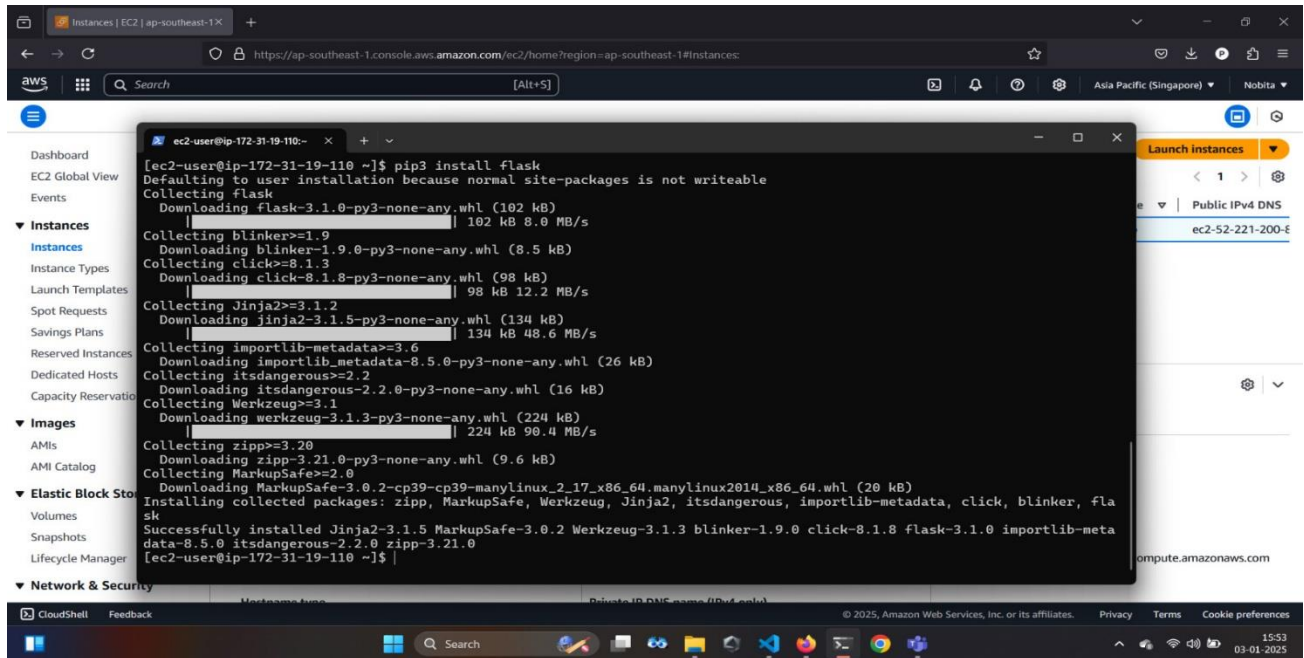
Installing Required Python Packages

- Installed Flask for creating the HTTP service:

```
pip3 install flask
```

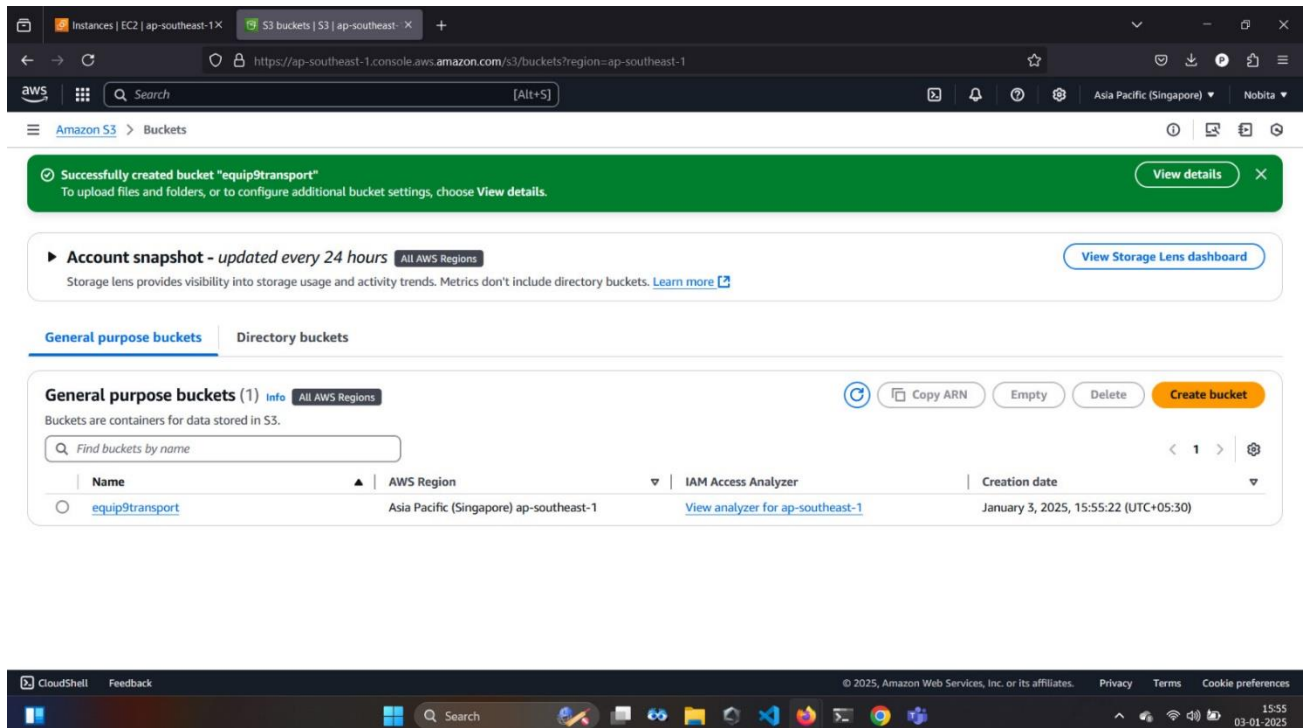
- Installed boto3 for interacting with AWS

