**Docker Lab Exercises**

**Exercise 1: Getting Started with Containers**

**Commands Used:**

docker run hello-world

docker run --name webserver -d -p 8080:80 nginx

docker run -it alpine sh

docker ps

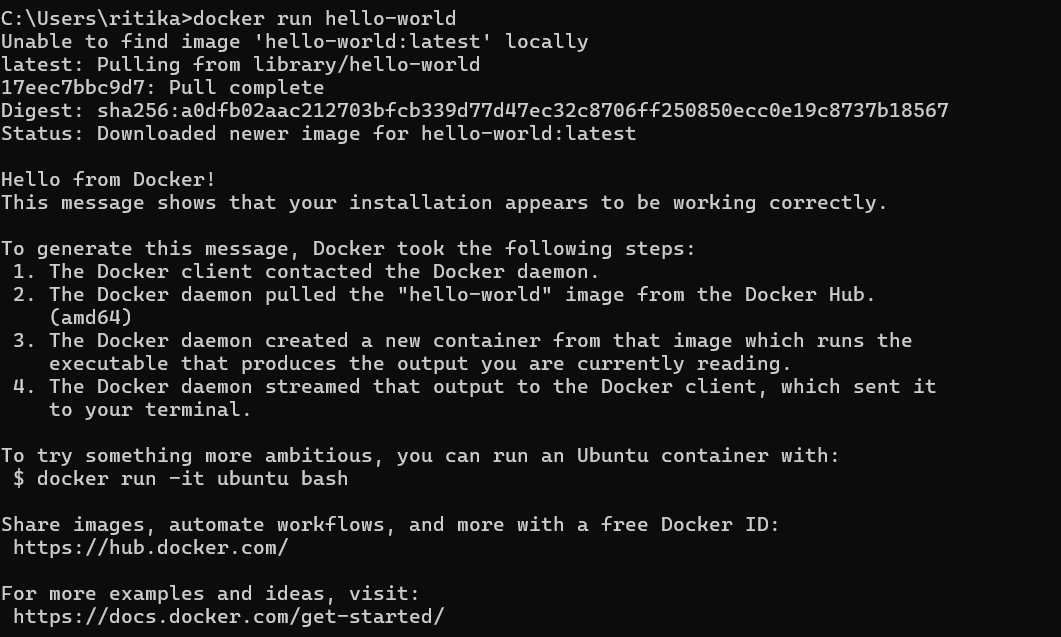
docker ps -a

docker exec -it 941524b369ee sh

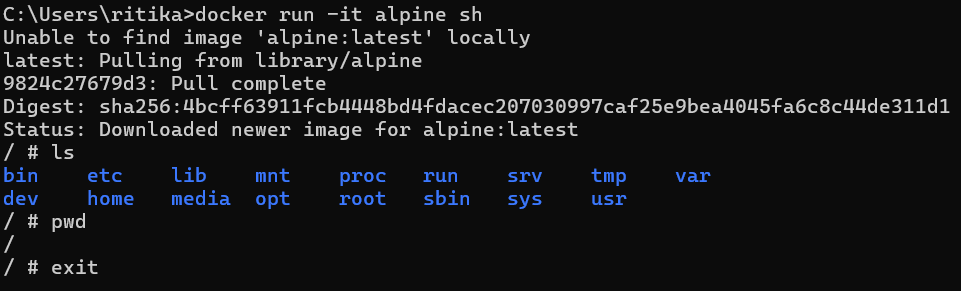
docker logs 941524b369ee

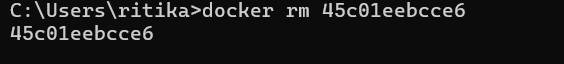
**● Run hello-world, nginx, and alpine containers.**

**● Explore docker run flags: --rm, -it, -d, -p.**



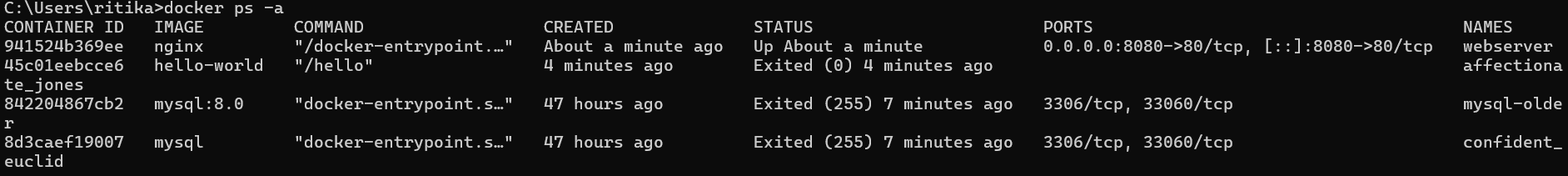




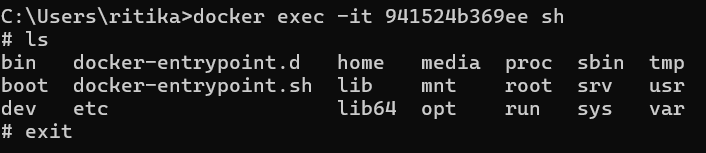


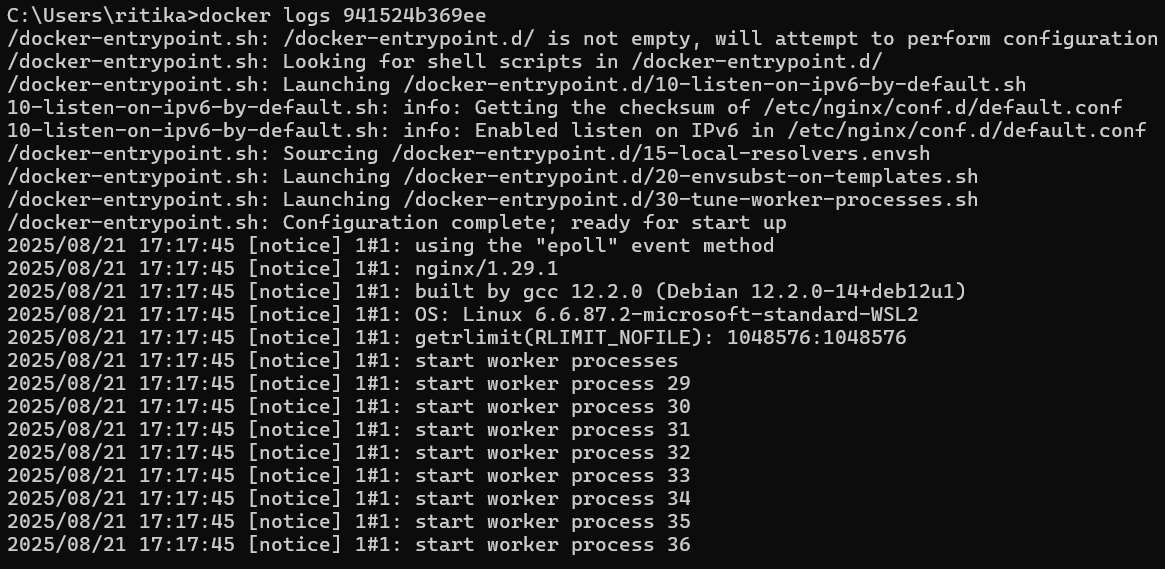
**● Use docker ps and docker ps -a to inspect states.**





**● Use docker exec and docker logs.**





What happens if a container doesn't run in detached mode? What if ports aren't mapped?

