Name: Kriti
College: Lovely Professional University
Week4 Assignment
Email: kritijulia@gmail.com

Q1. Introduction to containerization and Docker fundamentals, Basic Commands

Introduction to Docker

➔ Docker is a platform that allows developers to package applications and their dependencies into lightweight, portable containers. These containers ensure consistency across different environments, making application deployment seamless and efficient.

Introduction to Docker Hub

➔ Docker Hub is a cloud-based repository that allows users to store, share, and distribute Docker images. It acts as a central registry for pre-built container images that developers can pull and use.

Docker Architecture

➔ Docker follows a client-server architecture that consists of:

1. **Docker Client**: The interface where users interact with Docker.
2. **Docker Daemon (Server)**: Responsible for running and managing containers.
3. **Docker Image**: A lightweight, standalone package containing everything needed to run an application.
4. **Docker Containers**: Running instances of Docker images.
5. **Docker Registry**: Stores Docker images (e.g., Docker Hub).

Docker Life Cycle

1. Create an image (Dockerfile → Build → Image)
2. Push the image to Docker Hub
3. Pull the image and run a container
4. Modify a running container and commit changes
5. Stop and remove containers
6. Delete images when no longer needed

Concept of Hyper-V

➔ Hyper-V is a hypervisor technology by Microsoft that allows multiple virtual machines (VMs) to run on a single host. It is used in Docker for Windows to create lightweight virtual machines for container execution.

➔ **What is Docker?**

➔ **Docker** is a platform for developing, shipping, and running containers.

➔ Docker is an open-source tool which is used to containerize and run applications in isolated environments. It solves the issue of versions and installation multiple dependencies.

➔ It uses **Docker Engine**, which supports:

- Image building (via `Dockerfile`)

- Container execution (via `docker run`)
- Volume and network management

What is Containerization?

➔ **Containerization** is a lightweight form of virtualization.

➔ It allows you to **package applications** with all dependencies (libraries, configs) into a **container**.

➔ Containers share the host OS kernel but are isolated from each other.

Why Containers?

➔ Run **consistently** across environments (dev → test → prod).

➔ **Faster** than VMs (no full OS boot).

➔ Use **less resources** (share OS kernel).

➔ **Portable**, scalable, and reproducible.

Component	Description
Image	Read-only template (e.g. OS + app)
Container	Running instance of an image
Dockerfile	Script to build Docker images
DockerHub	Cloud repository to share images
Volume	Persistent storage for containers
Network	Connect containers securely

Basic Commands

IMAGES:

- List all Local images: docker images
- Delete an image: docker rmi <image name>
- Remove unused images: docker image prune
- Build an image from a Dockerfile: docker build -t <image name>:<version> . (version is optional)
- Build an image without cache: docker build -t <image name>:<version> . --no-cache

CONTAINER:

- List all Local containers (running & stopped): docker ps -a
- List all running containers: docker ps
- Create & run a new container: docker run <image name> (If the image is not available locally, it will be downloaded from DockerHub)
- Run container in background: docker run -d <image name>
- Run container with a custom name: docker run --name <container name> <image name>
- Port Binding in a container: docker run -p <host port>:<container port> <image name>
- Set environment variables in a container: docker run -e <var name>=<var value>
- Running an existing container in Docker is quite straightforward. Here are the steps you can follow:
- **List all containers**: First, you might want to see the list of all containers, including those that are stopped. You can do this with the following command: docker ps -a
- **Start the container**: To start an existing container, use the docker start command followed by the container ID or name. For example:

docker start <container_id_or_name>

- **Attach to the container (optional)**: If you want to interact with the container after starting it, you can attach to it using:
  docker attach <container_id_or_name>
- **Run the container in the background**: If you prefer to run the container in the background (detached mode), you can use:
  docker start -d <container_id_or_name>
- **Execute a command in the running container (optional)**: If you need to run a specific command inside the running container, you can use:
  docker exec -it <container_id_or_name> <command>
  For example, to open a bash shell inside the container:
  docker exec -it <container_id_or_name> /bin/bash
  These commands should help you manage and run your existing Docker containers efficiently. If you have any specific needs or run into issues, feel free to ask!

DOCKER HUB:

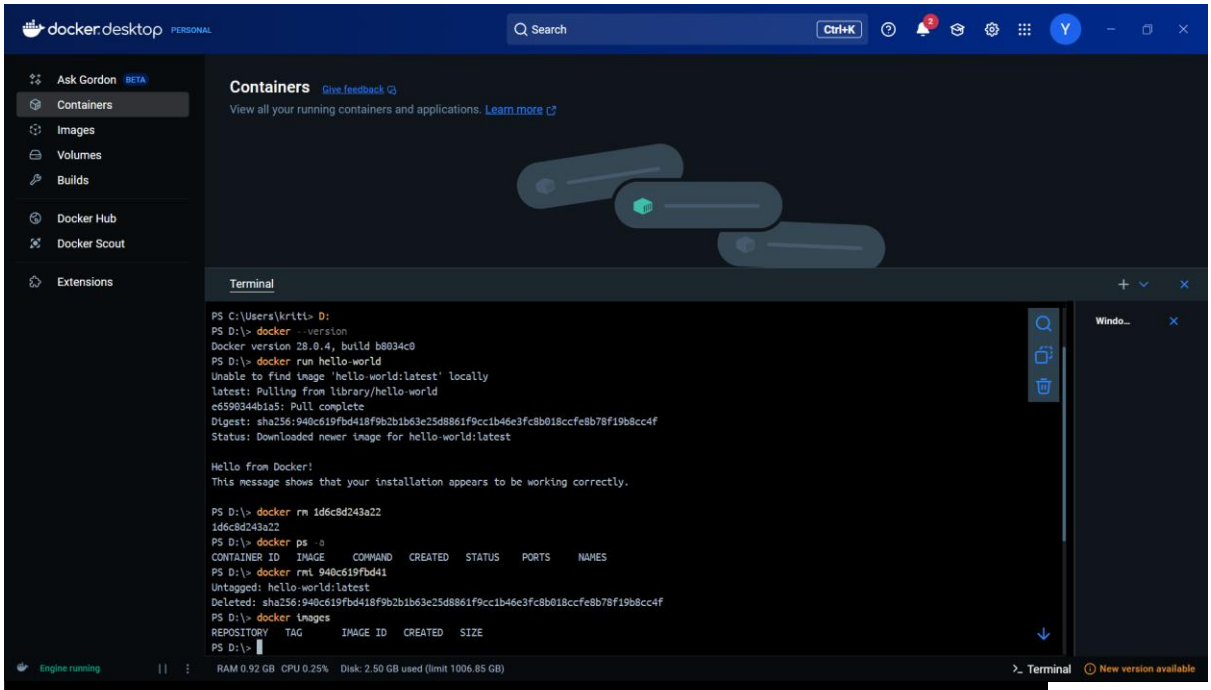
- Pull an image from DockerHub: docker pull <image name>
- Publish an image to DockerHub: docker push <image name>
- Login to DockerHub: docker login -u <username> or docker login
- Logout from DockerHub: docker logout
- Search for an image on DockerHub: docker search <image name>