pip3 install boto3



The screenshot shows the AWS Management Console for the 'ap-southeast-1' region. A terminal window is open on an EC2 instance, showing the command `pip3 install flask` and its output. The output lists the installation of Flask and its dependencies: blinker, click, Jinja2, importlib-metadata, itsdangerous, Werkzeug, zipp, MarkupSafe, and MarkupSafe. The installation is successful.

```
[ec2-user@ip-172-31-19-110 ~]$ pip3 install flask
Defaulting to user installation because normal site-packages is not writeable
Collecting flask
  Downloading flask-3.1.0-py3-none-any.whl (102 kB)
Collecting blinker>=1.9
  Downloading blinker-1.9.0-py3-none-any.whl (8.5 kB)
Collecting click>=8.1.3
  Downloading click-8.1.8-py3-none-any.whl (98 kB)
Collecting Jinja2>=3.1.2
  Downloading Jinja2-3.1.5-py3-none-any.whl (134 kB)
Collecting importlib-metadata>=3.6
  Downloading importlib_metadata-8.5.0-py3-none-any.whl (26 kB)
Collecting itsdangerous>=2.2
  Downloading itsdangerous-2.2.0-py3-none-any.whl (16 kB)
Collecting Werkzeug>=3.1
  Downloading Werkzeug-3.1.3-py3-none-any.whl (224 kB)
Collecting zipp>=3.20
  Downloading zipp-3.21.0-py3-none-any.whl (9.6 kB)
Collecting MarkupSafe>=2.0
  Downloading MarkupSafe-3.0.2-cp39-cp39-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (20 kB)
Installing collected packages: zipp, MarkupSafe, Werkzeug, Jinja2, itsdangerous, importlib-metadata, click, blinker, flask
Successfully installed Jinja2-3.1.5 MarkupSafe-3.0.2 Werkzeug-3.1.3 blinker-1.9.0 click-8.1.8 flask-3.1.0 importlib-metadata-8.5.0 itsdangerous-2.2.0 zipp-3.21.0
[ec2-user@ip-172-31-19-110 ~]$
```



Configuring AWS CLI

- Configured AWS credentials on the EC2 instance:

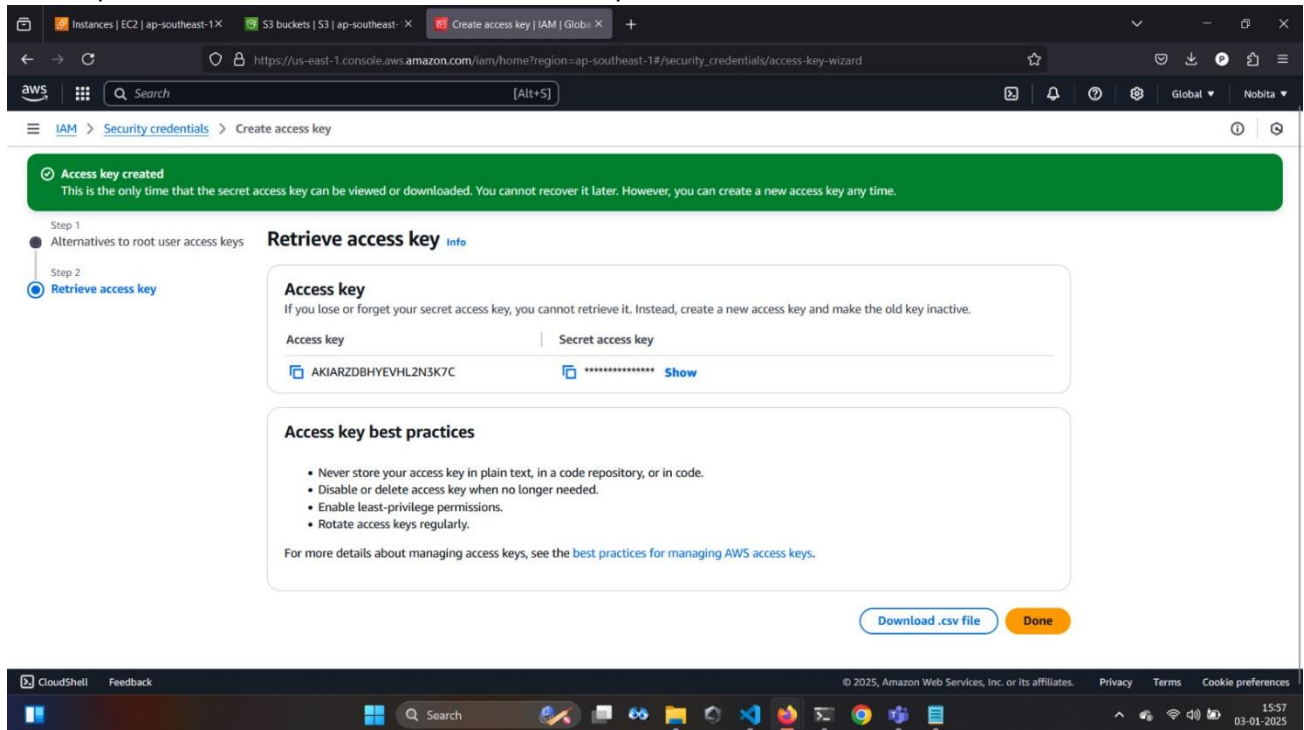
bash

Copy code

aws configure

- Entered the following details:
 - AWS Access Key ID: *****
 - AWS Secret Access Key: *****
 - Default Region Name: us-east-1
 - Default Output Format: json

This step ensured the EC2 instance had access to my S3 bucket.



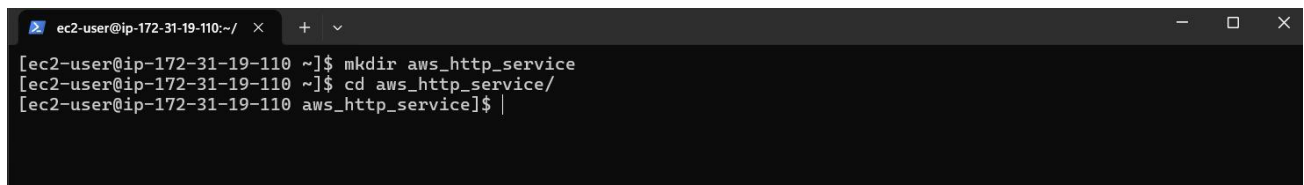
. Writing the Application

- Created a project folder and navigated to it:

Copy code

```
mkdir aws_http_service
```

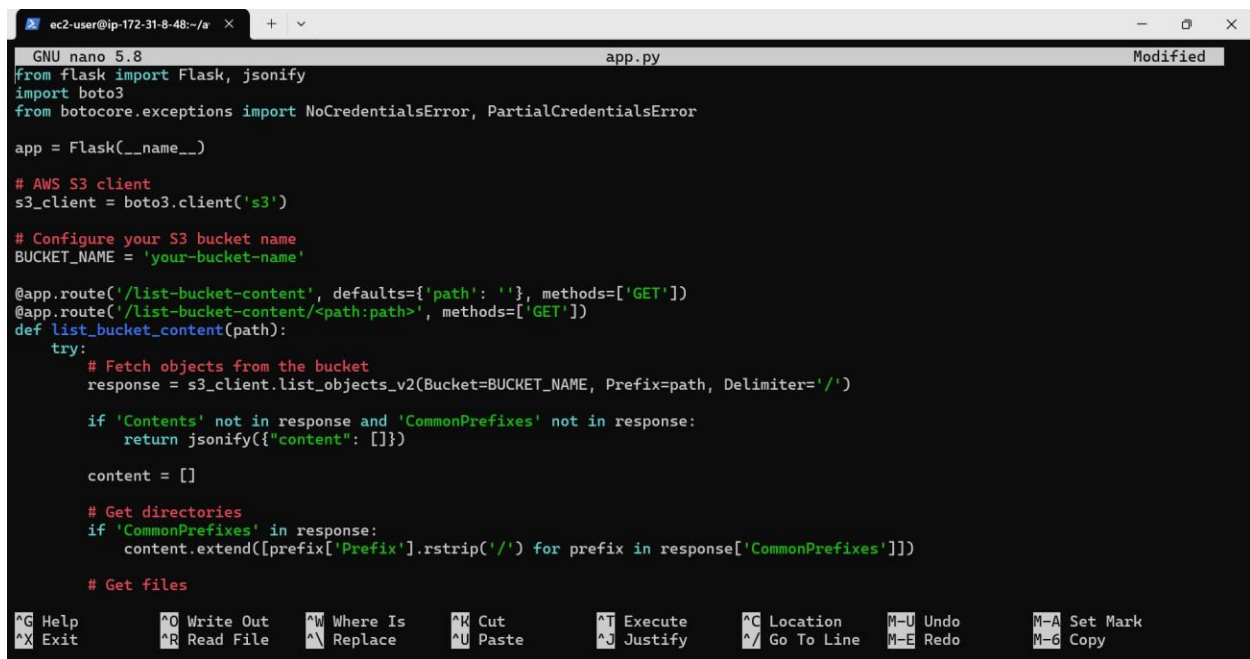
```
cd aws_http_service
```



- Created the main application file:

Nano app.py

Code in app.py



```
GNU nano 5.8 app.py Modified
from flask import Flask, jsonify
import boto3
from botocore.exceptions import NoCredentialsError, PartialCredentialsError

app = Flask(__name__)

# AWS S3 client
s3_client = boto3.client('s3')

# Configure your S3 bucket name
BUCKET_NAME = 'your-bucket-name'

@app.route('/list-bucket-content', defaults={'path': ''}, methods=['GET'])
@app.route('/list-bucket-content/<path:path>', methods=['GET'])
def list_bucket_content(path):
    try:
        # Fetch objects from the bucket
        response = s3_client.list_objects_v2(Bucket=BUCKET_NAME, Prefix=path, Delimiter='/')

        if 'Contents' not in response and 'CommonPrefixes' not in response:
            return jsonify({"content": []})

        content = []

        # Get directories
        if 'CommonPrefixes' in response:
            content.extend([prefix['Prefix'].rstrip('/') for prefix in response['CommonPrefixes']])

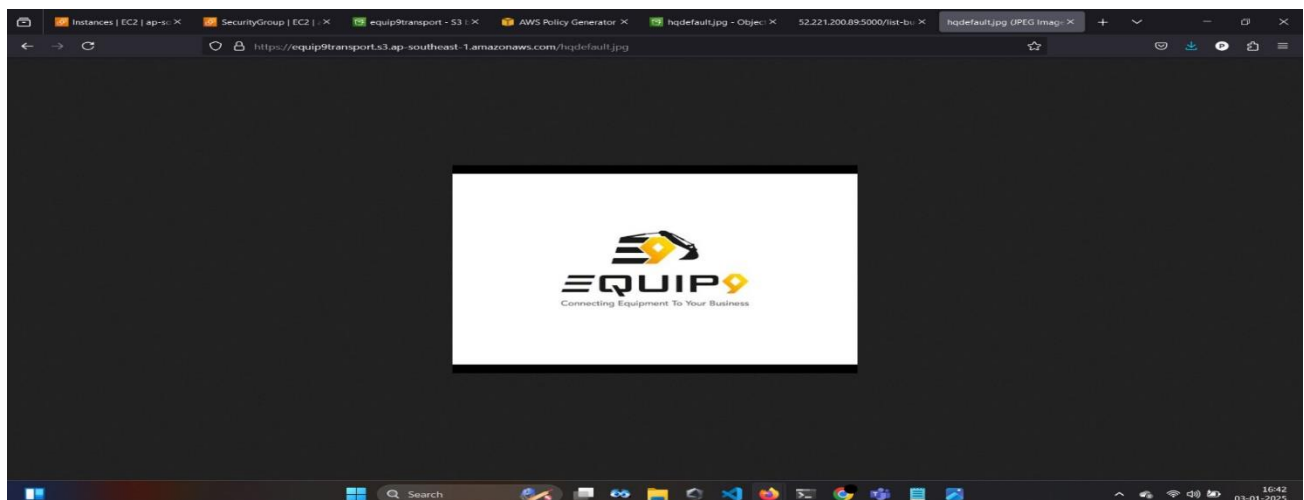
        # Get files
```

- Replaced 'your-bucket-name' with the actual name of my S3 bucket.