If a container doesn’t run in detached mode (-d), it will run in the foreground and occupy your terminal until stopped. If ports aren’tmapped (-p), services inside the container can’t be accessed from the host machine, even though they’re running inside the container.

**Exercise 2: Working with Container State**

**Commands Used:**

docker run -it ubuntu sh

apt-get update

apt-get install -y curl vim

exit

docker commit 6d759b1f83ab ubuntu-tools

docker run -it ubuntu-tools sh

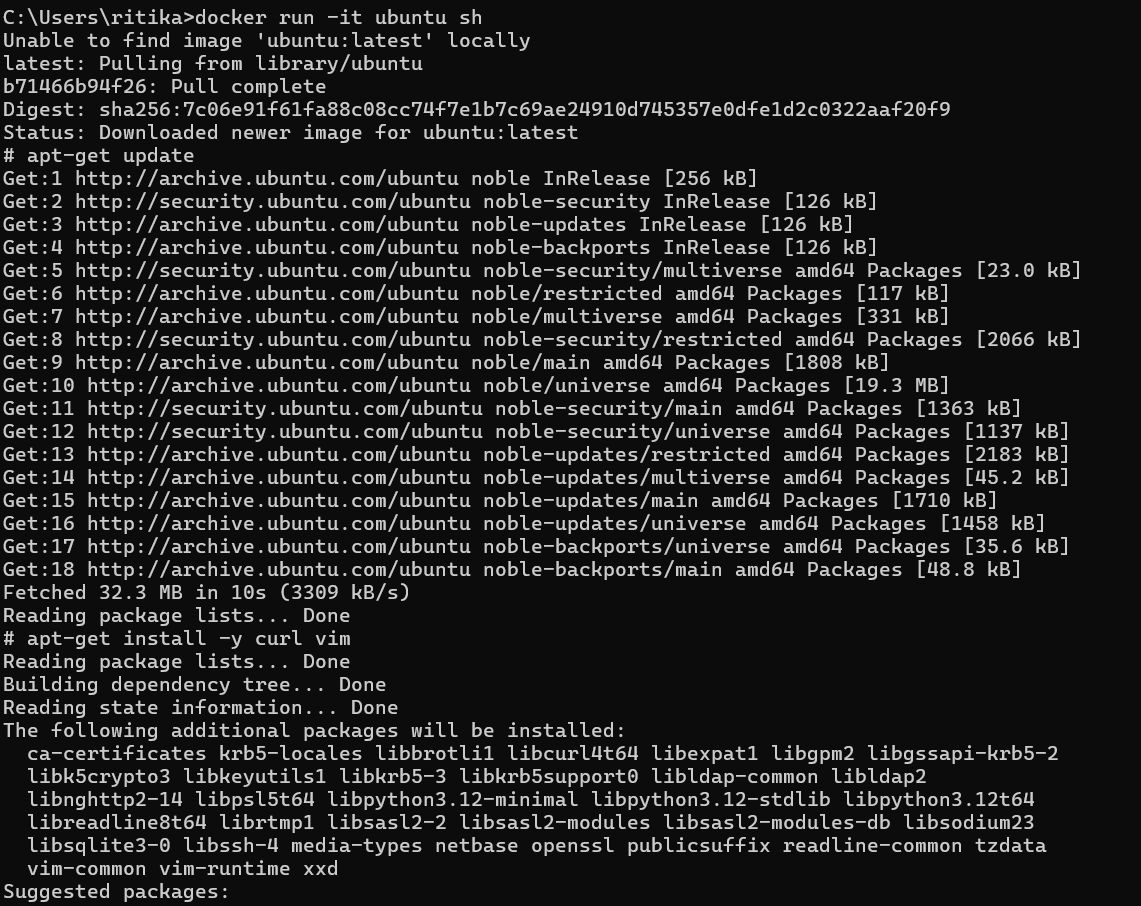
curl --version

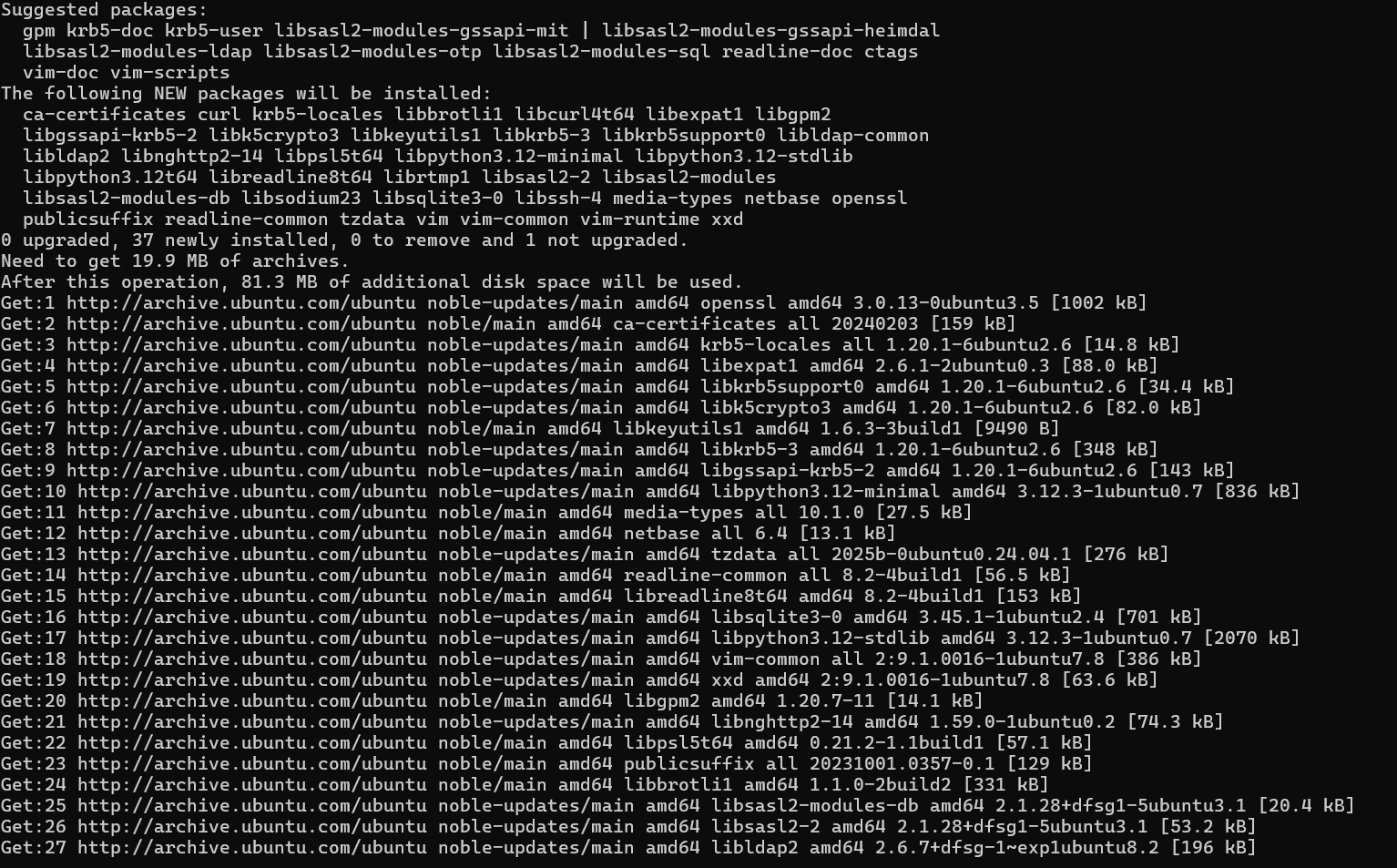
vim –version

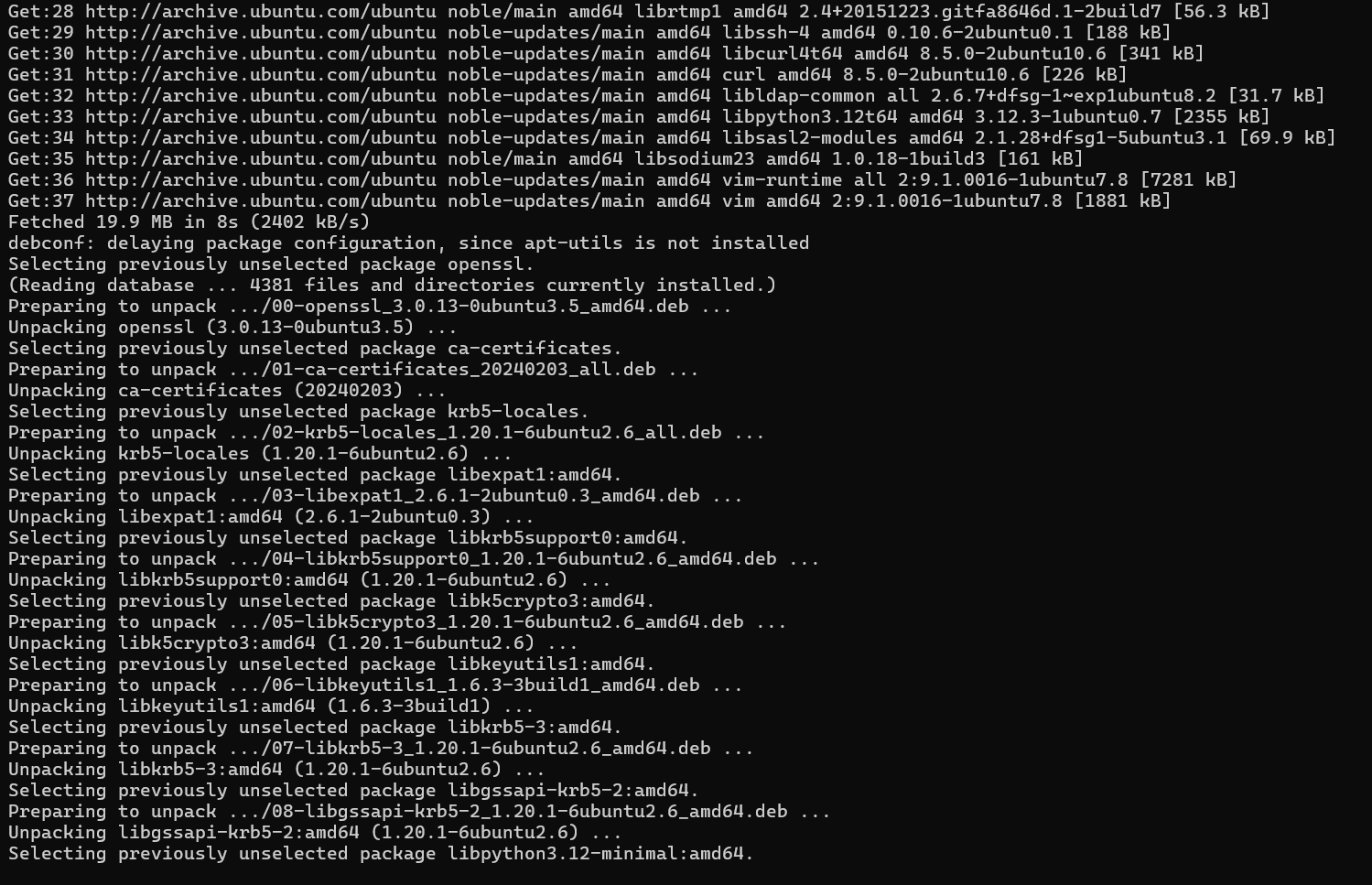
docker tag ubuntu-tools ritika/ubuntu-tools:v1

