Docker Weekend Task – 2

1. **Write a brief explanation of what Docker volumes are and why they are used in containerized environments.**

   State different types of volumes in Docker and also make a note on difference between them.

   Docker Volumes are storage for long-term data created and consumed by Docker containers. Volumes differ from storage in containers because they are controlled by Docker and persist even after the container is removed. Volumes are typically used to share data between containers, or to store database files, logs, uploads, etc. Volumes persist in a Docker-specific location and provide improved performance, portability, and flexibility compared to bind mounts, which are more difficult to use.

   Why Use Docker Volumes

   1. Data Persistence
      Volumes ensure that data remains intact even if a container is stopped, deleted, or recreated.
      For example, in a container running a database like MySQL or PostgreSQL, using a volume ensures that the actual data stored in the database is not lost when the container is updated or restarted.

   2. Data Sharing Between Containers
      Volumes allow multiple containers to access the same data simultaneously. This is especially useful in microservices architecture or when you have a separate container for logging, caching, or analytics that needs access to the same files or datasets.
      Example: A container uploads user files to a volume, and another container processes or scans those files from the same volume.

   3. Clean Separation of Concerns
      With volumes, you can move the application code and its data into separate entities, keeping your container structure cleaner and more modular.
      Your app is stateless, and the data is managed separately.

   4. Backup & Restore
      Since volumes are managed by Docker, it's easier to back up or migrate them. You can archive a volume's content and transfer it across environments (e.g., dev to prod) or host machines.

5. Performance
   Volumes are optimized for performance, especially compared to bind mounts. Docker can manage how volumes are stored on disk to ensure fast read/write speeds and reduce I/O overhead.

   Volumes also avoid some file syncing issues seen with bind mounts on Windows/macOS.

Types of Docker Volumess

Type	Description
Named Volumes	Created and managed by Docker. Stored in /var/lib/docker/volumes/. Useful for long-term persistence and sharing across containers.
Anonymous Volumes	Similar to named volumes but without a name. Created when a container runs and removed when the container is removed (unless retained).
Bind Mounts	Mounts a specific path from the host system into the container. Allows more control but less portability.
tmpfs Mounts	Stores data in the host system's memory only (RAM). Ideal for sensitive or temporary data (non-persistent).

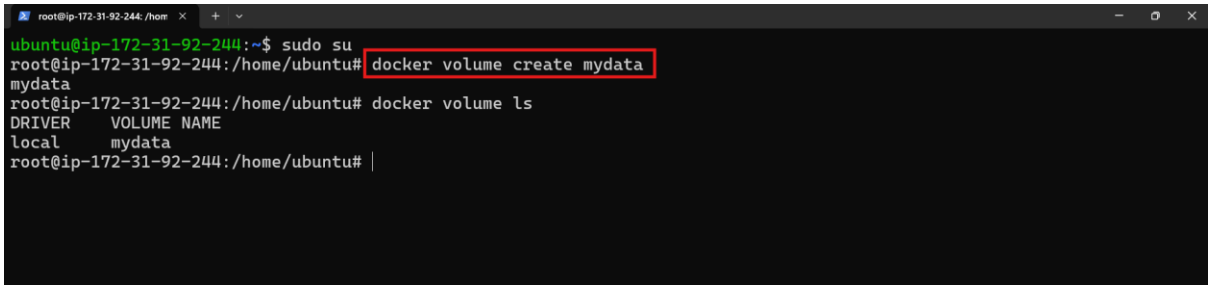
Key Differences Between Volume Types

Feature	Named Volume	Anonymous Volume	Bind Mount	tmpfs Mount
Persistence	Yes	Yes (short-term)	Yes	No
Location	Managed by Docker	Managed by Docker	Host-defined	Memory
Sharing	Yes	Possible but hard	Yes	No
Ease of Use	High	Medium	Low	Medium
Security	High	Medium	Low	High
Use Case Example	Databases, volumes across containers	Ephemeral containers	Mounting config files	Sensitive credentials or cache

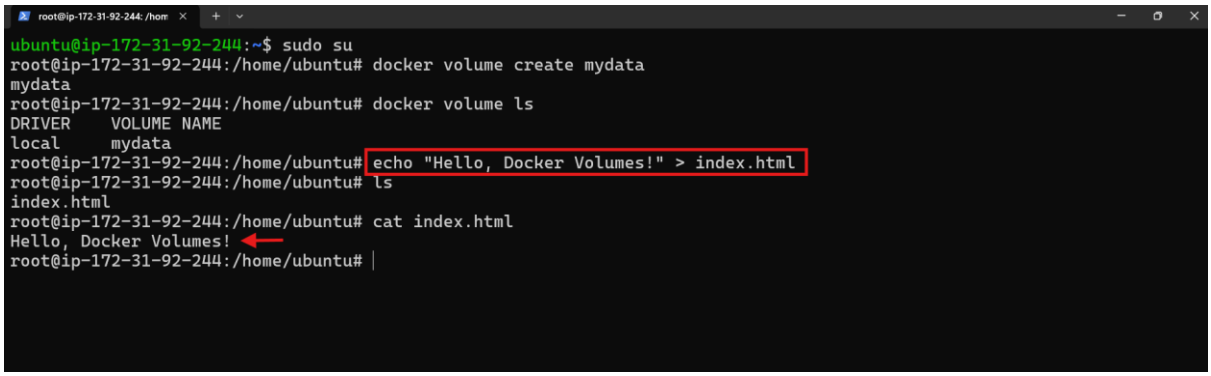
- **Named Volume**: Persistent and Docker-managed, ideal for sharing data like databases across containers.
- **Anonymous Volume**: Temporary Docker-managed storage, used when persistence isn't a priority.
- **Bind Mount**: Maps host directories into containers, useful for development or config files, but less secure.
- **tmpfs Mount**: Stores data in memory only, perfect for sensitive or temporary data that must not persist.

2. **Demonstrate the use of Named Volume.**
    - **Create a Docker Named volume named mydata.**
    - **Attach volume to a Nginx Container**
    - **Create an HTML file named index.html with some content (e.g., "Hello, Docker Volumes!") on your host machine. Copy this file into the mydata.**
    - **Verify that the index.html file is accessible from within the container by starting a simple HTTP request.**

