NAME : VADLA LAKSHMI SWETHA

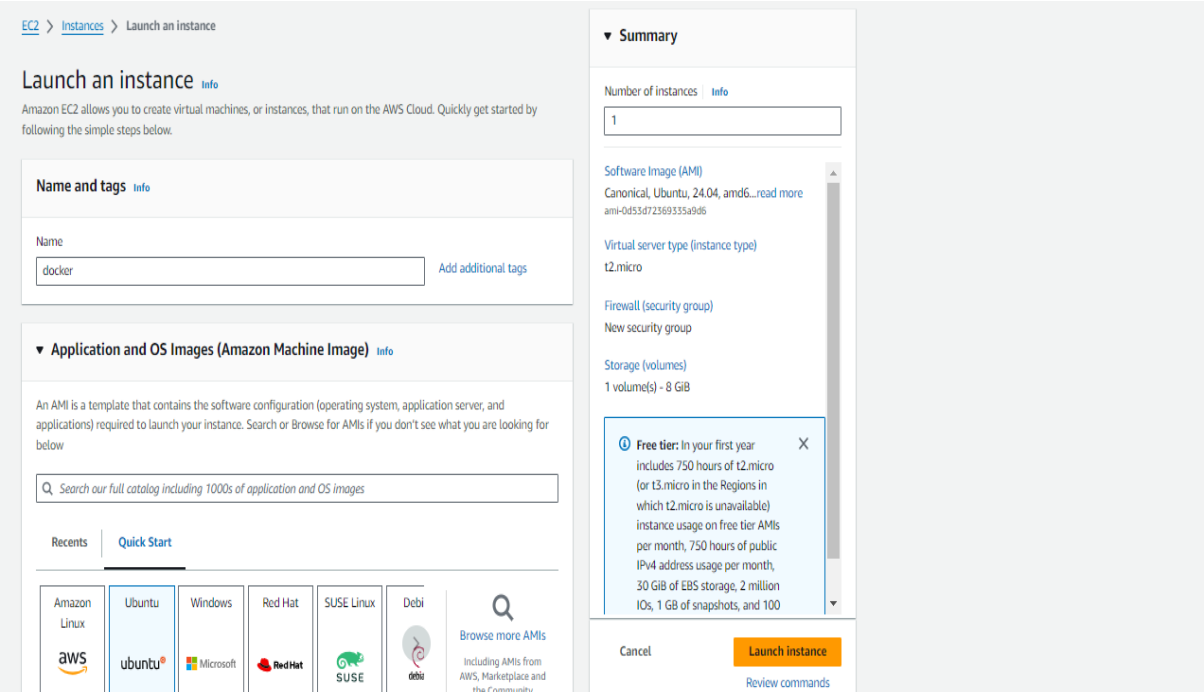
BATCH NO. : 126

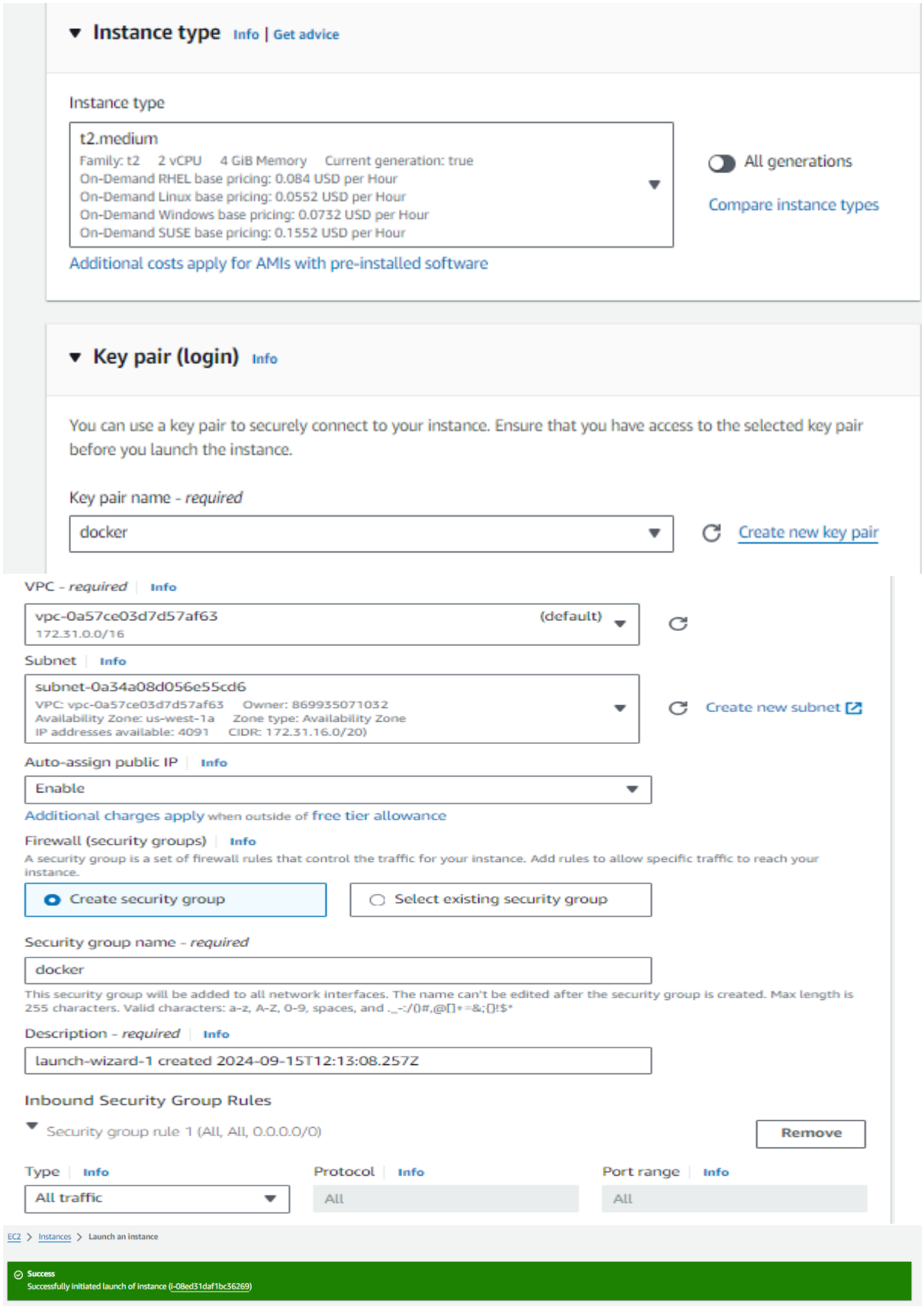