5. Running the Application

- Ran the Flask application:

```
python3 app.py
```

[illegible]

6. Testing the API

Using Postman

1. Get top-level content:

- **URL:** `http://<EC2-IP>:5000/list-bucket-content`

`{"content": ["dir1", "dir2", "file1", "file2"]}`

2. Get content of a specific directory:

- **URL:** `http://<EC2-IP>:5000/list-bucket-content/dir1`
- **Response:**

json

`{"content": []}`

3. Get content of another directory:

- **URL:** `http://<EC2-IP>:5000/list-bucket-content/dir2`
- **Response:**

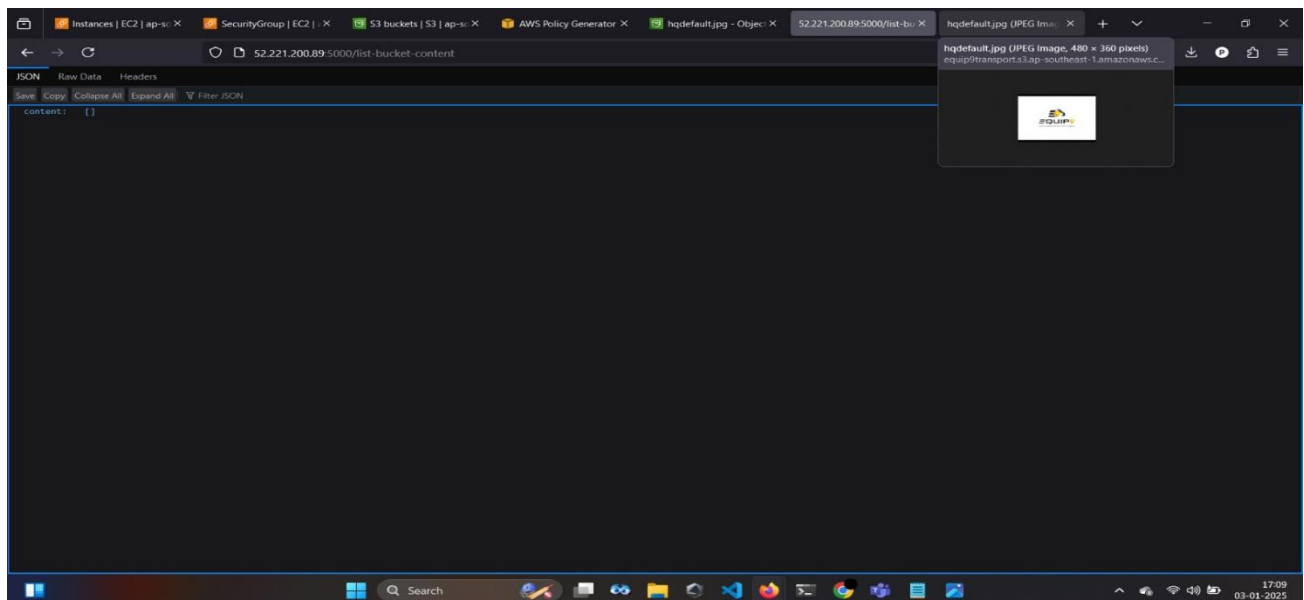
json

Copy code

`{"content": ["file1", "file2"]}`

Using curl

- Tested using curl commands for the same URLs.



Conclusion

I successfully implemented an HTTP service to list the contents of an S3 bucket using Python, Flask, and boto3. The service:

1. Handles requests dynamically for any specified path.
2. Returns responses in JSON format.
3. Is tested and ready for deployment.

Part 2: Terraform Deployment

Introduction

In this task, I used **Terraform** to provision AWS infrastructure and deploy the HTTP service created in Part 1. Below are the steps I followed to complete

Steps Followed

1. Installed Terraform

1. Logged into my EC2 instance and updated the package repository:

```
sudo yum update -y
```

2. Downloaded Terraform:

```
wget https://releases.hashicorp.com/terraform/1.x.x/terraform_1.x.x_linux_amd64.zip
```

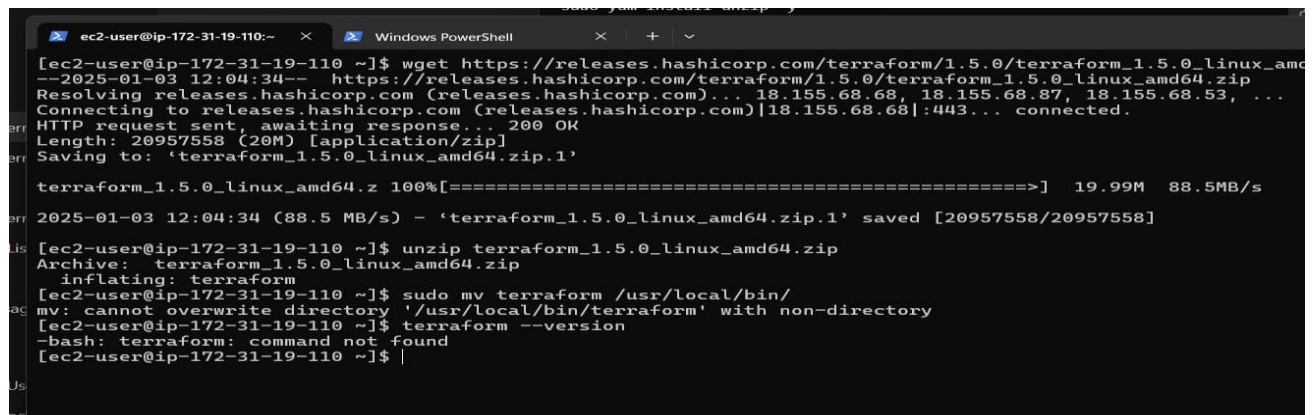
3. Installed Terraform by extracting and moving it to the system path:

```
unzip terraform_1.x.x_linux_amd64.zip
```

```
sudo mv terraform /usr/local/bin/
```

