Labs 6-9

Student ID: 23905652

Student Name: Zhe Han

Lab 6 Web Application

Set up an EC2 instance

[1] Create an EC2 micro instance with Ubuntu and SSH into it

1. Similar to lab 5, write the following script to create EC2 instances. Since the steps later in this lab will use the ALB, and AWS strongly recommends configuring the ALB across at least two subnets in different availability zones for high availability, I reused the code for EC2 creating from lab 5 to create EC2 instances in different availability zones. The code is as follows, with detailed explanations included in the comments:

```
import boto3
import os

student_number = "23905652"
region = "ap-southeast-1"
# Initialize the EC2 client
ec2 = boto3.client('ec2', region_name=region)

def create_security_group():
    # Create a new security group with a name based on the student number
    response = ec2.create_security_group(
        GroupName=f"{student_number}-sg",
        Description="Security group for Web Application"
    )
    security_group_id = response['GroupId']

    # Configure inbound rules to allow HTTP (port 80) and SSH (port 22) traffic
    ec2.authorize_security_group_ingress(
        GroupId=security_group_id,
        IpPermissions=[
        {
            'IpProtocol': 'tcp',
            'FromPort': 80,
            'ToPort': 80,
            'IpRanges': [{'CidrIp': '0.0.0.0/0'}] # Allow HTTP traffic from any IP
        },
        {
            'IpProtocol': 'tcp',
            'FromPort': 22,
            'ToPort': 22,
            'IpRanges': [{'CidrIp': '0.0.0.0/0'}] # Allow SSH traffic from any IP
        }
    ]
    )
    print(f"Created Security Group: {security_group_id}")
    return security_group_id
```

```
def create_key_pair():
    # Generate a new key pair for SSH access to the EC2 instances
    key_name = f"{student_number}-key"
    response = ec2.create_key_pair(KeyName=key_name)
    private_key = response['KeyMaterial']

    # Save the private key to a file with the key name
    private_key_file = f"{key_name}.pem"

    # Allow writing to the private key file
    with open(private_key_file, 'w') as key_file:
        key_file.write(private_key)

    # Set the correct permissions for the private key file (read-only by owner)
    os.chmod(private_key_file, 0o400)

    print(f"Private key saved to {private_key_file}")
    return key_name

def get_availability_zones():
    # Retrieve the first two availability zones in the specified region
    response = ec2.describe_availability_zones(
        Filters=[
            {
                'Name': 'region-name',
                'Values': [region]
            },
        ]
    )
    return [az['ZoneName'] for az in response['AvailabilityZones'][:2]]

def create_ec2_instance(security_group_id, key_name, availability_zone,
instance_number):
    # Launch a new EC2 instance with specified parameters
    response = ec2.run_instances(
        ImageId="ami-0497a974f8d5dcef8", # Amazon Linux 2 AMI ID for ap-southeast-1
        InstanceType="t2.micro", # Free tier instance type
        KeyName=key_name, # Key pair for SSH access
        MaxCount=1, # Launch a single instance
        MinCount=1, # Launch a single instance
        SecurityGroupIds=[security_group_id], # Attach the created security group
        Placement={
            'AvailabilityZone': availability_zone # Specify the AZ for the instance
        },
        TagSpecifications=[
            {
                'ResourceType': 'instance',
                'Tags': [
                    {'Key': 'Name', 'Value': f'{student_number}-vm{instance_number}'}
                ]
            }
        ]
```

```
    )
    instance_id = response['Instances'][0]['InstanceId']
    print(f"Created EC2 instance: {instance_id} in {availability_zone}")
    return instance_id

def main():
    # Create security group
    security_group_id = create_security_group()

    # Create key pair
    key_name = create_key_pair()

    # Get availability zones
    availability_zones = get_availability_zones()

    instance_ids = []
    for i, az in enumerate(availability_zones, start=1):
        instance_id = create_ec2_instance(security_group_id, key_name, az, i)
        instance_ids.append(instance_id)

    # Wait for all instances to reach the 'running' state
    print("Waiting for instances to enter 'running' state...")
    waiter = ec2.get_waiter('instance_running')
    waiter.wait(InstanceIds=instance_ids)
    print("Instances are now running.")

    # Retrieve and display public IP addresses of the created instances
    for i, instance_id in enumerate(instance_ids, start=1):
        response = ec2.describe_instances(InstanceIds=[instance_id])
        instance = response['Reservations'][0]['Instances'][0]
        public_ip = instance.get('PublicIpAddress', 'N/A')
        az = instance['Placement']['AvailabilityZone']
        print(f"Instance {i} (ID: {instance_id}) created in {az} with Public IP:
{public_ip}")

if __name__ == "__main__":
    main()
```

Main functionalities and steps:

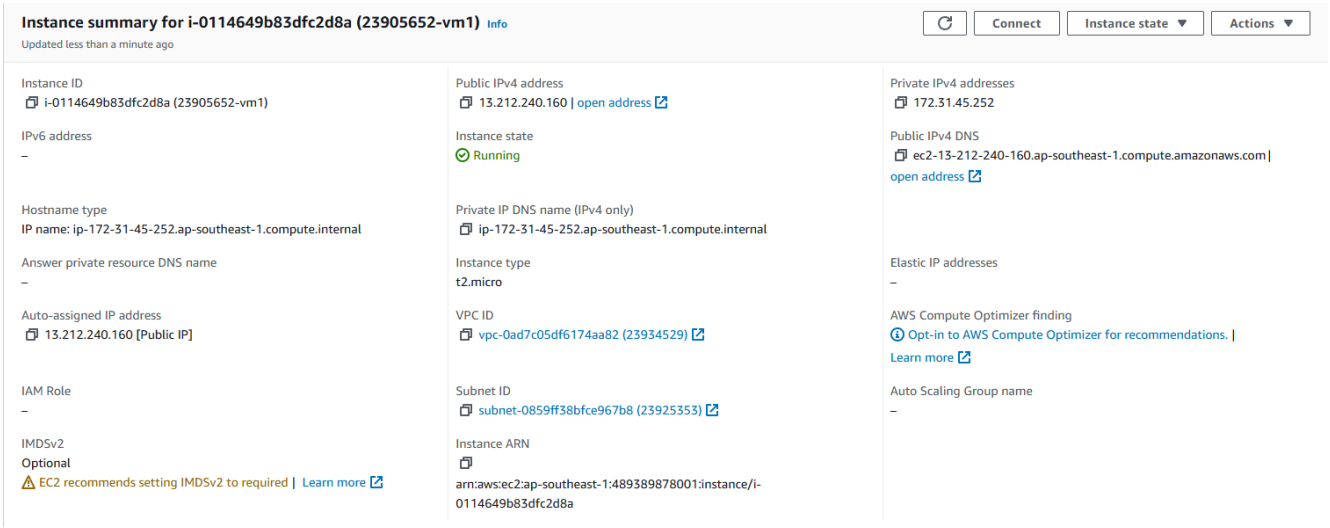
1. **Initialization:** Set the student identifier and AWS region; Create the EC2 client.
2. **Security Settings:** Create a security group that allows HTTP (port 80) and SSH (port 22) traffic; Generate a new key pair for SSH access and save the private key to a local file.
3. **Network Configuration:** Retrieve the first two availability zones in the specified region.
4. **Instance Creation:** Create an EC2 instance in each selected availability zone; Use the Amazon Linux 2 AMI and t2.micro instance type; Associate the instance with the previously created security group and key pair; Add tags to each instance for identification.
5. **Wait and Verify:** Wait for all instances to enter the "running" state; Retrieve and display the public IP address of each instance.

Save the script as "create2EC2.py" and run the code.

Result:

```
4aa82'
mayhan@DESKTOP-KEJ3P3J:~/lab/lab06$ python3 create2EC2.py
Created Security Group: sg-05c6c36862582b514
Private key saved to 23905652-key.pem
Created EC2 instance: i-0114649b83dfc2d8a in ap-southeast-1a
Created EC2 instance: i-046e7533e262ada51 in ap-southeast-1b
Waiting for instances to enter 'running' state...
Instances are now running.
Instance 1 (ID: i-0114649b83dfc2d8a) created in ap-southeast-1a with Public IP: 13.212.240.160
Instance 2 (ID: i-046e7533e262ada51) created in ap-southeast-1b with Public IP: 47.129.32.113
```

From the output, we can see that we have established EC2 instances in two different availability zones. We will select Instance 1 as an example for the Web Application setup steps after. Confirm in the AWS console:



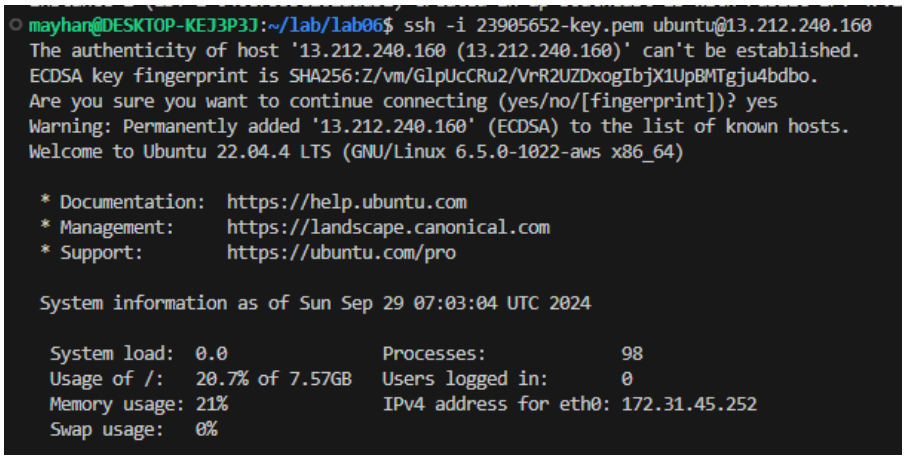
2. Next, we use the private key and public IP address outputted in step 1 to SSH into the EC2 instance. Execute the following command in the terminal, which uses the specified private key file (23905652-key.pem) to connect via SSH as the ubuntu user to the EC2 instance with the IP address 13.212.240.160:

```
ssh -i 23905652-key.pem ubuntu@13.212.240.160
```

Explanation:

- `ssh`: The command-line client for Secure Shell, used to securely connect to remote servers.
- `-i 23905652-key.pem`: The `-i` option specifies the identity file to use, in this case, it's the private key file. `23905652-key.pem` is the name of the private key file. This file contains the private key that we saved locally in step 1.
- `ubuntu`: The username for logging into the remote server. In this case, it indicates that it is an Ubuntu EC2 instance.
- `@`: This symbol is used to separate the username from the host address.
- `13.212.240.160`: This is the public IP address of the EC2 instance, which is the public IP address outputted after successfully creating the EC2 in step 1.

Result:



According to the output, we have successfully accessed the instance via SSH.

[2] Install the Python 3 virtual environment package

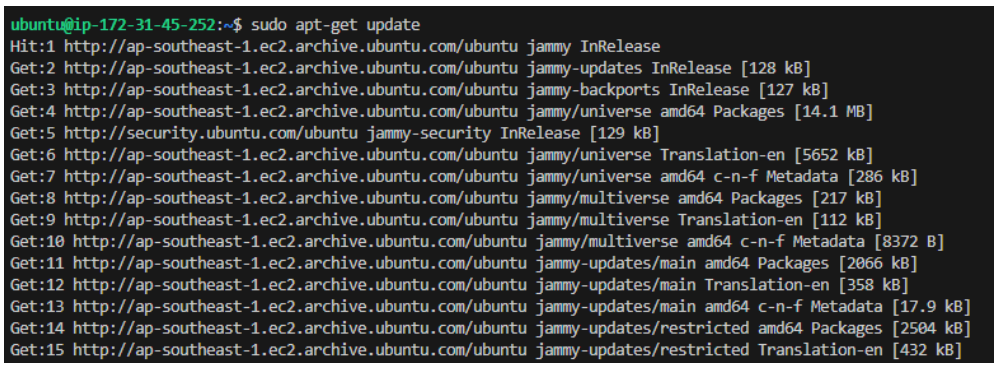
After connecting to the instance via SSH, install the Python 3 virtual environment package using the following commands:

```
sudo apt-get update
sudo apt-get upgrade
sudo apt-get install python3-venv
```

Explanation:

- `sudo apt-get update`: Updates the system's information about available packages and versions. It's an important step before installing or upgrading any software.
- `sudo apt-get upgrade`: Upgrades all installed packages to their latest versions without removing any or installing new ones to meet dependencies.
- `sudo apt-get install python3-venv`: Installs the Python 3 virtual environment module, allowing the creation of isolated Python environments. This is useful for managing dependencies for different projects.

Result:



```
ubuntu@ip-172-31-45-252:~$ sudo apt-get upgrade
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
Calculating upgrade... Done
The following packages have been kept back:
  linux-aws linux-headers-aws linux-image-aws
The following packages will be upgraded:
  apparmor apport apt apt-utils base-files bind9-dnsutils bind9-host bind9-libs busybox-initramfs busybox-static
  libcom-err2 libcurl3-gnutls libcurl4 libexpat1 libext2fs2 libgssapi-krb5-2 libk5crypto3 libkrb5-3 libkrb5suppo
  libpython3.10-minimal libpython3.10-stdlib libss2 libssl3 logsave motd-news-config openssl python-apt-common p
  python3-pkg-resources python3-problem-report python3-setuptools python3-twisted python3-update-manager python3
  ubuntu-pro-client ubuntu-pro-client-l10n ubuntu-release-upgrader-core ubuntu-server ubuntu-standard update-mar
71 upgraded, 0 newly installed, 0 to remove and 3 not upgraded.
Need to get 66.7 MB of archives.
After this operation, 273 kB of additional disk space will be used.
Do you want to continue? [Y/n] Y
```

```
ubuntu@ip-172-31-45-252:~$ sudo apt-get install python3-venv
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
The following additional packages will be installed:
  python3-pip-whl python3-setuptools-whl python3.10-venv
The following NEW packages will be installed:
  python3-pip-whl python3-setuptools-whl python3-venv python3.10-venv
0 upgraded, 4 newly installed, 0 to remove and 3 not upgraded.
Need to get 2475 kB of archives.
After this operation, 2891 kB of additional disk space will be used.
Do you want to continue? [Y/n] Y
Get:1 http://ap-southeast-1.ec2.archive.ubuntu.com/ubuntu jammy-updates/universe amd64 python3-pip-whl all 22.0.2+dfsg-1ubuntu0.4 [1680 kB]
Get:2 http://ap-southeast-1.ec2.archive.ubuntu.com/ubuntu jammy-updates/universe amd64 python3-setuptools-whl all 59.6.0-1.2ubuntu0.22.04.2 [788 kB]
Get:3 http://ap-southeast-1.ec2.archive.ubuntu.com/ubuntu jammy-updates/universe amd64 python3.10-venv amd64 3.10.12-1~22.04.6 [5722 B]
Get:4 http://ap-southeast-1.ec2.archive.ubuntu.com/ubuntu jammy-updates/universe amd64 python3-venv amd64 3.10.6-1~22.04.1 [1042 B]
Fetched 2475 kB in 5s (461 kB/s)
```

At this point, run the `sudo bash` command to start a new bash shell session with root privileges. This new session runs as the root user instead of a regular user account, granting full control over the system.

Result:

```
ubuntu@ip-172-31-45-252:~$ sudo bash
root@ip-172-31-45-252:/home/ubuntu#
```

[3] Access a directory

Create a directory at the path `/opt/wwc/mysites` using the following command and then `cd` into the directory.

```
mkdir -p /opt/wwc/mysites
cd /opt/wwc/mysites
```

[4] Set up a virtual environment

After navigating to the `/opt/wwc/mysites` directory, run the following command to set up a virtual environment. Virtual environments create an isolated workspace for each Python project, enabling to manage dependencies and Python versions without conflicts between projects, ensuring portability and consistency.

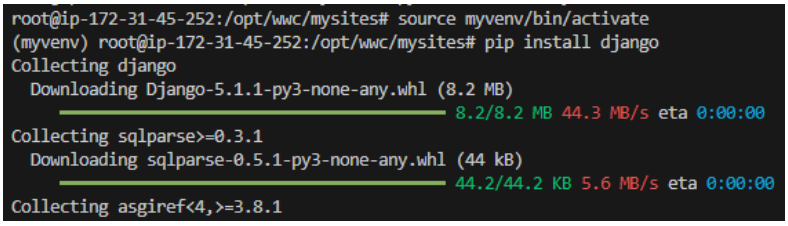
```
python3 -m venv myvenv
```

[5] Activate the virtual environment

Run the following command to activate the virtual environment. After activation, use `pip install django` to install Django in the project environment. **Django is a Python framework for quickly building and developing web applications. It provides developers with a set of tools and structures for creating internet projects ranging from simple websites to complex web applications.**

```
root@ip-172-31-45-252:/opt/wwc/mysites# source myvenv/bin/activate
(myvenv) root@ip-172-31-45-252:/opt/wwc/mysites# pip install django
```

Result:



Next, execute the following commands to create a new Django project:

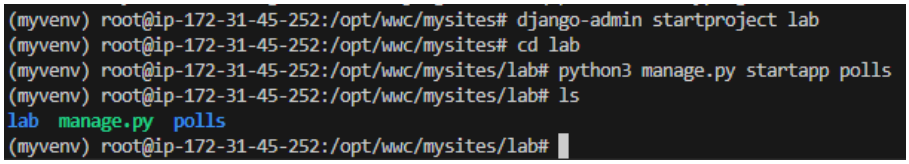
```
(myvenv) root@ip-172-31-45-252:/opt/wwc/mysites# django-admin startproject lab
(myvenv) root@ip-172-31-45-252:/opt/wwc/mysites# cd lab
(myvenv) root@ip-172-31-45-252:/opt/wwc/mysites/lab# python3 manage.py startapp polls
```

Explanation:

- `django-admin startproject lab`: Creates a new Django project called "lab," generating the basic structure and files of the Django project, including configuration files and URLs.
- `cd lab`: Switches the directory to the newly created "lab" project folder.
- `python3 manage.py startapp polls`: Creates a new application in the Django project called "polls." In Django, a project can contain multiple applications, each responsible for specific functionality.

Use the `ls` command here to check the created files.

Result:



The created files are as follows:

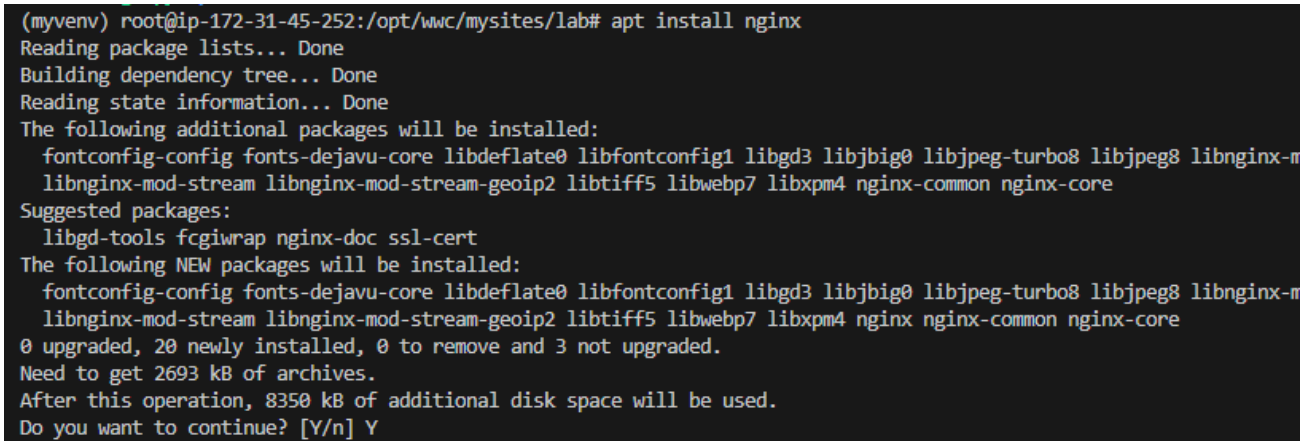
- `lab`: The main directory of the Django project, typically containing project-level settings and configuration files.
- `manage.py`: A command-line tool for interacting with the Django project. We can use it to run the server, create databases, run tests, and more. We used it to create the 'polls' application.
- `polls`: The Django application we created earlier.

[6] Install nginx

Now, use the following command to install Nginx. It is a high-performance web server and reverse proxy server, mainly used for serving static files, load balancing, and acting as a frontend proxy to application servers, improving website performance, scalability, and security.

```
apt install nginx
```

Result:



Press “Y” and hit “Enter” to complete the installation.

[7] Configure nginx

Next, we configure Nginx. Use the following command to edit the `/etc/nginx/sites-enabled/default` file:

```
nano /etc/nginx/sites-enabled/default
```

Replace the content with the configuration shown below. This configuration sets up a reverse proxy that listens to all requests on port 80 and forwards them to the backend server running on port 8000 on localhost (127.0.0.1), while preserving the original request's hostname and client IP address.

```
  GNU nano 6.2                                        /etc/nginx/sites-enabled/default *
##
# You should look at the following URL's in order to grasp a solid understanding
# of Nginx configuration files in order to fully unleash the power of Nginx.
# https://www.nginx.com/resources/wiki/start/
# https://www.nginx.com/resources/wiki/start/topics/tutorials/config_pitfalls/
# https://wiki.debian.org/Nginx/DirectoryStructure
#
# In most cases, administrators will remove this file from sites-enabled/ and
# leave it as reference inside of sites-available where it will continue to be
# updated by the nginx packaging team.
#
# This file will automatically load configuration files provided by other
# applications, such as Drupal or Wordpress. These applications will be made
# available underneath a path with that package name, such as /drupal8.
#
# Please see /usr/share/doc/nginx-doc/examples/ for more detailed examples.
##

# Default server configuration
#
server {
  listen 80 default_server;
  listen [::]:80 default_server;

  location / {
    proxy_set_header X-Forwarded-Host $host;
    proxy_set_header X-Real-IP $remote_addr;

    proxy_pass http://127.0.0.1:8000;
  }
}
```

After editing, exit with `Ctrl + X`, confirm the save with `Y`, and press `Enter` to save the filename.