VOLUMES:

- List all volumes: docker volume ls
- Create a new named volume: docker volume create <volume name>
- Delete a named volume: docker volume rm <volume name>
- Mount a named volume with a running container: docker run --volume <volume name>:<container path>
- Using --mount option: docker run --mount type=volume,src=<volume name>,dest=<container path>
- Mount an anonymous volume with a running container: docker run --volume <container path>
- Create a bind mount: docker run --mount type=bind,src=<host path>,dest=<container path>
- Remove unused local volumes: docker volume prune (for anonymous volumes)

NETWORK:

- List all networks: docker network ls
- Create a network: docker network create <network name>
- Remove a network: docker network rm <network name>
- Remove all unused networks: docker network prune




```
PS D:\> docker pull ubuntu
Using default tag: latest
latest: Pulling from library/ubuntu
d9d352c11bbd: Pull complete
Digest: sha256:b59d21599a2b151e23eea5f6602f4af4d7d31c4e236d22bf0b62b86d2e386b8f
Status: Downloaded newer image for ubuntu:latest
docker.io/library/ubuntu:latest
PS D:\> docker run -it ubuntu bash
root@1490ebebb68b:/# ls
bin  boot  dev  etc  home  lib  lib64  media  mnt  opt  proc  root  run  sbin  srv  sys  tmp  usr  var
root@1490ebebb68b:/# exit
exit
PS D:\>
```

Q2. Docker installation and basic container operations, Build an image from Dockerfile

Installing Docker on Windows/Linux

- **Windows:** Install Docker Desktop from the official Docker website and enable Hyper-V.
- **Linux:** Install using package managers (e.g., apt for Ubuntu, yum for CentOS).

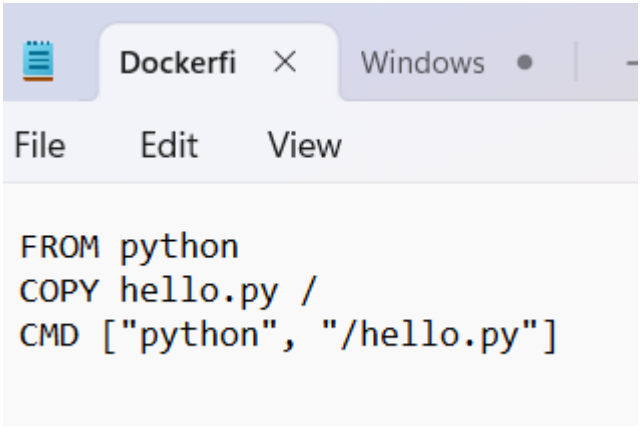
What is Dockerfile?

➔ Dockerfile is a simple text file with instructions to build image Automation of Docker Image Creation.

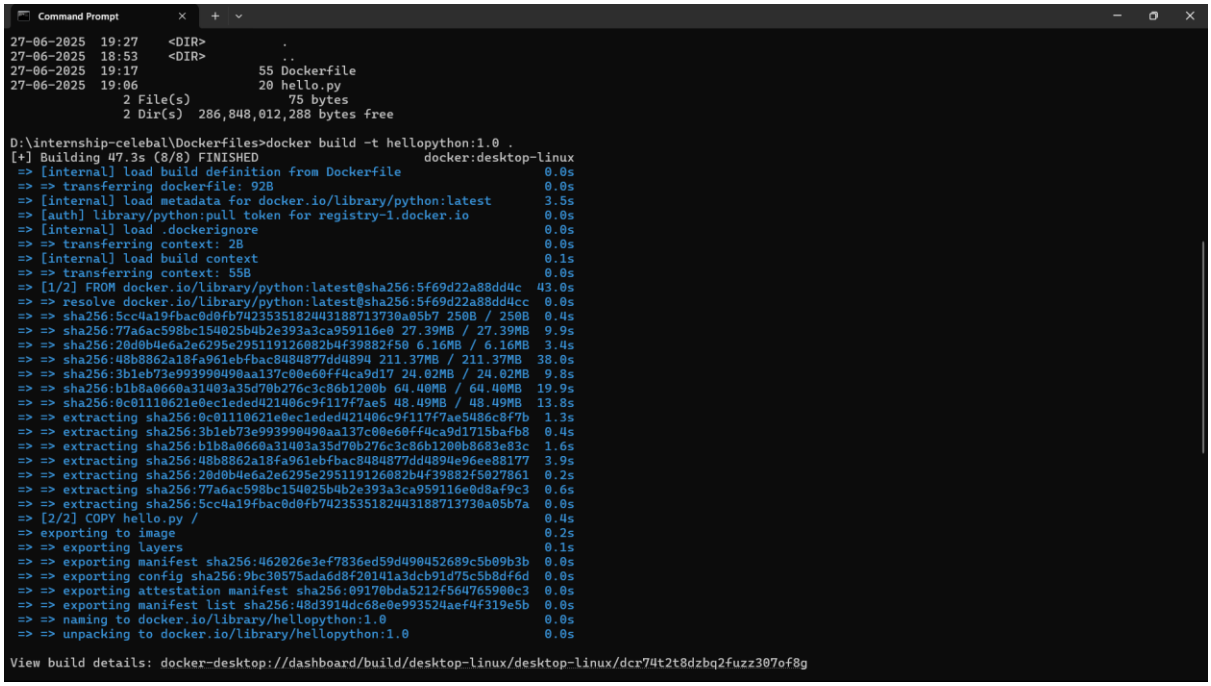
➔ **Dockerfile contains some instructions like:**