MAIL ID: lakshmiswethavadla@gmail.com

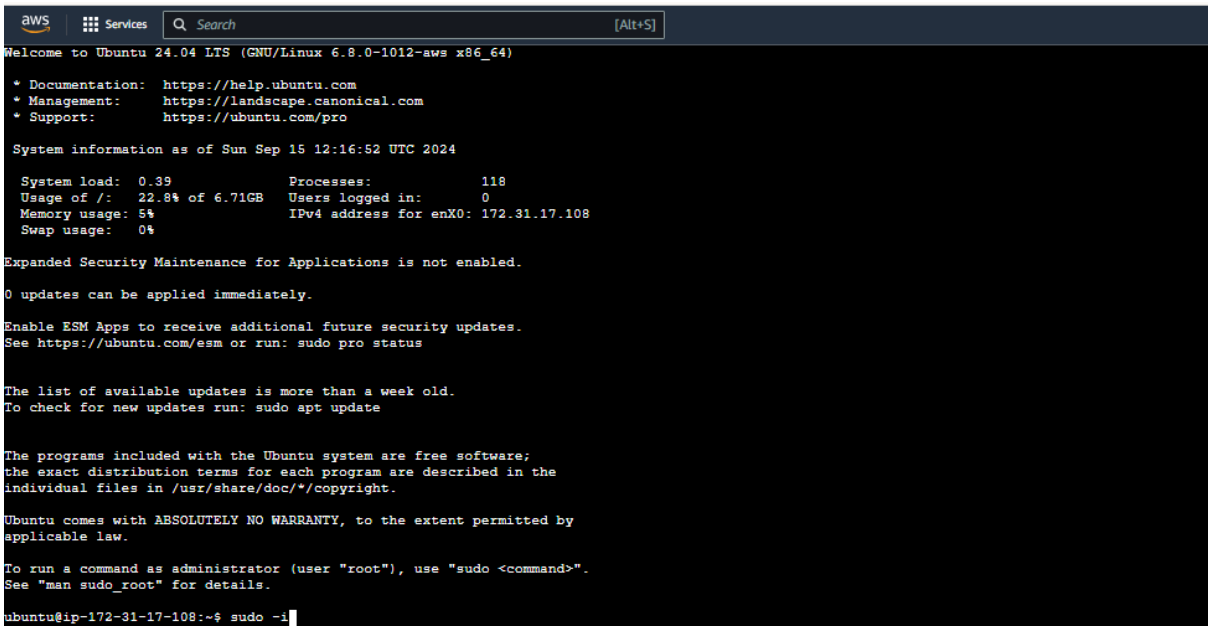
PROJECT :