Step 1: Create a Named Volume

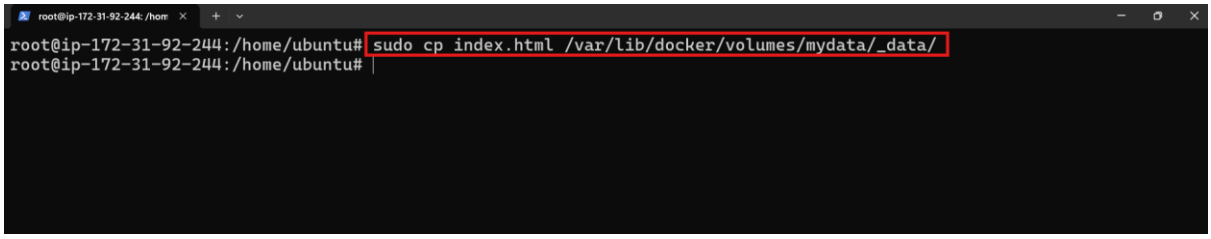


Step 2: Create index.html File on Host



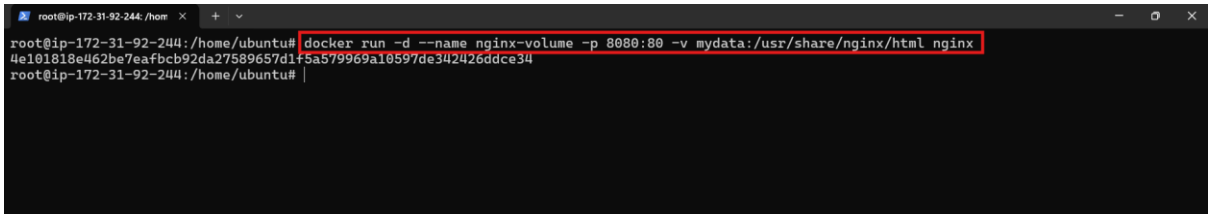
Step 3: Copy index.html into the Volume

Docker named volumes are stored under /var/lib/docker/volumes/<volume-name>/_data.



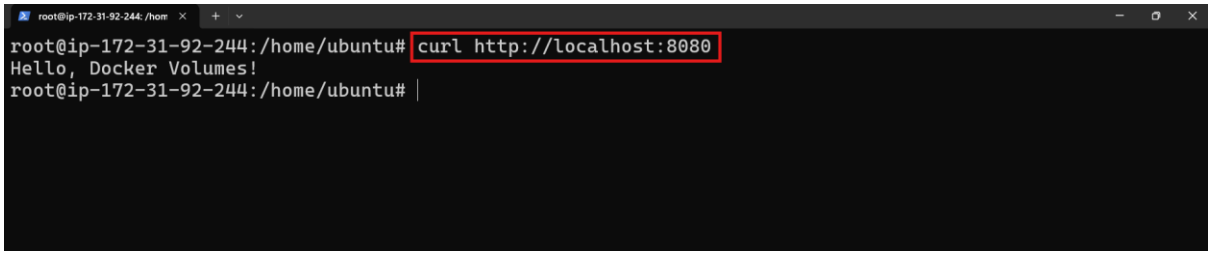
Step 4: Run Nginx Container with the Volume

Mount the volume to NGINX's default web directory /usr/share/nginx/html



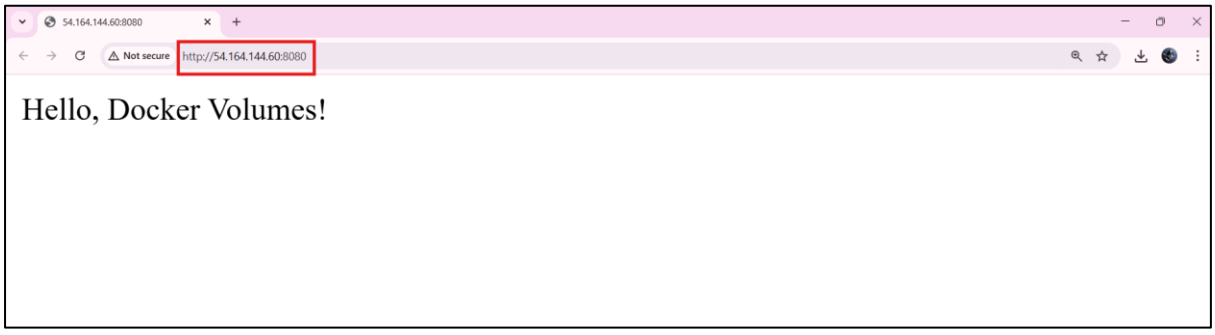
Step 5: Verify the Setup

- Option 1: Using curl (Command Line)



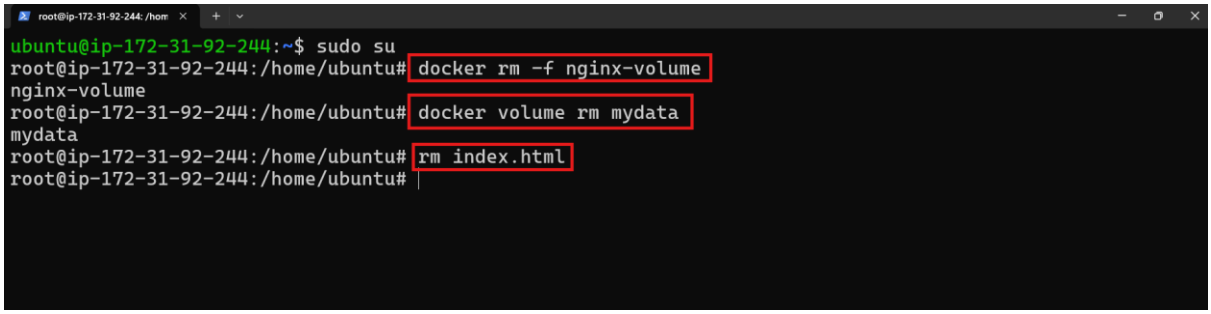
- Option 2: Open in Web Browser

Must allow port 8080 in the EC2 security group's inbound rules if want to access **http://<EC2-public-IP>:8080** from your browser (on local machine).



Step 6: Clean Up

To stop and remove the container and volume



3. **Write a brief explanation of what Docker networks.**

   Write the difference between host network and bridge network.

   Docker networks enable communication between containers, and between containers and the host machine or external systems. They provide isolated, secure communication channels so containers can interact with each other or services outside the container environment.

   When Docker runs a container, it connects it to a network — which can be default or custom, and can have different configurations depending on the use case.

   Types of Docker Networks

   - **Bridge (default) –** Used by default for standalone containers.
   - **Host** – Container shares the host's networking stack**.**
   - **None –** No network access (isolated container).
   - **Overlay –** Used in Docker Swarm for multi-host communication**.**
   - **Macvlan –** Assigns a MAC address to a container, making it appear as a physical device.

   Difference Between Host and Bridge Network

Feature	**Bridge Network**	**Host Network**
Isolation	Containers have their own network namespace	Shares the host's network namespace
IP Address	Container gets a private IP (e.g., 172.x.x.x)	Container uses the host's IP address
Port Mapping	Required (e.g., -p 8080:80)	Not required (container uses host ports directly)
Use Case	Default for container-to-container communication	When performance is critical (e.g., low latency)

Security	More secure due to isolation	Less secure; container has access to host network
Performance	Slight overhead due to network translation	Better performance, no network translation
Docker Compose Support	Yes	Limited (must be manually configured)

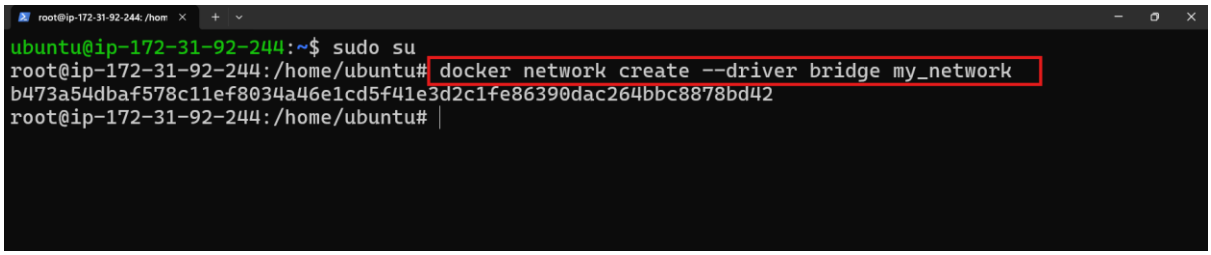
Example

- **Bridge:** Microservices app where containers need to talk via service names.
- **Host**: High-performance apps (like game servers or monitoring tools) that need raw access to network interfaces.