- FROM: Sets the base image (e.g., FROM ubuntu:20.04)
- RUN: Executes shell commands during build time
- CMD: Provides default command to run at container start
- ENTRYPOINT: Defines a fixed command, often combined with CMD for args
- COPY/ADD: Copies files into the image (ADD supports URLs and archives)
- ENV: Sets environment variables
- EXPOSE: Documents which ports the container will listen on
- WORKDIR: Sets the working directory for following instructions

- VOLUME: Declares mount points for persistent or shared data
- ARG: Defines build-time variables, usable during RUN




```
Command Prompt

D:\internship-celebal>mkdir Dockerfiles

D:\internship-celebal>cd Dockerfiles

D:\internship-celebal\Dockerfiles>notepad Dockerfile

D:\internship-celebal\Dockerfiles>notepad Dockerfile

D:\internship-celebal\Dockerfiles>notepad hello.py

D:\internship-celebal\Dockerfiles>docker build -t hellopython:1.0 .
[+] Building 0.1s (1/1) FINISHED                          docker:desktop-linux
 => [internal] load build definition from Dockerfile                      0.0s
 => => transferring dockerfile: 2B                                        0.0s
ERROR: failed to solve: failed to read dockerfile: open Dockerfile: no such file or directory

View build details: docker-desktop://dashboard/build/desktop-linux/desktop-linux/tbpw65d18lb1ezmqw0z3w1mzv

D:\internship-celebal\Dockerfiles>dir
 Volume in drive D is New Volume
 Volume Serial Number is 3E17-594E

 Directory of D:\internship-celebal\Dockerfiles

27-06-2025  19:06    <DIR>          .
27-06-2025  18:53    <DIR>          ..
27-06-2025  19:17                55 Dockerfile.txt
27-06-2025  19:06                20 hello.py
               2 File(s)             75 bytes
               2 Dir(s)  286,848,012,288 bytes free

D:\internship-celebal\Dockerfiles>rename Dockerfile.txt Dockerfile

D:\internship-celebal\Dockerfiles>dir
 Volume in drive D is New Volume
 Volume Serial Number is 3E17-594E

 Directory of D:\internship-celebal\Dockerfiles

27-06-2025  19:27    <DIR>          .
27-06-2025  18:53    <DIR>          ..
27-06-2025  19:17                55 Dockerfile
27-06-2025  19:06                20 hello.py
```


Command Prompt
27-06-2025 19:27 <DIR> .
27-06-2025 18:53 <DIR> ..
27-06-2025 19:17 55 Dockerfile
27-06-2025 19:06 20 hello.py
2 File(s) 75 bytes
2 Dir(s) 286,848,012,288 bytes free
D:\internship-celebal\Dockerfiles>docker build -t hellopython:1.0 .
[+] Building 47.3s (8/8) FINISHED docker:desktop-linux
=> [internal] load build definition from Dockerfile 0.0s
=> => transferring dockerfile: 92B 0.0s
=> [internal] load metadata for docker.io/library/python:latest 3.5s
=> [auth] library/python:pull token for registry-1.docker.io 0.0s
=> [internal] load .dockerignore 0.0s
=> => transferring context: 2B 0.0s
=> [internal] load build context 0.1s
=> => transferring context: 55B 0.0s
=> [1/2] FROM docker.io/library/python:latest@sha256:5f69d22a88dd4c 43.0s
=> => resolve docker.io/library/python:latest@sha256:5f69d22a88dd4cc 0.0s
=> => sha256:5cc4a19fbac0d0fb74235351824431887137 30a05b7 250B / 250B 0.4s
=> => sha256:77a6ac598bc154025b4b2e393a3ca959116e0 27.39MB / 27.39MB 9.9s
=> => sha256:20d0b4e6a2e6295e295119126082b4f39882f50 6.16MB / 6.16MB 3.4s
=> => sha256:48b8862a18fa961ebfbac8484877dd4894 211.37MB / 211.37MB 38.0s
=> => sha256:3b1eb73e993990490aa137c00e60ff4ca9d17 24.02MB / 24.02MB 9.8s
=> => sha256:b1b8a0660a31403a35d70b276c3c86b1200b 64.40MB / 64.40MB 19.9s
=> => sha256:0c01110621e0ec1eded421406c9f117f7ae5 48.49MB / 48.49MB 13.8s
=> => extracting sha256:0c01110621e0ec1eded421406c9f117f7ae5486c8f7b 1.3s
=> => extracting sha256:3b1eb73e993990490aa137c00e60ff4ca9d1715bafb8 0.4s
=> => extracting sha256:b1b8a0660a31403a35d70b276c3c86b1200b8683e83c 1.6s
=> => extracting sha256:48b8862a18fa961ebfbac8484877dd4894e96ee88177 3.9s
=> => extracting sha256:20d0b4e6a2e6295e295119126082b4f39882f5027861 0.2s
=> => extracting sha256:77a6ac598bc154025b4b2e393a3ca959116e0d8af9c3 0.6s
=> => extracting sha256:5cc4a19fbac0d0fb74235351824431887137 30a05b7a 0.0s
=> [2/2] COPY hello.py / 0.4s
=> exporting to image 0.2s
=> => exporting layers 0.1s
=> => exporting manifest sha256:462026e3ef7836ed59d490452689c5b09b3b 0.0s
=> => exporting config sha256:9bc30575ada6d8f20141a3dcb91d75c5b8df6d 0.0s
=> => exporting attestation manifest sha256:09170bda5212f564765900c3 0.0s
=> => exporting manifest list sha256:48d3914dc68e0e993524aef4f319e5b 0.0s
=> => naming to docker.io/library/hellopython:1.0 0.0s
=> => unpacking to docker.io/library/hellopython:1.0 0.0s
View build details: docker-desktop://dashboard/build/desktop-linux/desktop-linux/dcr74t2t8dzbq2fuzz307of8g

Q3. Docker Registry, DockerHub, Create a Multi-Stage Build

What is a Docker Registry?