[8] Restart nginx

After completing the editing, restart Nginx using the following command to apply the changes:

```
service nginx restart
```

[9] Access your EC2 instance

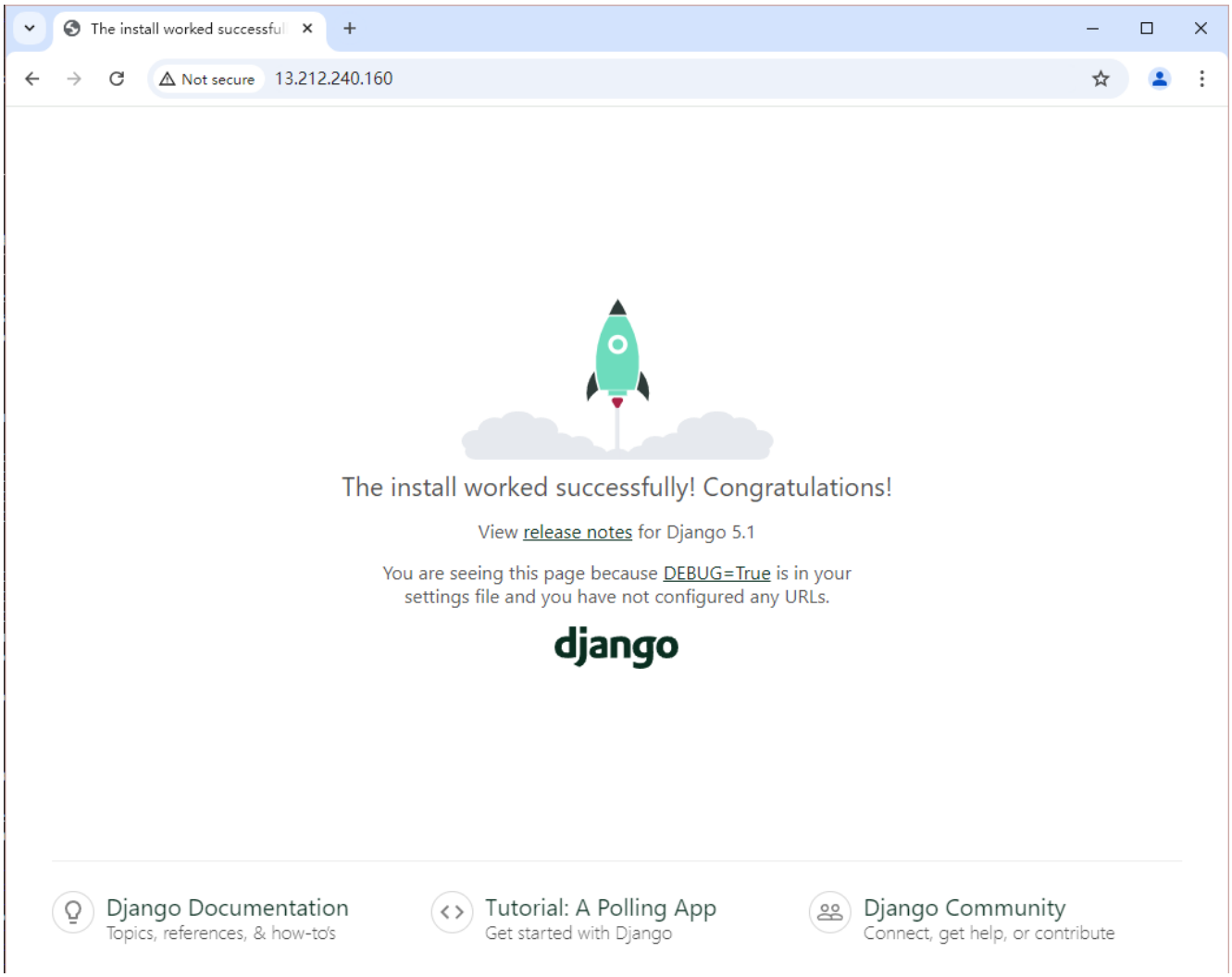
Next, run the following command in the `/opt/wwc/mysites/lab` directory to start Django's development web server on port 8000:

```
python3 manage.py runserver 8000
```

Result:

```
(myvenv) root@ip-172-31-45-252:/opt/wwc/mysites/lab# python3 manage.py runserver 8000
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).

You have 18 unapplied migration(s). Your project may not work properly until you apply the migrations for app(s): admin, auth, contenttypes, sessions.
Run 'python manage.py migrate' to apply them.
September 29, 2024 - 07:27:49
Django version 5.1.1, using settings 'lab.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

The server is now running, and we can open a browser and access the EC2 instance's IP address (13.212.240.160, the IP output after creating the EC2 instance):



We successfully see the initial Django UI, showing that the web server is up and running. Use `ctrl + C` to stop the server as per the terminal prompt.

Set up Django inside the created EC2 instance

[1] Edit the following files (create them if not exist)

Edit the files using the following commands in the current folder:

```
(myvenv) root@ip-172-31-45-252:/opt/wwc/mysites/lab# nano polls/views.py
(myvenv) root@ip-172-31-45-252:/opt/wwc/mysites/lab# nano polls/urls.py
(myvenv) root@ip-172-31-45-252:/opt/wwc/mysites/lab# nano lab/urls.py
```

Add the following content:

polls/views.py:

This file imports `HttpResponse` to return HTTP responses. It defines a view function called `index` that returns the text "Hello, world." when called.

polls/urls.py:

This file imports the `path` function to define URL patterns. It imports the `views` module from the current directory, which contains the view function defined earlier. In the `urlpatterns` list, it defines a path: the empty string `''` represents the root URL of the app, pointing to the `views.index` function. The `name='index'` assigns the URL pattern a name of 'index', which can be referenced in templates.

lab/urls.py (due to missing screenshot, the modified content is presented in text format):

```
from django.urls import include, path
from django.contrib import admin

urlpatterns = [
    path('polls/', include('polls.urls')),
    path('admin/', admin.site.urls),
]
```

This is the main URL configuration file for the project. It imports Django's admin interface. And in `urlpatterns`, it defines two paths: the `polls/` path includes all the URLs from the polls app, and the `admin/` path directs to Django's admin interface.

Together, these three files define the structure and routing of a simple Django application, where visiting `/polls/` routes the request to the `index` view of the polls app and displays "Hello, world." Visiting `/admin/` brings up Django's admin interface.

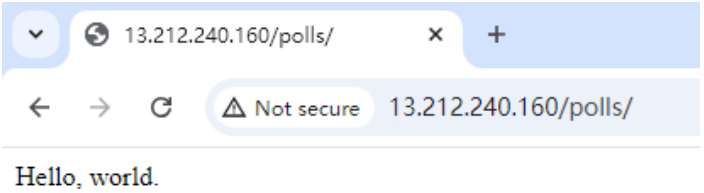
[2] Run the web server again

Run the web server again using the same command from step 9 of the previous section:

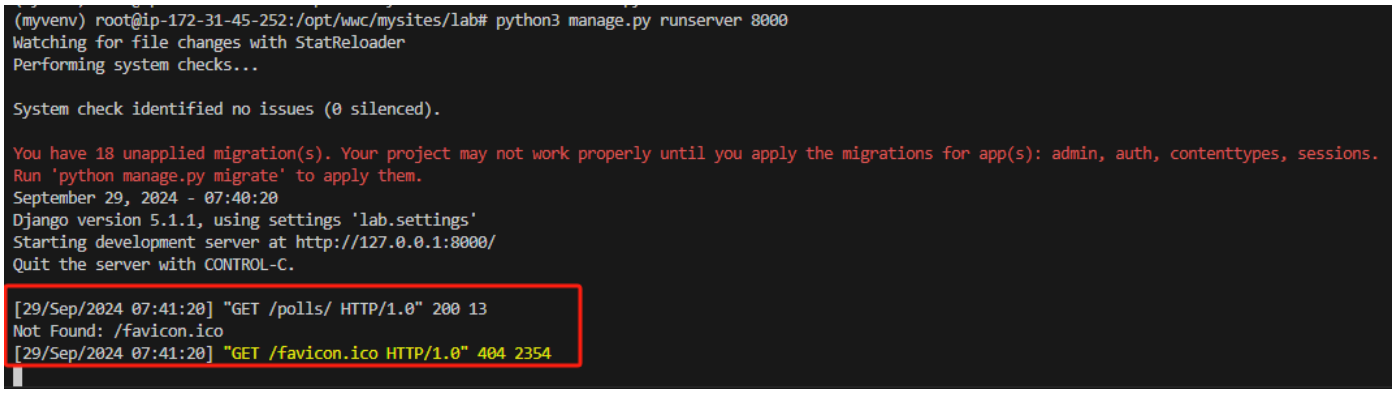
```
python3 manage.py runserver 8000
```

[3] Access the EC2 instance

Visit the URL: http://13.212.240.160/polls/ (where 13.212.240.160 is the EC2 instance's IP address), and the output looks like this:



We can see the text output "Hello, world.", from the view function edited earlier.



From the terminal output, we can also see a successful GET request to the `/polls/` path, returning a 200 OK status code, indicating that the polls app and URL configuration are working correctly.

Set up an ALB

[1] Create an application load balancer

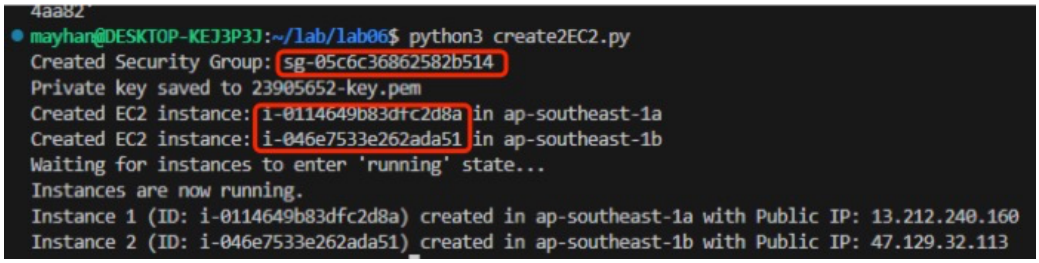
Then, write a Python script to create an ALB. This script uses Python and AWS SDK (boto3) to create and configure an Application Load Balancer (ALB). Detailed explanations of the code are provided in the comments:

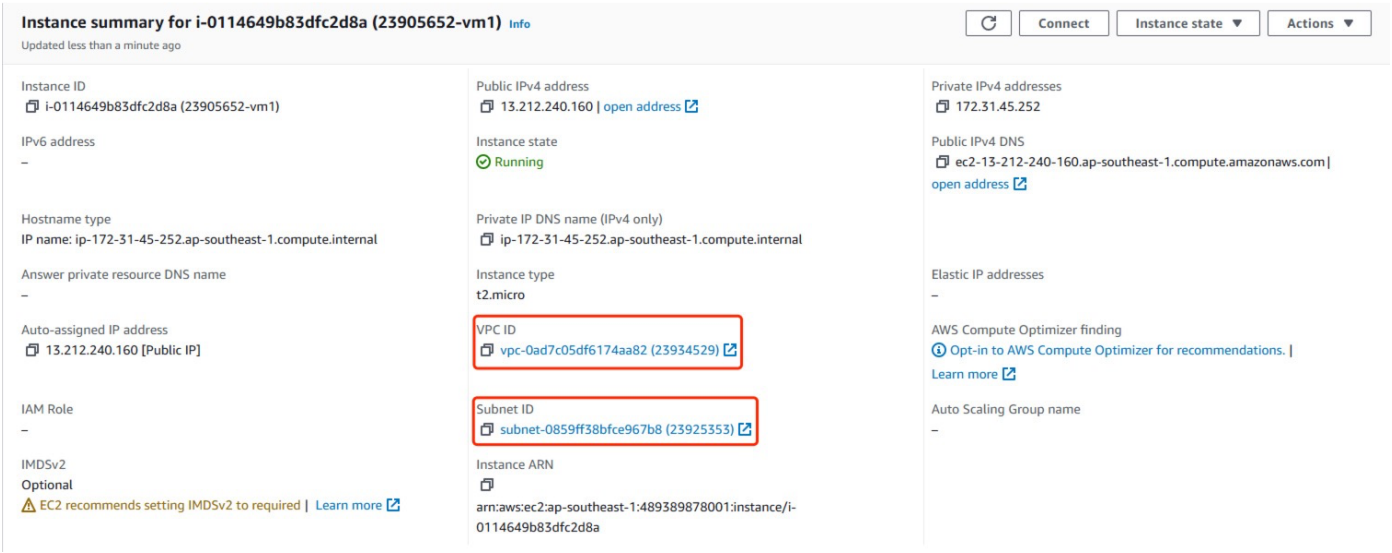
```
import boto3
import botocore
import os

student_number = "23905652"
region = "ap-southeast-1"
# Initialize AWS service clients
ec2 = boto3.client('ec2', region_name=region)
elbv2 = boto3.client('elbv2', region_name=region)

def create_load_balancer(security_group_id, subnet_ids):
    # Create an ALB in the specified subnets with the given security group
    response = elbv2.create_load_balancer(
        Name=f'{student_number}-alb', # Unique name for the ALB
        Subnets=subnet_ids, # Subnets where the ALB will be placed
        SecurityGroups=[security_group_id], # Security group for the ALB
        Scheme='internet-facing', # Makes the ALB accessible from the internet
        Type='application',  # Specifies an Application Load Balancer
        Tags=[{'Key': 'Name', 'Value': f'{student_number}-alb'},]
    )
    return response['LoadBalancers'][0]['LoadBalancerArn']

def create_target_group(vpc_id):
    # Create a target group for the ALB to route requests to EC2 instances
    response = elbv2.create_target_group(
        Name=f'{student_number}-tg',  # Unique name for the target group
        Protocol='HTTP',  # Protocol used by the targets
        Port=80,  # Port on which the targets receive traffic
```

```
        VpcId=vpc_id,  # VPC where the target group is created
        HealthCheckPort='80',  # Port used for health checks
        HealthCheckProtocol='HTTP',  # Protocol used for health checks
        HealthCheckPath='/polls/',  # Path used for health checks
        HealthCheckIntervalSeconds=30,  # Time between health checks
        HealthCheckTimeoutSeconds=5,  # Timeout for health check responses
        HealthyThresholdCount=5,  # Number of successful checks to consider healthy
        UnhealthyThresholdCount=2,  # Number of failed checks to consider unhealthy
        Matcher={
            'HttpCode': '200'  # HTTP status code indicating a healthy target
        },
        TargetType='instance'  # Targets are EC2 instances
    )
    return response['TargetGroups'][0]['TargetGroupArn']

def create_listener(load_balancer_arn, target_group_arn):
    # Create a listener for the ALB to forward traffic to the target group.
    response = elbv2.create_listener(
        LoadBalancerArn=load_balancer_arn,  # ALB to attach the listener to
        Protocol='HTTP',  # Protocol for incoming traffic
        Port=80,  # Port for incoming traffic
        DefaultActions=[{
                'Type': 'forward',  # Action type to forward traffic
                'TargetGroupArn': target_group_arn  # Target group to forward traffic to
        }]
    )
    return response['Listeners'][0]['ListenerArn']

def register_targets(target_group_arn, instance_ids):
    # Register EC2 instances as targets in the target group.
    targets = [{'Id': instance_id} for instance_id in instance_ids]

    elbv2.register_targets(TargetGroupArn=target_group_arn, Targets=targets)

def main():
    # VPC and subnet IDs where the ALB and targets will be created
    vpc_id = 'vpc-0ad7c05df6174aa82'
    subnet_ids = ['subnet-0859ff38bfce967b8', 'subnet-056a5c6c3bd465883']

    # Security group ID for the ALB
    security_group_id = 'sg-05c6c36862582b514'

    # EC2 instance IDs to be registered as targets
    instance_ids = ['i-0114649b83dfc2d8a', 'i-046e7533e262ada51']

    print("Creating Application Load Balancer...")
    load_balancer_arn = create_load_balancer(security_group_id, subnet_ids)
    print(f"Load Balancer created: {load_balancer_arn}")

    print("Creating Target Group...")
    target_group_arn = create_target_group(vpc_id)
    print(f"Target Group created: {target_group_arn}")
```

```
    print("Creating listener...")
    listener_arn = create_listener(load_balancer_arn, target_group_arn)
    print(f"Listener created: {listener_arn}")

    print("Registering targets...")
    register_targets(target_group_arn, instance_ids)
    print("Targets registered")

    print("Setup complete!")

if __name__ == "__main__":
    main()
```

The information about EC2 in the code, such as `vpc_id`, `subnet_ids`, `security_group_id`, and `instance_ids`, comes from the output in Step 1 of the previous section and the corresponding EC2 console:





(Subnet ID of Instance 2 is retrieved similarly)

Main functionality of the script:

The script automates the deployment and configuration of an Application Load Balancer (ALB) on AWS, providing the foundational infrastructure for load balancing and high availability of a web application.

1. **Create an Application Load Balancer (ALB):** Sets up an internet-facing ALB; Specifies security groups and subnets.
2. **Create a Target Group:** Configures the HTTP protocol and port 80; Sets health check parameters, including checking the path `/polls/`; Defines 'instance' as the target type.

3. **Create a Listener:** Configures the ALB to listen on port 80 using the HTTP protocol; Sets the default action to forward traffic to the target group.
4. **Register Target Instances:** Registers the specified EC2 instances to the target group.
5. **Automated Deployment:** Uses predefined VPC, subnets, security groups, and instance IDs; Executes the steps in sequence to complete the ALB setup.
6. **Feedback:** Provides feedback printing after each major step is completed.

Save the script as "createALB" and run it.

Result:

```
mayhan@DESKTOP-KEJ3P3J:~/lab/lab06$ python3 createALB.py
Creating Application Load Balancer...
Load Balancer created: arn:aws:elasticloadbalancing:ap-southeast-1:489389878001:loadbalancer/app/23905652-alb/fb6fec4f0242c455
Creating Target Group...
Target Group created: arn:aws:elasticloadbalancing:ap-southeast-1:489389878001:targetgroup/23905652-tg/54019a52b30f3681
Creating listener...
Listener created: arn:aws:elasticloadbalancing:ap-southeast-1:489389878001:listener/app/23905652-alb/fb6fec4f0242c455/03ed8b7a68b3f23b
Registering targets...
Targets registered
Setup complete!
```

From the output, we can see that the ALB has been created and set up successfully, including the target group, listener, and registered target instances.We can see the id of ALB.

We can also verify this in the ALB console, matching the output with the console details.

[2] Health check

In the previous script, we have already set up the health check:

```
def create_target_group(vpc_id):
    response = elbv2.create_target_group(
        Name=f'{student_number}-tg',
        Protocol='HTTP',
        Port=80,
        VpcId=vpc_id,
        HealthCheckPort='80',  # Port used for health checks
        HealthCheckProtocol='HTTP',  # Protocol used for health checks
        HealthCheckPath='/polls/',  # Path used for health checks
        HealthCheckIntervalSeconds=30,  # Time between health checks
        HealthCheckTimeoutSeconds=5,  # Timeout for health check responses
        HealthyThresholdCount=5,  # Number of successful checks to consider healthy
        UnhealthyThresholdCount=2,  # Number of failed checks to consider unhealthy
        Matcher={
            'HttpCode': '200'  # HTTP status code indicating a healthy target
        },
        TargetType='instance'
    )
    return response['TargetGroups'][0]['TargetGroupArn']
```

These settings define how the load balancer determines if the EC2 instances are healthy:

- Every 30 seconds, the load balancer sends an HTTP request to the `/polls/` path on port 80 of each registered target.

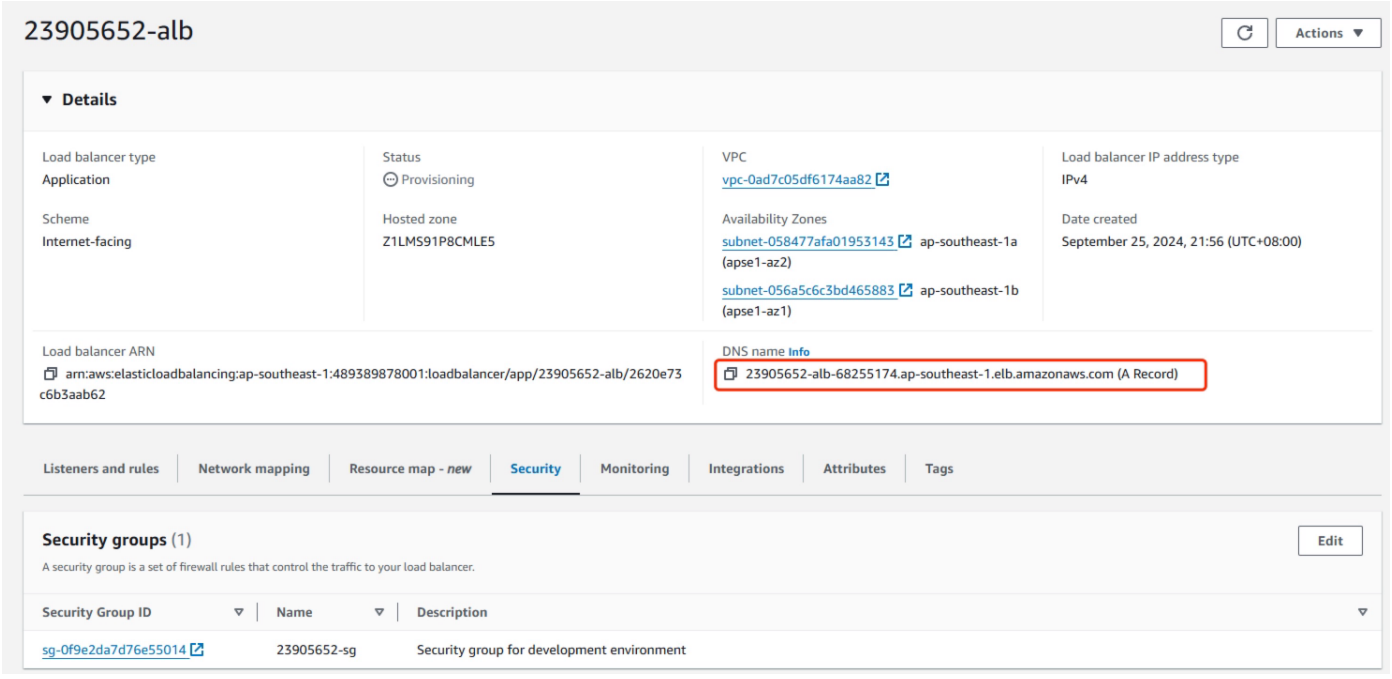
- If the target responds within 5 seconds with an HTTP 200 status code, the health check is considered successful.
- If a target passes 5 consecutive health checks, it is marked as healthy.
- If a target fails 2 consecutive health checks, it is marked as unhealthy.

After running the `python3 manage.py runserver 8000` command to start the web server, we can see the GET requests being sent every 30 seconds, indicating that the health check has been successfully set up:

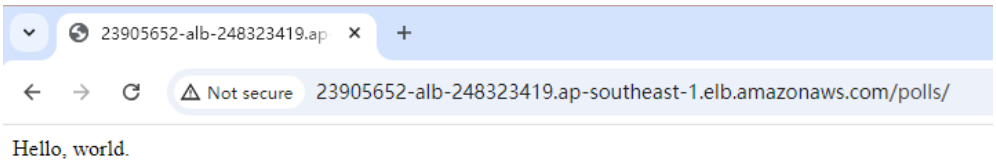
```
^C(myvenv) root@ip-172-31-45-252:/opt/wwc/mysites/lab# python3 manage.py runserver 8000
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).

You have 18 unapplied migration(s). Your project may not work properly until you apply the migrations for app(s): admin, auth, contenttypes, sessions.
Run 'python manage.py migrate' to apply them.
September 29, 2024 - 08:03:47
Django version 5.1.1, using settings 'lab.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.