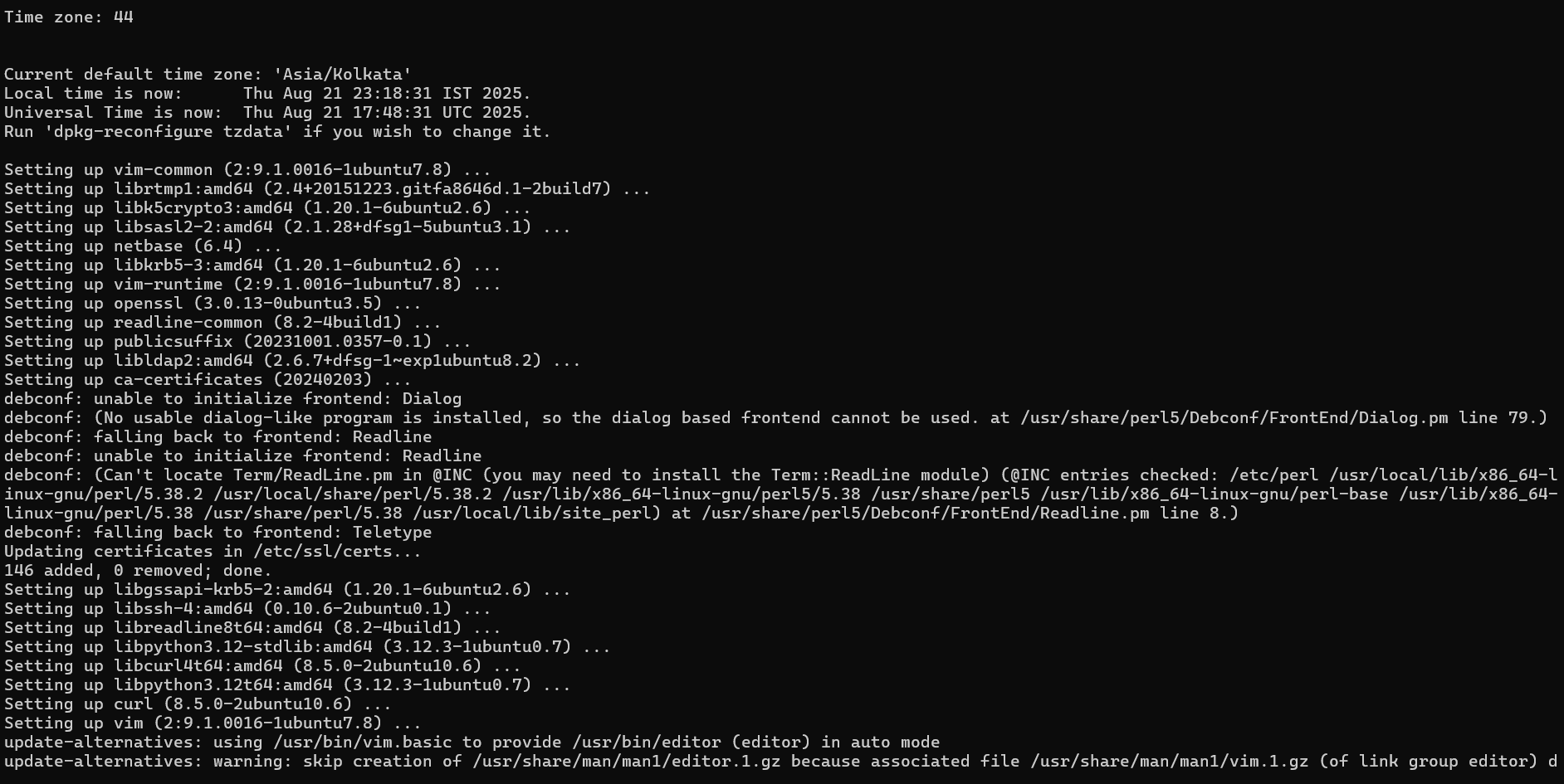
docker images

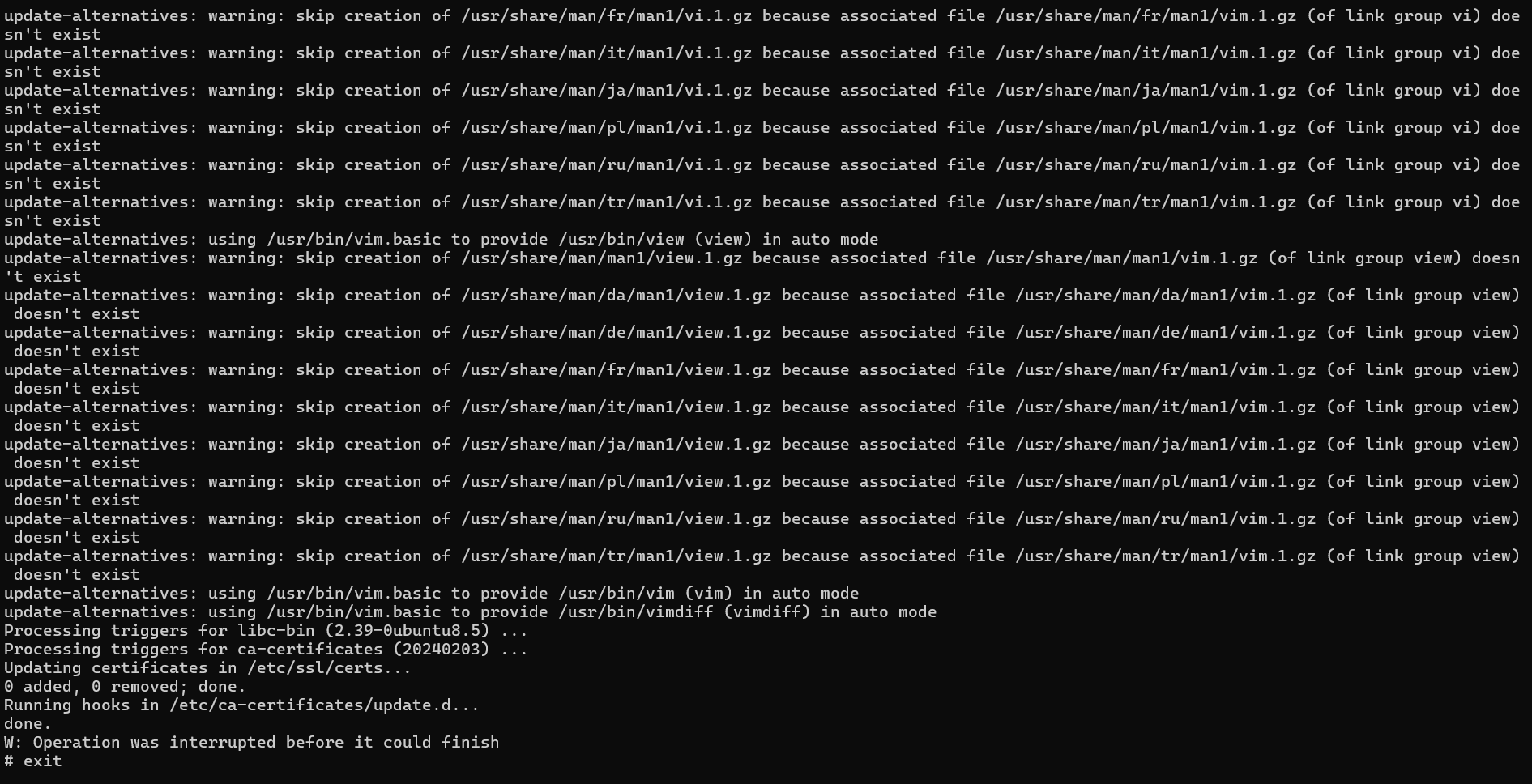
**● Run an Ubuntu container, install curl and vim.**



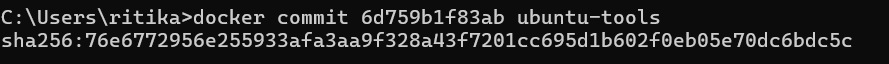




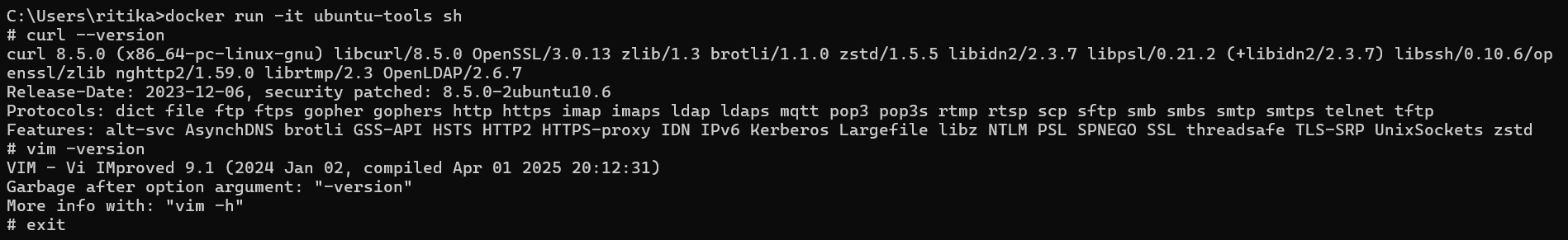




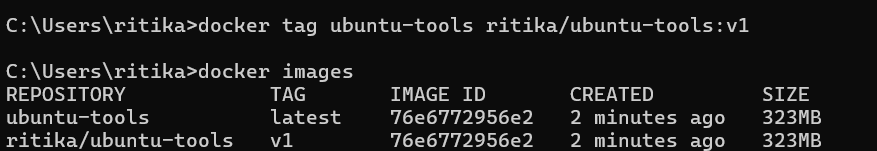
**● Exit and commit the image as ubuntu-tools.**