➔ A Docker Registry is a storage and distribution system for Docker images. It allows developers to push (upload) and pull (download) container images.

Public vs Private Registries

Type	Examples
Public	**DockerHub, GitHub Container Registry**
Private	**Azure Container Registry (ACR), Amazon ECR, Harbor**

What is DockerHub?

- Official Docker registry (https://hub.docker.com)
- Provides:
  - Prebuilt official images (e.g., nginx, node, python)
  - Personal or team repositories
  - Public or private image storage

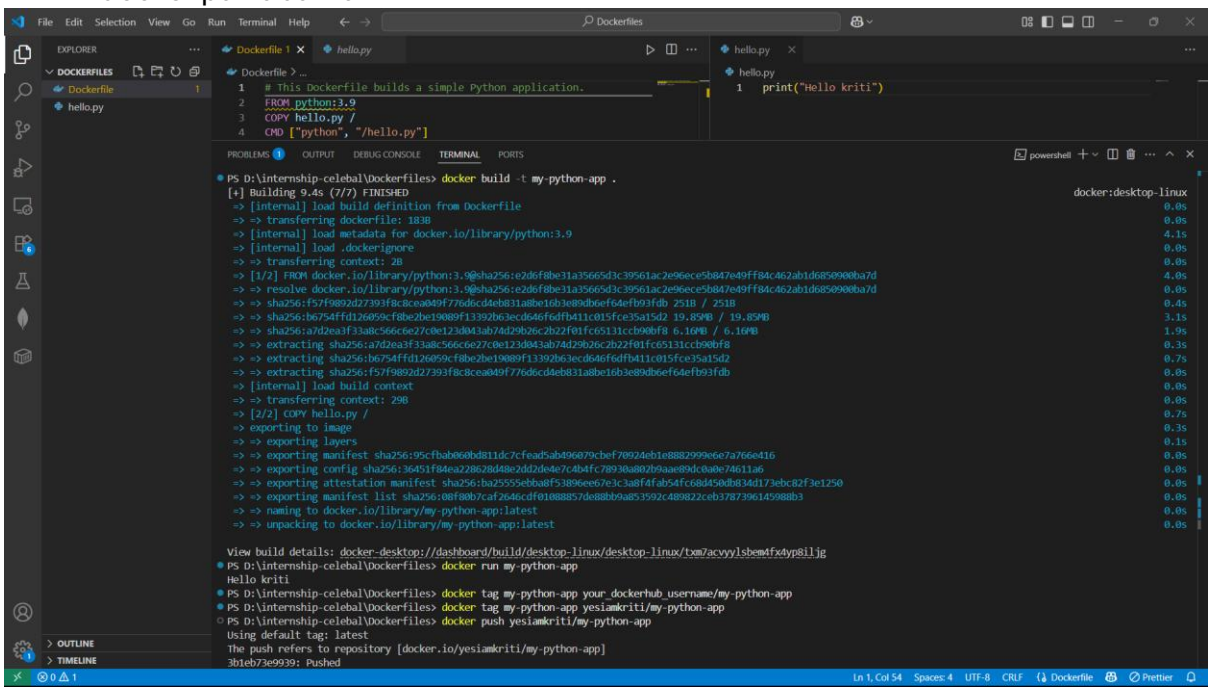
Common DockerHub Commands

➔ docker login

➔ Tag your local image to match DockerHub format
docker tag my-image your_dockerhub_username/my-image

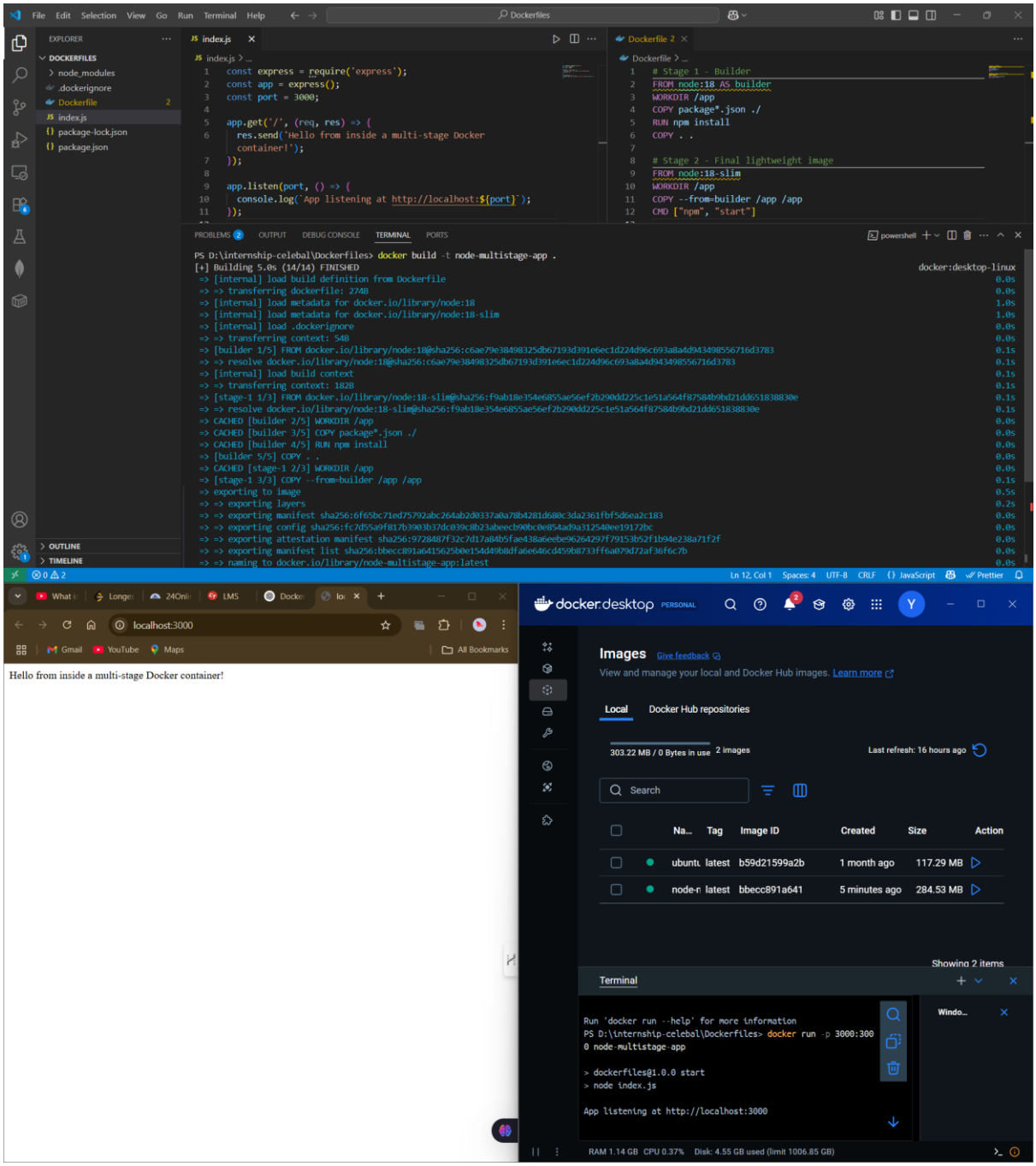
➔ Push it to DockerHub
docker push your_dockerhub_username/my-image