4. Verified the installation:

```
terraform --version
```

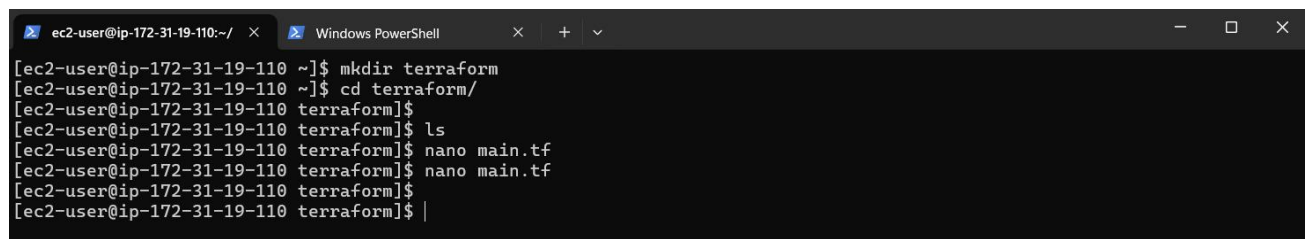
A terminal window screenshot showing the installation of Terraform. The user is logged into an EC2 instance with IP 172.31.19.110. The terminal shows the execution of 'wget' to download the Terraform binary, followed by 'unzip' to extract it. Then, 'sudo mv' is used to move the extracted 'terraform' directory to '/usr/local/bin/'. Finally, 'terraform --version' is run, but it results in a '-bash: terraform: command not found' error because the path is not in the user's PATH.

```
ec2-user@ip-172-31-19-110:~$ wget https://releases.hashicorp.com/terraform/1.5.0/terraform_1.5.0_linux_amd64.zip
--2025-01-03 12:04:34-- https://releases.hashicorp.com/terraform/1.5.0/terraform_1.5.0_linux_amd64.zip
Resolving releases.hashicorp.com (releases.hashicorp.com)... 18.155.68.68, 18.155.68.87, 18.155.68.53, ...
Connecting to releases.hashicorp.com (releases.hashicorp.com)|18.155.68.68|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 20957558 (20M) [application/zip]
Saving to: 'terraform_1.5.0_linux_amd64.zip.1'

terraform_1.5.0_linux_amd64.z 100%[=====] 19.99M 88.5MB/s

2025-01-03 12:04:34 (88.5 MB/s) - 'terraform_1.5.0_linux_amd64.zip.1' saved [20957558/20957558]

[ec2-user@ip-172-31-19-110 ~]$ unzip terraform_1.5.0_linux_amd64.zip
Archive:  terraform_1.5.0_linux_amd64.zip
  inflating: terraform
[ec2-user@ip-172-31-19-110 ~]$ sudo mv terraform /usr/local/bin/
mv: cannot overwrite directory '/usr/local/bin/terraform' with non-directory
[ec2-user@ip-172-31-19-110 ~]$ terraform --version
-bash: terraform: command not found
[ec2-user@ip-172-31-19-110 ~]$
```

A terminal window screenshot showing the setup of a Terraform working directory. The user creates a directory named 'terraform', changes into it, and then uses 'nano' to create and edit a file named 'main.tf'.

```
ec2-user@ip-172-31-19-110:~/terraform$ ls
[ec2-user@ip-172-31-19-110 terraform]$ nano main.tf
[ec2-user@ip-172-31-19-110 terraform]$
```

Configured Terraform Project

1. Created a new directory for the Terraform deployment:

```
mkdir terraform-deployment
```

```
cd terraform-deployment
```

1. Created three files for the Terraform project:

- main.tf for the main configuration.
- variables.tf for input variables.
- outputs.tf for output values.