****

**● Run a new container from the committed image.**

****

**● Tag the image and list it with docker images.**

****

**Exercise 3: Build Custom Images Using Dockerfile**

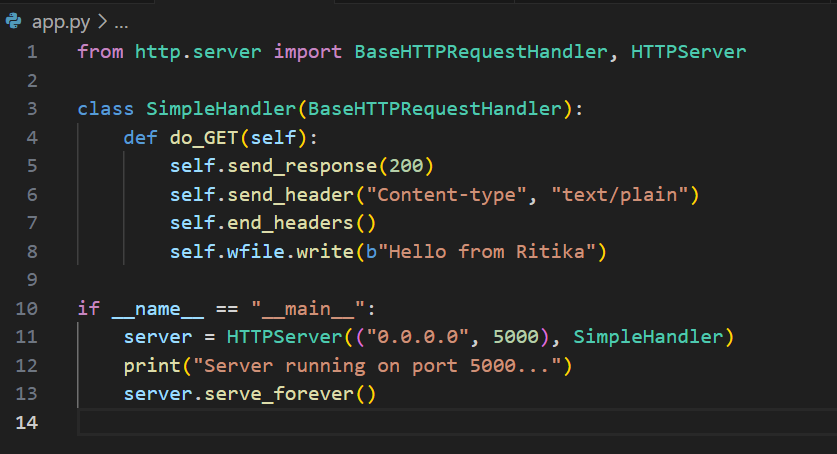
**Commands Used:**

docker build -t python-webserver .

docker run -d -p 8080:5000 python-webserver

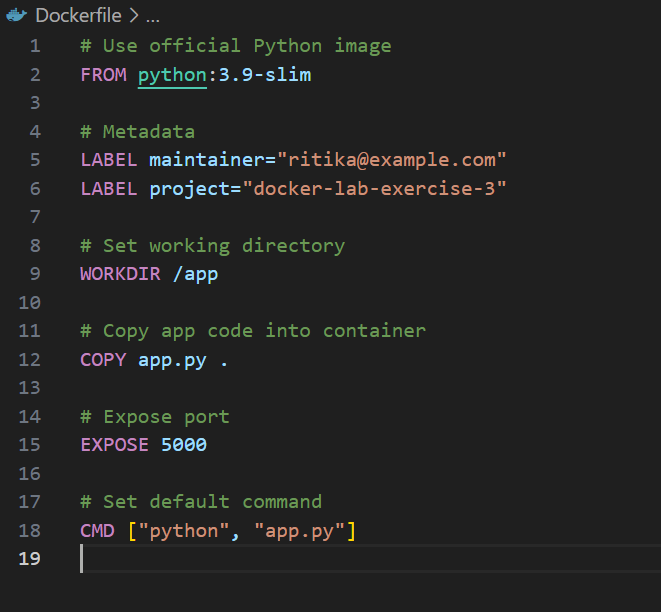
curl http://localhost:8080

**● Create a simple Node or Python web server.**

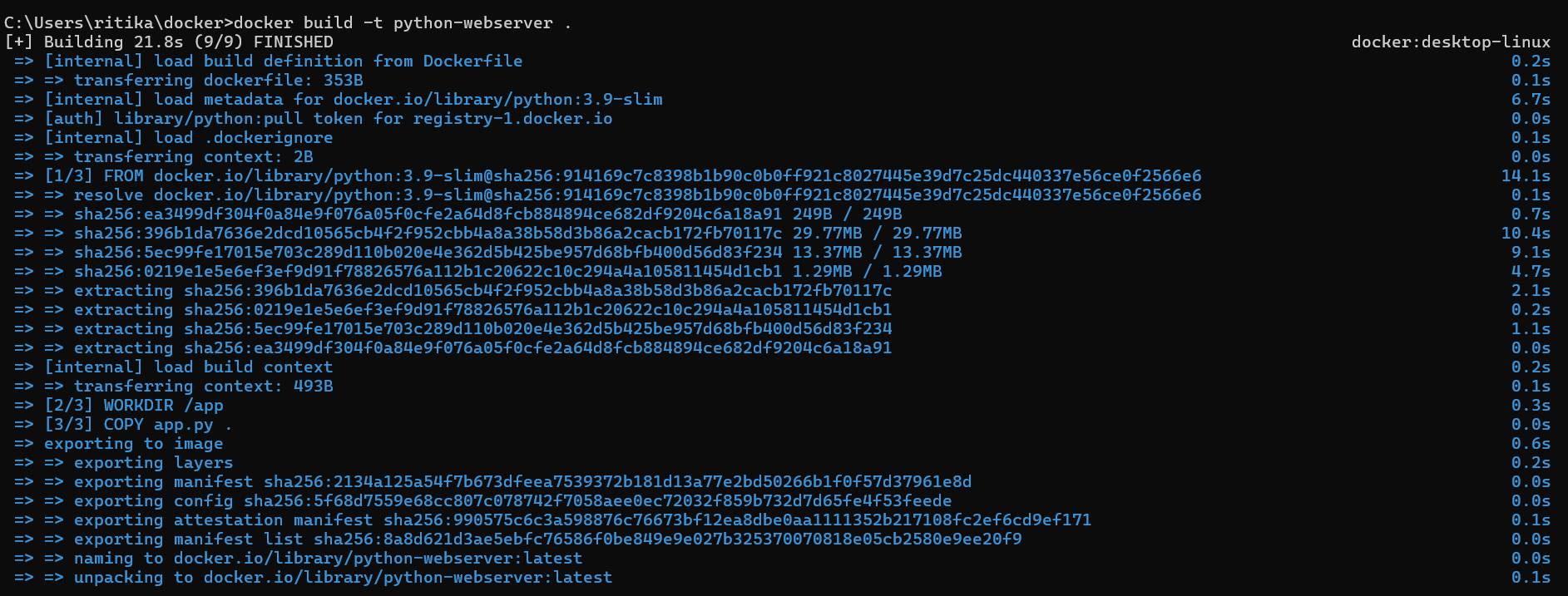


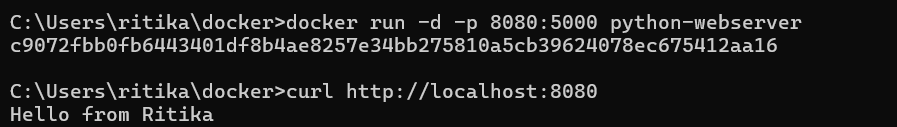
**● Write a Dockerfile to copy the code and expose a port.**

**● Add metadata using LABEL and set CMD or ENTRYPOINT.**



**● Build and run the image. Test with curl.**

****

****

**Exercise 4: Sharing Images**

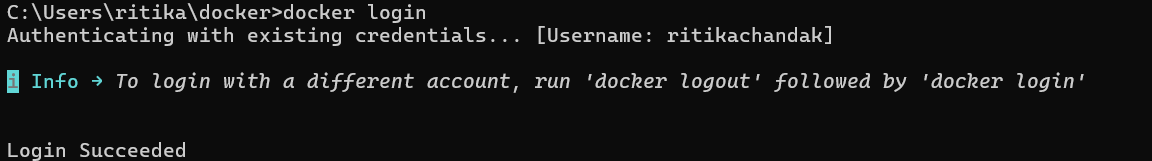
**Commands Used:**

docker login

docker tag python-webserver ritikachandak/python-webserver:v1

docker push ritikachandak/python-webserver:v1

**● Create a Docker Hub account.**

****

**● Tag your custom image.**



**● Push it to Docker Hub.**



**● Pull it from Docker Hub.**

****

Why is image tagging important, and what sort of tagging strategy can we use?