4. Demonstrate the use of Custom Network

- **Create a custom bridge network named my_network.**
- **Start two containers, one using the nginx image and another using the httpd image.**
- **Attach both containers to the my_network network.**
- **Test Network Connectivity: Ensure that the nginx container can communicate with the httpd container over the custom network. You can do this by sending an HTTP request from one container to another using tools like curl.**

Step 1: Create a Custom Bridge Network



Step 2: Start the Containers on the Custom Network

```
b473a54dbaf578c11ef8034a46e1cd5f41e3d2c1fe86390dac264bbc8878bd42
root@ip-172-31-92-244:/home/ubuntu#
root@ip-172-31-92-244:/home/ubuntu# docker run -d --name my-nginx nginx
Unable to find image 'nginx:latest' locally
latest: Pulling from library/nginx
3da95a905ed5: Pull complete
6c8e51cf0087: Pull complete
9bbbd7ee45b7: Pull complete
48670a58a68f: Pull complete
ce7132063a56: Pull complete
23e05839d684: Pull complete
ee95256df030: Pull complete
Digest: sha256:93230cd54060f497430c7a120e2347894846a81b6a5dd2110f7362c5423b4abc
Status: Downloaded newer image for nginx:latest
90b53729d8bfce93b612b3649f3260bf5812aeb6878fc9b434e869c2cb22da43
root@ip-172-31-92-244:/home/ubuntu# docker run -d --name my-httpd httpd
Unable to find image 'httpd:latest' locally
latest: Pulling from library/httpd
3da95a905ed5: Already exists
1a50d37c990c: Pull complete
4f4fb700ef54: Pull complete
365277d54dac: Pull complete
d812016dda12: Pull complete
816d153af128: Pull complete
Digest: sha256:1ae8051591a5ded56e4a3d7399c423e940e8475ad0e5adb82e6e10893fe9b365
Status: Downloaded newer image for httpd:latest
37443a62227346bd5e206e4b62b363f7b047bcab9423f6238605803a3b75ced0
root@ip-172-31-92-244:/home/ubuntu#
```

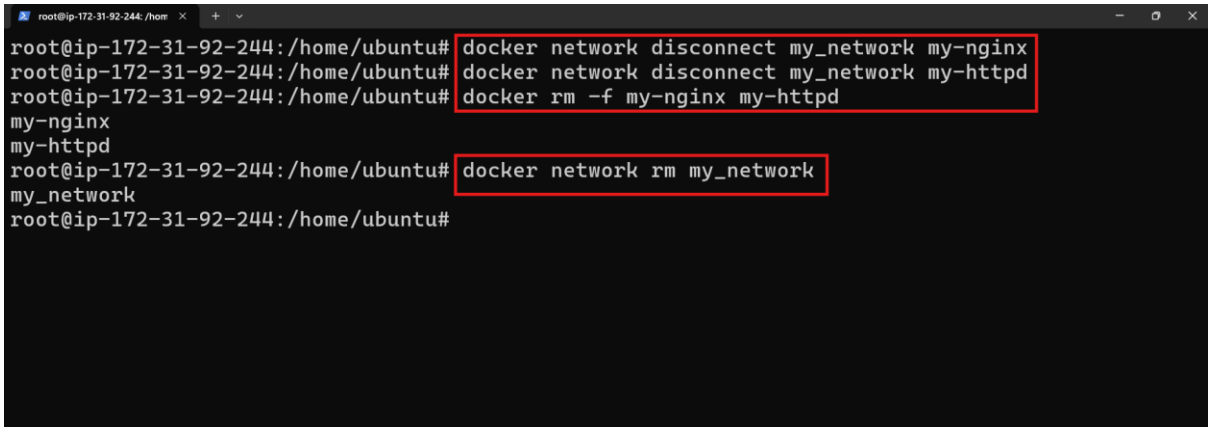
Step 3: Connect Containers to the Custom Network

```
37443a62227346bd5e206e4b62b363f7b047bcab9423f6238605803a3b75ced0
root@ip-172-31-92-244:/home/ubuntu#
root@ip-172-31-92-244:/home/ubuntu# docker network connect my_network my-nginx
root@ip-172-31-92-244:/home/ubuntu# docker network connect my_network my-httpd
root@ip-172-31-92-244:/home/ubuntu#
```

Step 4: Test Network Connectivity

```
root@ip-172-31-92-244:/home/ubuntu# docker network connect my_network my-nginx
root@ip-172-31-92-244:/home/ubuntu# docker network connect my_network my-httpd
root@ip-172-31-92-244:/home/ubuntu#
root@ip-172-31-92-244:/home/ubuntu# docker exec -it my-nginx sh
# curl http://my-httpd
<html><body><h1>It works!</h1></body></html>
#
```

Step 5: Cleanup (Disconnect & Remove)



5. Write a note on Dockerfile with usage of its attributes

A Dockerfile is a text file that contains step-by-step instructions on how to build a Docker image. It defines the base image, sets up the environment, copies application code, installs dependencies, exposes ports, and specifies the command to run the application.

It is the blueprint for creating Docker images in a repeatable, automated way.

Instruction	Description
`FROM`	Sets the **base image**. It must be the first instruction (except for optional ARGs). Example: `FROM ubuntu:20.04`
`LABEL`	Adds metadata. Example: `LABEL version="1.0"`
`ENV`	Sets environment variables. Example: `ENV PATH="/app/bin:$PATH"`
`WORKDIR`	Sets the working directory for subsequent commands. Example: `WORKDIR /app`
`COPY`	Copies files from host to container. Example: `COPY . /app`
`ADD`	Like `COPY`, but supports remote URLs and auto-extracts archives.
`RUN`	Executes commands during image build. Example: `RUN apt-get update && apt-get install -y nginx`
`CMD`	Sets default command to run when the container starts. Example: `CMD ["nginx", "-g", "daemon off;"]`
`ENTRYPOINT`	Sets the main command that cannot be overridden easily.
`EXPOSE`	Documents the port the app uses (does not publish it).
`ARG`	Defines build-time variables. Example: `ARG version=1.0`
`VOLUME`	Creates a mount point for persistent data.
`USER`	Sets the user to run the image as.
`HEALTHCHECK`	Defines how Docker should check if the container is healthy.
`MAINTAINER`	To indicate who built or maintains the image.