[29/Sep/2024 08:03:53] "GET /polls/ HTTP/1.0" 200 13
[29/Sep/2024 08:03:59] "GET /polls/ HTTP/1.0" 200 13
[29/Sep/2024 08:04:23] "GET /polls/ HTTP/1.0" 200 13
[29/Sep/2024 08:04:29] "GET /polls/ HTTP/1.0" 200 13
[29/Sep/2024 08:04:53] "GET /polls/ HTTP/1.0" 200 13
[29/Sep/2024 08:04:59] "GET /polls/ HTTP/1.0" 200 13
```

[3] Access

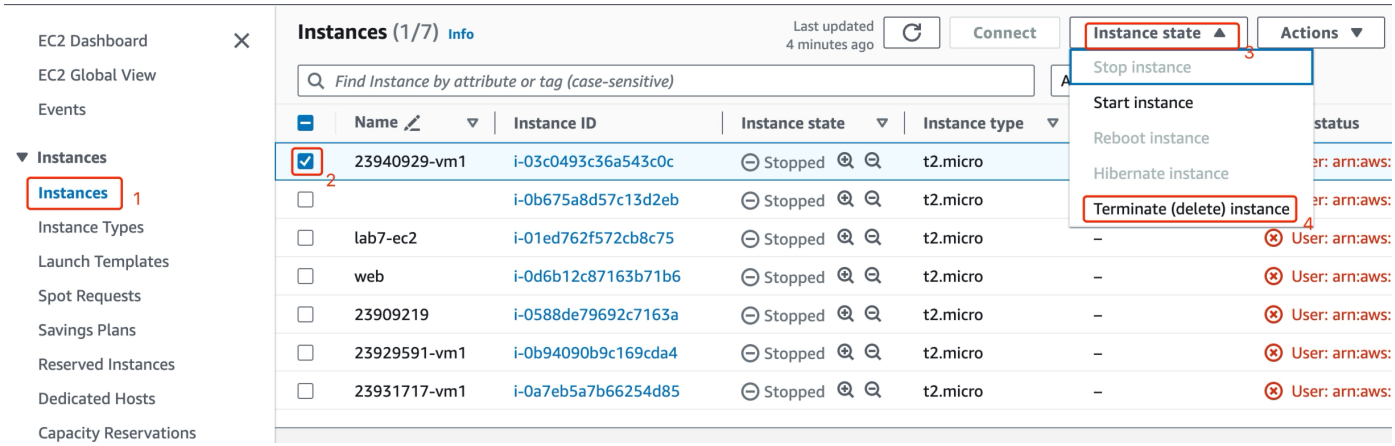
With the web server running, open a browser and access the DNS name (we can find the DNS name in the AWS console. Since the ALB console screenshot for this lab is missing, the following image is a sample screenshot showing where to find the DNS name. The DNS name in the screenshot is different from the one used in this lab):



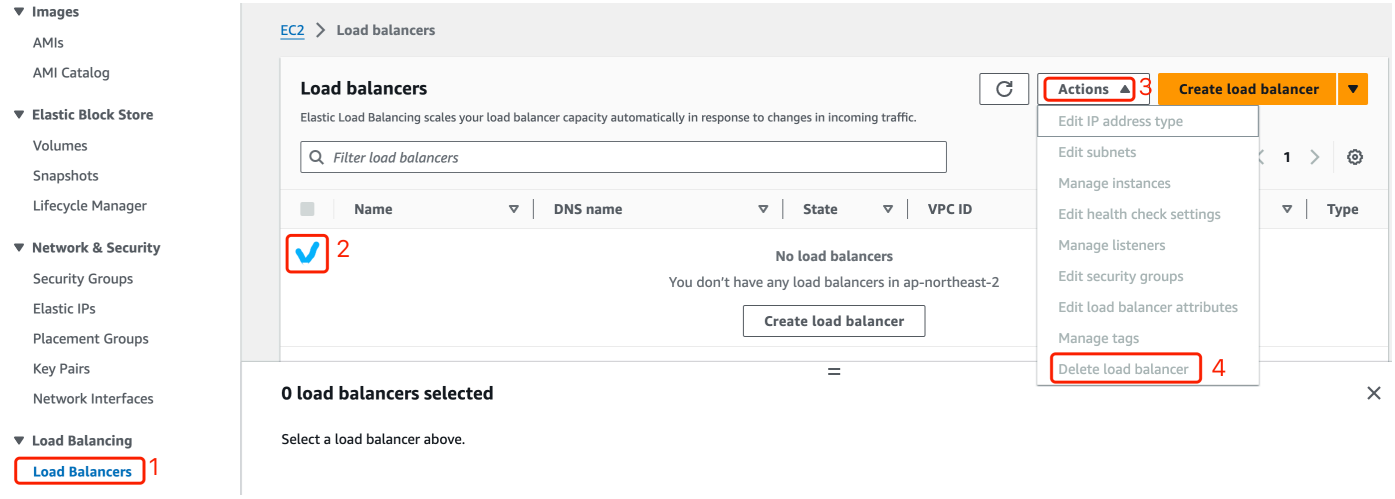
Upon entering the DNS name, the output is as follows:



After finishing the lab, we have to manually delete the created instances (23905652-vm1/23905652-vm2). In the EC2 console under the "Instances" tab, select the instance to delete, click "Instance state," and choose "Terminate(delete) instance" (the following screenshot uses another stopped instance as an example to show the steps for manually stopping the instance):



Then, navigate to the "Load Balancers" tab, and similarly, select the ALB to delete. Under the "Actions" dropdown menu, choose "Delete load balancer" to delete the ALB.



Lab 7 DevOps

Create an EC2 instance

We first use the code from lab02 to create an EC2 instance:

```
import boto3
import os

student_number = "23905652"
region = "ap-southeast-1"
# Initialize the EC2 client
ec2 = boto3.client('ec2', region)

# Create a security group
response = ec2.create_security_group(
    GroupName=f"{student_number}-sg",
    Description="security group for development environment"
)
security_group_id = response['GroupId']

# Allows HTTP (port 80) and SSH (port 22) traffic from any IP (0.0.0.0/0)
ec2.authorize_security_group_ingress(
    GroupId=security_group_id,
    IpPermissions=[
      {'IpProtocol': 'tcp', 'FromPort': 80, 'ToPort': 80, 'IpRanges': [{'CidrIp':
'0.0.0.0/0'}]},
      {'IpProtocol': 'tcp', 'FromPort': 22, 'ToPort': 22, 'IpRanges': [{'CidrIp':
'0.0.0.0/0'}]}
    ]
)
print(f"Created Security Group: {security_group_id}")

# Create a key pair for SSH access to EC2 instances
response = ec2.create_key_pair(KeyName=f"{student_number}-key")
private_key = response['KeyMaterial']
private_key_file = f"{student_number}-key.pem"
with open(private_key_file, 'w') as key_file:
    key_file.write(private_key)

# Set the correct permissions for the private key file
os.chmod(private_key_file, 0o400)

# Launch an EC2 instance
response = ec2.run_instances(
    ImageId="ami-0497a974f8d5dcef8",
    SecurityGroupIds=[security_group_id],
    MinCount=1,
    MaxCount=1,
    InstanceType="t2.micro",
```

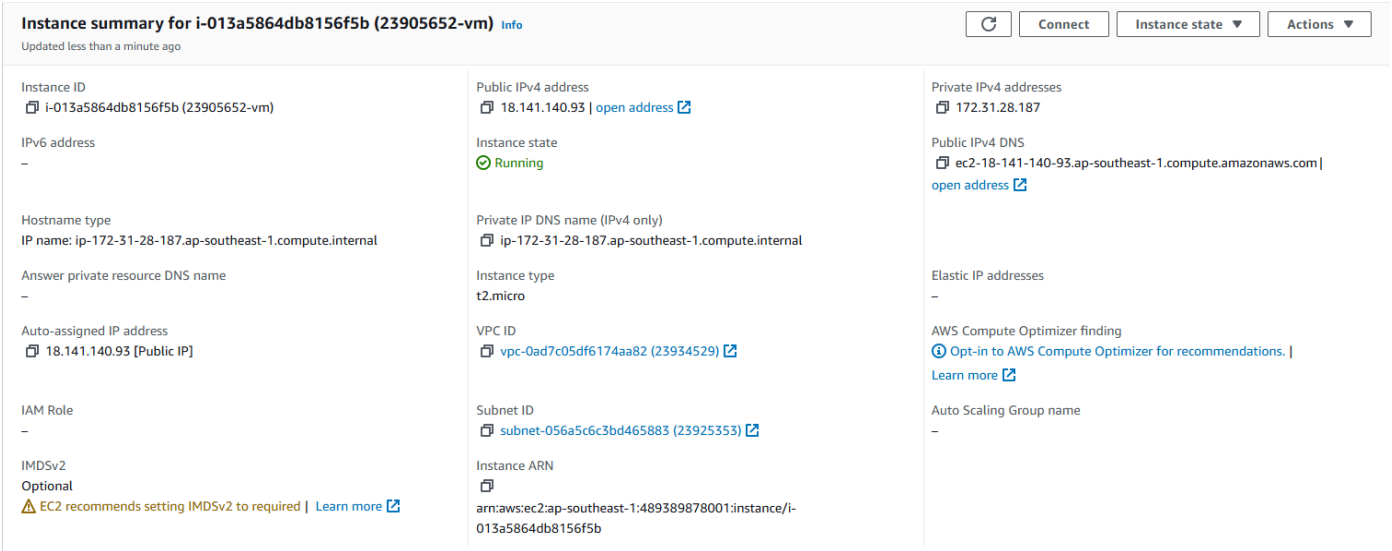
```
    KeyName=f"{student_number}-key",
    TagSpecifications=[
      { 'ResourceType': 'instance',
        'Tags': [{'Key': 'Name', 'Value': f"{student_number}-vm"}]
      }
    ]
)
print("Instance created successfully")

# Get the ID of the newly created instance
instance_id = response['Instances'][0]['InstanceId']

# Wait for the instance to enter the 'running' state
ec2.get_waiter('instance_running').wait(InstanceIds=[instance_id])

# Get the public IP address of the instance
response = ec2.describe_instances(InstanceIds=[instance_id])
public_ip_address = response['Reservations'][0]['Instances'][0]['PublicIpAddress']
print(f"Instance created successfully with Public IP: {public_ip_address}")
```

This code sets up an EC2 instance based on the student number and the corresponding AWS region. It includes creating a security group and configuring inbound rules for HTTP and SSH traffic, creating a key pair, saving the private key to a local file with appropriate permissions, and launching a new EC2 instance using the specified AMI, security group, instance type, and key pair. Upon success, it retrieves and prints the public IP address of the instance. Save the script as “createEC2.py” and run the code:

Result:

```
mayhan@DESKTOP-KEJ3P3J:~/lab/lab07$ python3 createEC2.py
Created Security Group: sg-084351f92c59d040c
Instance created successfully
Instance created successfully with Public IP: 18.141.140.93
mayhan@DESKTOP-KEJ3P3J:~/lab/lab07$
```

The EC2 instance is successfully created, and its public IP address is 18.141.140.93. We can also verify the successful creation and view detailed information in the EC2 console:

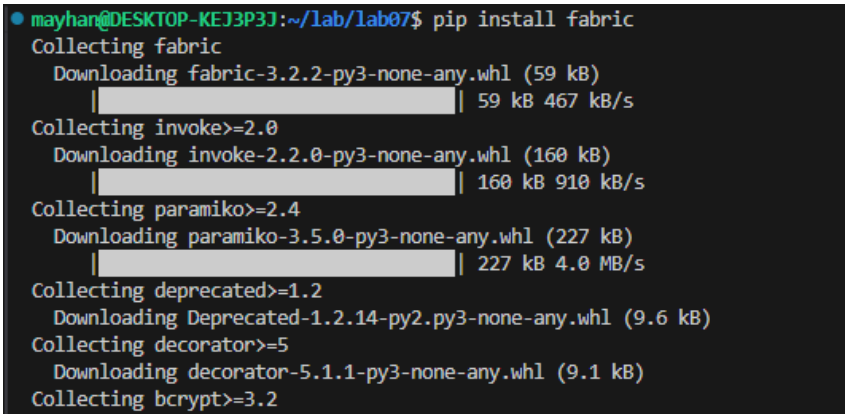


Install and configure Fabric

Install Fabric by running the following command in the terminal. Fabric is a Python library that simplifies system administration and application deployment tasks. It provides a high-level API for executing local or remote shell commands via SSH, uploading and downloading files, automating server configuration, application deployment, and routine system management tasks:

```
pip install fabric
```

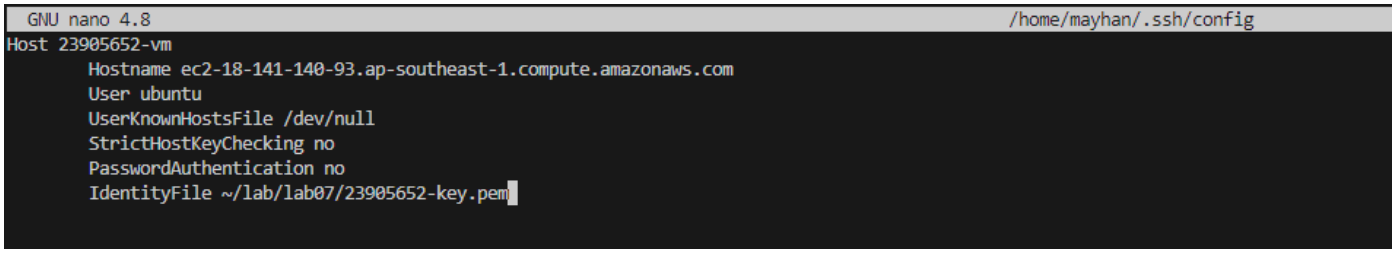
Result:



Use the following command to create a configuration file in the `~/.ssh` directory:

```
nano ~/.ssh/config
```

In edit mode, add the following content to the file as shown in the screenshot. After editing, use `Ctrl + X` to exit, press `Y` to confirm saving, and press `Enter` to save the filename:



This `.ssh/config` file is an SSH client configuration file that simplifies and customizes SSH connections. With this configuration, we can connect to the EC2 instance simply by using the `ssh 23905652-vm` command without needing to specify the full hostname, username, and key file.

Explanation:

1. `Host 23905652-vm`: This line defines an alias. We can use this alias (23905652-vm) to connect to the specified server without remembering the full hostname. The `23905652-vm` is the tag name of the EC2 instance created earlier.
2. `Hostname ec2-18-141-140-93.ap-southeast-1.compute.amazonaws.com`: This is the actual hostname or IP address of the server. Here, `ec2-18-141-140-93.ap-southeast-1.compute.amazonaws.com` is the public IPv4 DNS of the newly created instance.
3. `User ubuntu`: Specifies the username to use for the connection as "ubuntu."
4. `UserKnownHostsFile /dev/null`: Disables known host checking, so the host key will not be stored in the `known_hosts` file.

5. `StrictHostKeyChecking no`: Disables strict host key checking, meaning SSH will not ask if you trust new or changed host keys.
6. `PasswordAuthentication no`: Disables password authentication, only allowing key-based authentication.
7. `IdentityFile ~/lab/lab07/23905652-key.pem`: Specifies the path to the private key file for authentication. In this case, `~/lab/lab07/23905652-key.pem` is the path to the key file saved earlier.

After configuring, we can test the Fabric connection by running the following commands in the terminal:

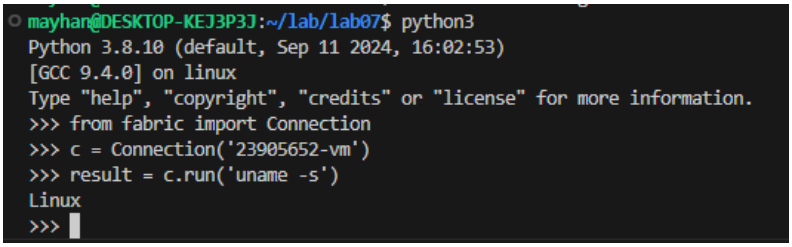
First, use the `python3` command to launch the Python 3 interpreter in the terminal, and then enter the following commands to establish an SSH connection to the remote server:

```
>>> from fabric import Connection
>>> c = Connection('23905652-vm')
>>> result = c.run('uname -s')
```

Explanation:

- `Connection`: The `Connection` class is used to establish SSH connections to remote servers.
- `c.run('uname -s')`: Executes the `uname -s` command on the remote server. The `uname -s` command is used to display the operating system name.

Result:



The command outputs `Linux`, indicating that the remote server is running a Linux operating system, and confirming that the connection using Fabric was successful.

Use Fabric for automation

Next, we will write a Python script to replicate the steps from lab06. It will automatically set up a Python 3 virtual environment, Nginx, and a Django application inside the newly created EC2 instance. We will then run the Django development server in the background on port 8000.

In the Python script, we will create functions for each step and call them sequentially in the `main` function.

[1] Set up a virtual environment

In the script, we use the Fabric library to automate the process of setting up a Python 3 virtual environment on the remote server, which includes updating the system, installing necessary packages, creating directories, setting permissions, creating a virtual environment, and installing Django inside the virtual environment. Detailed explanations of the code implementation are provided in the comments:

```
from fabric import Connection
```

```
def setup_virtual_environment(c):
    # This function sets up a Python 3 virtual environment on a remote server
    # 'c' is expected to be a Fabric Connection object
    print("Setting up Python 3 virtual environment...")

    # Update the package lists for upgrades and new package installations using sudo
    c.sudo('apt-get update')

    # Upgrade all installed packages to their latest versions using sudo
    # The -y flag automatically answers "yes" to all prompts
    c.sudo('apt-get upgrade -y')

    # Install the Python 3 venv package, which is required for creating virtual
environments
    c.sudo('apt-get install -y python3-venv')

    # Create a directory for the project
    # The -p flag creates parent directories as needed
    c.sudo('mkdir -p /opt/wwc/mysites')

    # Change the ownership of the project directory to the 'ubuntu' user
    # This allows the 'ubuntu' user to write to this directory
    c.sudo('chown ubuntu:ubuntu /opt/wwc/mysites')

    # Change the current working directory to the project directory
    # All commands inside this block will be executed in this directory
    with c.cd('/opt/wwc/mysites'):
        # Create a new Python virtual environment named 'myvenv'
        c.run('python3 -m venv myvenv')

        # Activate the virtual environment and install Django
        # The '&&' operator allows multiple commands to be chained
        # If the first command succeeds, the second one will be executed
        c.run('source myvenv/bin/activate && pip install django')
```

Save the script as "fabric_deploy_django.py" and call this function at the script's entry point to set up the virtual environment:

```
if __name__ == '__main__':
    c = Connection('23905652-vm') # The name set in the SSH config
    setup_virtual_environment(c)
```

Result (the screenshot shows the result of executing all the scripts, the first line of the virtual environment setup output is highlighted):

```
mayhan@DESKTOP-KEJ3P3J:~/lab/lab07$ python3 fabric_deploy_django.py
Setting up Python 3 virtual environment...
Hit:1 http://ap-southeast-1.ec2.archive.ubuntu.com/ubuntu jammy InRelease
Hit:2 http://ap-southeast-1.ec2.archive.ubuntu.com/ubuntu jammy-updates InRelease
Hit:3 http://ap-southeast-1.ec2.archive.ubuntu.com/ubuntu jammy-backports InRelease
Hit:4 http://security.ubuntu.com/ubuntu jammy-security InRelease
Reading package lists...
Reading package lists...
Building dependency tree...
Reading state information...
Calculating upgrade...
The following packages have been kept back:
  linux-aws linux-headers-aws linux-image-aws
0 upgraded, 0 newly installed, 0 to remove and 3 not upgraded.
Reading package lists...
Building dependency tree...
Reading state information...
python3-venv is already the newest version (3.10.6-1~22.04.1).
0 upgraded, 0 newly installed, 0 to remove and 3 not upgraded.
Requirement already satisfied: django in ./myvenv/lib/python3.10/site-packages (5.1.1)
Requirement already satisfied: sqlparse>=0.3.1 in ./myvenv/lib/python3.10/site-packages (from django) (0.5.1)
Requirement already satisfied: asgiref<4,>=3.8.1 in ./myvenv/lib/python3.10/site-packages (from django) (3.8.1)
Requirement already satisfied: typing-extensions>=4 in ./myvenv/lib/python3.10/site-packages (from asgiref<4,>=3.8.1->django) (4.12.2)
Setting up and configuring nginx...

WARNING: apt does not have a stable CLI interface. Use with caution in scripts.
```

[2] Install and configure Nginx

Next, we will write a function to set up and configure the Nginx web server. For security, we will first edit the configuration file locally and save it as "nginx.conf", then upload it to the server using the script. The file content is as follows, which matches lab06. It sets up a reverse proxy that listens to all requests on port 80 and forwards them to the backend server running on port 8000 on localhost (127.0.0.1), while preserving the original request's hostname and client IP address:

```
  GNU nano 4.8                                                    nginx.conf
server {
  listen 80 default_server;
  listen [::]:80 default_server;

  location / {
    proxy_set_header X-Forwarded-Host $host;
    proxy_set_header X-Real-IP $remote_addr;

    proxy_pass http://127.0.0.1:8000;
  }
}
```

After saving the configuration file in the same directory as the script, we add the following function to automate the process of setting up and configuring Nginx on the remote server. Detailed explanations of the code implementation are provided in the comments:

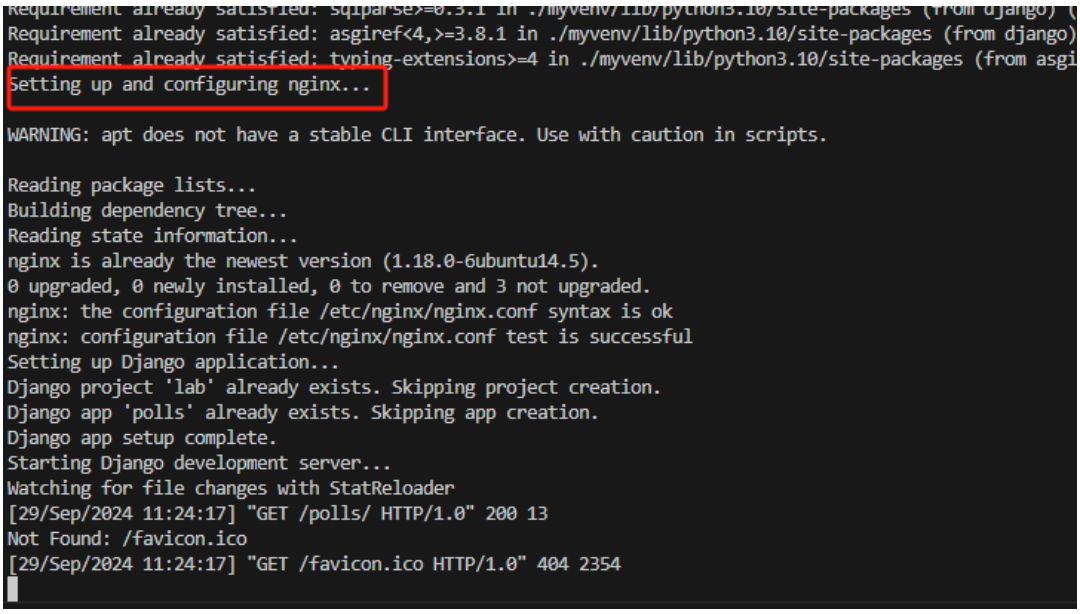
```python
def setup_nginx(c):
    # This function sets up and configures Nginx on a remote server
    # 'c' is expected to be a Fabric Connection object
    print("Setting up and configuring nginx...")

    # Install Nginx using apt-get
    c.sudo('apt install -y nginx')

    # Upload the local nginx configuration file to the remote server
    # The file is first uploaded to /tmp for security reasons
```

```
    c.put('nginx.conf', '/tmp/nginx.conf')

    # Move the uploaded configuration file to the Nginx sites-enabled directory
    c.sudo('mv /tmp/nginx.conf /etc/nginx/sites-enabled/default')

    # Test the Nginx configuration
    # The warn=True flag prevents Fabric from stopping execution if the command fails
    result = c.sudo('nginx -t', warn=True)

    # Check if the Nginx configuration test failed
    if result.failed:
        print("Nginx configuration test failed. ")
        return  # Exit the function if the configuration test failed

    # If the configuration test passed, restart Nginx to apply the new configuration
    c.sudo('systemctl restart nginx')
```

Add the `setup_nginx(c)` function call at the script's entry point and run the script:

```
if __name__ == '__main__':
    c = Connection('23905652-vm')
    setup_virtual_environment(c)
    setup_nginx(c)
```

Result（the screenshot shows the result of executing all the scripts, the first line of the Nginx setup output is highlighted）:



[3] Set up Django inside the created EC2 instance

Next, we will add the `setup_django_app()` function to automate the process of setting up a basic Django application on the remote server. This will include:

1. Creating a Django project named "lab."
2. Creating an application named "polls" inside the "lab" project.

3. Creating a simple "Hello, world" view in `index.py` inside the "polls" app.
4. Creating a `urls.py` file to set up a URL pattern for the index view.
5. Updating the main `urls.py` file in the project to include the "polls" app's URLs while retaining the admin URL.

The detailed explanation of the code implementation will be provided in the comments:

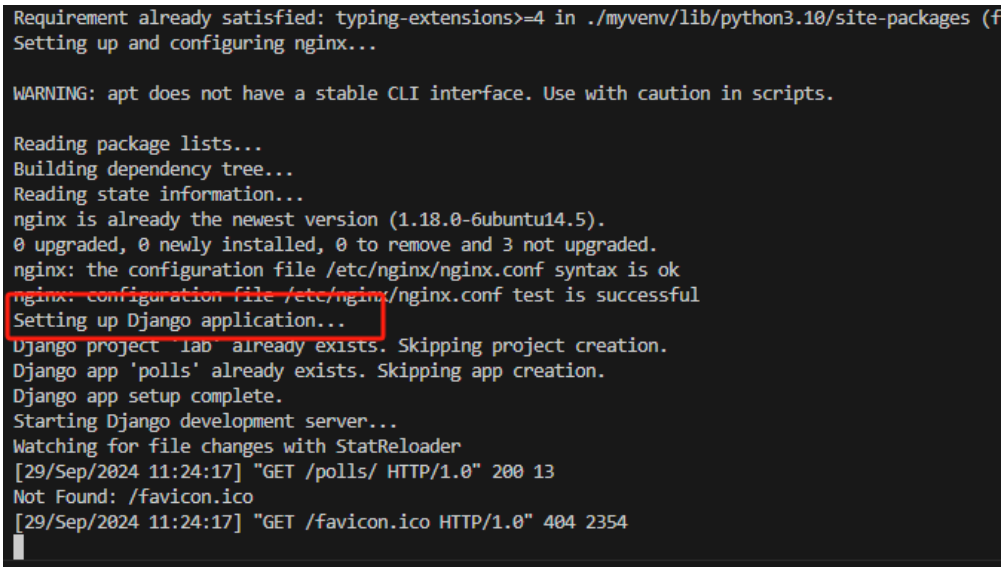
```
def setup_django_app(c):
    # This function sets up a Django application on a remote server
    # 'c' is expected to be a Fabric Connection object
    print("Setting up Django application...")