➔ docker pull ubuntu



What is Multi-Stage Build?

- A way to use multiple FROM statements in a Dockerfile.
- Allows separating build environment from final runtime image.
- Greatly reduces final image size and attack surface.
- It is used to avoid shipping build tools (e.g., compilers) in the final image.
- Keep images minimal and clean.



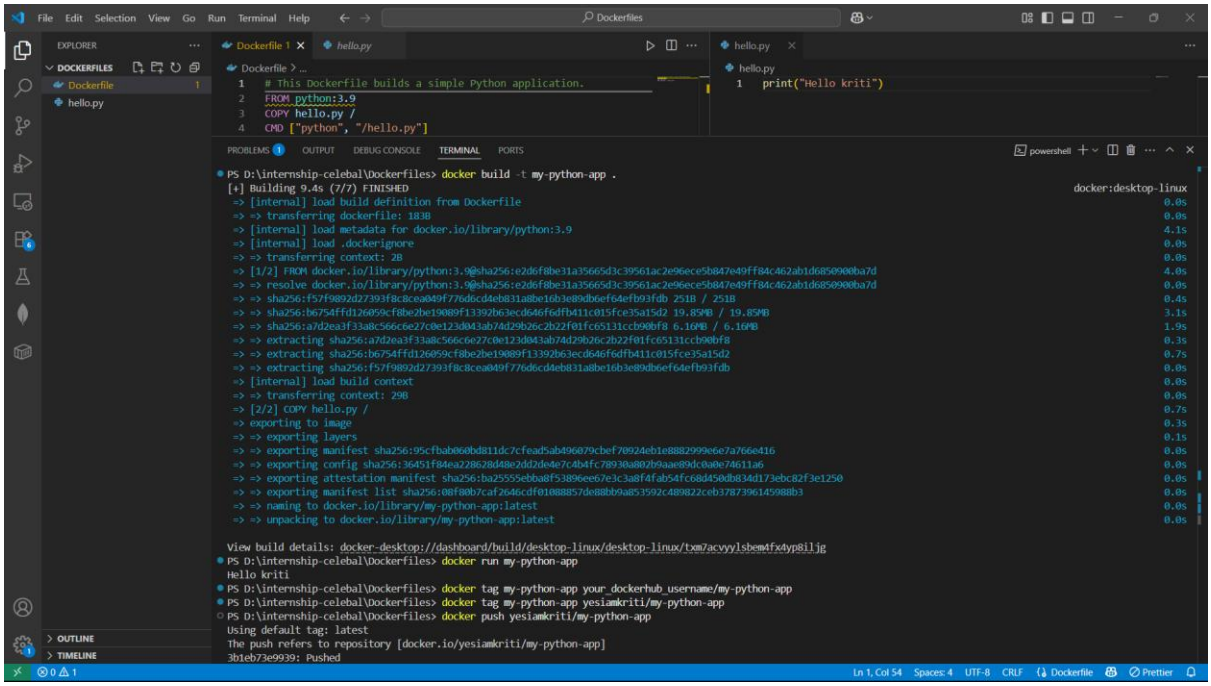
Q4. Create a docker image from multiple methods likes Dockerfile, running containers.

Method 1: Create Docker Image from a Dockerfile : it will automate and version-control our image setup.

Create a docker file for an application

Build the image using “docker build .” command

Run the image to make container using “docker run <image name>”