Tagging allows us to manage different versions of the same image (e.g., v1, v2, latest). It ensures consistency when sharing or deploying applications.  
A good strategy is to use both latest (for recent build) and semantic versioning (v1.0, v1.1). This way production environments can stick to stable versions, while development teams can test newer builds without confusion.

**Exercise 5: Data Persistence with Volumes**

**Commands Used:**

docker run -it --name testbox -v myvolume:/data busybox sh

echo "Hello Docker Volume" > /data/hello.txt

ls /data

cat /data/hello.txt

exit

docker rm testbox

docker run -it --name testbox2 -v myvolume:/data busybox sh

ls /data

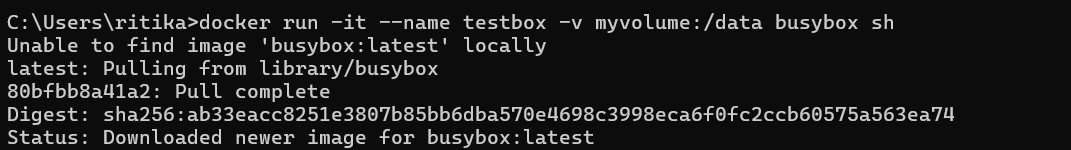
cat /data/hello.txt

mkdir C:\Users\ritika\docker\data

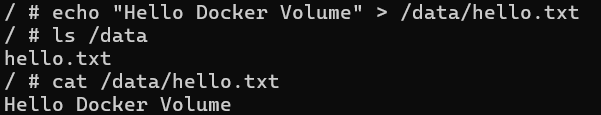
docker run -it -v C:\Users\ritika\docker\data:/data busybox sh

echo "Bind mount example" > /data/test.txt

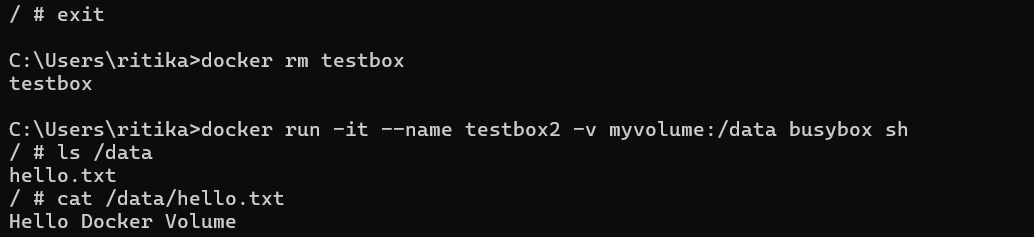
**● Launch a busybox container with a named volume.**



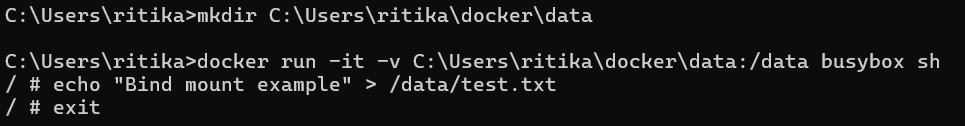
● **Insert sample data.**



● **Stop, remove, and relaunch to verify persistence**.



● **Try bind mount using -v $(pwd)/data:/data.**



Compare: Named volume vs bind mount. Pros/cons?

**Named Volumes -**Managed entirely by Docker (/var/lib/docker/volumes/...). Great for production when you don’t care about the exact host path. Easy to back up and migrate.

**Bind Mounts-**Map directly to a host directory. Useful for development (editing code locally and seeing changes inside container).But can cause permission and path issues and depends on host OS.

**Exercise 6: Container Networking Basics**

**Command Used:**

docker network create mynetwork

docker run -d --name webserver --network mynetwork nginx

docker run -it --name testbox --network mynetwork busybox sh

wget -qO- http://webserver

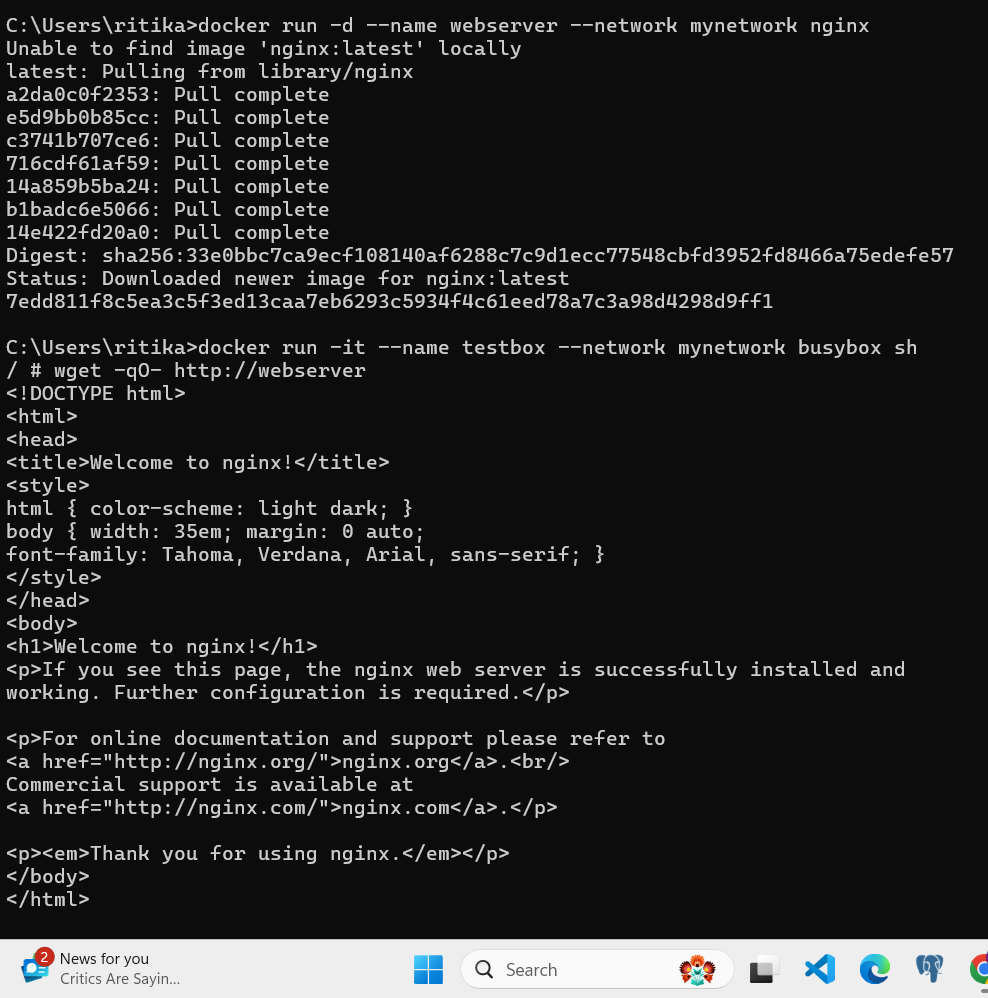
● **Create a user-defined bridge network.**