    # Change to the project directory
    with c.cd('/opt/wwc/mysites'):
        # Check if the Django project 'lab' already exists
        if c.run('test -d lab', warn=True).failed:
            # If the project doesn't exist, create a new Django project named 'lab'
            # We first activate the virtual environment to ensure we're using the correct
Python and Django versions
            c.run('source myvenv/bin/activate && django-admin startproject lab')
            print("Django project 'lab' created.")
        else:
            print("Django project 'lab' already exists. Skipping project creation.")

        # Change to the 'lab' project directory
        with c.cd('lab'):
            # Check if the 'polls' app already exists
            if c.run('test -d polls', warn=True).failed:
                # If the app doesn't exist, create a new Django app named 'polls'
                # Activate the virtual environment before running Django commands
                c.run('source ../myvenv/bin/activate && python3 manage.py startapp polls')
                print("Django app 'polls' created.")
            else:
                print("Django app 'polls' already exists. Skipping app creation.")

    # Update the views.py file with a simple view
    # This creates a basic view that returns "Hello, world" when accessed
    views_content = """
from django.http import HttpResponse

def index(request):
    return HttpResponse('Hello, world.')
    """
    # We use echo to write the content to the file
    # This is suitable for simple file modifications
    c.run(f'echo "{views_content}" > /opt/wwc/mysites/lab/polls/views.py')

    # Create the polls/urls.py file to set up URL routing for the polls app
    # This maps the root URL of the 'polls' app to the index view we just created
    urls_content = """
from django.urls import path
```

```
from . import views

urlpatterns = [
    path('', views.index, name='index'),
]
    """
    c.run(f'echo "{urls_content}" > /opt/wwc/mysites/lab/polls/urls.py')

    # Update the main urls.py file to include the polls app URLs
    # This connects the 'polls' app URLs to the main project URLconf
    # It also keeps the default admin URL
    main_urls_content = """
from django.contrib import admin
from django.urls import include, path

urlpatterns = [
    path('polls/', include('polls.urls')),
    path('admin/', admin.site.urls),
]
    """
    c.run(f'echo "{main_urls_content}" > /opt/wwc/mysites/lab/lab/urls.py')

    print("Django app setup complete.")
```

Add the above function `setup_django_app()` to the script, and include a function call in the script's entry point to set up Django:

```
if __name__ == '__main__':
    c = Connection('23905652-vm')
    setup_virtual_environment(c)
    setup_nginx(c)
    setup_django_app(c)
```

Result（the screenshot shows the result of executing all the scripts, the first line of the Django setup output is highlighted）:



[4] Run the web server

Finally, write a function to run the web server:

```
def run_django_server(c):
    # This function starts the Django development server on a remote machine
    # 'c' is expected to be a Fabric Connection object

    print("Starting Django development server...")
    c.run('source /opt/wwc/mysites/myvenv/bin/activate && cd /opt/wwc/mysites/lab &&
python3 manage.py runserver 8000 &')
```

Explanation:

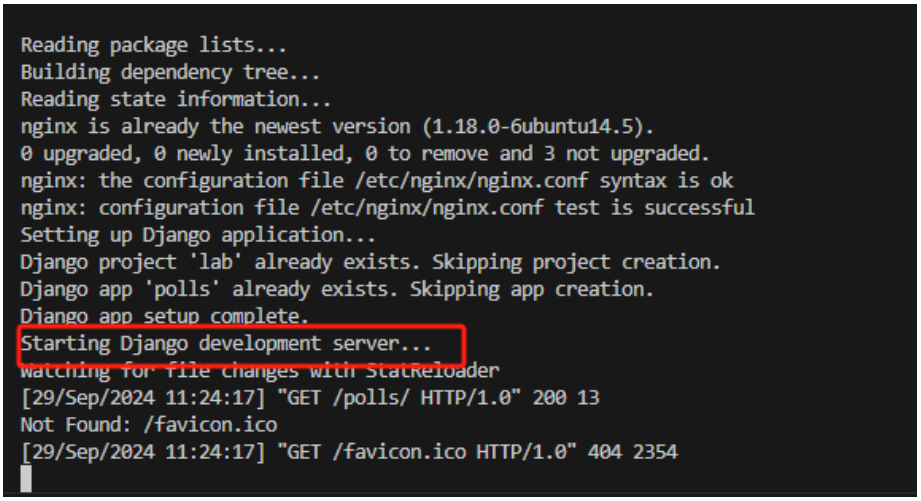
`source /opt/wwc/mysites/myvenv/bin/activate && cd /opt/wwc/mysites/lab && python3 manage.py runserver 8000 &` is a shell command composed of multiple parts. The `&&` operator ensures that the next command runs only if the previous one succeeds. The `&` operator at the end means running the command in the background, allowing the SSH session to end without terminating the server.

This combined command first sets up the correct environment (virtual environment and working directory) and then starts the application within this environment.

We will add the above function to the script and call it at the script's entry point to run the web server. At the end, add a `print` statement to indicate the successful completion of the entire script process:

```
if __name__ == '__main__':
    c = Connection('23905652-vm')
    setup_virtual_environment(c)
    setup_nginx(c)
    setup_django_app(c)
    run_django_server(c)

    print("Django deployment complete.")
```

Result（the screenshot shows the result of executing all the scripts, the first line of the web server output is highlighted）:



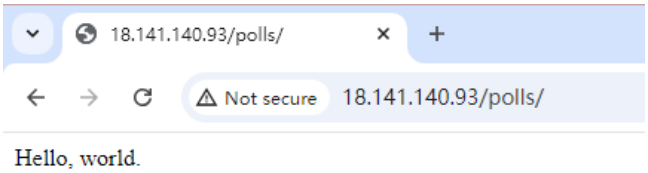
[5] Access the EC2 instance

After completing the entire script process, we access the URL in the browser: http://18.141.140.93/polls/ (where 18.141.140.93 is the EC2 IP address), and the output is as follows.

Process completed:

```
Reading package lists...
Building dependency tree...
Reading state information...
nginx is already the newest version (1.18.0-6ubuntu14.5).
0 upgraded, 0 newly installed, 0 to remove and 3 not upgraded.
nginx: the configuration file /etc/nginx/nginx.conf syntax is ok
nginx: configuration file /etc/nginx/nginx.conf test is successful
Setting up Django application...
Django project 'lab' already exists. Skipping project creation.
Django app 'polls' already exists. Skipping app creation.
Django app setup complete.
Starting Django development server...
Watching for file changes with StatReloader
[29/Sep/2024 11:24:17] "GET /polls/ HTTP/1.0" 200 13
Not Found: /favicon.ico
[29/Sep/2024 11:24:17] "GET /favicon.ico HTTP/1.0" 404 2354
```

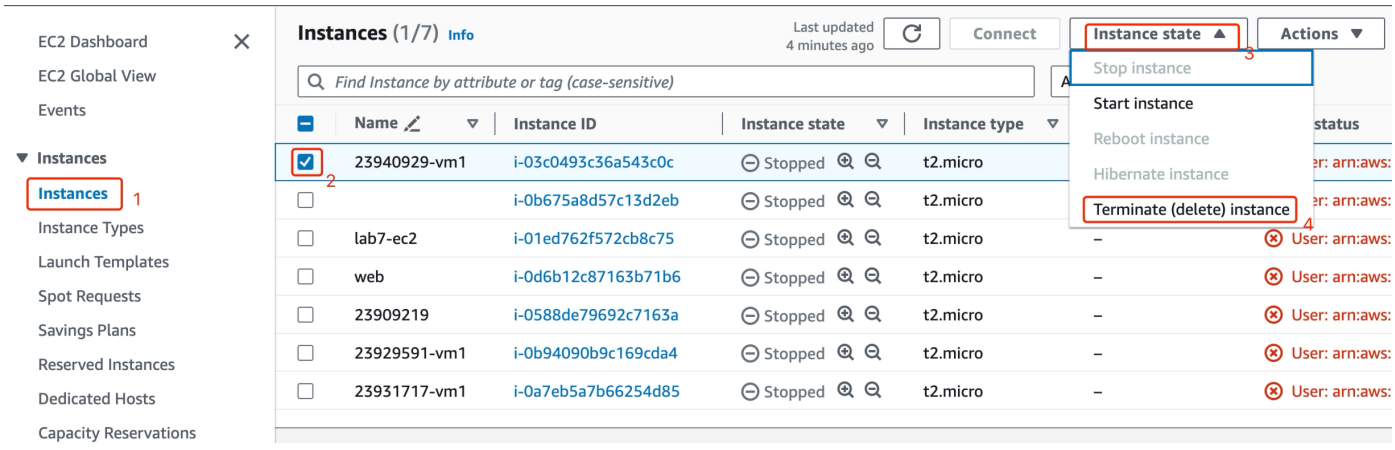
Browser output:



We successfully see the output "Hello, world." from the view function.

[6] Manually stop the EC2 instance

After the experiment, manually delete the created instance (23905652-vm). In the EC2 console under the "Instances" tab, select the instance to delete, click "Instance state," and choose "Terminate (delete) instance" (this screenshot shows another stopped instance as an example to demonstrate the manual stopping process):



Lab 8 AI

Install and run

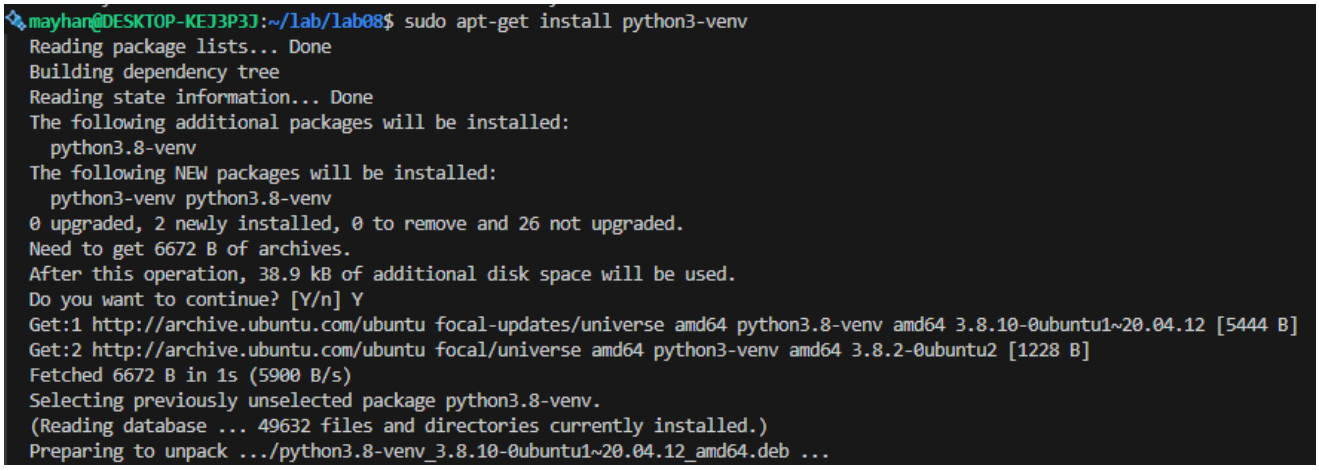
To install and run the notebook package, we first create a new Python virtual environment for this lab using `venv`.

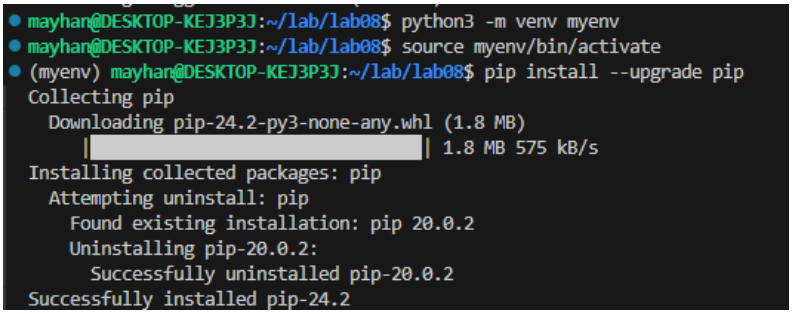
Run the following commands in the terminal:

```
mayhan@DESKTOP-KEJ3P3J:~/lab/lab08$ sudo apt-get install python3-venv
mayhan@DESKTOP-KEJ3P3J:~/lab/lab08$ python3 -m venv myenv
mayhan@DESKTOP-KEJ3P3J:~/lab/lab08$ source myenv/bin/activate
(myenv) mayhan@DESKTOP-KEJ3P3J:~/lab/lab08$ pip install --upgrade pip
```

These commands create and activate a Python virtual environment, ensuring that `pip` is up to date within that environment.

Result:

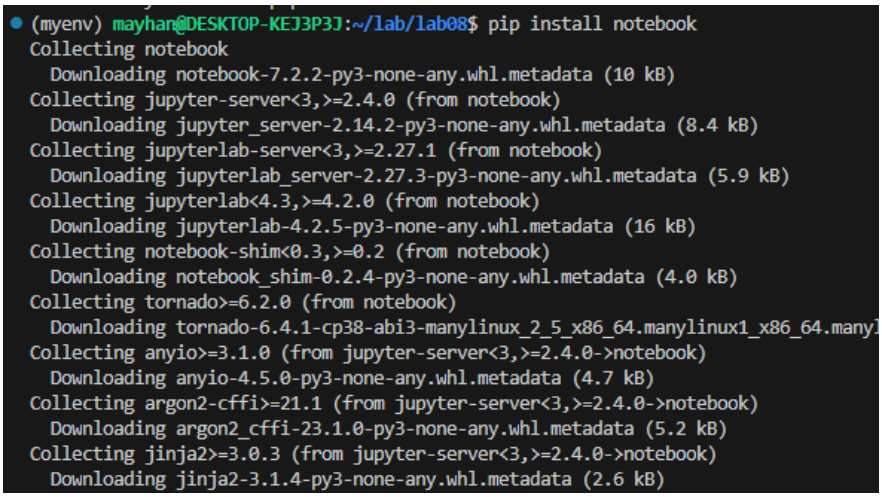




Now that the environment is set up, we install the `notebook` package using the following command:

```
(myenv) mayhan@DESKTOP-KEJ3P3J:~/lab/lab08$ pip install notebook
```

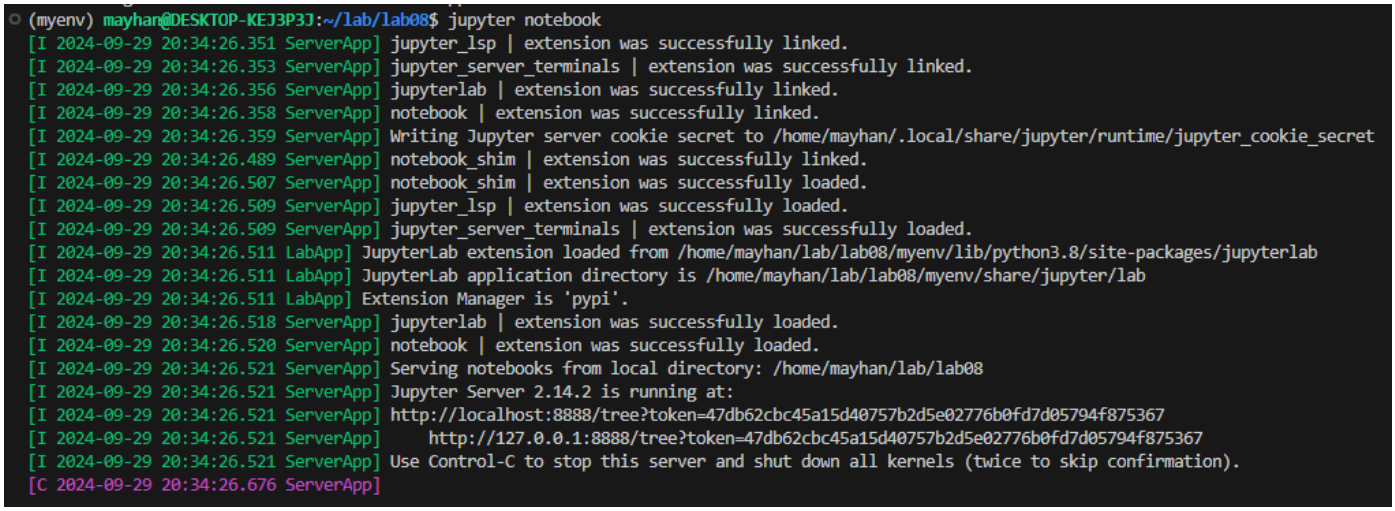
Result:



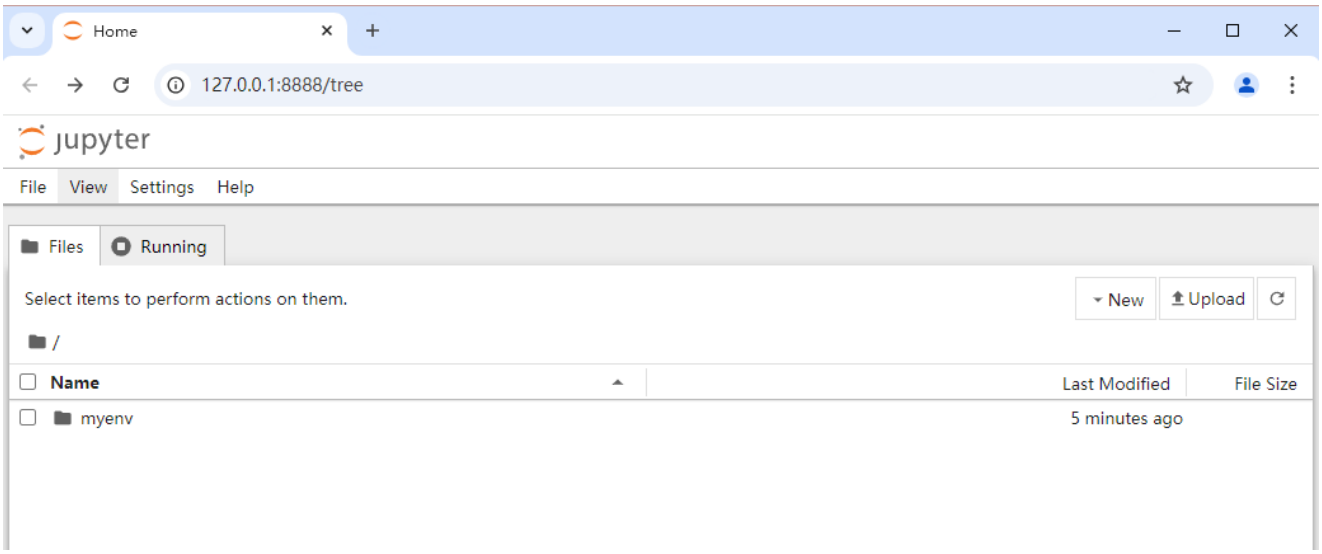
To run Jupyter Notebooks, which is an interactive development environment that allows users to combine code, text descriptions, mathematical equations, visualizations, and rich media in a single document, we execute the following command:

```
(myenv) mayhan@DESKTOP-KEJ3P3J:~/lab/lab08$ jupyter notebook
```

Result:



This command opens the browser and accesses http://localhost:8888/tree to open the Jupyter notebook home page, where we can see the contents of the folder.



Install ipykernel

Next, we can install `ipykernel` using the following command. IPykernel is a core component of the Jupyter ecosystem, providing the kernel for the Python programming language. It allows Jupyter Notebooks and other Jupyter frontends (like JupyterLab) to interact with the Python interpreter, enabling users to run Python code and retrieve results in a Jupyter environment.

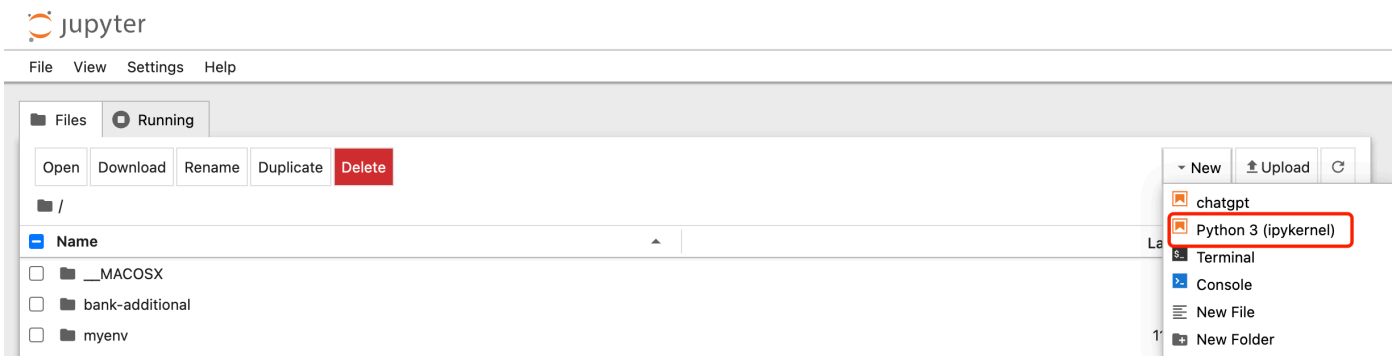
```
(myenv) mayhan@DESKTOP-KEJ3P3J:~/lab/lab08$ pip install ipykernel
```

Result:

```
(myenv) mayhan@DESKTOP-KEJ3P3J:~/lab/lab08$ pip install ipykernel
Requirement already satisfied: ipykernel in ./myenv/lib/python3.8/site-packages (6.29.5)
Requirement already satisfied: comm>=0.1.1 in ./myenv/lib/python3.8/site-packages (from ipykernel) (0.2.2)
Requirement already satisfied: debugpy>=1.6.5 in ./myenv/lib/python3.8/site-packages (from ipykernel) (1.8.6)
Requirement already satisfied: ipython>=7.23.1 in ./myenv/lib/python3.8/site-packages (from ipykernel) (8.12.3)
Requirement already satisfied: jupyter-client>=6.1.12 in ./myenv/lib/python3.8/site-packages (from ipykernel) (8.6.3)
Requirement already satisfied: jupyter-core!=5.0.*,>=4.12 in ./myenv/lib/python3.8/site-packages (from ipykernel) (5.7.2)
Requirement already satisfied: matplotlib-inline>=0.1 in ./myenv/lib/python3.8/site-packages (from ipykernel) (0.1.7)
Requirement already satisfied: nest-asyncio in ./myenv/lib/python3.8/site-packages (from ipykernel) (1.6.0)
Requirement already satisfied: packaging in ./myenv/lib/python3.8/site-packages (from ipykernel) (24.1)
Requirement already satisfied: psutil in ./myenv/lib/python3.8/site-packages (from ipykernel) (6.0.0)
Requirement already satisfied: pyzmq>=24 in ./myenv/lib/python3.8/site-packages (from ipykernel) (26.2.0)
Requirement already satisfied: tornado>=6.1 in ./myenv/lib/python3.8/site-packages (from ipykernel) (6.4.1)
```

Run hyperparameter tuning jobs

We will use Jupyter Notebook to complete the upcoming tasks. In the Jupyter Notebook homepage in the browser, create a new “Python 3 (ipykernel)” notebook and name it “lab07.ipynb.”



[1] Install libraries

Enter the following commands in a cell to install the `sagemaker`, `pandas`, and `numpy` libraries:

```
[1]: # Install SageMaker via jupyter notebook
!pip install sagemaker
# Install pandas and numpy jupyter notebook
!pip install pandas
!pip install numpy
```