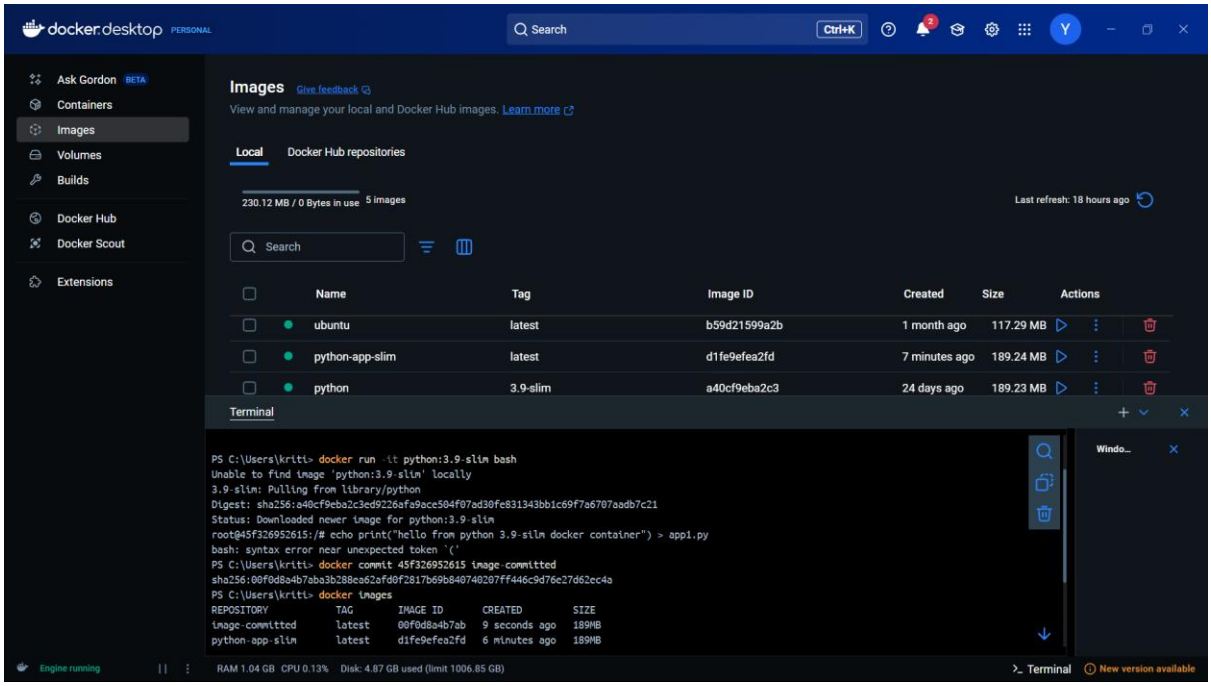


Method 2: Create Docker Image from Running Container

➔ Used for testing/debugging interactively and want to save the container's state as a new image.

- **Start a container interactively** using “docker run -it python:3.9 bash”
- **Inside the container**, create a file: echo 'print("Hello from committed image!")' > hello.py
- Then “exit”
- **Find the container ID**: docker ps -a
- **Commit the container to a new image**: docker commit <container_id> image-committed
- **Run the new image**: docker run image-committed python hello.py

Method	**Dockerfile**	**Commit from Container**
Reproducible	✅ Yes	❌ No, manual state only
Version-controlled	✅ Yes (via Git)	❌ No
Good for Production	✅ Recommended	❌ Avoid
Use case	App deployment, CI/CD	One-off manual experiments



Q5. Push and pull image to Docker hub and ACR

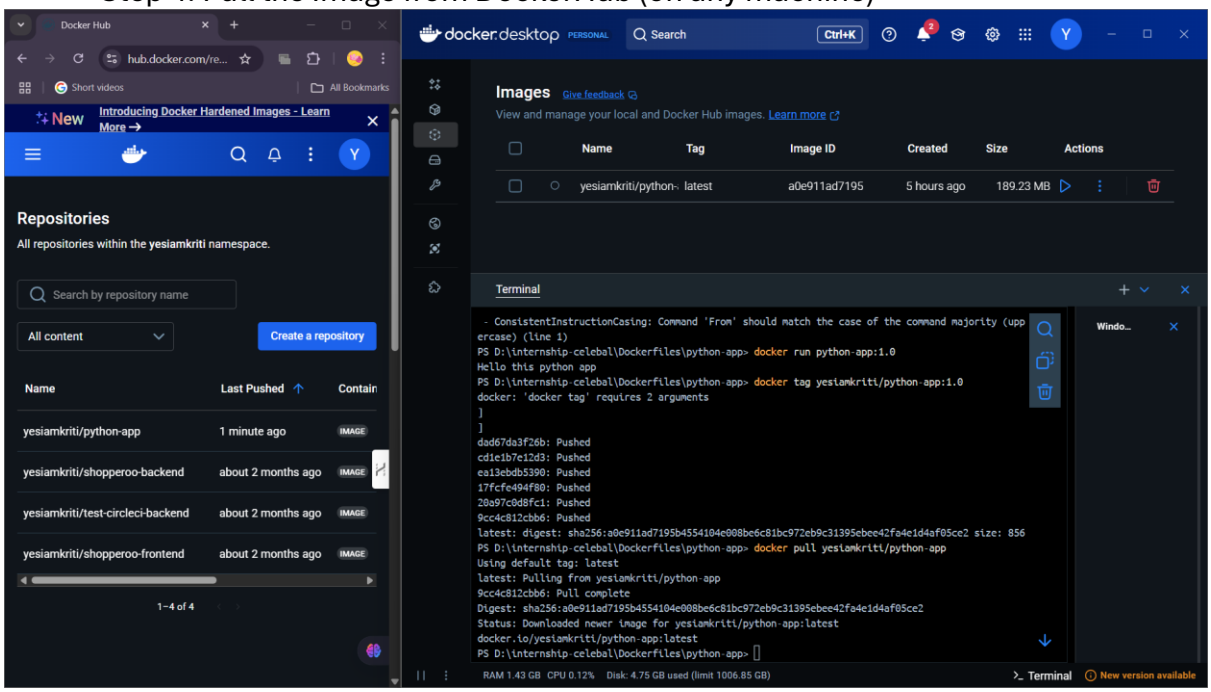
Part 1: Push & Pull Docker Image to DockerHub

Step 1: Login to DockerHub : docker login

Step 2: Tag the Image

Step 3: Push the Image to DockerHub

Step 4: Pull the Image from DockerHub (on any machine)



Part 2: Push & Pull Docker Image to Azure Container Registry (ACR)

Setup ACR

Step 1: Login to Azure CLI : az login

Step 2: Create ACR: az acr create --name myacrregistry --resource-group myResourceGroup --sku Basic

Push Image to ACR

Login to ACR: az acr login --name myacrregistry

Step 3: Tag the Docker Image: docker tag myapp myacrregistry.azurecr.io/myapp:v1

Step 4: Push the Image: docker push myacrregistry.azurecr.io/myapp:v1

Pull the Image from ACR : docker pull myacrregistry.azurecr.io/myapp:v1

Q6. Create a Custom Docker Bridge Network

Why Custom Bridge Network?