Create your own Image by using required Dockerfile instructions

Launch an instance and connect to it →

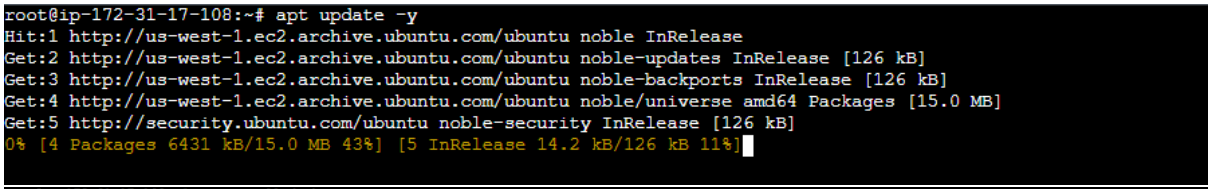


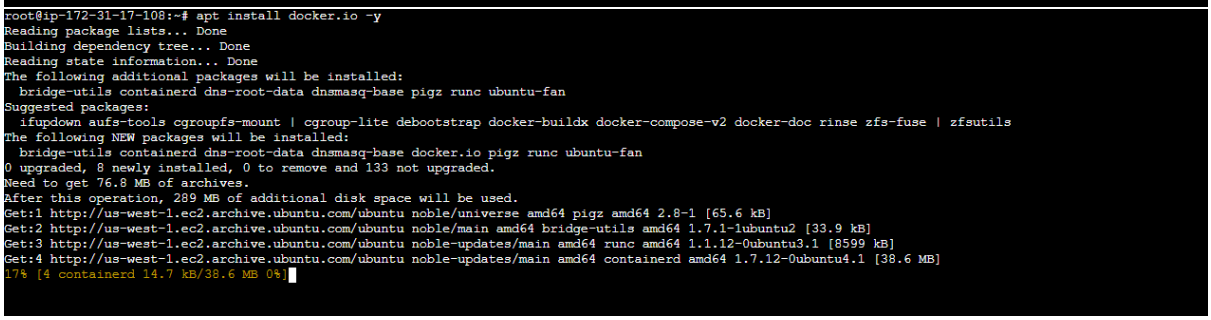

Instance type Info | Get advice
Instance type
t2.medium
Family: t2 2 vCPU 4 GiB Memory Current generation: true
On-Demand RHEL base pricing: 0.084 USD per Hour
On-Demand Linux base pricing: 0.0552 USD per Hour
On-Demand Windows base pricing: 0.0732 USD per Hour
On-Demand SUSE base pricing: 0.1552 USD per Hour
All generations
Compare instance types
Additional costs apply for AMIs with pre-installed software
Key pair (login) Info
You can use a key pair to securely connect to your instance. Ensure that you have access to the selected key pair before you launch the instance.
Key pair name - required
docker
Create new key pair
VPC - required Info
vpc-0a57ce03d7d57af63 (default)
172.31.0.0/16
Subnet Info
subnet-0a34a08d056e55cd6
VPC: vpc-0a57ce03d7d57af63 Owner: 869935071032
Availability Zone: us-west-1a Zone type: Availability Zone
IP addresses available: 4091 CIDR: 172.31.16.0/20)
Create new subnet
Auto-assign public IP Info
Enable
Additional charges apply when outside of free tier allowance
Firewall (security groups) Info
A security group is a set of firewall rules that control the traffic for your instance. Add rules to allow specific traffic to reach your instance.
Create security group
Select existing security group
Security group name - required
docker
This security group will be added to all network interfaces. The name can't be edited after the security group is created. Max length is 255 characters. Valid characters: a-z, A-Z, 0-9, spaces, and ._-:/()#,@[]+=&;{}!$*
Description - required Info
launch-wizard-1 created 2024-09-15T12:13:08.257Z
Inbound Security Group Rules
Security group rule 1 (All, All, 0.0.0.0/0)
Remove
Type Info
All traffic
Protocol Info
All
Port range Info
All
EC2 > Instances > Launch an instance
Success
Successfully initiated launch of instance (i-08ed31daf1bc36269)

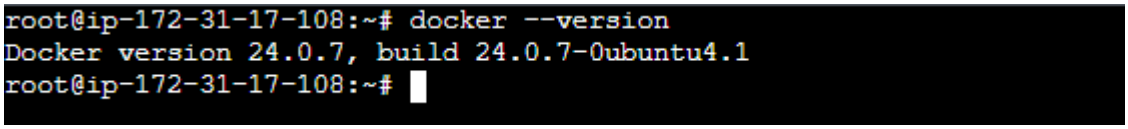


Step 1: Install Docker

Make sure Docker is installed on your machine.