****

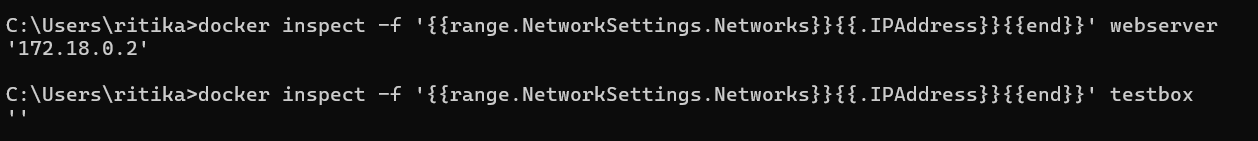
● **Start an nginx container and a busybox container.**

**● Attach both containers to the network.**

**● From busybox, use wget or curl to access nginx.**



View IPs with docker inspect. Try without a custom network—what’s different?





**Without custom network**

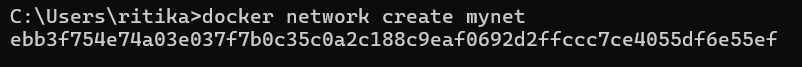
**Commands used:**

docker network create mynet

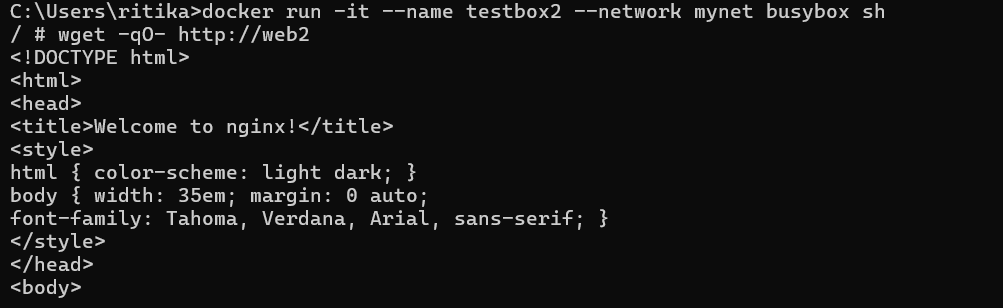
docker run -d --name web2 --network mynet nginx

docker run -it --name testbox2 --network mynet busybox sh

wget -qO- http://web2







**User-defined bridge network**: Containers can resolve each other by name (DNS), making communication easy.

**Default bridge network**: Requires using IP addresses (no built-in DNS), which makes it harder and less reliable because IPs may change.

**Exercise 7: Building a Two-Tier App**

**Commands Used:**

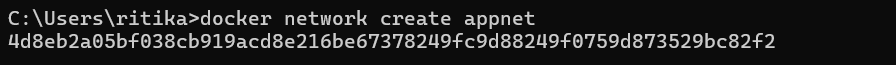
docker network create appnet

docker run -d --name mydb --network appnet -e POSTGRES\_USER=ritika -POSTGRES\_PASSWORD=pass123 -e POSTGRES\_DB=myappdb -v pgdata:/var/lib/postgresql/data postgres:13

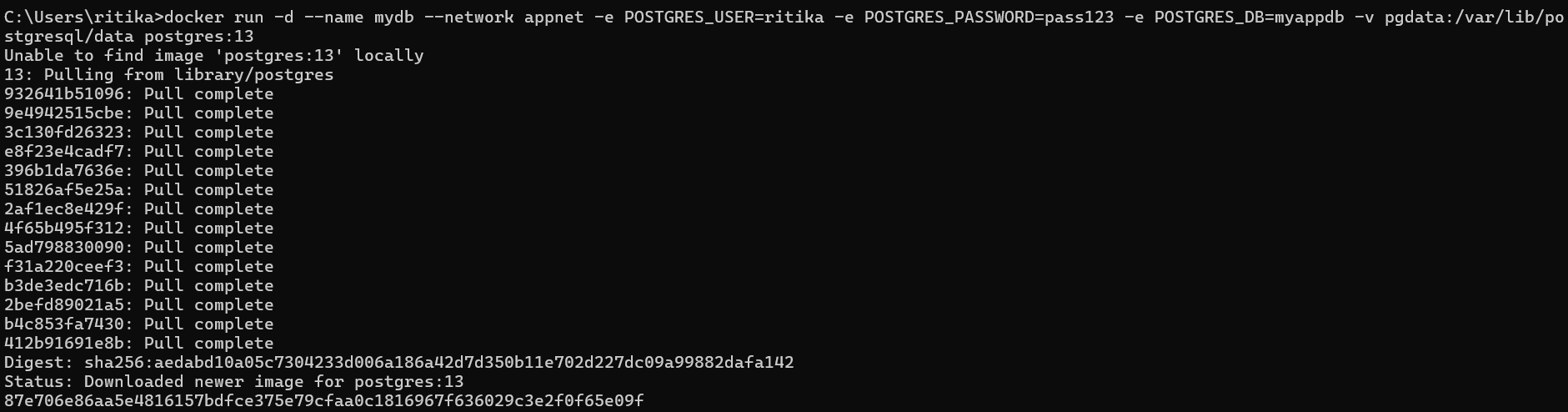
docker build -t flask-app .