- By default, Docker uses a bridge network (e.g., bridge, host, none).
- Custom bridge networks allow:
    - Container-to-container communication by name
    - Isolation from other apps
    - Easier debugging and security

Step-by-Step: Create and Use Custom Docker Bridge Network

Step 1: Create a Custom Bridge Network

➔ docker network create my_network

Step 2: Verify the Network

➔ docker network ls

Step 3: Run Containers in the Custom Network

➔ docker run -dit --name container1 --network my_network alpine sh

➔ docker run -dit --name container2 --network my_network alpine sh

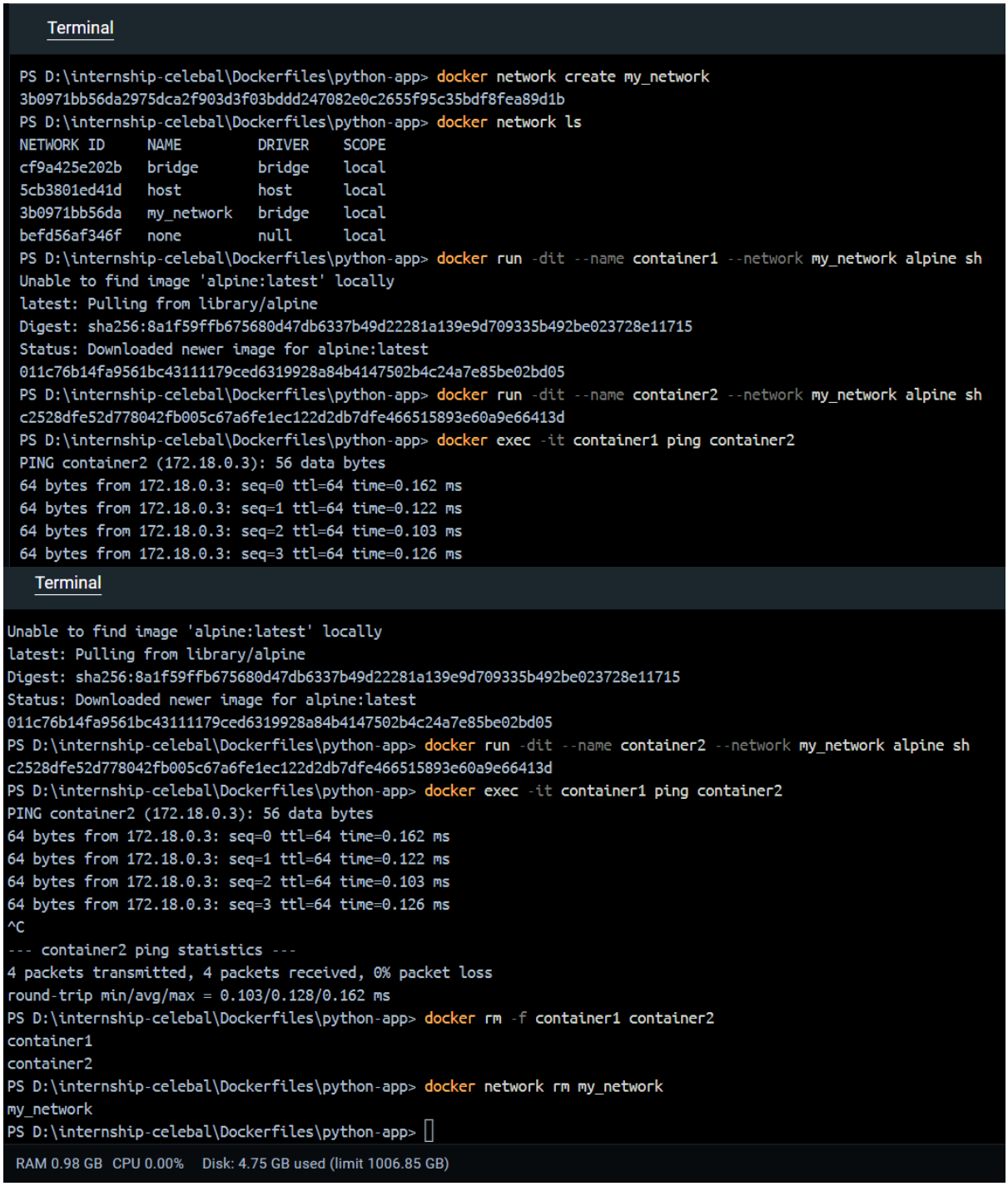
Step 4: Test Communication by Container Name

➔ docker exec -it container1 ping container2

Step 5: Clean Up

➔ docker rm -f container1 container2

➔ docker network rm my_network



Q7. Create a Docker volume and mount it to a container.

What is a Docker Volume?

- A Docker volume is a persistent storage mechanism.
- Data in a volume:
  - Persists even if the container is removed.
  - Is stored outside the container layer, inside /var/lib/docker/volumes/.

Step-by-Step: Create and Mount a Docker Volume

Step 1: Create a Volume

➔ docker volume create my_data_volume

➔ docker volume ls

Step 2: Mount the Volume to a Container

→ docker run -dit \
 --name my_volume_container \
 -v my_data_volume:/data \
 alpine sh

Step 3: Store Some Data in the Volume

➔ docker exec -it my_volume_container sh

Inside the container:

➔ echo "Hello from volume!" > /data/hello.txt

➔ exit

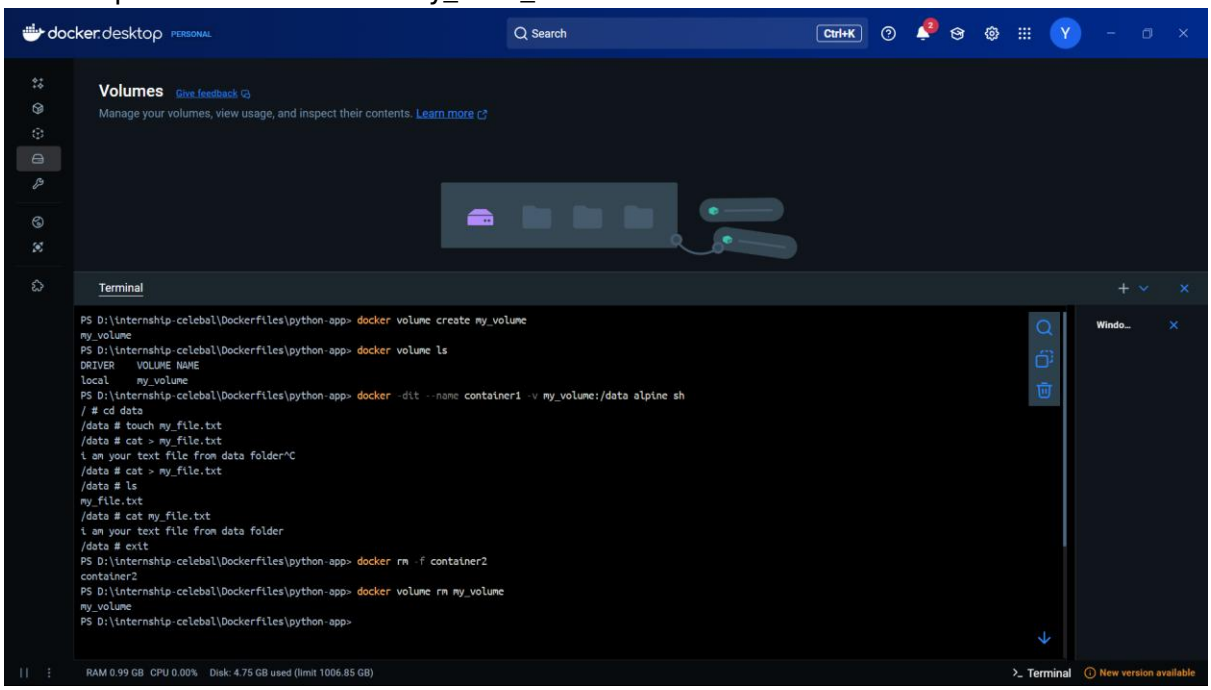
Step 4: Remove the Container (Volume Data Persists!)