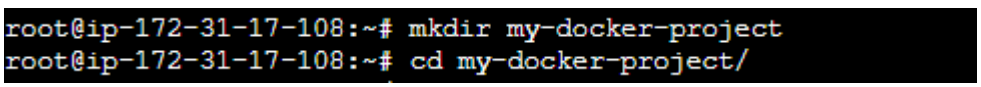




Step 2: Create a Project Directory

Create a new directory for your project. This will hold your `Dockerfile` and any other files you need.

- **mkdir my-docker-project**
- **cd my-docker-project**



Step 3: Create the Dockerfile

Inside your project directory, create a file named `Dockerfile` (with no extension). This file will contain the instructions for building your Docker image.

- **vi Dockerfile**

```
root@ip-172-31-17-108:~/my-docker-project# vi Dockerfile
root@ip-172-31-17-108:~/my-docker-project#
```

Step 4: Write Dockerfile Instructions

Open the `Dockerfile` with a text editor and add the necessary instructions. Below is an example `Dockerfile` for creating a simple Python application image.

Example: Dockerfile for a Python Application

- Step 1: Use an official Python runtime as a parent image
  FROM python:3.9-slim
- Step 2: Set the working directory in the container
  WORKDIR /app
- Step 3: Copy the current directory contents into the container at /app
  COPY . /app
- Step 4: Install any needed packages specified in requirements.txt
  RUN pip install --no-cache-dir -r requirements.txt
- Step 5: Make port 80 available to the world outside this container
  EXPOSE 80
- Step 6: Define environment variable
  ENV NAME World
- Step 7: Run app.py when the container launches
  CMD ["python", "app.py"]

```
FROM python:3.9-slim
WORKDIR /app
COPY . /app
RUN pip install --no-cache-dir -r requirements.txt
EXPOSE 80
ENV NAME World
CMD ["python", "app.py"]
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
:wq
```

Explanation of Dockerfile Instructions

1. FROM: Specifies the base image to use. In this case, `python:3.9-slim` is used, which is a lightweight version of the Python image.

2. WORKDIR: Sets the working directory inside the container. All subsequent instructions will be run in this directory.

3. COPY: Copies files from your local directory (where the `Dockerfile` is) to the working directory inside the container.

4. RUN: Executes commands inside the container. Here, it installs Python packages listed in `requirements.txt`.

5. EXPOSE: Documents the port on which the container will listen for connections.

6. ENV: Sets an environment variable inside the container.

7. CMD: Specifies the command to run when the container starts. Here, it runs `app.py` using Python.

Step 5: Create Additional Files

For the example Dockerfile above, you would need an `app.py` and a `requirements.txt` file in the same directory.

app.py:

python

```
root@ip-172-31-17-108:~/my-docker-project# vi app.py
```

```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def hello_world():
    return 'Hello, World!'

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=80)
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
:wq
```

requirements.txt:

```
root@ip-172-31-17-108:~/my-docker-project# vi requirements.txt
root@ip-172-31-17-108:~/my-docker-project#
```

Step 6: Build the Docker Image

Use the `docker build` command to build the image from your Dockerfile.

docker build -t my-python-app .

```
root@ip-172-31-17-108:~/my-docker-project# docker build -t my-python-app .
DEPRECATED: The legacy builder is deprecated and will be removed in a future release.
            Install the buildx component to build images with BuildKit:
            https://docs.docker.com/go/buildx/

Sending build context to Docker daemon  4.096kB
Step 1/7 : FROM python:3.9-slim
 ---> 397ed8d31636
Step 2/7 : WORKDIR /app
 ---> Using cache
 ---> 04868a926717
Step 3/7 : COPY . /app
 ---> b2321905484a
Step 4/7 : RUN pip install --no-cache-dir -r requirements.txt
 ---> Running in 7084774d9b7d
```

- `-t my-python-app`: Tags the image with the name `my-python-app`.

- `.`: Specifies the build context, which is the current directory.

Step 7: Run the Docker Container

Run a container from your newly created image to test it.

docker run -p 4000:80 my-python-app

```
root@ip-172-31-17-108:~/my-docker-project# docker run -p 4000:80 my-python-app
 * Serving Flask app 'app'
 * Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on all addresses (0.0.0.0)
 * Running on http://127.0.0.1:80
 * Running on http://172.17.0.2:80
Press CTRL+C to quit
```

- `-p 4000:80`: Maps port 80 in the container to port 4000 on your host machine.

- `my-python-app`: The name of the image to run.

Step 8: Test Your Application

Open a web browser and go to `http://localhost:4000`. You should see "Hello, World!" displayed.

Step 9: Clean Up

To stop the container, press `Ctrl+C` in the terminal. To remove the container and image, use the following commands:

docker ps -a # List all containers

docker rm <container_id> # Remove a container

docker rmi my-python-app # Remove an image