Benefits of Using Dockerfile

- Automation of image creation
- Consistency across environments
- Easier version control and debugging
- Reusability and modular builds

6. What is difference between CMD and ENTRYPOINT?

Dockerfile instructions CMD and ENTRYPOINT specify which command should be executed when a container starts. But they behave differently and have different functions.

Feature	CMD	ENTRYPOINT
Purpose	Sets a **default command or arguments**	Defines a **fixed command to run**
Overridable	Yes, completely overridden by docker run	No, only additional arguments are appended
Command Type	Can be used as **default parameters**	Treated as the **main executable**
Typical Use Case	Specify default behavior that can be changed	Enforce consistent behavior (e.g., always run app)
Preferred Format	Shell or exec (["arg1", "arg2"])	**Exec format** (["executable", "param"])
Flexibility	More flexible (fully replaceable)	Less flexible (command is fixed)
Ignored if ENTRYPOINT is set	Yes	No (always runs)
Example	CMD ["npm", "start"]	ENTRYPOINT ["nginx"]

7. What is difference between ADD and COPY?

To move files or directories from the host computer to the Docker image, use the ADD and COPY Dockerfile instructions. Their use cases and functionalities vary.

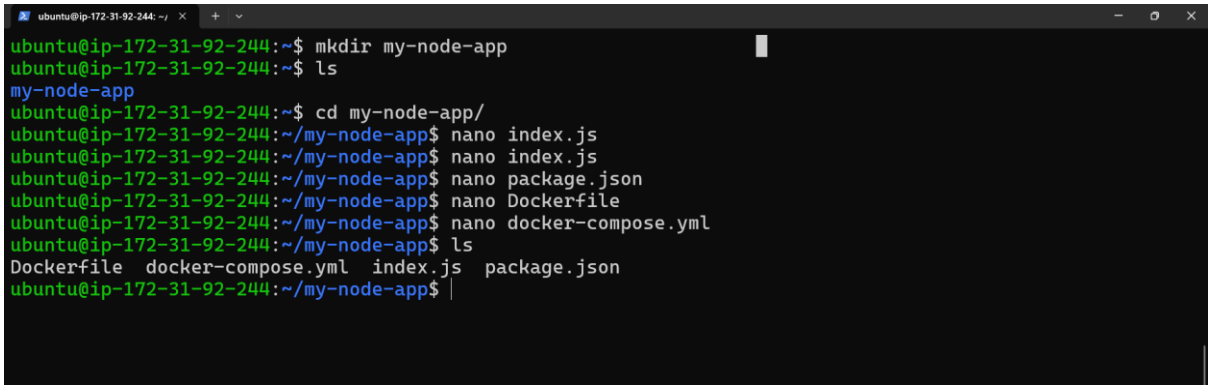
Feature	COPY	ADD
Basic Function	Copies files/directories to container image	Copies files/directories **with extra features**

Archive Extraction	No	Yes, extracts .tar, .tar.gz, etc. automatically
Remote URL Support	No	Yes, can fetch files from remote URLs
Clarity & Simplicity	Simple and explicit	Can behave differently depending on source
Preferred Use	For basic file/folder copying	When archive extraction or URL download is needed
Best Practice	Recommended for most use cases	Use only when its special features are required
Security	More secure due to limited functionality	Slightly riskier (downloads and auto-extraction)

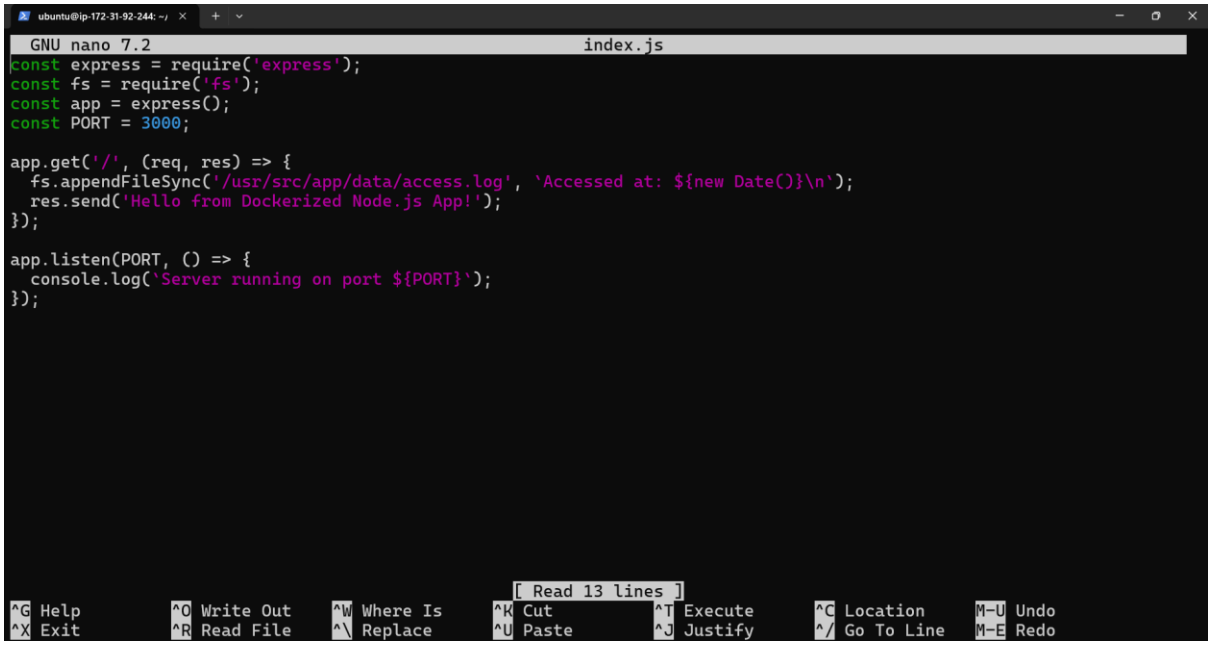
8. **Write a Dockerfile to run Nodejs application build an image from it and create a container using that image (also include persistent volume and network in Dockerfile).**