```
Collecting sagemaker
  Downloading sagemaker-2.232.2-py3-none-any.whl.metadata (16 kB)
Collecting attrs<24,>=23.1.0 (from sagemaker)
  Using cached attrs-23.2.0-py3-none-any.whl.metadata (9.5 kB)
Collecting boto3<2.0,>=1.34.142 (from sagemaker)
  Downloading boto3-1.35.38-py3-none-any.whl.metadata (6.7 kB)
Collecting cloudpickle==2.2.1 (from sagemaker)
  Downloading cloudpickle-2.2.1-py3-none-any.whl.metadata (6.9 kB)
Collecting docker (from sagemaker)
  Downloading docker-7.1.0-py3-none-any.whl.metadata (3.8 kB)
Collecting google-pasta (from sagemaker)
  Downloading google_pasta-0.2.0-py3-none-any.whl.metadata (814 bytes)
Collecting importlib-metadata<7.0,>=1.4.0 (from sagemaker)
  Downloading importlib_metadata-6.11.0-py3-none-any.whl.metadata (4.9 kB)
Requirement already satisfied: jsonschema in ./myenv/lib/python3.12/site-packages (from sagemaker) (4.23.0)
Collecting numpy<2.0,>=1.9.0 (from sagemaker)
```

Explanation:

- `!pip install sagemaker` installs the SageMaker library. SageMaker is Amazon AWS's machine learning platform, and this library allows you to use SageMaker's features in Python.
- `!pip install pandas` installs the pandas library. Pandas is a powerful data analysis and manipulation library, providing efficient data structures like DataFrames for working with structured data.
- `!pip install numpy` installs the NumPy library. NumPy is a foundational library for scientific computing, providing multidimensional arrays and various associated tools.
- The `!` at the beginning indicates this is a shell command, not Python code. This allows us to run system commands like pip installations directly in a notebook.

[2] Prepare a SageMaker session

Continue by entering the following Python script in the cell to set up the environment for the machine learning project using Amazon SageMaker and other AWS services.

In the script, we import necessary libraries, create AWS service clients, retrieve the SageMaker role ARN, and set up variables for AWS resources. Finally, we create an S3 bucket and a folder within the bucket to store project-related data. The code implementation is explained in the comments:

```
# Import necessary libraries
import sagemaker  # Amazon SageMaker library for machine learning tasks
import boto3  # AWS SDK for Python, used to interact with various AWS services

# Import data processing libraries
import numpy as np  # NumPy: For matrix operations and numerical processing
import pandas as pd  # Pandas: For handling and analyzing structured data

# Import time-related functions and OS module
from time import gmtime, strftime  # For working with timestamps
import os  # For interacting with the operating system

# Set up AWS service clients
```

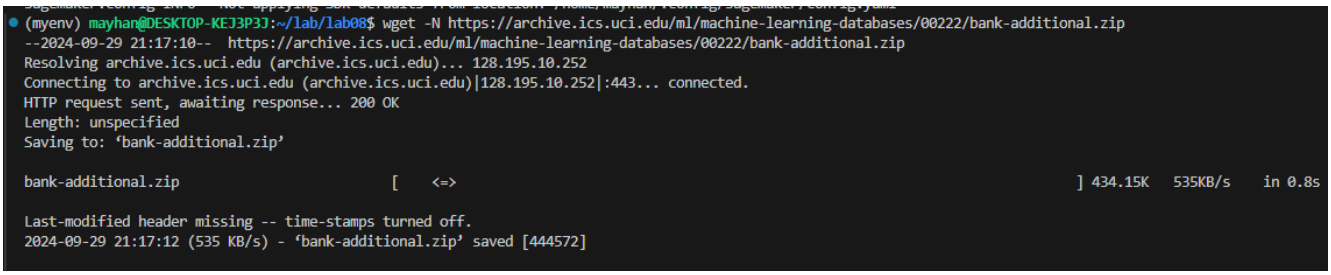
```
smclient = boto3.Session().client("sagemaker")  # Create a SageMaker client
iam = boto3.client('iam')  # Create an IAM (Identity and Access Management) client

# Retrieve the ARN (Amazon Resource Name) for the SageMaker role
sagemaker_role = iam.get_role(RoleName='SageMakerRole')['Role']['Arn']

# Set up variables for AWS resources
region = 'ap-southeast-1'  # Specify the AWS region (Singapore in this case)
student_id = "23905652"  # Set the student ID
bucket = '23905652-lab8'  # Define the S3 bucket name
prefix = f"sagemaker/{student_id}-hpo-xgboost-dm"  # Define the S3 object prefix

# Create an S3 bucket
s3 = boto3.resource('s3')  # Create an S3 resource
s3.create_bucket(
    Bucket=bucket,
    CreateBucketConfiguration={'LocationConstraint': region}
)  # Create the bucket in the specified region

# Create a folder (object) in the S3 bucket
s3.Object(bucket, prefix + '/').put(Body='')  # Create an empty object (folder) in the
bucket
```

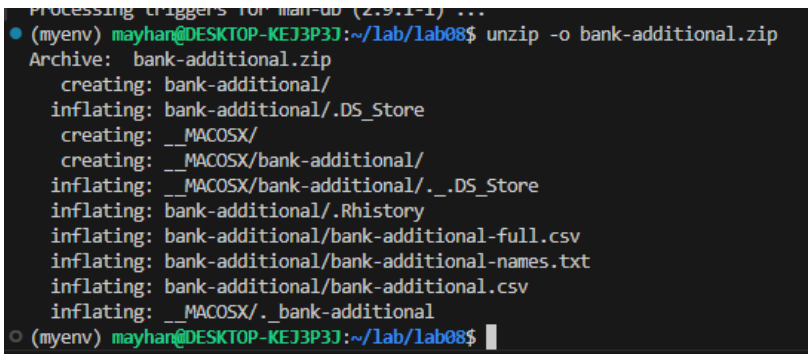
[3] Download a dataset

1. Download the marketing dataset from UCI's ML Repository and unzip it using the following commands:

```
wget -N https://archive.ics.uci.edu/ml/machine-learning-databases/00222/bank-
additional.zip
unzip -o bank-additional.zip
```

Result:





2. After unzipping, enter the following script in a Jupyter Notebook cell to load the CSV file into a pandas DataFrame, allow more columns (up to 500) and rows (up to 50) to be displayed, and then display the DataFrame.

```
import sagemaker
import boto3

import numpy as np
import pandas as pd
from time import gmtime, strftime
import os

# Read the CSV file into a pandas DataFrame
# The file is located at "./bank-additional/bank-additional-full.csv" and uses
semicolon as separator
data = pd.read_csv("./bank-additional/bank-additional-full.csv", sep=";")

# Configure pandas display options
pd.set_option("display.max_columns", 500)  # Set maximum number of columns to display
to 500
pd.set_option("display.max_rows", 50)  # Set maximum number of rows to display to 50

# Display the DataFrame
data
```

Result:

```
import sagemaker
import boto3

import numpy as np
import pandas as pd
from time import gmtime, strftime
import os

data = pd.read_csv("./bank-additional/bank-additional-full.csv", sep=";")
pd.set_option("display.max_columns", 500)  # Make sure we can see all of the columns
pd.set_option("display.max_rows", 50)  # Keep the output on one page
data
```

```
sagemaker.config INFO - Not applying SDK defaults from location: /etc/xdg/sagemaker/config.yaml
sagemaker.config INFO - Not applying SDK defaults from location: /home/mayhan/.config/sagemaker/config.yaml
```

	age	job	marital	education	default	housing	loan	contact	month	day_of_week	duration	campaign	pdays	previous	poutcor
0	56	housemaid	married	basic.4y	no	no	no	telephone	may	mon	261	1	999	0	nonexiste
1	57	services	married	high.school	unknown	no	no	telephone	may	mon	149	1	999	0	nonexiste
2	37	services	married	high.school	no	yes	no	telephone	may	mon	226	1	999	0	nonexiste
3	40	admin.	married	basic.6y	no	no	no	telephone	may	mon	151	1	999	0	nonexiste
4	56	services	married	high.school	no	no	yes	telephone	may	mon	307	1	999	0	nonexiste
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
41183	73	retired	married	professional.course	no	yes	no	cellular	nov	fri	334	1	999	0	nonexiste
41184	46	blue-collar	married	professional.course	no	no	no	cellular	nov	fri	383	1	999	0	nonexiste
41185	56	retired	married	university.degree	no	yes	no	cellular	nov	fri	189	2	999	0	nonexiste
41186	44	technician	married	professional.course	no	no	no	cellular	nov	fri	442	1	999	0	nonexiste
41187	74	retired	married	professional.course	no	yes	no	cellular	nov	fri	239	3	999	1	failu

41188 rows × 21 columns

From this result, we observe that the dataset has 41188 rows and 21 columns. It contains various types of data, including numerical (age, duration, campaign, etc.) and categorical (job, marital, education, etc.). This dataset seems to be related to a bank marketing campaign, likely for predicting whether customers will subscribe to a bank product such as a term deposit.

3. To further confirm which variables are categorical and which are numerical in the dataset, we write the following script:

```
# Find numerical columns
numerical_columns = data.select_dtypes(include=[np.number]).columns.tolist()
print("numerical_colums:", numerical_columns)

# Find categorical columns
categorical_columns = data.select_dtypes(include=[object]).columns.tolist()
print("categorical_columns:", categorical_columns)
```

Explanation:

- `select_dtypes()` is used to select columns of a specific data type.
- `include=[np.number]` selects all numerical columns (like integers and floats).
- `include=[object]` selects all object type columns, typically "string" or combined type, representing categorical data.
- `.columns` gets the names of the selected columns.
- `.tolist()` converts the column names to a Python list.

Result:

```
# Find numerical columns
numerical_columns = data.select_dtypes(include=[np.number]).columns.tolist()
print("numerical_colums:", numerical_columns)

# Find categorical columns
categorical_columns = data.select_dtypes(include=[object]).columns.tolist()
print("categorical_columns:", categorical_columns)
```

```
numerical_colums: ['age', 'duration', 'campaign', 'pdays', 'previous', 'emp.var.rate', 'cons.price.idx', 'cons.conf.idx', 'euribor3m', 'nr.employed']
categorical_columns: ['job', 'marital', 'education', 'default', 'housing', 'loan', 'contact', 'month', 'day_of_week', 'poutcome', 'y']
```

Answer the questions:

1. **Which variables in the dataset are categorical? Give at least four variables.**

   A: 'age', 'duration', 'campaign', 'pdays', 'previous'...

2. **Which variables in the dataset are numerical? Give at least four variables.**

   A: 'job', 'marital', 'education', 'default', 'housing'...

4. Use the following code to create new binary features:

```
data["no_previous_contact"] = np.where(
    data["pdays"] == 999, 1, 0
)  # Indicator variable to capture when pdays takes a value of 999
data["not_working"] = np.where(
    np.in1d(data["job"], ["student", "retired", "unemployed"]), 1, 0
)  # Indicator for individuals not actively employed
model_data = pd.get_dummies(data)  # Convert categorical variables to sets of
indicators
model_data
```

Explanation:

- `data["no_previous_contact"] = np.where(data["pdays"] == 999, 1, 0)`: Creates a new binary feature called "no_previous_contact". If the "pdays" column has a value of 999, the new feature is 1; otherwise, it's 0.
- `data["not_working"] = np.where(np.in1d(data["job"], ["student", "retired", "unemployed"]), 1, 0)`: Creates another binary feature called "not_working". If the "job" column has a value of "student", "retired", or "unemployed", the new feature is 1; otherwise, it's 0. This feature identifies individuals who are not actively employed.
- `model_data = pd.get_dummies(data)`: Uses the `get_dummies()` function from pandas to perform one-hot encoding on all categorical variables in the DataFrame. This creates a new binary column for each unique value of the categorical variables.

Result:

[3]:
```
data["no_previous_contact"] = np.where(
    data["pdays"] == 999, 1, 0
)  # Indicator variable to capture when pdays takes a value of 999
data["not_working"] = np.where(
    np.in1d(data["job"], ["student", "retired", "unemployed"]), 1, 0
)  # Indicator for individuals not actively employed
model_data = pd.get_dummies(data)  # Convert categorical variables to sets of indicators
model_data
```

[3]:

	age	duration	campaign	pdays	previous	emp.var.rate	cons.price.idx	cons.conf.idx	euribor3m	nr.employed	no_previous_contact	not_working	job
0	56	261	1	999	0	1.1	93.994	-36.4	4.857	5191.0	1	0	
1	57	149	1	999	0	1.1	93.994	-36.4	4.857	5191.0	1	0	
2	37	226	1	999	0	1.1	93.994	-36.4	4.857	5191.0	1	0	
3	40	151	1	999	0	1.1	93.994	-36.4	4.857	5191.0	1	0	
4	56	307	1	999	0	1.1	93.994	-36.4	4.857	5191.0	1	0	
...	...	...	...	...	...	...	...	...	...	...	...	...	...
41183	73	334	1	999	0	-1.1	94.767	-50.8	1.028	4963.6	1	1	
41184	46	383	1	999	0	-1.1	94.767	-50.8	1.028	4963.6	1	0	
41185	56	189	2	999	0	-1.1	94.767	-50.8	1.028	4963.6	1	1	
41186	44	442	1	999	0	-1.1	94.767	-50.8	1.028	4963.6	1	0	
41187	74	239	3	999	1	-1.1	94.767	-50.8	1.028	4963.6	1	1	

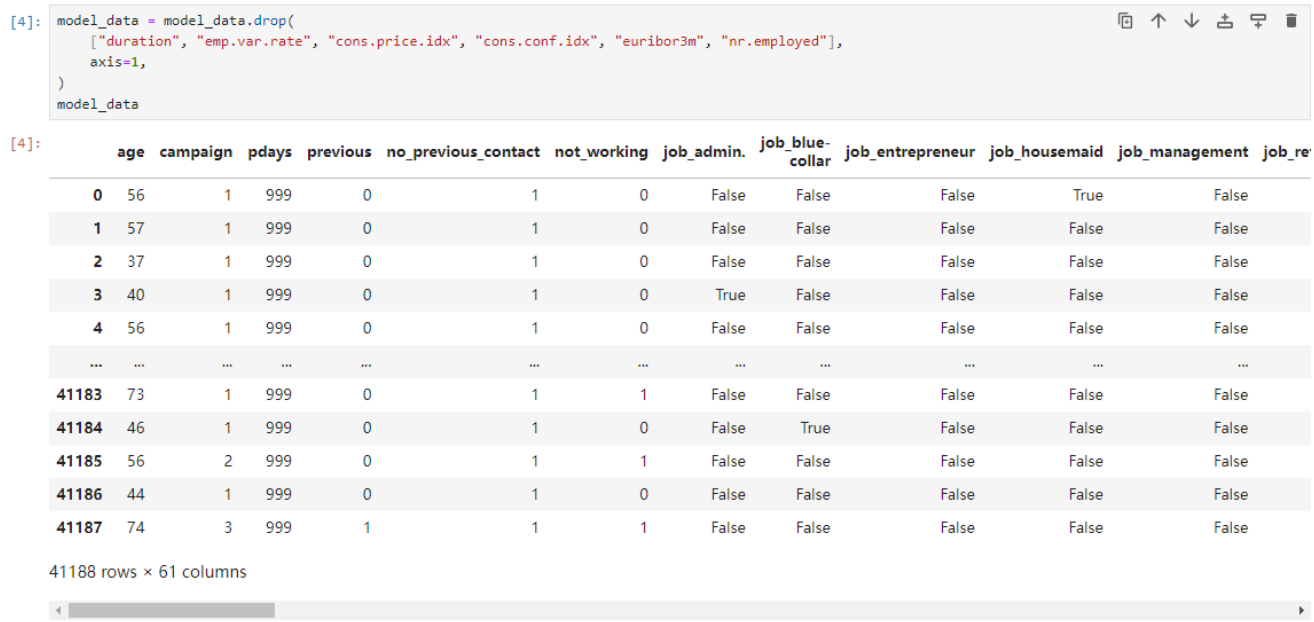
41188 rows × 67 columns

5. Next, remove certain columns from the dataset using the following code:

```
model_data = model_data.drop(
    ["duration", "emp.var.rate", "cons.price.idx", "cons.conf.idx", "euribor3m",
"nr.employed"],
    axis=1,
)
model_data
```

The `model_data.drop()` method is used to remove the specified columns from the `model_data` DataFrame. Here, the columns "duration", "emp.var.rate", "cons.price.idx", "cons.conf.idx", "euribor3m", and "nr.employed" are removed. The `axis=1` parameter specifies that we are deleting columns, not rows.

Result:



[4]:
```
model_data = model_data.drop(
    ["duration", "emp.var.rate", "cons.price.idx", "cons.conf.idx", "euribor3m", "nr.employed"],
    axis=1,
)
model_data
```

[4]:

	age	campaign	pdays	previous	no_previous_contact	not_working	job_admin.	job_blue-collar	job_entrepreneur	job_housemaid	job_management	job_re
0	56	1	999	0	1	0	False	False	False	True	False	
1	57	1	999	0	1	0	False	False	False	False	False	
2	37	1	999	0	1	0	False	False	False	False	False	
3	40	1	999	0	1	0	True	False	False	False	False	
4	56	1	999	0	1	0	False	False	False	False	False	
...	...	...	...	...	...	...	...	...	...	...	...	
41183	73	1	999	0	1	1	False	False	False	False	False	
41184	46	1	999	0	1	0	False	True	False	False	False	
41185	56	2	999	0	1	1	False	False	False	False	False	
41186	44	1	999	0	1	0	False	False	False	False	False	
41187	74	3	999	1	1	1	False	False	False	False	False	

41188 rows × 61 columns

[4] Split the data into training, validation and test

1. Next, we split the data into a training dataset (70%), a validation dataset (20%), and a test dataset (10%) and convert the datasets into the appropriate format. We will use the training and validation datasets during training, and the test dataset will be used for model evaluation after deployment.

   Amazon SageMaker's XGBoost algorithm requires data in either libSVM or CSV format. In this lab, we will use the CSV format.

   Use the following code to split the data:

```
# Split the dataset
train_data, validation_data, test_data = np.split(
    model_data.sample(frac=1, random_state=1729),
    [int(0.7 * len(model_data)), int(0.9 * len(model_data))],
)

# Save the training set
pd.concat([train_data["y_yes"], train_data.drop(["y_no", "y_yes"], axis=1)],
axis=1).to_csv(
```

```
    "train.csv", index=False, header=False
)

# Save the validation set
pd.concat(
    [validation_data["y_yes"], validation_data.drop(["y_no", "y_yes"], axis=1)],
axis=1
).to_csv("validation.csv", index=False, header=False)

# Save the test set
pd.concat([test_data["y_yes"], test_data.drop(["y_no", "y_yes"], axis=1)],
axis=1).to_csv(
    "test.csv", index=False, header=False
)
```

Explanation:

1. Split the dataset:
    - `model_data.sample(frac=1, random_state=1729)` randomly shuffles the entire dataset.
    - The `np.split()` function splits the shuffled dataset into three parts: the training set (70% of the data), the validation set (20% of the data), and the test set (10% of the data).
2. Save the training set:
    - Select the "y_yes" column (likely the positive class label), then drop the "y_no" and "y_yes" columns. Concatenate the remaining feature columns with "y_yes" and save the result as "train.csv", without including the index and column names.
3. Save the validation set:
    - Perform the same operation as with the training set and save the validation set as "validation.csv".
4. Save the test set:
    - Repeat the same process for the test set and save it as "test.csv".

2. Use the following code to upload the training and validation set files to the previously created Amazon S3 bucket:

```
# Upload the training set
boto3.Session().resource("s3").Bucket(bucket).Object(
    os.path.join(prefix, "train/train.csv")
).upload_file("train.csv")

# Upload the validation set
boto3.Session().resource("s3").Bucket(bucket).Object(
    os.path.join(prefix, "validation/validation.csv")
).upload_file("validation.csv")
```

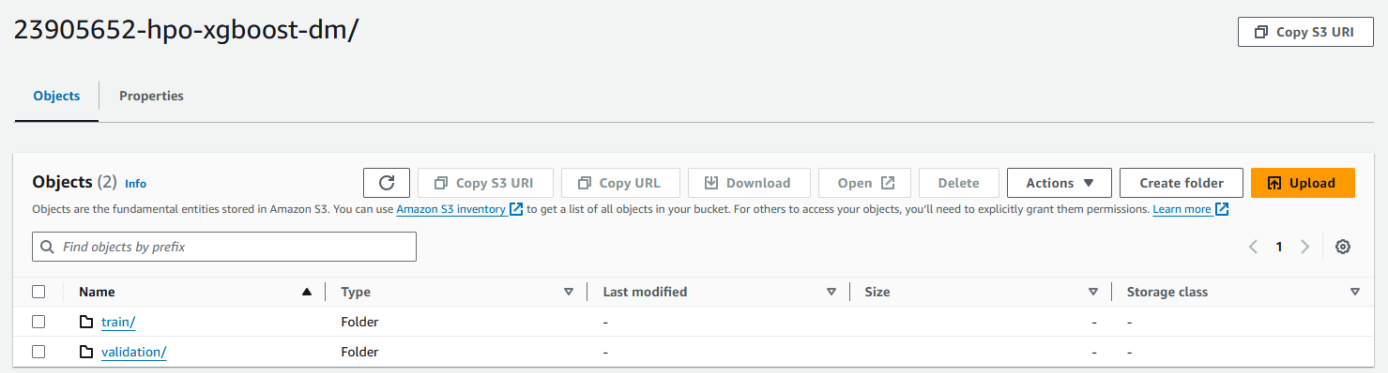
Explanation:

1. Upload the training set:
    - A new boto3 session is created, and the S3 resource is retrieved.

- The target bucket (defined by the `bucket` variable) is specified.
- A new object is created in the bucket with the path `prefix/train/train.csv`.
- The `upload_file()` method uploads the local "train.csv" file to this S3 object.

2. Upload the validation set:
   - Similarly, the local "validation.csv" file is uploaded, with the target path in S3 being `prefix/validation/validation.csv`.

We can view the created folders and corresponding files in the "23905652-lab8" bucket under the "sagemaker/23905652-hpo-xgboost-dm" path in the S3 Console:



[5] Setup hyperparameter tuning

Next, we set up the hyperparameter tuning job for the XGBoost model using the following code. We define the ranges for hyperparameters, set resource limits, specify the optimization strategy, and configure the objective metric for the tuning process. Detailed explanations are provided in the comments:

```
from time import gmtime, strftime, sleep

# Define a unique tuning job name. Names have to be unique.
tuning_job_name = "23905652-xgboost-tuningjob-01"
print(tuning_job_name)

# Configuration for the tuning job
tuning_job_config = {
    "ParameterRanges": {
        # No categorical parameters are defined for this tuning job
        "CategoricalParameterRanges": [],

        # Continuous parameters to be tuned
        "ContinuousParameterRanges": [
            {
                "MaxValue": "1",
                "MinValue": "0",
                "Name": "eta",  # Learning rate
            },
            {
                "MaxValue": "10",
                "MinValue": "1",
```

```
                "Name": "min_child_weight",  # Minimum sum of instance weight needed in a
child
            },
            {
                "MaxValue": "2",
                "MinValue": "0",
                "Name": "alpha",  # L1 regularization term
            },
        ],

        # Integer parameters to be tuned
        "IntegerParameterRanges": [
            {
                "MaxValue": "10",
                "MinValue": "1",
                "Name": "max_depth",  # Maximum depth of a tree
            }
        ],
    },

    # Limits on the maximum number of training jobs and the number of jobs running in
parallel, both set to 2
    "ResourceLimits": {
        "MaxNumberOfTrainingJobs": 2,  # Total number of training jobs to run
        "MaxParallelTrainingJobs": 2   # Maximum number of parallel training jobs
    },

    # The strategy used for hyperparameter optimization
    "Strategy": "Bayesian",

    # The objective metric to be optimized during the tuning job
    "HyperParameterTuningJobObjective": {
        "MetricName": "validation:auc",  # Area Under the Curve on validation set
        "Type": "Maximize"  # We want to maximize this metric
    },
}
```

Next, we write the script to configure the SageMaker training job for the XGBoost model. We define the location of the data, specify the compute resources, set static hyperparameters for the XGBoost algorithm, and outline the job constraints. Detailed code explanations will be included in the comments:

```
from sagemaker.image_uris import retrieve

# Get the URI for the XGBoost training image
# The image contains the XGBoost algorithm implementation that SageMaker will use for
training
training_image = retrieve(framework="xgboost", region=region, version="latest")

# Define S3 URIs for training and validation data
# These create the full S3 path strings where the training and validation datasets are
stored
```

```
s3_input_train = "s3://{}/{}/train".format(bucket, prefix)
s3_input_validation = "s3://{}/{}/validation/".format(bucket, prefix)

# Define the training job configuration
training_job_definition = {
    # Specify the algorithm to use
    # This tells SageMaker which algorithm to use (XGBoost) and how to read the input data
    "AlgorithmSpecification": {"TrainingImage": training_image, "TrainingInputMode":
"File"},