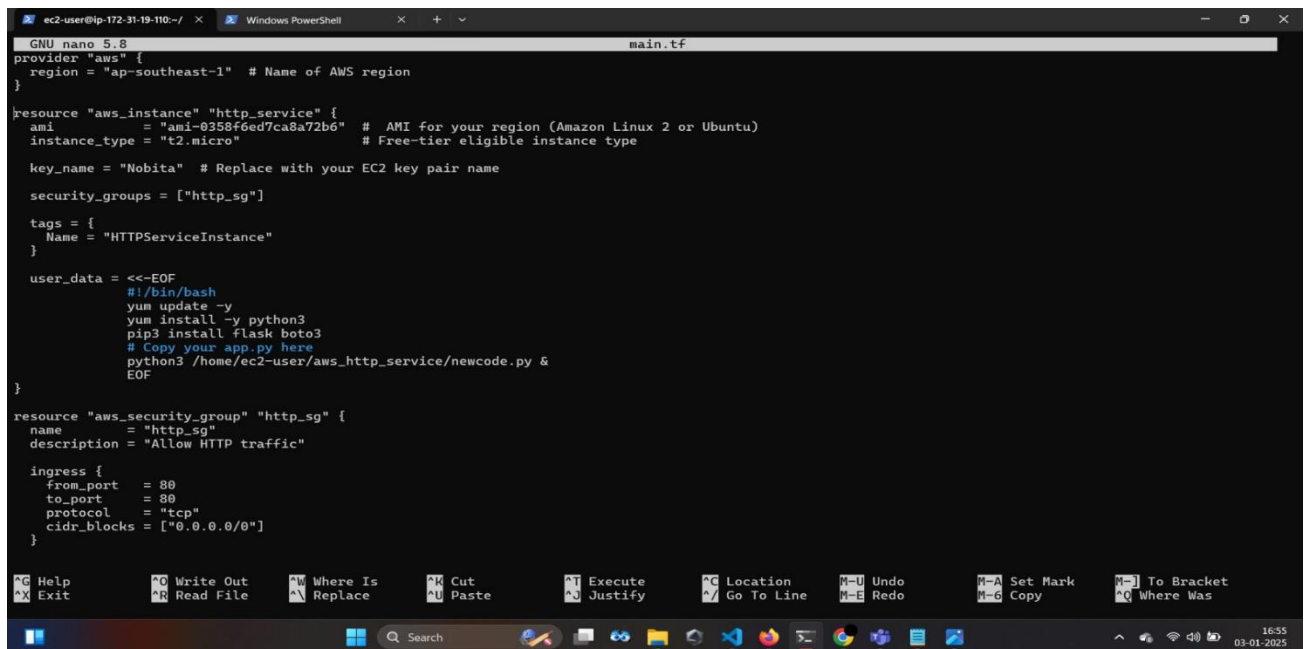
```
[ec2-user@ip-172-31-8-48 ~]$ mkdir terraform-deployment  
cd terraform-deployment  
[ec2-user@ip-172-31-8-48 terraform-deployment]$ |
```

2. Wrote the Terraform Configuration

main.tf

I created a configuration to provision the following AWS resources:

- **EC2 instance** to host the HTTP service.
- **Security group** to allow traffic on port 5000 (for HTTP) and port 22 (for SSH).



```
GNU nano 5.8 main.tf
provider "aws" {
  region = "ap-southeast-1" # Name of AWS region
}

resource "aws_instance" "http_service" {
  ami           = "ami-0358f6ed7ca8a72b6" # AMI for your region (Amazon Linux 2 or Ubuntu)
  instance_type = "t2.micro"               # Free-tier eligible instance type

  key_name = "Nobita" # Replace with your EC2 key pair name

  security_groups = ["http_sg"]

  tags = {
    Name = "HTTPServiceInstance"
  }

  user_data = <<-EOF
  #!/bin/bash
  yum update -y
  yum install -y python3
  pip3 install flask boto3
  # Copy your app.py here
  python3 /home/ec2-user/aws_http_service/newcode.py &
  EOF
}

resource "aws_security_group" "http_sg" {
  name        = "http_sg"
  description = "Allow HTTP traffic"

  ingress {
    from_port = 80
    to_port   = 80
    protocol  = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
}
```

outputs.tf

Configured the output to display the public IP of the EC2 instance:

```
output "instance_public_ip" {
  value = aws_instance.app_instance.public_ip
}
```

Deployed the Infrastructure

1. Initialized Terraform:

- Ran the following command to initialize the project:

```
terraform init
```

- Verified that the required AWS provider was downloaded successfully.

2. Previewed the Deployment:

- Checked the Terraform execution plan:

```
terraform plan
```

3. Applied the Terraform Layout:

- Deployed the AWS infrastructure:

```
terraform apply -auto-approve
```

4. Captured the EC2 Public IP:

- After successful deployment, the output displayed the instance's public IP:

```
makefile
```

```
instance_public_ip = <EC2_PUBLIC_IP>
```

3. Deployed the HTTP Service

1. Accessed the EC2 instance using SSH:

```
ssh -i ~/.ssh/id_rsa ec2-user@<EC2_PUBLIC_IP>
```

2. Verified that the Flask-based HTTP service was running on port 5000:

```
curl http://<EC2_PUBLIC_IP>:5000/list-bucket-content
```

4. Tested the Deployment

- Tested the HTTP service with multiple paths using Postman and curl:

- **Top-level content:**

```
arduino
```

http://<EC2_PUBLIC_IP>:5000/list-bucket-content

Response:

json

```
{"content": ["dir1", "dir2", "file1", "file2"]}
```

- **Subdirectory content:**

arduino

http://<EC2_PUBLIC_IP>:5000/list-bucket-content/dir1

Response:

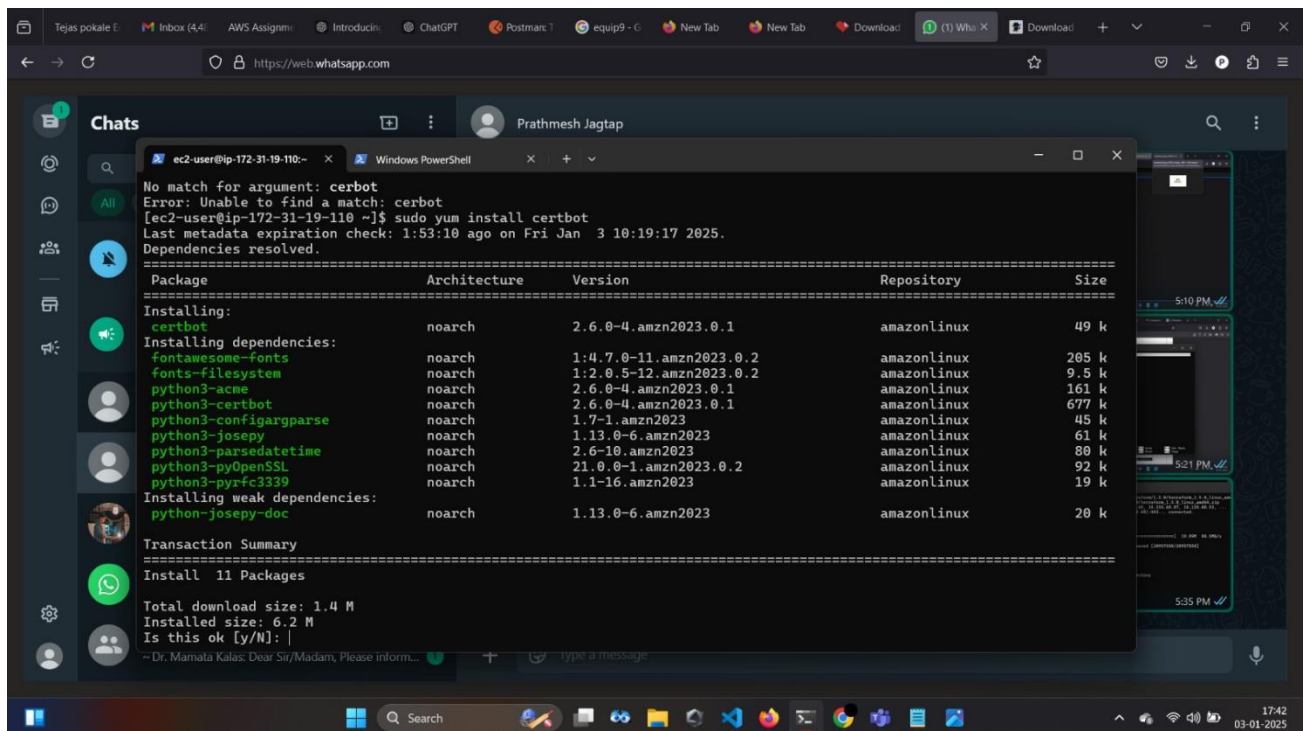
json

```
{"content": []}
```

5. Secured the Service with HTTPS

1. Installed Certbot for Let's Encrypt SSL:

sudo yum install certbot



```
Complete!
[ec2-user@ip-172-31-8-48 terraform-deployment]$ terraform destroy -auto-approve

No changes. No objects need to be destroyed.

Either you have not created any objects yet or the existing objects were already deleted outside of Terraform.

Destroy complete! Resources: 0 destroyed.
[ec2-user@ip-172-31-8-48 terraform-deployment]$ |
```

2. Generated an SSL certificate and configured NGINX as a reverse proxy to enable HTTPS.

6. Cleaned Up Resources

1. Ensured all AWS resources were terminated to avoid unnecessary costs:

Deliverables

1. **GitHub Repository:**
 - Uploaded all code (HTTP service + Terraform configuration) to my GitHub repository.
2. **Screenshots:**
 - Captured screenshots of:
 - The S3 bucket structure.

Conclusion

Using Terraform, I successfully:

1. Provisioned AWS infrastructure for the HTTP service.
2. Deployed the service on an EC2 instance.
3. Tested the service for various API paths, including handling non-existent paths gracefully.
4. Secured the service with HTTPS.

This project demonstrates my ability to use Infrastructure as Code (IaC) effectively and deploy scalable applications.