Step 1: Create Project Structure

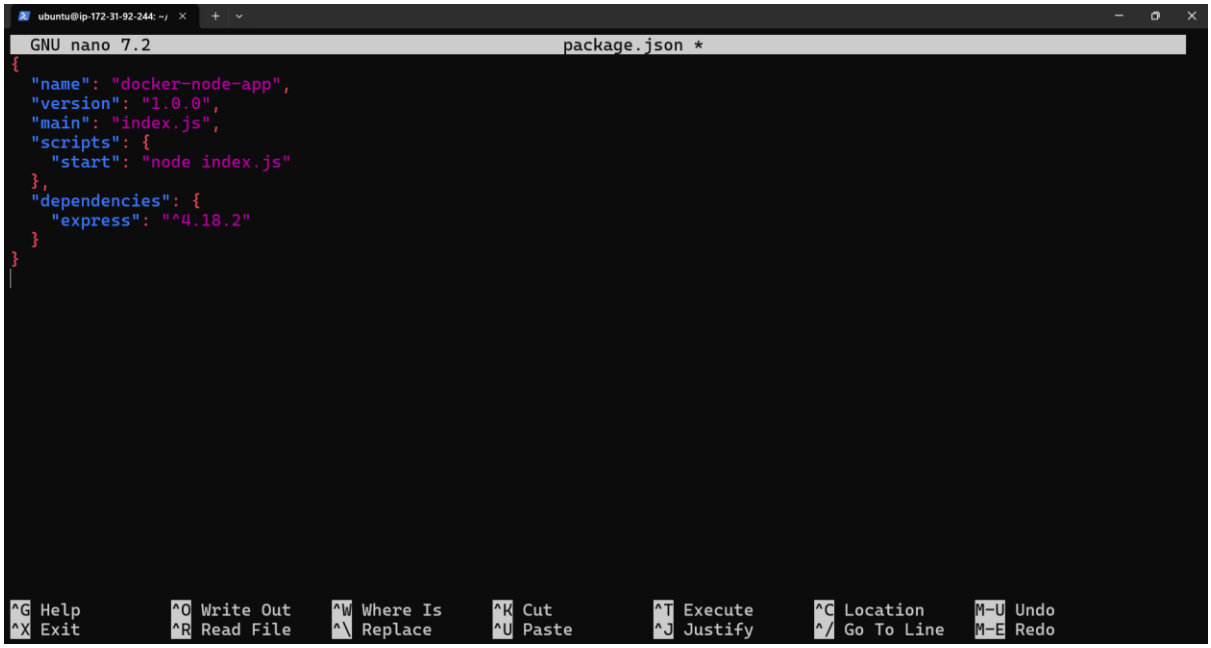
```
my-node-app/
├── Dockerfile
├── docker-compose.yml
├── index.js
└── package.json
```



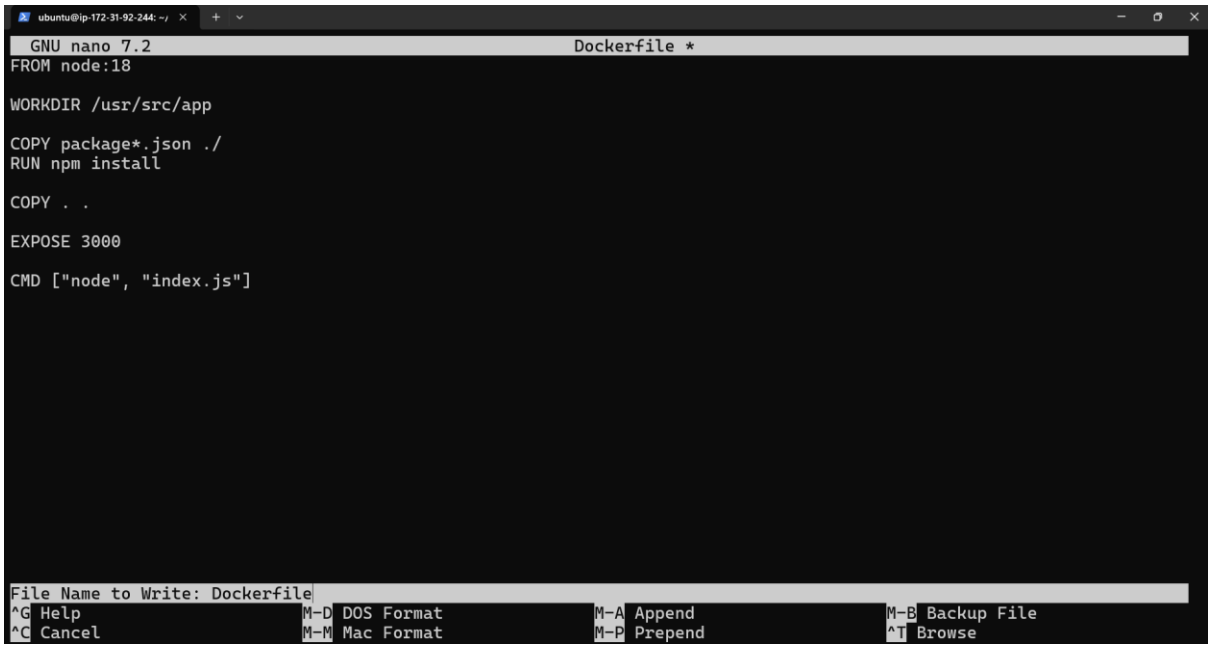
Step 2: Create index.js (Node.js App)