    # Configure input data channels
    "InputDataConfig": [
        {
            "ChannelName": "train",  # Name of the training data channel
            "CompressionType": "None",  # Data is not compressed
            "ContentType": "csv",  # Data is in CSV format
            "DataSource": {
                "S3DataSource": {
                    "S3DataDistributionType": "FullyReplicated",  # Each machine gets a
full copy of the data
                    "S3DataType": "S3Prefix",  # The S3 location specified is a prefix,
not a specific file
                    "S3Uri": s3_input_train,  # Location of the training data in S3
                }
            },
        },
        {
            # Similar configuration for validation data
            "ChannelName": "validation",
            "CompressionType": "None",
            "ContentType": "csv",
            "DataSource": {
                "S3DataSource": {
                    "S3DataDistributionType": "FullyReplicated",
                    "S3DataType": "S3Prefix",
                    "S3Uri": s3_input_validation,
                }
            },
        },
    ],

    # Specify where to save the output
    "OutputDataConfig": {"S3OutputPath": "s3://{}/{}/output".format(bucket, prefix)},

    # Configure the compute resources for training
    # This specifies the type and number of EC2 instances to use for training
    "ResourceConfig": {
        "InstanceCount": 1,  # Number of EC2 instances to use
        "InstanceType": "ml.m5.xlarge",  # Type of EC2 instance
        "VolumeSizeInGB": 10  # Size of the EBS volume in GB
    },
```

```
    # Specify the IAM role for SageMaker to assume
    # This role gives SageMaker the necessary permissions to access resources (like S3)
    "RoleArn": sagemaker_role,

    # Set static hyperparameters for XGBoost
    # These are algorithm-specific parameters that remain constant during training
    "StaticHyperParameters": {
        "eval_metric": "auc",  # Evaluation metric to use
        "num_round": "1",  # Number of rounds for boosting
        "objective": "binary:logistic",  # Specifies that this is a binary classification
problem
        "rate_drop": "0.3",  # Dropout rate for dropout boosting
        "tweedie_variance_power": "1.4",  # Parameter for Tweedie regression
    },

    # Set a maximum runtime for the job
    "StoppingCondition": {"MaxRuntimeInSeconds": 43200},  # 12 hours maximum runtime
}
```

After the configuration, we can launch the tuning job using the following code:

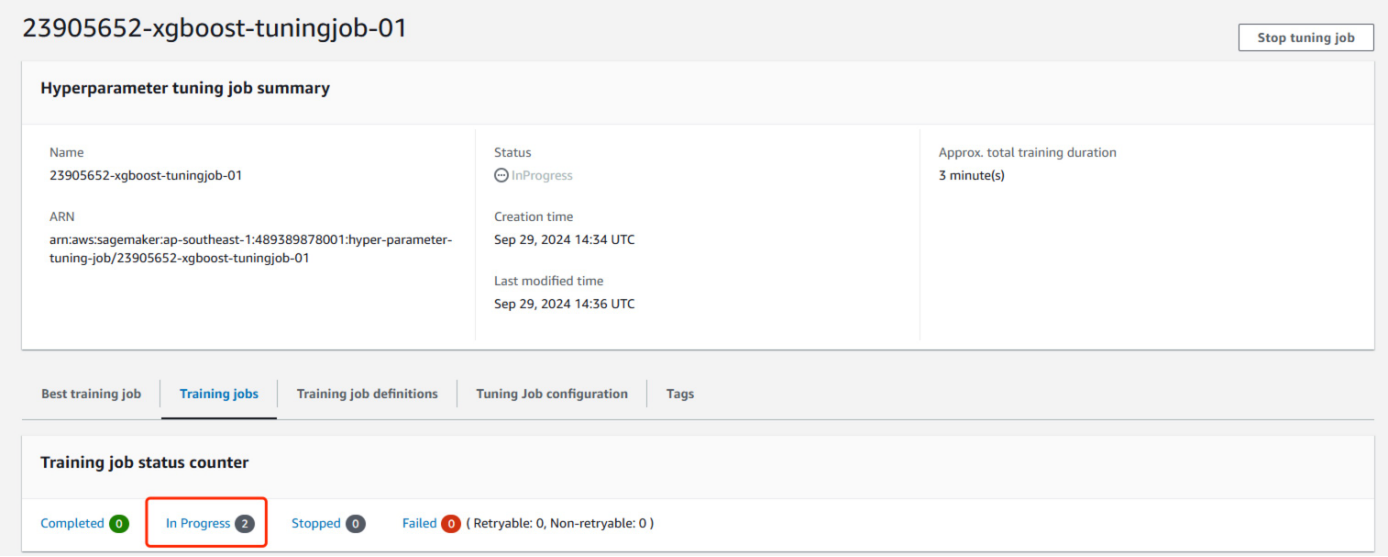
```
#Launch Hyperparameter Tuning Job
smclient.create_hyper_parameter_tuning_job(
    # Specify the name of the hyperparameter tuning job
    HyperParameterTuningJobName=tuning_job_name,

    # Provide the configuration for the hyperparameter tuning job
    HyperParameterTuningJobConfig=tuning_job_config,

    # Define the training job configuration
    TrainingJobDefinition=training_job_definition,
)
```

Result:

```
[3]: #Launch Hyperparameter Tuning Job
     smclient.create_hyper_parameter_tuning_job(
         HyperParameterTuningJobName=tuning_job_name,
         HyperParameterTuningJobConfig=tuning_job_config,
         TrainingJobDefinition=training_job_definition,
     )
```

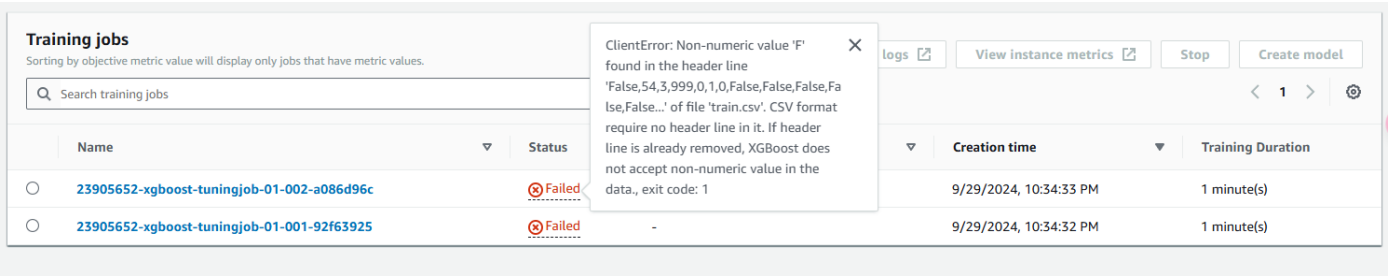
```
[3]: {'HyperParameterTuningJobArn': 'arn:aws:sagemaker:ap-southeast-1:489389878001:hyper-parameter-tuning-job/23905652-xgboost-tuningjob-01',
      'ResponseMetadata': {'RequestId': '6c1c42d9-21f4-4a0b-a3ef-9c33cfc749dd',
       'HTTPStatusCode': 200,
       'HTTPHeaders': {'x-amzn-requestid': '6c1c42d9-21f4-4a0b-a3ef-9c33cfc749dd',
        'content-type': 'application/x-amz-json-1.1',
        'content-length': '135',
        'date': 'Sun, 29 Sep 2024 14:34:27 GMT'},
       'RetryAttempts': 0}}
```

The Hyperparameter Tuning Job has been successfully launched. Next, we can monitor the progress of the hyperparameter tuning job through the AWS Console (SageMaker -> Training -> Hyperparameter tuning jobs). The job is currently in the "In Progress" state. We can check the the progress by looking at the position indicated in the screenshot:



A few minutes later, the job failed. This error indicated that there was a header line in the 'train.csv' file containing non-numeric data, specifically the character 'F'.

We need to convert non-numeric values to numeric values (e.g., True/False to 1/0). Then, rename the tuning job and run it again .



We go back to the data processing section and add the following code before splitting the dataset to convert `True` values to `1` and `False` values to `0`:

```
boolean_columns = model_data.select_dtypes(include=['bool']).columns
model_data[boolean_columns] = model_data[boolean_columns].astype(int)
```

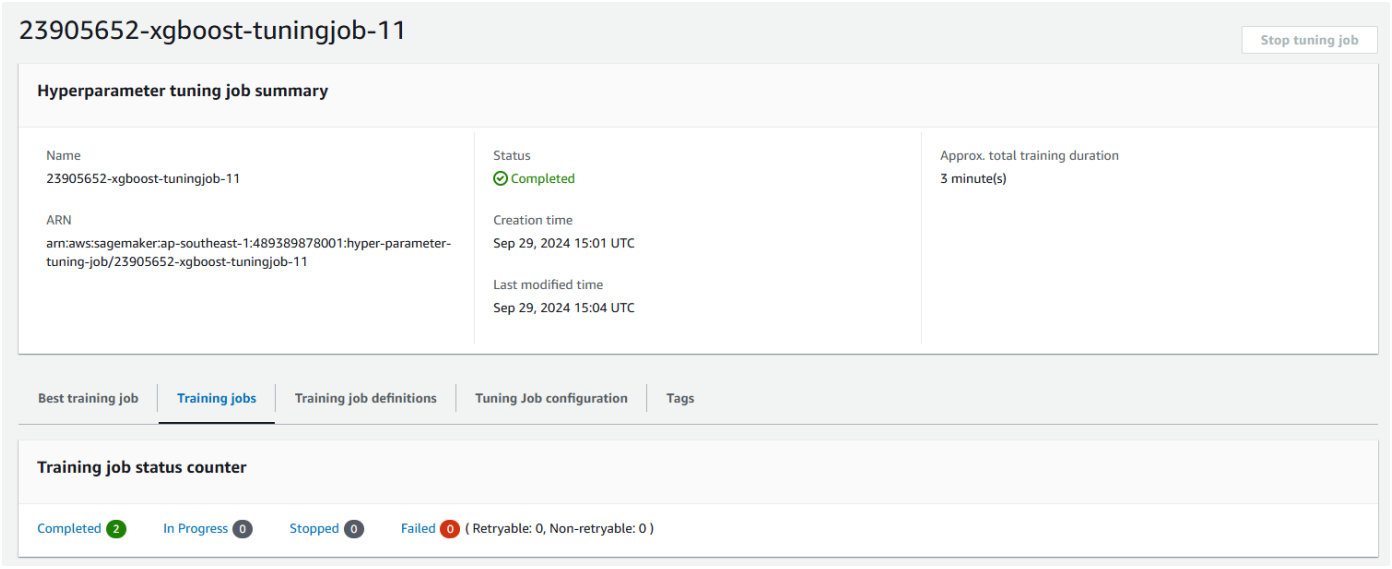
Explanation:

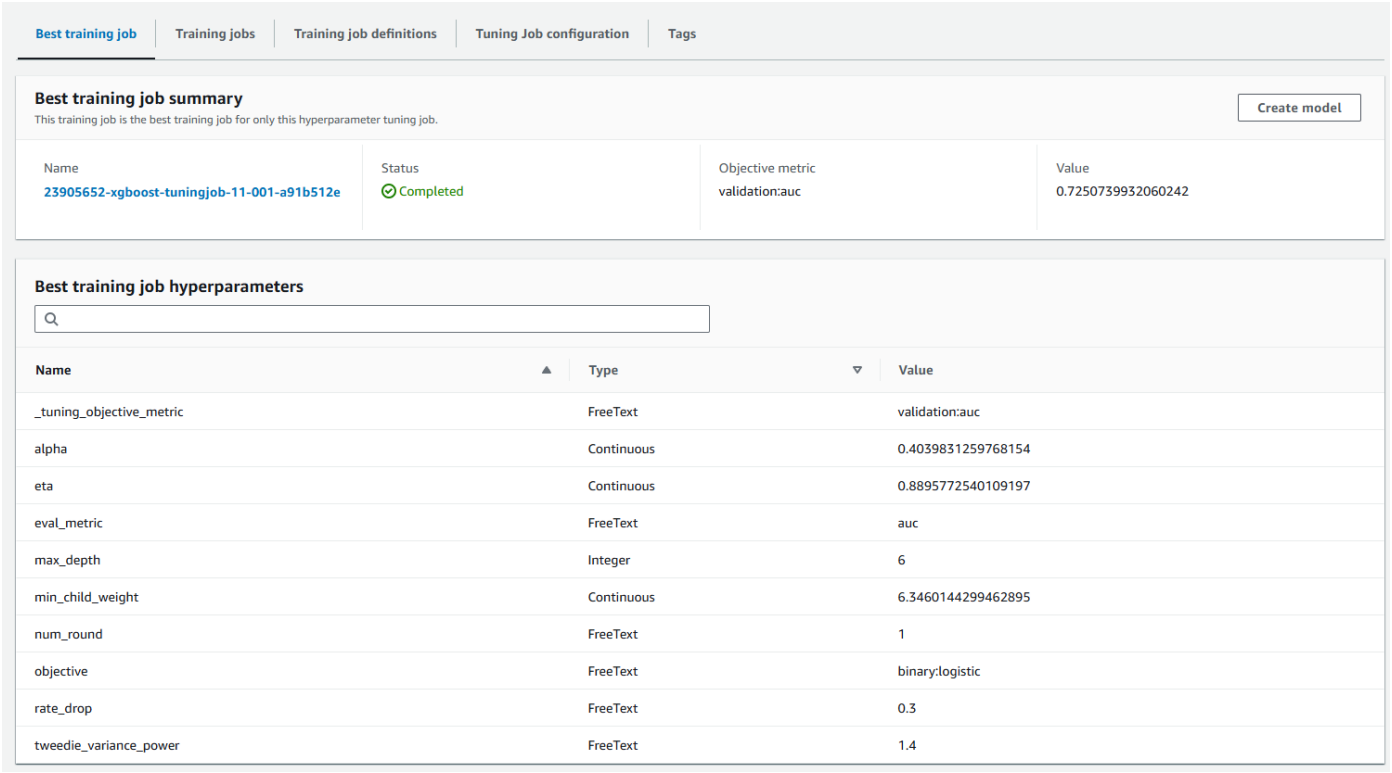
- `model_data.select_dtypes(include=['bool'])`: Selects all columns in the `model_data` DataFrame that have a boolean data type.
- `.columns`: Retrieves the names of these boolean columns and stores them in the `boolean_columns` variable.
- `model_data[boolean_columns]`: Selects all boolean columns.
- `.astype(int)`: Converts the data type of these columns from boolean to integer.
- The converted results are assigned back to the corresponding columns in the original `model_data` DataFrame.

Afterward, repeat all the steps starting from splitting the dataset, and change the `tuning_job_name` to a new name. In this case, we rename it to "23905652-xgboost-tuningjob-11":

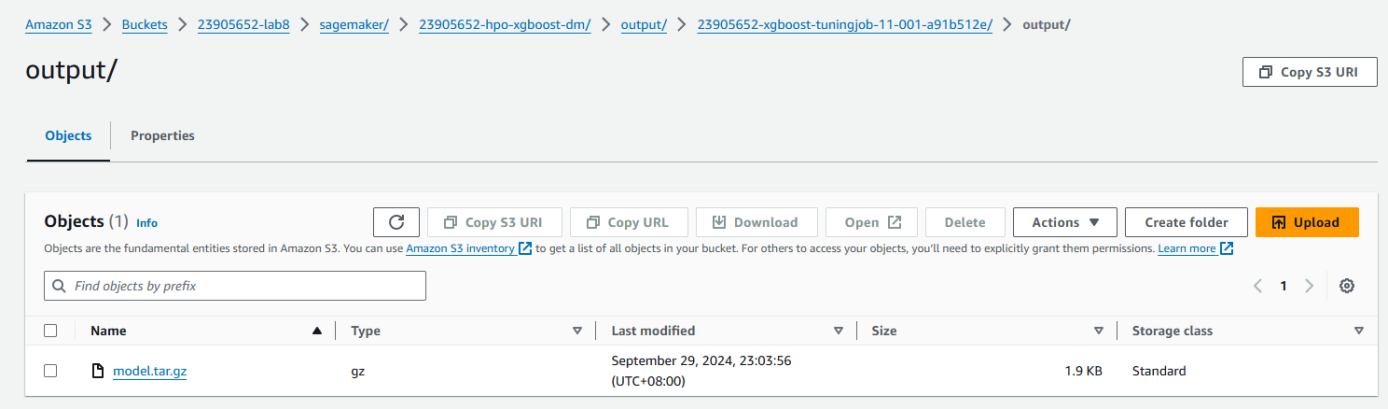
```
from time import gmtime, strftime, sleep

# Names have to be unique. You will get an error if you reuse the same name
tuning_job_name = "23905652-xgboost-tuningjob-11"

print(tuning_job_name)
```

Once again, monitor the progress of the hyperparameter tuning job in the AWS Console (SageMaker -> Training -> Hyperparameter tuning jobs). This time, it completed successfully.





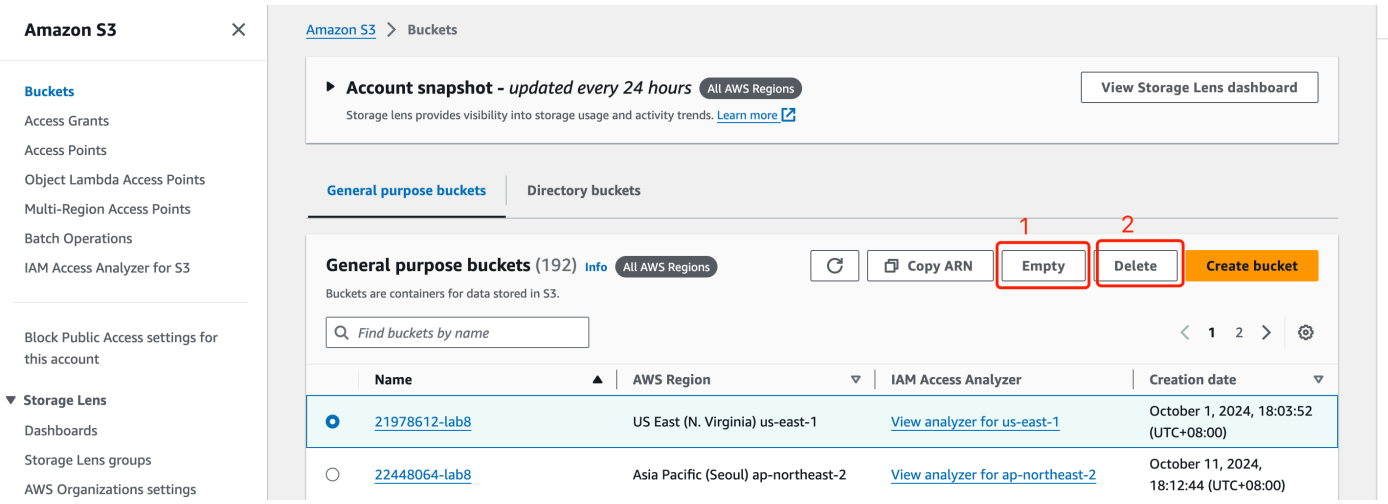
We can view the output results in the "/output" folder in the S3 console.



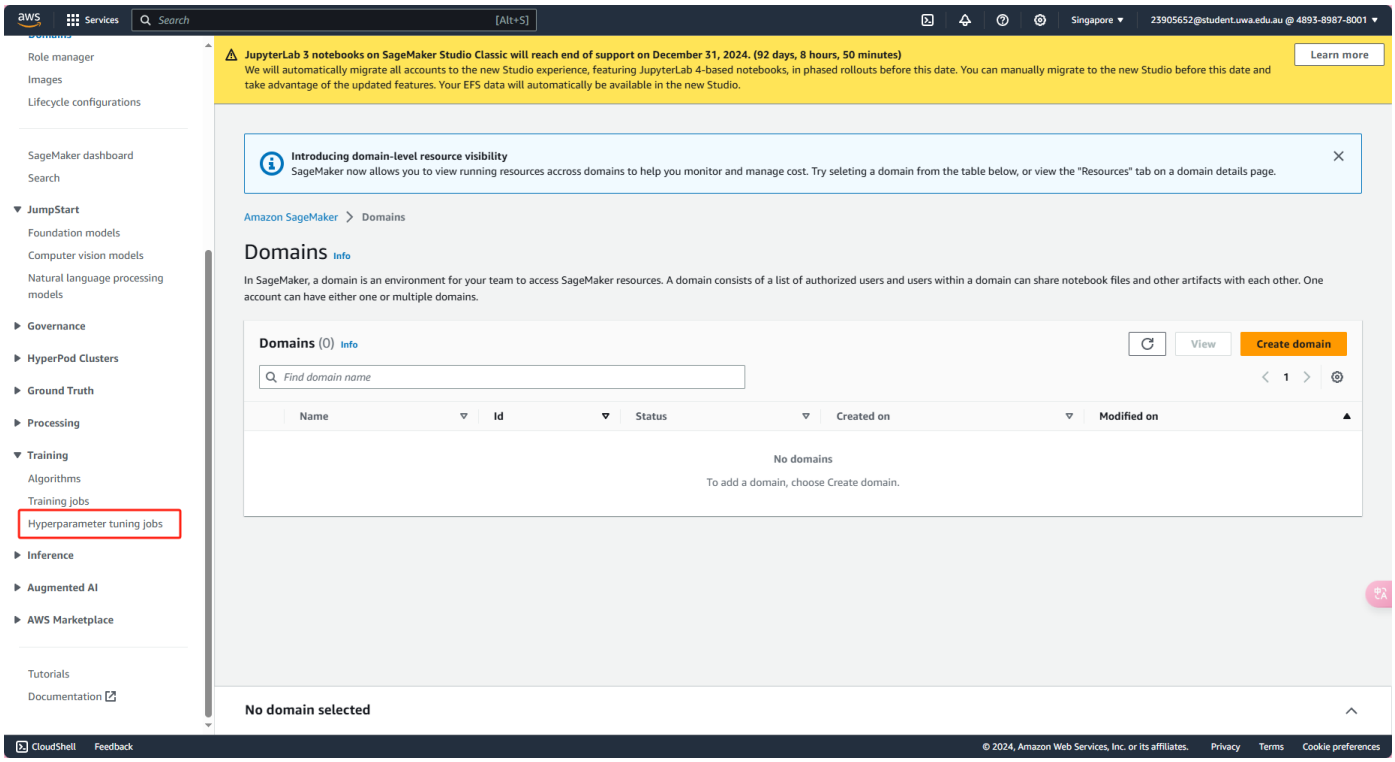
[6] Manually shut down created AWS resources

After finishing the lab, we have to manually shut down created AWS resources.

First, delete the created S3 bucket in the S3 console. Select the resource "23905652-lab8" to delete, then click "Empty" to clear the bucket, and follow the prompts to complete the operation. After emptying, click "Delete" to remove the bucket, and follow the prompts to complete the deletion (the screenshot is for demonstration purposes only).



AWS Hyperparameter tuning jobs do not need to be manually shut down. They automatically manage their lifecycle, and the tuning job will stop automatically. Once the tuning job is complete, AWS automatically releases the related computing resources.


aws
Services
Search
[Alt+S]
Singapore
23905652@student.uwa.edu.au @ 4893-8987-8001
Role manager
Images
Lifecycle configurations
SageMaker dashboard
Search
JumpStart
Foundation models
Computer vision models
Natural language processing models
Governance
HyperPod Clusters
Ground Truth
Processing
Training
Algorithms
Training jobs
Hyperparameter tuning jobs
Inference
Augmented AI
AWS Marketplace
Tutorials
Documentation
JupyterLab 3 notebooks on SageMaker Studio Classic will reach end of support on December 31, 2024. (92 days, 8 hours, 50 minutes)
We will automatically migrate all accounts to the new Studio experience, featuring JupyterLab 4-based notebooks, in phased rollouts before this date. You can manually migrate to the new Studio before this date and take advantage of the updated features. Your EFS data will automatically be available in the new Studio.
Learn more
Introducing domain-level resource visibility
SageMaker now allows you to view running resources accross domains to help you monitor and manage cost. Try seleting a domain from the table below, or view the "Resources" tab on a domain details page.
Amazon SageMaker
Domains
Domains Info
In SageMaker, a domain is an environment for your team to access SageMaker resources. A domain consists of a list of authorized users and users within a domain can share notebook files and other artifacts with each other. One account can have either one or multiple domains.
Domains (0) Info
View
Create domain
Find domain name
1
Name
Id
Status
Created on
Modified on
No domains
To add a domain, choose Create domain.
No domain selected
CloudShell
Feedback
© 2024, Amazon Web Services, Inc. or its affiliates.
Privacy
Terms
Cookie preferences

Lab 9 More AI

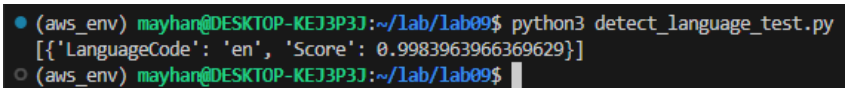
AWS Comprehend

Detect Languages from text

First, try running the sample code (save it as "detect_language_test.py"):