➔ docker rm -f my_volume_container

Step 5: Reuse the Volume in Another Container

➔ docker run --rm -v my_data_volume:/data alpine cat /data/hello.txt

Clean Up: docker volume rm my_data_volume



Q8. Docker Compose for multi-container applications, Docker security best practices

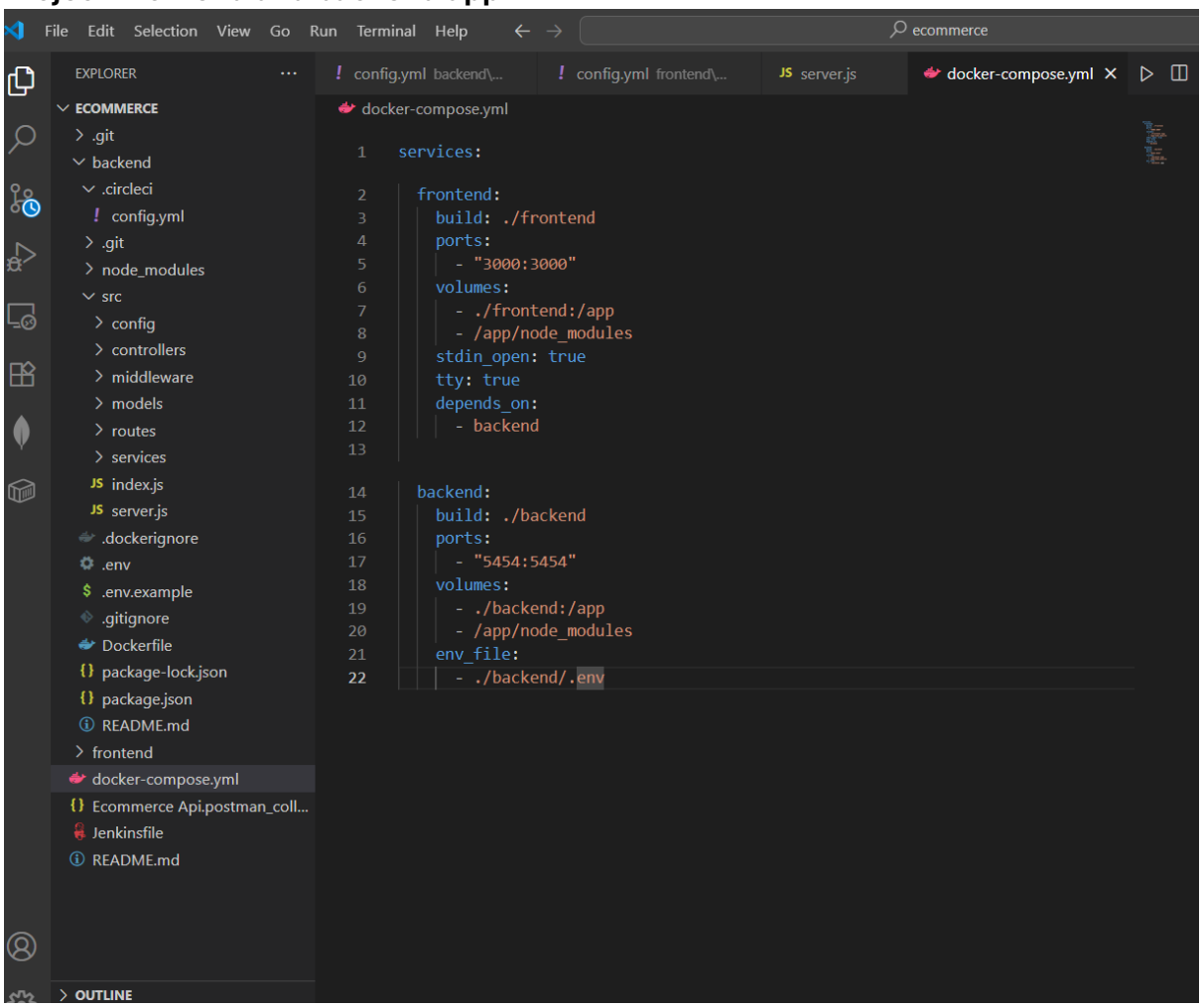
PART 1: Docker Compose for Multi-Container Applications

What is Docker Compose?

Docker Compose lets you define and run multi-container applications using a single YAML file.

- **Benefits:**
  - **Manage multi-service apps with one command**
  - **Define volumes, networks, environment variables, and build steps**
  - **Portable, versionable setup**

Project: frontend and backend app



PART 2: Docker Security Best Practices

Why Security in Docker Matters?

Containers isolate apps, but misconfiguration can expose your host system, secrets, or networks.

Secure dockerfile

```
FROM node:18-slim
WORKDIR /app
COPY . .
RUN npm ci
USER node
CMD ["node", "app.js"]
```