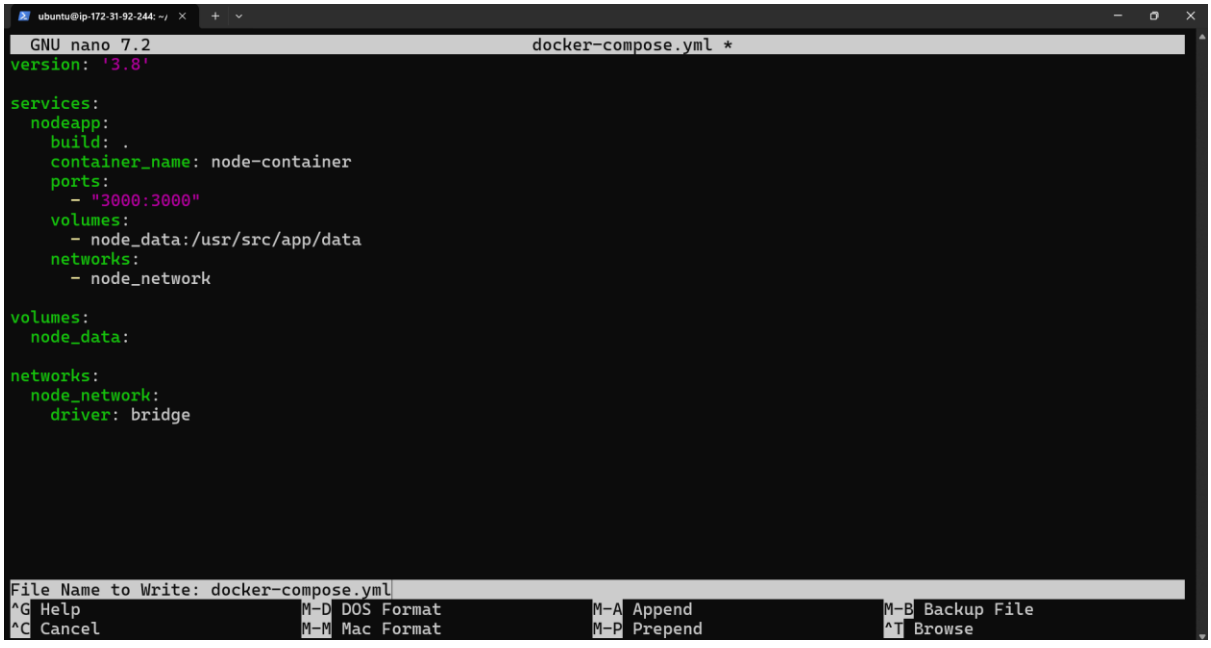
Step 3: Create package.json



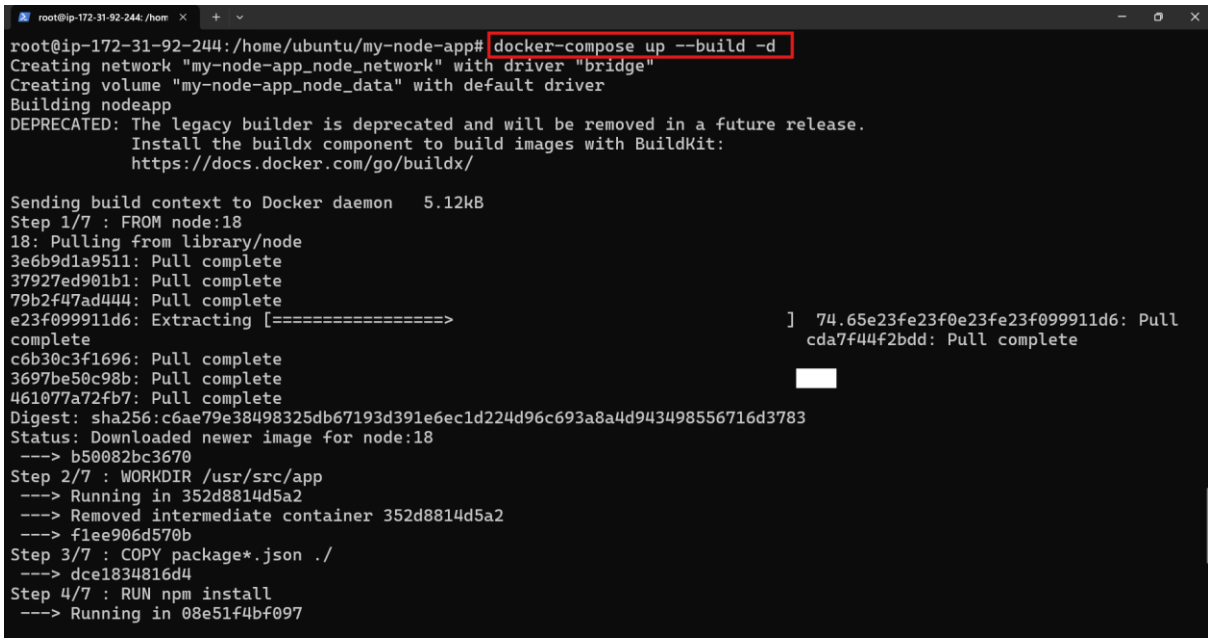
Step 4: Create Dockerfile



Step 5: Create docker-compose.yml

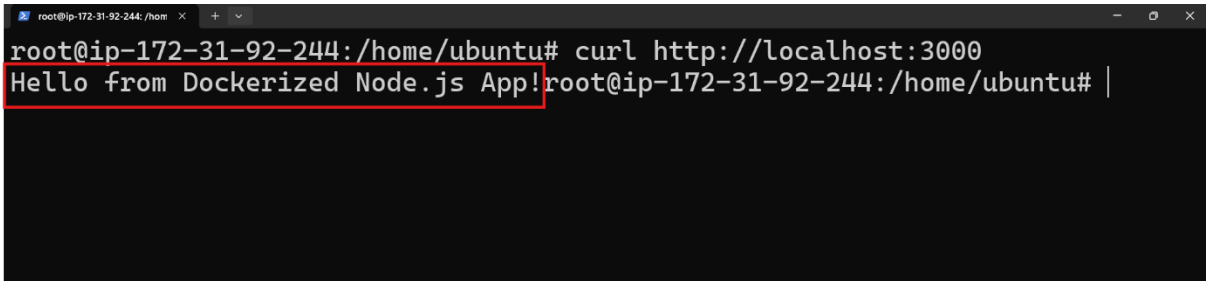


Step 6: Build and Run the Application

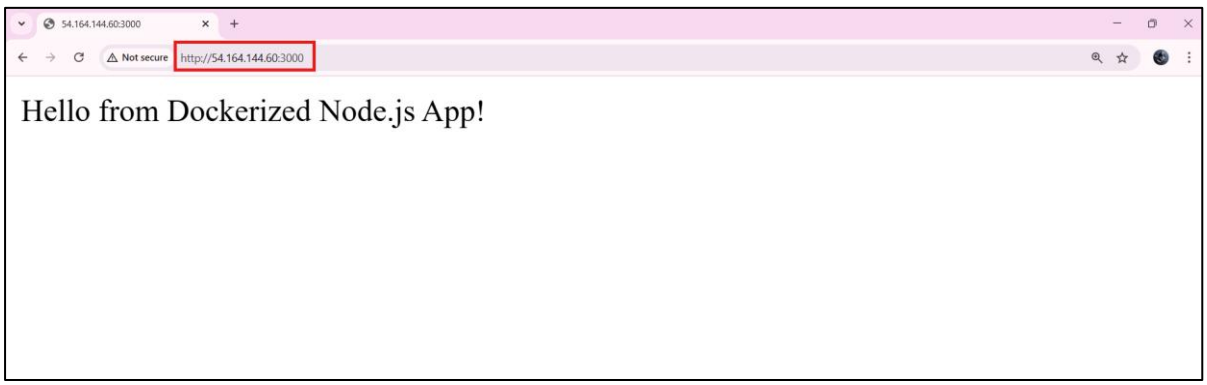


Step 7: Test the Application

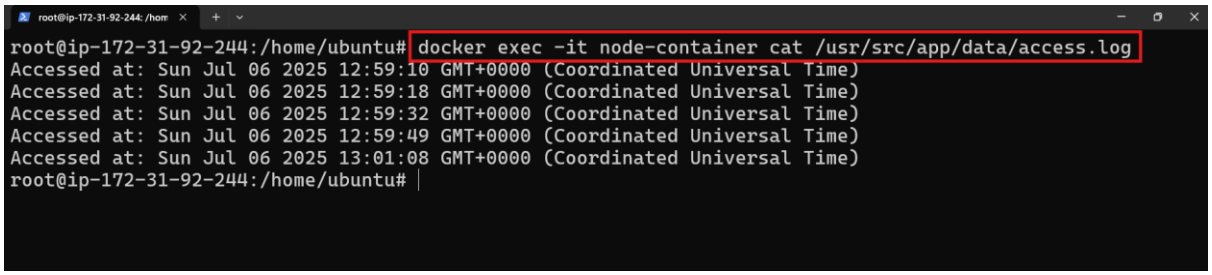
- Option 1: Using curl (Command Line)



- Option 2: Open in Web Browser



Step 8: Check Persistent Log Volume



9. **Write a Dockerfile to create a python application build image from it and push that image to private repository of Docker hub.**