```
import boto3
client = boto3.client('comprehend')

# Detect dominant language
response = client.detect_dominant_language(
    Text="The French Revolution was a period of social and political upheaval in France and its colonies beginning in 1789 and ending in 1799.",
)

print(response['Languages'])
```

Result:



This means that the detected language is "en" (English) with a confidence score greater than 0.99.

[1] Modify the code above

We modify the provided example to write a script that detects different languages and provides the confidence score. The code explanation is included in the comments:

```
import boto3

def detect_language(text):
    # Create a boto3 client for AWS Comprehend service
    client = boto3.client('comprehend')

    # Use the client to detect the dominant language in the given text
    response = client.detect_dominant_language(Text=text)

    # Extract language information from the response
    language = response['Languages'][0]
    language_code = language['LanguageCode']
    confidence = language['Score'] * 100  # Convert confidence score to percentage

    # Map of language codes to full language names
    language_map = {
        'en': 'English', 'es': 'Spanish', 'fr': 'French', 'it': 'Italian'
    }
```

```
    # Get the full language name
    language_name = language_map.get(language_code, language_code)

    # Print the detected language and confidence
    print(f"{language_name} detected with {confidence:.0f}% confidence")

# List of test texts in different languages
texts = [
    "The French Revolution was a period of social and political upheaval in France and its colonies beginning in 1789 and ending in 1799.",
    "El Quijote es la obra más conocida de Miguel de Cervantes Saavedra. Publicada su primera parte con el título de El ingenioso hidalgo don Quijote de la Mancha a comienzos de 1605, es una de las obras más destacadas de la literatura española y la literatura universal, y una de las más traducidas. En 1615 aparecería la segunda parte del Quijote de Cervantes con el título de El ingenioso caballero don Quijote de la Mancha.",
    "Moi je n'étais rien Et voilà qu'aujourd'hui Je suis le gardien Du sommeil de ses nuits Je l'aime à mourir Vous pouvez détruire Tout ce qu'il vous plaira Elle n'a qu'à ouvrir L'espace de ses bras Pour tout reconstruire Pour tout reconstruire Je l'aime à mourir",
    "L'amor che move il sole e l'altre stelle."
]

# Test the function with each text in the list
for text in texts:
    detect_language(text)
```

Explanation:

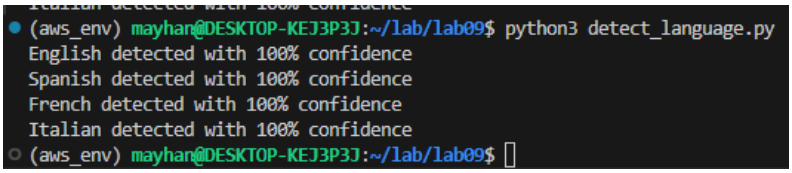
This script uses AWS Comprehend to detect the dominant language in a given text. **Steps:**

1. The `detect_language` function takes text as input and performs the following steps:
    - Creates an AWS Comprehend client.
    - Calls the `client.detect_dominant_language` method to analyze the text.
    - Extracts the detected language code and confidence score.
    - Maps the language code to a full language name.
    - Prints the detected language and confidence.
2. A list of test texts in different languages (English, Spanish, French, and Italian) is placed in the `texts` list.
3. The script iterates through each text in the list and calls the `detect_language` function to detect the dominant language.

[2] Test your code with other languages

In step 1, we have already written a Python script with different language test texts in the `texts` list to detect the language for each one. Save the script as "detect_language.py" and run the code:

Result:



Based on the order of the test texts in the list, we can verify that the output results are correct.

Analyze sentiment

Sentiment analysis uses NLP to determine whether the data is positive, negative, neutral, or mixed. It is often applied to text data to help businesses monitor customer feedback about brands and products and understand customer needs.

We write a Python script using `boto3` and AWS Comprehend for sentiment analysis, applying the four texts from earlier as test cases. The explanation of the script is provided in the comments. The test texts are omitted here:

```
import boto3

def analyze_sentiment(text):
    client = boto3.client('comprehend')

    # First, detect the dominant language of the text
    response = client.detect_dominant_language(Text=text)
    language_code = response['Languages'][0]['LanguageCode']

    # Now, analyze sentiment with the detected language
    response = client.detect_sentiment(Text=text, LanguageCode=language_code)

    # Extract sentiment and confidence score
    sentiment = response['Sentiment']
    confidence = response['SentimentScore'][sentiment.capitalize()] * 100  # Convert to
percentage

    # Print the sentiment analysis result
    print(f"The sentiment of the text is {sentiment} with {confidence:.0f}% confidence")

texts = ["...", "...", "...", "..."] # Same as the test texts list in detect_language,
omitted here

for text in texts:
    print("\nText:", text)
    analyze_sentiment(text)
```

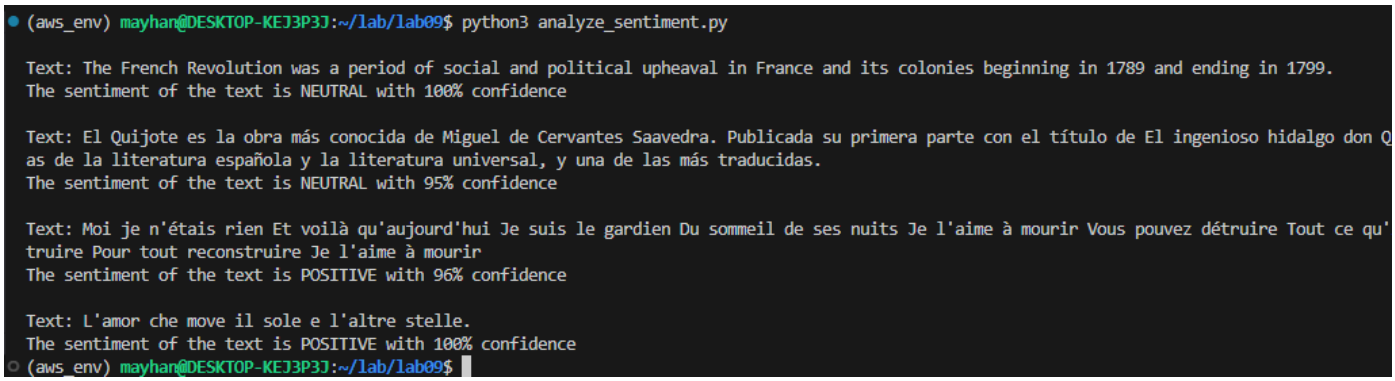
This script uses AWS Comprehend to perform sentiment analysis on text in different languages. The main steps:

1. The `analyze_sentiment` function is defined, which takes a text as input and performs the following:
    - Creates an AWS Comprehend client.

- Detects the dominant language of the input text (necessary for further analysis).
- Uses the `client.detect_sentiment` method to analyze the sentiment in the detected language.
- Extracts the overall sentiment and confidence score. The sentiment can be one of the following: POSITIVE, NEGATIVE, NEUTRAL, or MIXED.
- Prints the sentiment analysis result.

2. A list of test texts in different languages (English, Spanish, French, and Italian) is placed in the `texts` list.
3. The script iterates over each text in the list, printing the text and calling the `analyze_sentiment` function to analyze its sentiment.

Result:



Based on the results, we can see that the sentiment for each text was detected with high confidence.

Detect entities

Next, we write a script for entity detection and test it with the previous four texts. The test texts are omitted here:

```
import boto3

def detect_entities(text):
    client = boto3.client('comprehend')

    response = client.detect_dominant_language(Text=text)
    language_code = response['Languages'][0]['LanguageCode']

    # Use the detected language to perform entity recognition
    response = client.detect_entities(Text=text, LanguageCode=language_code)
    entities = response['Entities']
    print("Detected entities:")
    if not entities:
        print("No entities detected")
        return

    # Print each detected entity, its type, and confidence score
    for entity in entities:
        confidence = entity['Score'] * 100
```

```
        print(f"Entity: {entity['Text']}, Type: {entity['Type']}, with {confidence:.0f}%
confidence")

texts = ["...", "...", "...", "..."] # Same as the test texts list in detect_language,
omitted here

for text in texts:
    print("\nText:", text)
    detect_entities(text)
    print("\n")
```

This script uses AWS Comprehend to perform entity detection on text in different languages. The main steps:

1. The `detect_entities` function is defined, which takes a text as input and performs the following:
    - Creates an AWS Comprehend client.
    - Detects the dominant language of the input text (necessary for further analysis).
    - Uses the `client.detect_entities` method to recognize entities in the detected language. Entity types can include PERSON, LOCATION, ORGANIZATION, DATE, QUANTITY, and others, depending on the content of the text.
    - Extracts the list of detected entities.
    - Prints each detected entity, its type, and confidence score. If no entities are detected, it prints "No entities detected".
2. A list of test texts in different languages (English, Spanish, French, and Italian) is placed in the `texts` list.
3. The script iterates over each text in the list, printing the text and calling the `detect_entities` function to perform entity detection.

Result:

```
(aws_env) mayhan@DESKTOP-KEJ3P3J:~/lab/lab09$ python3 detect_entity.py

Text: The French Revolution was a period of social and political upheaval in France and its colonies beginning in 1789 and ending in 1799.
Detected entities:
Entity: French Revolution, Type: EVENT
Entity: France, Type: LOCATION
Entity: 1789, Type: DATE
Entity: 1799, Type: DATE



Text: El Quijote es la obra más conocida de Miguel de Cervantes Saavedra. Publicada su primera parte con el título de El ingenioso hidalgo do
as de la literatura española y la literatura universal, y una de las más traducidas. En 1615 aparecería la segunda parte del Quijote de Cerva
Detected entities:
Entity: El Quijote, Type: TITLE
Entity: Miguel de Cervantes Saavedra, Type: PERSON
Entity: primera parte, Type: QUANTITY
Entity: El ingenioso hidalgo don Quijote de la Mancha, Type: TITLE
Entity: 1605, Type: DATE
Entity: una de, Type: QUANTITY
Entity: española, Type: OTHER
Entity: una de las más, Type: QUANTITY
Entity: 1615, Type: DATE
Entity: segunda parte, Type: QUANTITY
Entity: Quijote de Cervantes, Type: TITLE
Entity: El ingenioso caballero don Quijote de la Mancha, Type: TITLE



Text: Moi je n'étais rien Et voilà qu'aujourd'hui Je suis le gardien Du sommeil de ses nuits Je l'aime à mourir Vous pouvez détruire Tout ce
truire Pour tout reconstruire Je l'aime à mourir
Detected entities:
Entity: aujourd'hui, Type: DATE
Entity: Tout ce qu', Type: QUANTITY



Text: L'amor che move il sole e l'altre stelle.
Detected entities:
No entities detected


(aws_env) mayhan@DESKTOP-KEJ3P3J:~/lab/lab09$
```

The output shows that the script successfully processed English, Spanish, French, and Italian texts, automatically detecting the language of each text and applying the appropriate entity recognition model.

Answer this question: describe what entities are in your own words.

A: Entities are words or phrases in the text with specific meanings, such as names of people, locations, organizations, dates, and other unique objects or concepts. They represent specific pieces of information that have meaning in the real world and are important for understanding the content. Entities form the basic building blocks of a sentence's structure.

Detect keyphrases

Next, we write a script for keyphrase detection and test it with the previous four texts. The test texts are omitted here:

```python
import boto3

def detect_key_phrases(text):
    client = boto3.client('comprehend')

    response = client.detect_dominant_language(Text=text)
    language_code = response['Languages'][0]['LanguageCode']

    # Use the detected language to perform key phrase detection
```

```
    response = client.detect_key_phrases(Text=text, LanguageCode=language_code)
    key_phrases = response['KeyPhrases']

    print("Detected key phrases:")
    if not key_phrases:
        print("No key phrases detected")
        return

    for phrase in key_phrases:
        confidence = phrase['Score'] * 100
        print(f"Text: {phrase['Text']}, with {confidence:.0f}% confidence")

texts = ["...", "...", "...", "..."] # Same as the test texts list in detect_language,
omitted here

for text in texts:
    print("\nText:", text)
    detect_key_phrases(text)
    print("\n")
```

This script uses AWS Comprehend to perform key phrase detection on text in different languages. **The main steps are:**

1. The `detect_key_phrases` function is defined, which takes text as input and performs the following:
    - Creates an AWS Comprehend client
    - Detects the dominant language of the input text (required before performing further detections)
    - Uses the `client.detect_key_phrases` method to detect key phrases in the detected language
    - Extracts the list of detected key phrases
    - Prints each detected key phrase and its confidence score. If no key phrases are detected, it prints "No key phrases detected."
2. A list of test texts in different languages (including English, Spanish, French, and Italian) is placed in the `texts` list.
3. The script iterates over each text in the list, prints the text, and calls the `detect_key_phrases` function to perform entity detection.

Result:

text1:

```
(aws_env) mayhan@DESKTOP-KEJ3P3J:~/lab/lab09$ python3 detect_key_phrases.py

Text: The French Revolution was a period of social and political upheaval in France and its colonies beginning in 1789 and ending in 1799.
Detected key phrases:
Text: The French Revolution, with 100% confidence
Text: a period, with 100% confidence
Text: social and political upheaval, with 100% confidence
Text: France, with 100% confidence
Text: its colonies, with 100% confidence
Text: 1789, with 100% confidence
Text: 1799, with 100% confidence
```

text2:

```
Text: El Quijote es la obra más conocida de Miguel de Cervantes Saavedra. Publicada su primera parte con el título de El ingenioso hidalgo don Quijote
as de la literatura española y la literatura universal, y una de las más traducidas. En 1615 aparecería la segunda parte del Quijote de Cervantes con
Detected key phrases:
Text: El Quijote, with 100% confidence
Text: la obra, with 100% confidence
Text: más conocida, with 100% confidence
Text: Miguel de Cervantes Saavedra, with 100% confidence
Text: su primera parte, with 100% confidence
Text: el título, with 100% confidence
Text: El ingenioso hidalgo don Quijote de la Mancha, with 96% confidence
Text: comienzos, with 100% confidence
Text: 1605, with 100% confidence
Text: las obras, with 100% confidence
Text: más destacadas, with 100% confidence
Text: la literatura española, with 100% confidence
Text: la literatura universal, with 100% confidence
Text: las más traducidas, with 100% confidence
Text: la segunda parte, with 100% confidence
Text: Quijote de Cervantes, with 100% confidence
Text: el título, with 100% confidence
Text: ingenioso caballero don Quijote de la Mancha, with 94% confidence
```

text3:

```
Text: Moi je n'étais rien Et voilà qu'aujourd'hui Je suis le gardien Du sommeil de ses nuits Je l'aime à mourir Vous pouvez détruire Tout ce
truire Pour tout reconstruire Je l'aime à mourir
Detected key phrases:
Text: Moi, with 100% confidence
Text: je, with 96% confidence
Text: n'étais rien, with 96% confidence
Text: aujourd'hui, with 95% confidence
Text: Je suis le gardien Du sommeil de ses nuits, with 94% confidence
Text: Je, with 100% confidence
Text: l', with 100% confidence
Text: Vous, with 100% confidence
Text: Tout ce, with 98% confidence
Text: qu', with 96% confidence
Text: il, with 100% confidence
Text: vous, with 100% confidence
Text: Elle, with 100% confidence
Text: L'espace de ses bras, with 99% confidence
Text: tout, with 89% confidence
Text: tout, with 96% confidence
Text: Je, with 100% confidence
Text: l', with 100% confidence
```

text4:

```
Text: L'amor che move il sole e l'altre stelle.
Detected key phrases:
Text: L'amor, with 100% confidence
Text: che, with 100% confidence
Text: il sole, with 100% confidence
Text: l'altre stelle, with 100% confidence
```

Answer this question: describe what keyphrases are in your own words.

A: Key phrases are important phrases or segments in a text that capture its main topics or ideas. They are the most important or representative word groups. They serve as condensed pieces of information, summarizing or highlighting the core thoughts expressed in a piece of writing. They reveal the subject of the sentence and help grasp the main idea and focus quickly.

Detect syntaxes

Now, let's write a script for syntax detection and use the previous 4 texts as test objects. The test texts are omitted here:

```
import boto3
```

```
def detect_syntax(text):
    client = boto3.client('comprehend')

    response = client.detect_dominant_language(Text=text)
    language_code = response['Languages'][0]['LanguageCode']

    # Use the detected language to perform syntax analysis
    response = client.detect_syntax(Text=text, LanguageCode=language_code)
    syntax_tokens = response['SyntaxTokens']

    print("Detected syntax:")
    # Print each word, its part of speech, and the confidence score
    for token in syntax_tokens:
        confidence = token['PartOfSpeech']['Score'] * 100
        print(f"Word: {token['Text']}, Part of Speech: {token['PartOfSpeech']['Tag']},
with {confidence:.0f}% confidence")

texts = ["...", "...", "...", "..."] # Same as the test texts list in detect_language,
omitted here

for text in texts:
    print("\nText:", text)
    detect_syntax(text)
    print("\n")
```

This script uses AWS Comprehend to perform syntax detection on text in different languages. **The main steps are:**

1. The `detect_syntax` function is defined, which takes text as input and performs the following:
    - Creates an AWS Comprehend client
    - Detects the dominant language of the input text (required before performing further detection)
    - Uses the `client.detect_syntax` method to perform syntax analysis. Syntax analysis breaks down the text into individual words (tokens) and identifies their parts of speech (e.g., noun, verb, adjective).
    - Extracts the list of syntax tokens (words and their parts of speech)
    - Prints each word, its part of speech, and the confidence score
2. A list of test texts in different languages (including English, Spanish, French, and Italian) is placed in the `texts` list.
3. The script iterates over each text in the list, prints the text, and calls the `detect_syntax` function to perform entity detection.

Result:

text1:

```
(aws_env) mayhan@DESKTOP-KEJ3P3J:~/lab/lab09$ python3 detect_syntax.py

Text: The French Revolution was a period of social and political upheaval in France and its colonies beginning in 1789 and ending in 1799.
Detected syntax:
Word: The, Part of Speech: DET, with 100% confidence
Word: French, Part of Speech: PROPN, with 100% confidence
Word: Revolution, Part of Speech: PROPN, with 100% confidence
Word: was, Part of Speech: VERB, with 100% confidence
Word: a, Part of Speech: DET, with 100% confidence
Word: period, Part of Speech: NOUN, with 100% confidence
Word: of, Part of Speech: ADP, with 100% confidence
Word: social, Part of Speech: ADJ, with 100% confidence
Word: and, Part of Speech: CCONJ, with 100% confidence
Word: political, Part of Speech: ADJ, with 100% confidence
Word: upheaval, Part of Speech: NOUN, with 100% confidence
Word: in, Part of Speech: ADP, with 100% confidence
Word: France, Part of Speech: PROPN, with 100% confidence
Word: and, Part of Speech: CCONJ, with 100% confidence
Word: its, Part of Speech: PRON, with 100% confidence
Word: colonies, Part of Speech: NOUN, with 100% confidence
Word: beginning, Part of Speech: VERB, with 100% confidence
Word: in, Part of Speech: ADP, with 100% confidence
Word: 1789, Part of Speech: NUM, with 100% confidence
Word: and, Part of Speech: CCONJ, with 100% confidence
Word: ending, Part of Speech: VERB, with 100% confidence
Word: in, Part of Speech: ADP, with 100% confidence
Word: 1799, Part of Speech: NUM, with 100% confidence
Word: ., Part of Speech: PUNCT, with 100% confidence
```

text2:

```
Text: El Quijote es la obra más conocida de Miguel de Cervantes Saavedra. Publicada su primera parte con el título de El
as de la literatura española y la literatura universal, y una de las más traducidas. En 1615 aparecería la segunda parte
Detected syntax:
Word: El, Part of Speech: DET, with 100% confidence
Word: Quijote, Part of Speech: PROPN, with 100% confidence
Word: es, Part of Speech: VERB, with 100% confidence
Word: la, Part of Speech: DET, with 100% confidence
Word: obra, Part of Speech: NOUN, with 100% confidence
Word: más, Part of Speech: ADV, with 100% confidence
Word: conocida, Part of Speech: ADJ, with 69% confidence
Word: de, Part of Speech: ADP, with 100% confidence
Word: Miguel, Part of Speech: PROPN, with 100% confidence
Word: de, Part of Speech: ADP, with 100% confidence
Word: Cervantes, Part of Speech: PROPN, with 100% confidence
Word: Saavedra, Part of Speech: PROPN, with 100% confidence
Word: ., Part of Speech: PUNCT, with 100% confidence
Word: Publicada, Part of Speech: VERB, with 100% confidence
Word: su, Part of Speech: DET, with 100% confidence
Word: primera, Part of Speech: ADJ, with 100% confidence
Word: parte, Part of Speech: NOUN, with 100% confidence
Word: con, Part of Speech: ADP, with 100% confidence
Word: el, Part of Speech: DET, with 100% confidence
Word: título, Part of Speech: NOUN, with 100% confidence
Word: de, Part of Speech: ADP, with 100% confidence
Word: El, Part of Speech: DET, with 100% confidence
Word: ingenioso, Part of Speech: PROPN, with 100% confidence
Word: hidalgo, Part of Speech: PROPN, with 100% confidence
Word: don, Part of Speech: PROPN, with 100% confidence
Word: Quijote, Part of Speech: PROPN, with 100% confidence
Word: de, Part of Speech: ADP, with 100% confidence
Word: la, Part of Speech: DET, with 100% confidence
Word: Mancha, Part of Speech: PROPN, with 100% confidence
Word: a, Part of Speech: ADP, with 100% confidence
Word: comienzos, Part of Speech: NOUN, with 100% confidence
Word: de, Part of Speech: ADP, with 100% confidence
Word: 1605, Part of Speech: NUM, with 100% confidence
Word: ,, Part of Speech: PUNCT, with 100% confidence
Word: es, Part of Speech: VERB, with 100% confidence
Word: una, Part of Speech: PRON, with 100% confidence
Word: de, Part of Speech: ADP, with 100% confidence
Word: las, Part of Speech: DET, with 100% confidence
Word: obras, Part of Speech: NOUN, with 100% confidence
Word: más, Part of Speech: ADV, with 100% confidence
Word: destacadas, Part of Speech: ADJ, with 100% confidence
Word: de, Part of Speech: ADP, with 100% confidence
Word: la, Part of Speech: DET, with 100% confidence
```

text3:

```
Text: Moi je n'étais rien Et voilà qu'aujourd'hui Je suis le gardien Du sommeil de ses nuits Je l'aime à mourir
truire Pour tout reconstruire Je l'aime à mourir
Detected syntax:
Word: Moi, Part of Speech: PRON, with 100% confidence
Word: je, Part of Speech: PRON, with 100% confidence
Word: n', Part of Speech: ADV, with 100% confidence
Word: étais, Part of Speech: AUX, with 100% confidence
Word: rien, Part of Speech: PRON, with 100% confidence
Word: Et, Part of Speech: CCONJ, with 100% confidence
Word: voilà, Part of Speech: VERB, with 100% confidence
Word: qu', Part of Speech: SCONJ, with 100% confidence
Word: aujourd'hui, Part of Speech: NOUN, with 100% confidence
Word: Je, Part of Speech: PRON, with 100% confidence
Word: suis, Part of Speech: AUX, with 100% confidence
Word: le, Part of Speech: DET, with 100% confidence
Word: gardien, Part of Speech: NOUN, with 100% confidence
Word: Du, Part of Speech: ADP, with 100% confidence
Word: sommeil, Part of Speech: NOUN, with 100% confidence
Word: de, Part of Speech: ADP, with 100% confidence
Word: ses, Part of Speech: DET, with 99% confidence
Word: nuits, Part of Speech: NOUN, with 100% confidence
Word: Je, Part of Speech: PRON, with 100% confidence
Word: l', Part of Speech: PRON, with 100% confidence
Word: aime, Part of Speech: VERB, with 100% confidence
Word: à, Part of Speech: ADP, with 100% confidence
Word: mourir, Part of Speech: VERB, with 100% confidence
Word: Vous, Part of Speech: PRON, with 100% confidence
Word: pouvez, Part of Speech: VERB, with 100% confidence
Word: détruire, Part of Speech: VERB, with 100% confidence
Word: Tout, Part of Speech: DET, with 92% confidence
Word: ce, Part of Speech: PRON, with 99% confidence
Word: qu', Part of Speech: PRON, with 100% confidence
Word: il, Part of Speech: PRON, with 100% confidence
Word: vous, Part of Speech: PRON, with 100% confidence
Word: plaira, Part of Speech: VERB, with 100% confidence
Word: Elle, Part of Speech: PRON, with 100% confidence
Word: n', Part of Speech: ADV, with 100% confidence
Word: a, Part of Speech: VERB, with 100% confidence
Word: qu', Part of Speech: SCONJ, with 71% confidence
Word: à, Part of Speech: ADP, with 100% confidence
Word: ouvrir, Part of Speech: VERB, with 100% confidence
Word: L', Part of Speech: DET, with 100% confidence
Word: espace, Part of Speech: NOUN, with 100% confidence
Word: de, Part of Speech: ADP, with 100% confidence
Word: ses, Part of Speech: DET, with 99% confidence
Word: bras, Part of Speech: NOUN, with 100% confidence
Word: Pour, Part of Speech: ADP, with 100% confidence
```

text4:

```
Text: L'amor che move il sole e l'altre stelle.
Detected syntax:
Word: L', Part of Speech: DET, with 100% confidence
Word: amor, Part of Speech: NOUN, with 100% confidence
Word: che, Part of Speech: PRON, with 100% confidence
Word: move, Part of Speech: VERB, with 100% confidence
Word: il, Part of Speech: DET, with 100% confidence
Word: sole, Part of Speech: NOUN, with 100% confidence
Word: e, Part of Speech: CCONJ, with 100% confidence
Word: l', Part of Speech: DET, with 100% confidence
Word: altre, Part of Speech: ADJ, with 100% confidence
Word: stelle, Part of Speech: NOUN, with 100% confidence
Word: ., Part of Speech: PUNCT, with 100% confidence
```

Answer this question: describe what syntaxes are in your own words.

A: Syntax analysis refers to identifying the sentence structure, the part of speech of each word in a text (such as nouns, verbs, adjectives, etc.) and their grammatical roles in a sentence. Here, syntax mainly involves understanding different parts of speech (like nouns, verbs, adjectives, adverbs) and knowing the function of each word in the sentence. This helps language learners and researchers understand sentences better.

AWS Rekognition

AWS Rekognition is a service provided by AWS that allows you to perform machine learning tasks on images.

Add images

To use AWS Rekognition service, we need to find four test images and upload them to AWS S3. We need the following four images:

1. An image of an urban environment (named urban.jpg)

2. An image of a person on a beach (named beach.jpg). Here, we found an image with three people on a beach, which is more suitable for the upcoming test scenario.

3. An image showing a person's face (named faces.jpg).

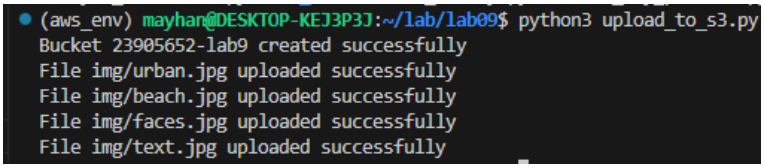
4. An image with text (named text.jpg).

After naming these images, save them to a folder called "img", and in the same directory, create a Python script to create a new AWS S3 bucket and upload the images:

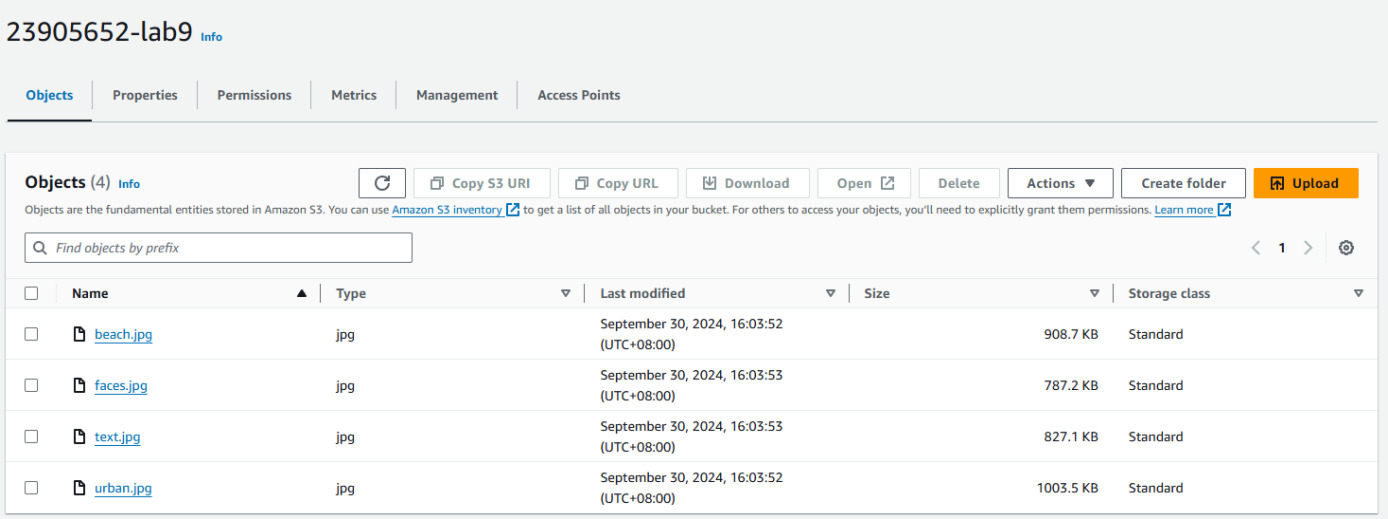
```
import boto3
from botocore.exceptions import ClientError
import os

# Initialize the S3 client
s3_client = boto3.client('s3')

def create_bucket(bucket_name, region):
    # Create an S3 bucket in a specified region
    try:
        # Define the location constraint for the bucket
        location = {'LocationConstraint': region}

        # Create the bucket with the specified name and region
        s3_client.create_bucket(Bucket=bucket_name, CreateBucketConfiguration=location)
        print(f"Bucket {bucket_name} created successfully")
    except ClientError as e:
        # Handle any errors that occur during bucket creation
        print(f"Error creating bucket: {e}")
```

```
        return False
    return True

def upload_file(file_name, bucket, object_name):
    # Upload a file to an S3 bucket
    try:
        # Upload the file to S3
        s3_client.upload_file(file_name, bucket, object_name)
        print(f"File {file_name} uploaded successfully")
    except ClientError as e:
        # Handle any errors that occur during file upload
        print(f"Error uploading file: {e}")
        return False
    return True

# Main execution
if __name__ == "__main__":
    # Set the bucket name and region
    bucket_name = "23905652-lab9"
    region = "ap-southeast-1"

    # Attempt to create the bucket
    if create_bucket(bucket_name, region):
        # If bucket creation is successful, proceed to upload images
        image_folder = "img"  # Folder containing the images
        images = ["urban.jpg", "beach.jpg", "faces.jpg", "text.jpg"]  # List of image
files to upload

        # Iterate through each image and upload it to the bucket
        for image in images:
            # Construct the full file path
            file_path = os.path.join(image_folder, image)
            # Upload the file to S3
            upload_file(file_path, bucket_name, image)
```

Result:



Based on the output, all images have been successfully uploaded, and we can confirm and view the uploaded images in the S3 Console:



Test AWS rekognition

[1] Label Recognition

Label Recognition: Automatically tags objects, concepts, scenes, and actions in images and provides confidence scores.

Update the Python script above with boto3 and AWS rekognition to test label recognition:

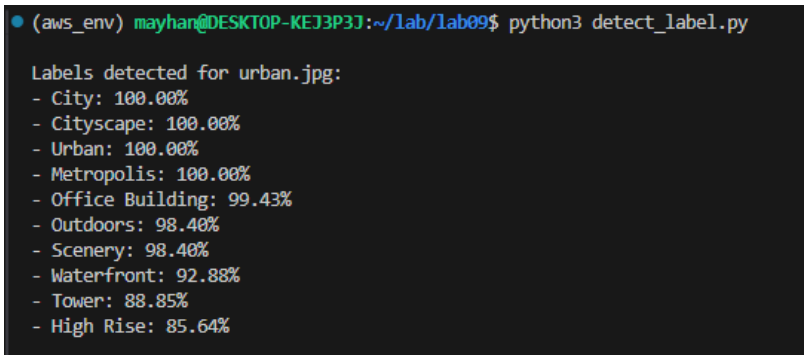
```
import boto3
from botocore.exceptions import ClientError

def detect_labels(bucket, image):
    # Detect labels in an image stored in an S3 bucket using AWS Rekognition.
    rekognition_client = boto3.client('rekognition')

    try:
        # Call the detect_labels API of AWS Rekognition
        response = rekognition_client.detect_labels(
            Image={'S3Object': {'Bucket': bucket, 'Name': image}},  # Specify the S3
object
            MaxLabels=10,  # Limit the number of labels to return
            MinConfidence=85  # Set the minimum confidence threshold for labels
        )

        # Print the detected labels
        print(f"\nLabels detected for {image}:")
        for label in response['Labels']:
            print(f"- {label['Name']}: {label['Confidence']:.2f}%")

    except ClientError as e:
        # Handle any errors that occur during the API call
        print(f"Error detecting labels: {e}")

if __name__ == "__main__":
    # Set the S3 bucket name
    bucket_name = "23905652-lab9"
```

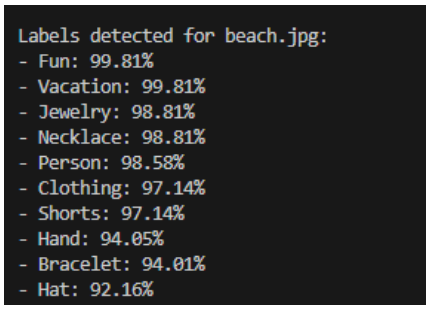
```
# List of images to analyze
images = ["urban.jpg", "beach.jpg", "faces.jpg", "text.jpg"]

# Iterate through each image and detect labels
for image in images:
    detect_labels(bucket_name, image)
```

This script uses AWS Rekognition to detect labels in images stored in an S3 bucket. **Key steps:**

1. `detect_labels` function:
    - Initializes the AWS Rekognition client.
    - Calls AWS Rekognition's `detect_labels` API:
        - Specifies the S3 bucket and image name.
        - Limits the return to a maximum of 10 labels.
        - Sets a minimum confidence threshold of 85%.
    - Prints the detected labels and their confidence scores.
2. Main execution block:
    - Defines the S3 bucket name.
    - Specifies the list of image files to analyze.
    - Iterates through each image, calling the `detect_labels` function.

Result:





```
Labels detected for faces.jpg:
- Head: 99.99%
- Person: 99.99%
- Face: 99.99%
- Happy: 99.99%
- Smile: 99.99%
- Adult: 98.88%
- Female: 98.88%
- Woman: 98.88%
- People: 96.30%
```

```
Labels detected for text.jpg:
- Crime Scene: 94.82%
```

From the output, we can see that the corresponding labels were detected for each image, matching the content.

[2] Image Moderation

Image Moderation: Automatically detects explicit or suggestive adult content or violent content in images and provides confidence scores.

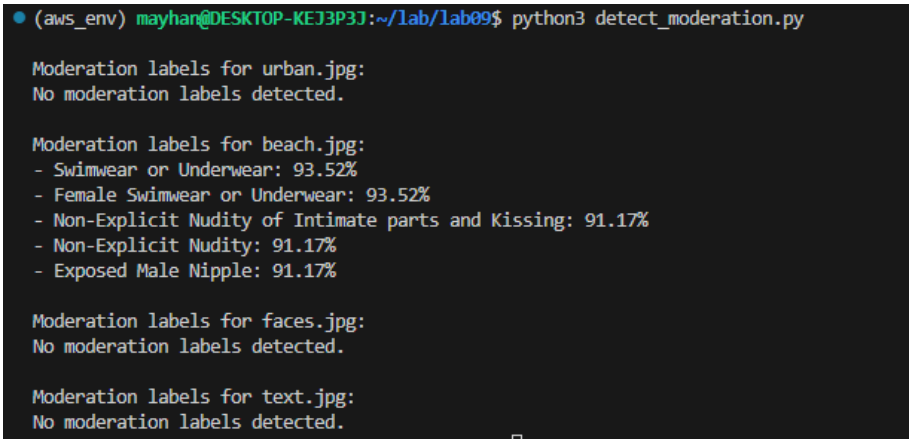
Update the Python script above with boto3 and AWS Rekognition to test image moderation:

```python
import boto3
from botocore.exceptions import ClientError

# Define a function to moderate an image using Amazon Rekognition
def moderate_image(bucket, image):
    rekognition_client = boto3.client('rekognition')
    try:
        # Call the detect_moderation_labels method of the Rekognition client
        # This method analyzes the image for inappropriate or offensive content
        response = rekognition_client.detect_moderation_labels(
            Image={'S3Object': {'Bucket': bucket, 'Name': image}},
            # Set the minimum confidence threshold for detected labels (75%)
            MinConfidence=75
        )
        print(f"\nModeration labels for {image}:")
        # Check if any moderation labels were detected
        if not response['ModerationLabels']:
            print("No moderation labels detected.")
        # Iterate through detected moderation labels and print them
        for label in response['ModerationLabels']:
            print(f"- {label['Name']}: {label['Confidence']:.2f}%")
    except ClientError as e:
        print(f"Error moderating image: {e}")

if __name__ == "__main__":
    bucket_name = "23905652-lab9"
    images = ["urban.jpg", "beach.jpg", "faces.jpg", "text.jpg"]

    for image in images:
        moderate_image(bucket_name, image)
```

This script uses AWS Rekognition to detect moderation labels in images stored in an S3 bucket. **Key steps:**

1. `moderate_image` function:
    - Initializes the AWS Rekognition client.
    - Calls AWS Rekognition's `detect_moderation_labels` API:
        - Specifies the S3 bucket and image name.
        - Sets a minimum confidence threshold of 75%.
    - Prints the detected moderation labels and their confidence scores.
2. Main execution block:
    - Defines the S3 bucket name.
    - Specifies the list of image files to analyze.
    - Iterates through each image, calling the `moderate_image` function.

Result:



According to the output, AWS Rekognition detected "exposed skin" in the "beach.jpg" image. Although the image only contains swimwear and no actual nudity, this proves that the detection is effective. The other images were correctly detected as having no moderation labels.

[3] Facial Analysis

Facial Analysis: Provides a full analysis of facial attributes with confidence scores.

Update the Python script above with boto3 and AWS Rekognition to test facial analysis.

```
import boto3
from botocore.exceptions import ClientError

# Define a function to analyze faces in an image using Amazon Rekognition
def analyze_faces(bucket, image):
    rekognition_client = boto3.client('rekognition')
    try:
        # Call the detect_faces method of the Rekognition client
        # This method analyzes the image for facial features and attributes
        response = rekognition_client.detect_faces(
            Image={'S3Object': {'Bucket': bucket, 'Name': image}},
```

```
            # Request all available attributes for each detected face
            Attributes=['ALL']
        )
        print(f"\nFaces detected in {image}:")

        # Check if any faces were detected
        if not response['FaceDetails']:
            print("No faces detected.")

        # Iterate through detected faces and print their attributes
        for face in response['FaceDetails']:
            print(f"- Age range: {face['AgeRange']['Low']}-{face['AgeRange']['High']}")
            print(f"  Gender: {face['Gender']['Value']} ({face['Gender']
['Confidence']:.2f}%)")
            print(f"  Top emotion: {face['Emotions'][0]['Type']} ({face['Emotions'][0]
['Confidence']:.2f}%)")
            print(f"  Smile: {'Yes' if face['Smile']['Value'] else 'No'} ({face['Smile']
['Confidence']:.2f}%)")
            print(f"  Sunglasses: {'Yes' if face['Sunglasses']['Value'] else 'No'}
({face['Sunglasses']['Confidence']:.2f}%)")
            print(f"  Beard: {'Yes' if face['Beard']['Value'] else 'No'} ({face['Beard']
['Confidence']:.2f}%)")

    except ClientError as e:
        print(f"Error analyzing faces: {e}")

if __name__ == "__main__":
    bucket_name = "23905652-lab9"
    images = ["urban.jpg", "beach.jpg", "faces.jpg", "text.jpg"]

    for image in images:
        analyze_faces(bucket_name, image)
```

This script uses AWS Rekognition to detect and analyze faces in images stored in an S3 bucket. **Key steps:**

1. `analyze_faces` function:
    - Initializes the AWS Rekognition client.
    - Calls AWS Rekognition's `detect_faces` API:
        - Specifies the S3 bucket and image name.
        - Requests all available attributes for each detected face.
    - Prints the detected facial attributes, including (selected from many available attributes that are more suitable for our chosen test images):
        - Age range
        - Gender
        - Top emotion
        - Presence of smile
        - Presence of sunglasses

        - Presence of beard

2. Main execution block:
    - Defines the S3 bucket name.
    - Specifies the list of image files to analyze.
    - Iterates through each image, calling the `analyze_faces` function.

If we modify the `analyze_faces` function to output all the "face" information, we can get the following result:

```
def analyze_faces(bucket, image):
    rekognition_client = boto3.client('rekognition')
    response = rekognition_client.detect_faces(
        Image={'S3Object': {'Bucket': bucket, 'Name': image}},
        Attributes=['ALL']
    )
    print(f"\nFaces detected in {image}:")
    for face in response['FaceDetails']:
        print(face)
```

Result:

We can see that many facial attributes can be detected: BoundingBox, AgeRange, Smile, Eyeglasses, Gender, Beard, Mustache, EyesOpen, MouthOpen, Emotions, Landmarks, Pose, Quality, Confidence, FaceOccluded, EyeDirection, Pitch...

I eventually selected the following attributes for testing, excluding similar ones and those that were not particularly useful for the test images: Age range, Gender, Top emotion, Presence of smile, Presence of sunglasses, Presence of beard.

Final results for the four test images are as follows:

```
(aws_env) mayhan@DESKTOP-KEJ3P3J:~/lab/lab09$ python3 facial_analysis.py

Faces detected in urban.jpg:
No faces detected.

Faces detected in beach.jpg:
- Age range: 19-25
  Gender: Female (99.97%)
  Top emotion: HAPPY (99.67%)
  Smile: Yes (99.72%)
  Sunglasses: No (100.00%)
  Beard: No (99.52%)
- Age range: 21-27
  Gender: Female (99.97%)
  Top emotion: HAPPY (100.00%)
  Smile: Yes (97.98%)
  Sunglasses: Yes (100.00%)
  Beard: No (80.79%)
- Age range: 21-27
  Gender: Male (99.97%)
  Top emotion: CALM (44.07%)
  Smile: No (89.87%)
  Sunglasses: No (99.95%)
  Beard: Yes (96.16%)
```

```
Faces detected in faces.jpg:
- Age range: 23-29
  Gender: Female (99.87%)
  Top emotion: HAPPY (97.07%)
  Smile: Yes (83.81%)
  Sunglasses: No (99.98%)
  Beard: No (97.06%)
- Age range: 25-33
  Gender: Female (99.65%)
  Top emotion: HAPPY (100.00%)
  Smile: Yes (99.91%)
  Sunglasses: No (99.94%)
  Beard: No (98.56%)
- Age range: 25-33
  Gender: Female (99.94%)
  Top emotion: CALM (99.16%)
  Smile: No (98.97%)
  Sunglasses: No (99.63%)
  Beard: No (98.14%)
- Age range: 21-27
  Gender: Female (99.83%)
  Top emotion: HAPPY (99.67%)
  Smile: Yes (99.54%)
  Sunglasses: No (100.00%)
  Beard: No (98.15%)

Faces detected in text.jpg:
No faces detected.
```

From the results, we can see that the number of faces was successfully detected. Other attributes are very accurate, except for the age range, which is difficult to judge even for humans.

[4] Extract Text from an image

Extract Text from an image: Automatically detects and extracts text from images.

Update the Python script above with boto3 and AWS Rekognition to test image text extraction:

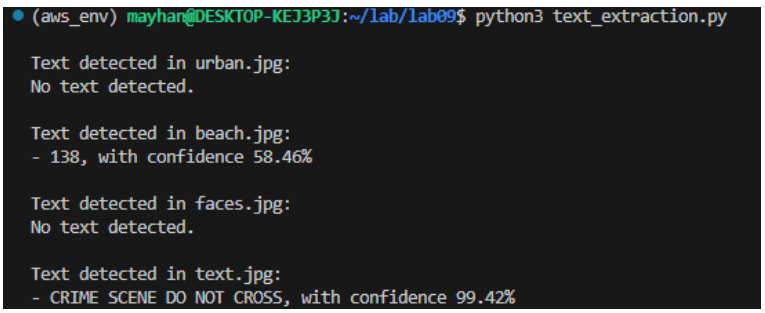
```python
import boto3
from botocore.exceptions import ClientError

# Define a function to extract text from an image using Amazon Rekognition
def extract_text(bucket, image):
    rekognition_client = boto3.client('rekognition')
    try:
        # Call the detect_text method of the Rekognition client
        # This method analyzes the image for text content
        response = rekognition_client.detect_text(
            Image={'S3Object': {'Bucket': bucket, 'Name': image}}
```

```
        )
        print(f"\nText detected in {image}:")
        # Check if any text was detected
        if not response['TextDetections']:
            print("No text detected.")
        # Iterate through detected text elements
        for text in response['TextDetections']:
            # Only print text elements of type 'LINE' (full lines of text)
            if text['Type'] == 'LINE':
                print(f"- {text['DetectedText']}, with confidence
{text['Confidence']:.2f}%")
    except ClientError as e:
        print(f"Error extracting text: {e}")

if __name__ == "__main__":
    bucket_name = "23905652-lab9"
    images = ["urban.jpg", "beach.jpg", "faces.jpg", "text.jpg"]

    for image in images:
        extract_text(bucket_name, image)
```

This script uses AWS Rekognition to detect and extract text from images stored in an S3 bucket. **Key steps:**

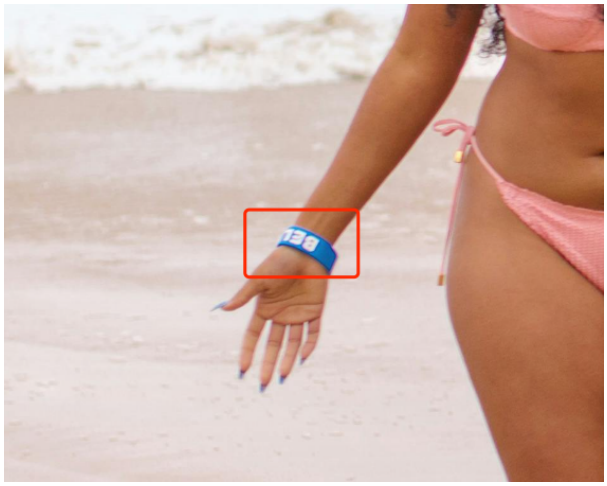
1. `extract_text` function:
    - Initializes the AWS Rekognition client.
    - Calls AWS Rekognition's `detect_text` API, specifying the S3 bucket and image name.
    - Prints the detected text:
        - Checks if any text is detected.
        - Iterates through the detected text elements.
        - Only prints text elements of type "LINE" (full lines of text).
        - Displays the detected text and its confidence score.
2. Main execution block:
    - Defines the S3 bucket name.
    - Specifies the list of image files to analyze.
    - Iterates through each image, calling the `extract_text` function.

Result:



The image "text.jpg" is mainly used to test text detection, and the results show that the detection was very accurate.

For the image "beach.jpg," it even detected the numbers on the girl's wrist (which I initially didn't notice). It looks like "138" or maybe "BEL". It showed a confidence of 58.46%, which seems reasonable.



[3] Manually delete the S3 bucket

After finishing the lab, we have to manually delete the created S3 bucket in the S3 console. Select the resource "23905652-lab9" to delete, then click "Empty" to clear the bucket, and follow the prompts to complete the operation. After emptying, click "Delete" to remove the bucket, and follow the prompts to complete the deletion (the screenshot is for demonstration purposes only).