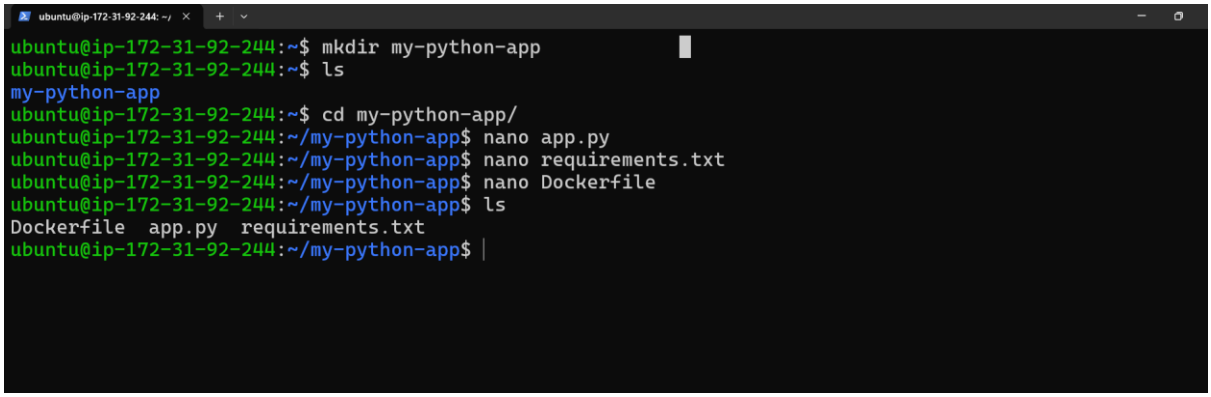
Step 1: Create Your Python Project

my-python-app/

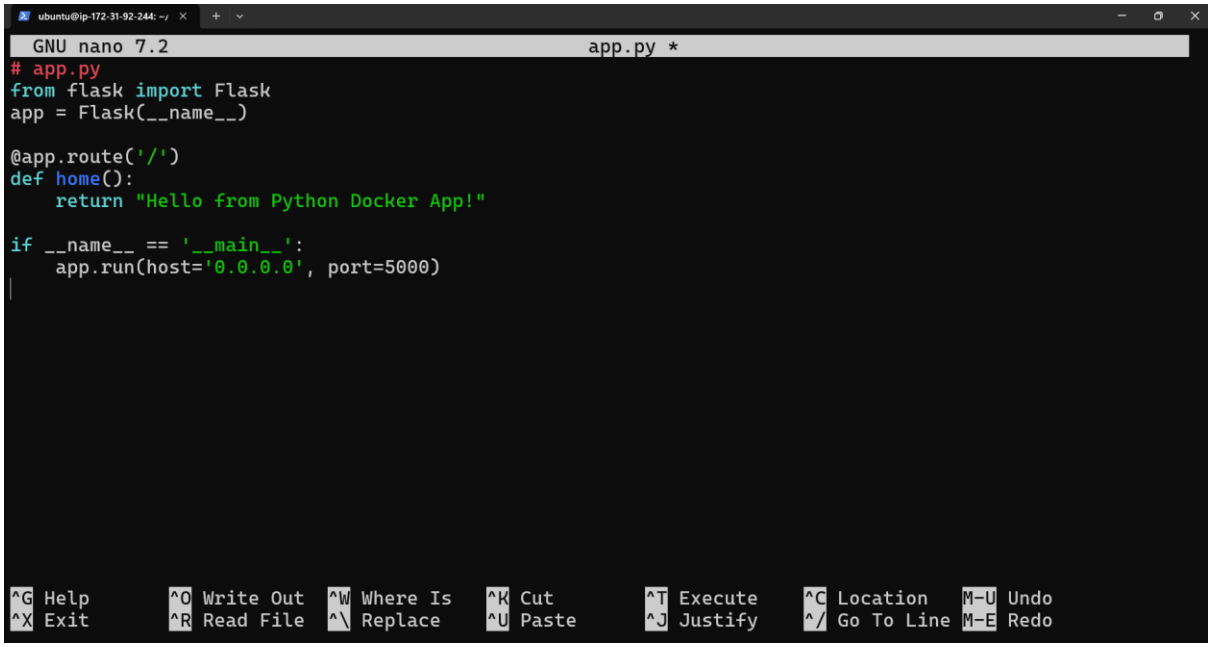
├── app.py

├── requirements.txt

└── Dockerfile



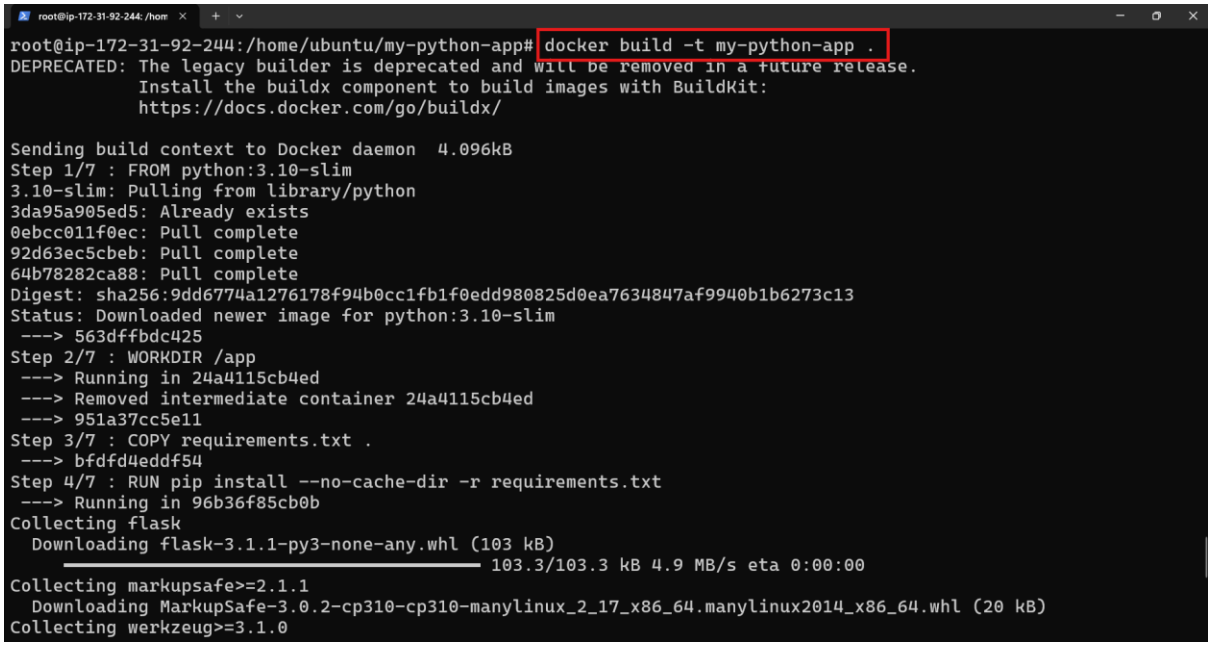
Step 2: Create Python App (app.py)



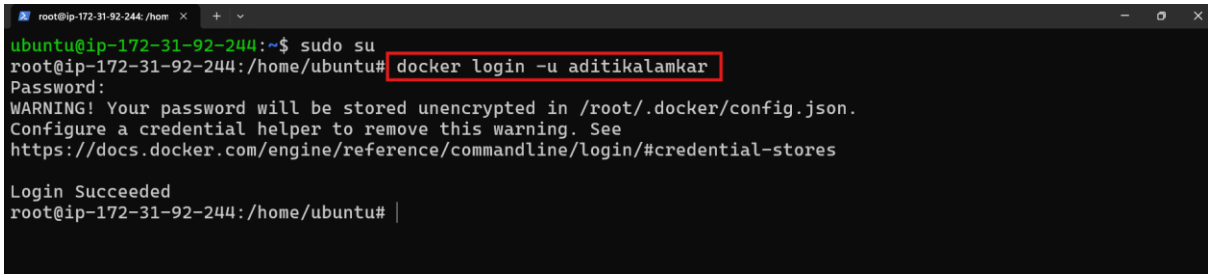
Step 3: Create requirements.txt

Step 4: Create Dockerfile

Step 5: Build Docker Image

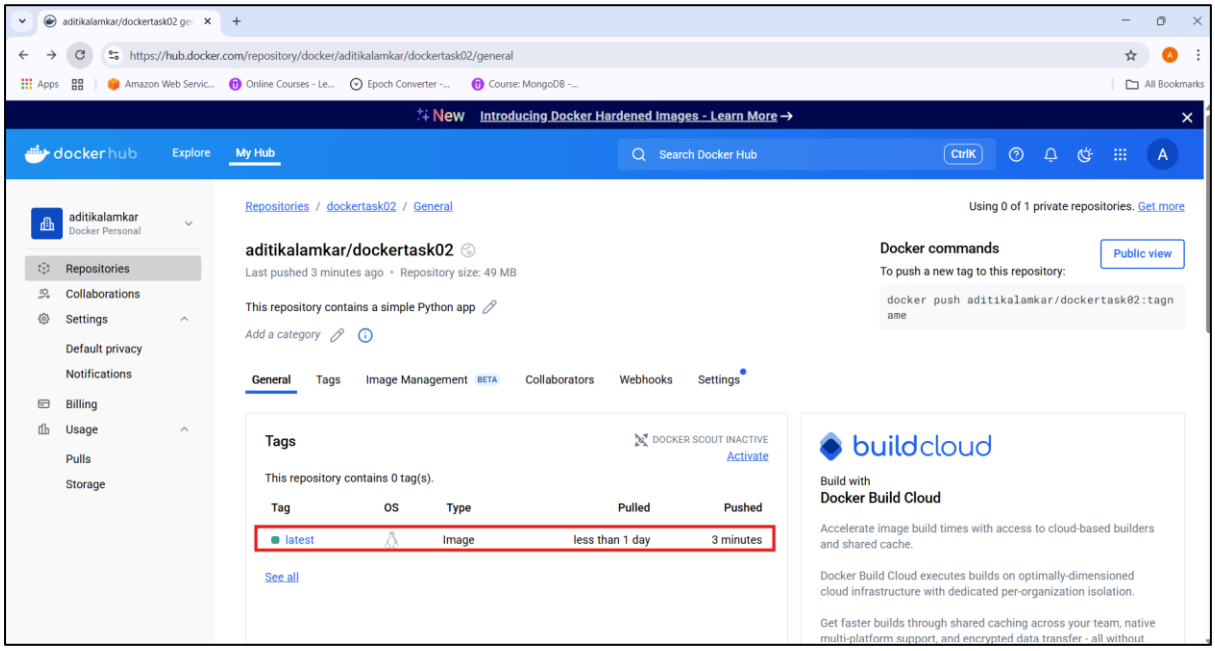


Step 6: Log In to Docker Hub



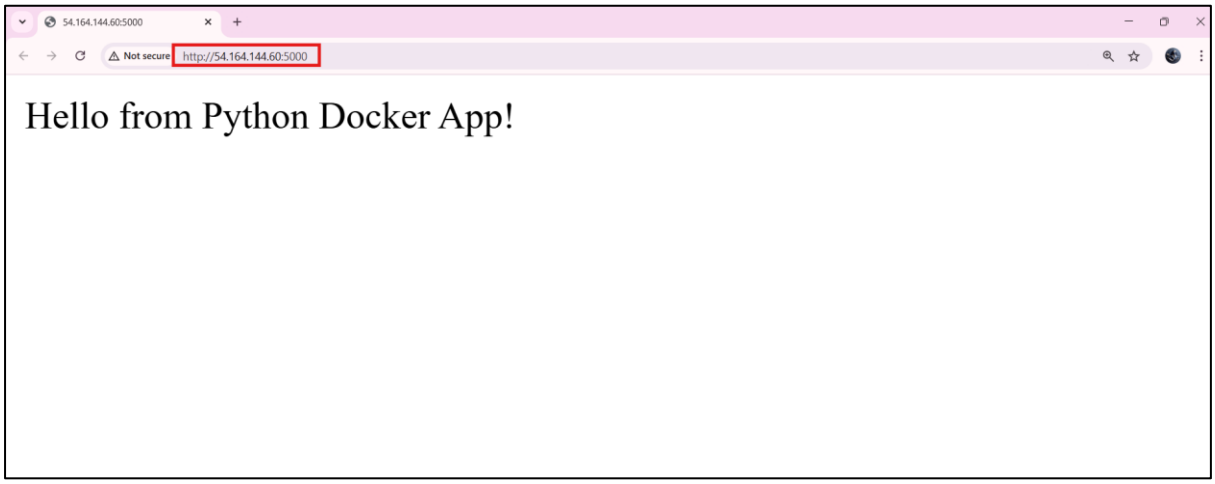
Step 7: Push the Image to Docker Hub

```
Upload an image to a registry
root@ip-172-31-92-244:/home/ubuntu# docker push aditikalamkar/dockertask02:latest
The push refers to repository [docker.io/aditikalamkar/dockertask02]
fb5f8849b975: Pushed
bbd4a4b99cb0: Pushed
e51b8b5c3cd7: Pushed
22fdf9cb1488: Pushed
31abbb538de0: Mounted from library/python
a2ccf7527b3c: Mounted from library/python
abd92ba2021e: Mounted from library/python
1bb35e8b4de1: Mounted from library/python
latest: digest: sha256:95d2c0fe30f460da287a7a8594b3d752e57304f6244b17872b4b0af9a603c9b1 size: 1990
root@ip-172-31-92-244:/home/ubuntu#
root@ip-172-31-92-244:/home/ubuntu#
```



Step 8: Run the Image

```
root@ip-172-31-92-244:/home/ubuntu# docker run -d -p 5000:5000 aditikalamkar/dockertask02:latest
e8959204392e472c7db345df39b7cd600c8e48fc40e86dde3a1df2759112e07f
root@ip-172-31-92-244:/home/ubuntu#
```



10. **Prepare a documentation in README.md with proper screenshots and push that file to your Git hub repository.**

https://github.com/aditikalamkar/FCTDevOpsLearning.git