docker run -d --name webapp --network appnet -p 5000:5000 flask-app

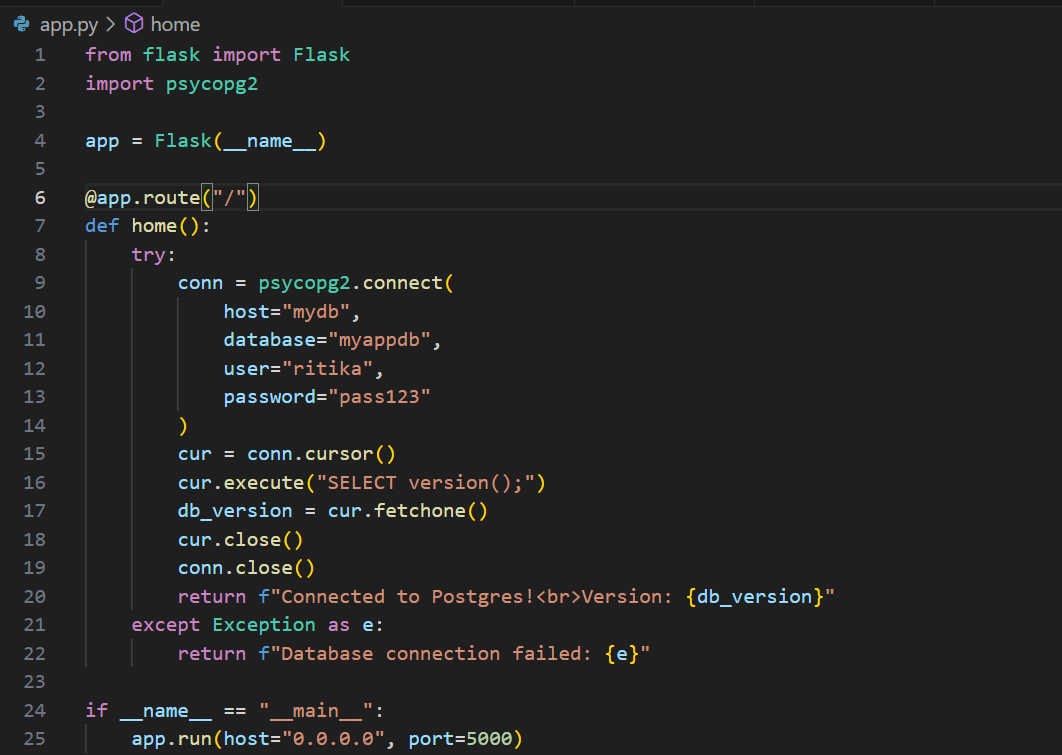
* **Create a Docker network**

****

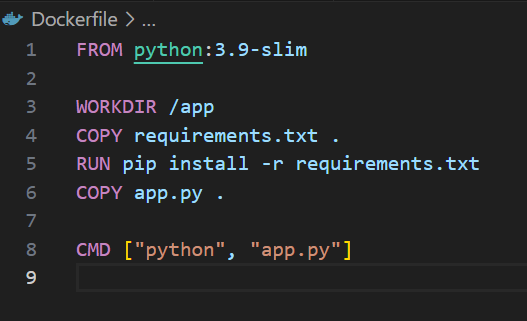
* **Run Postgres container with volume**

****

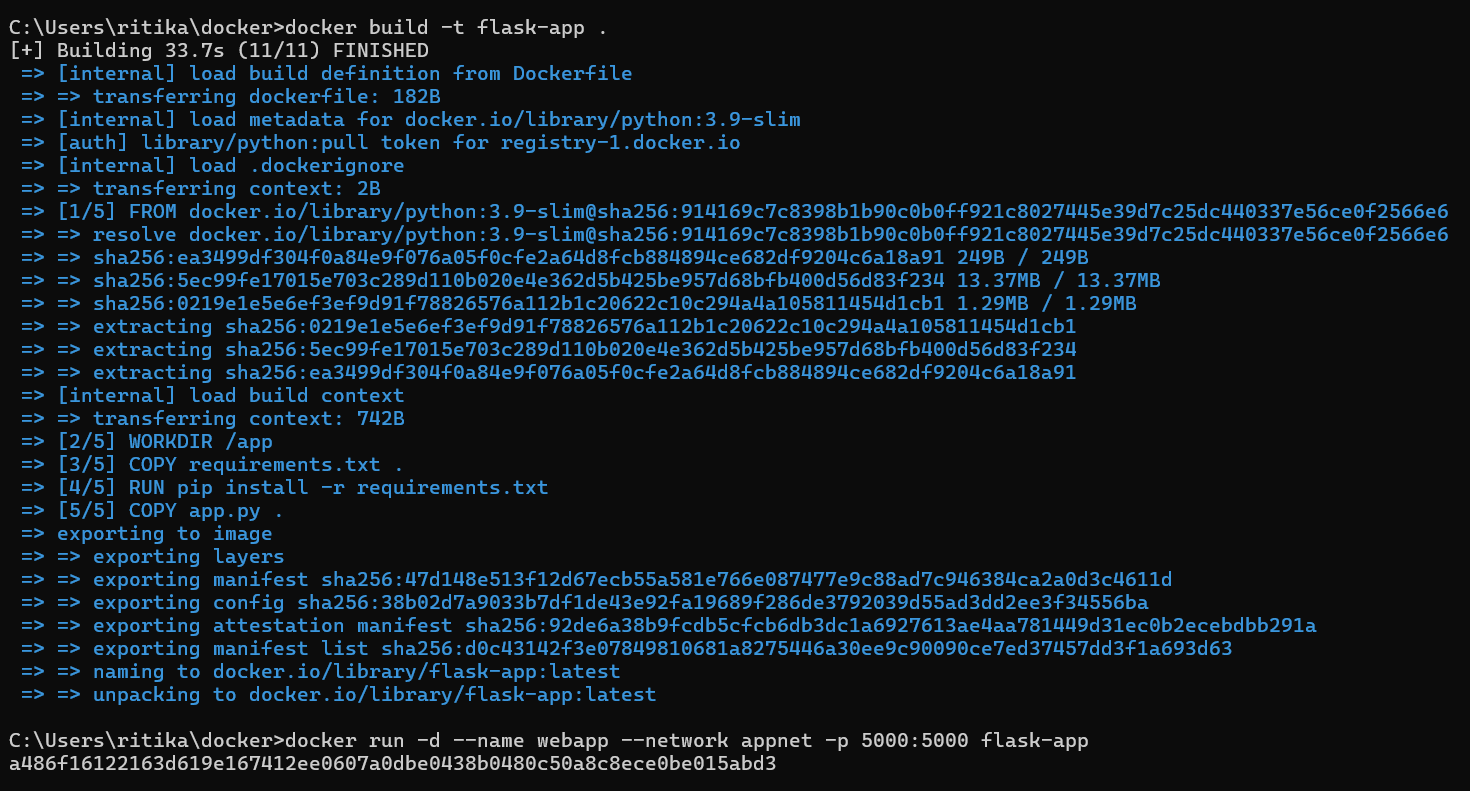
* **Create a simple Flask app**

****

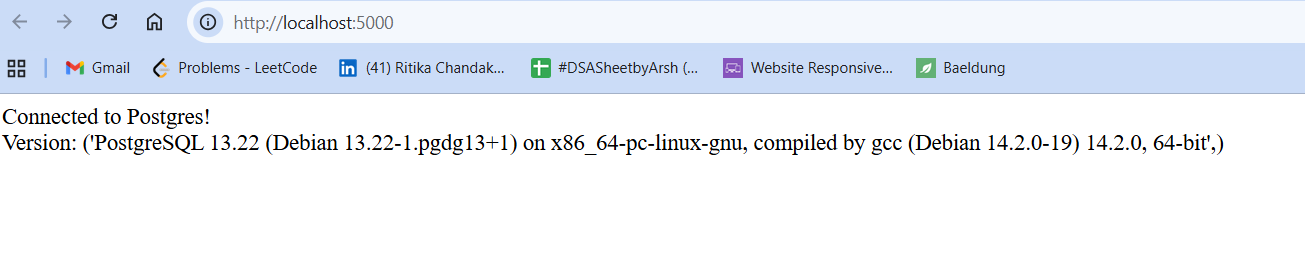
* **Create a Dockerfile for Flask app**

****

* **Build and run Flask container**



* **Test**



**Exercise 8: Docker Compose Basics**

**Command Used:**

docker compose up -d

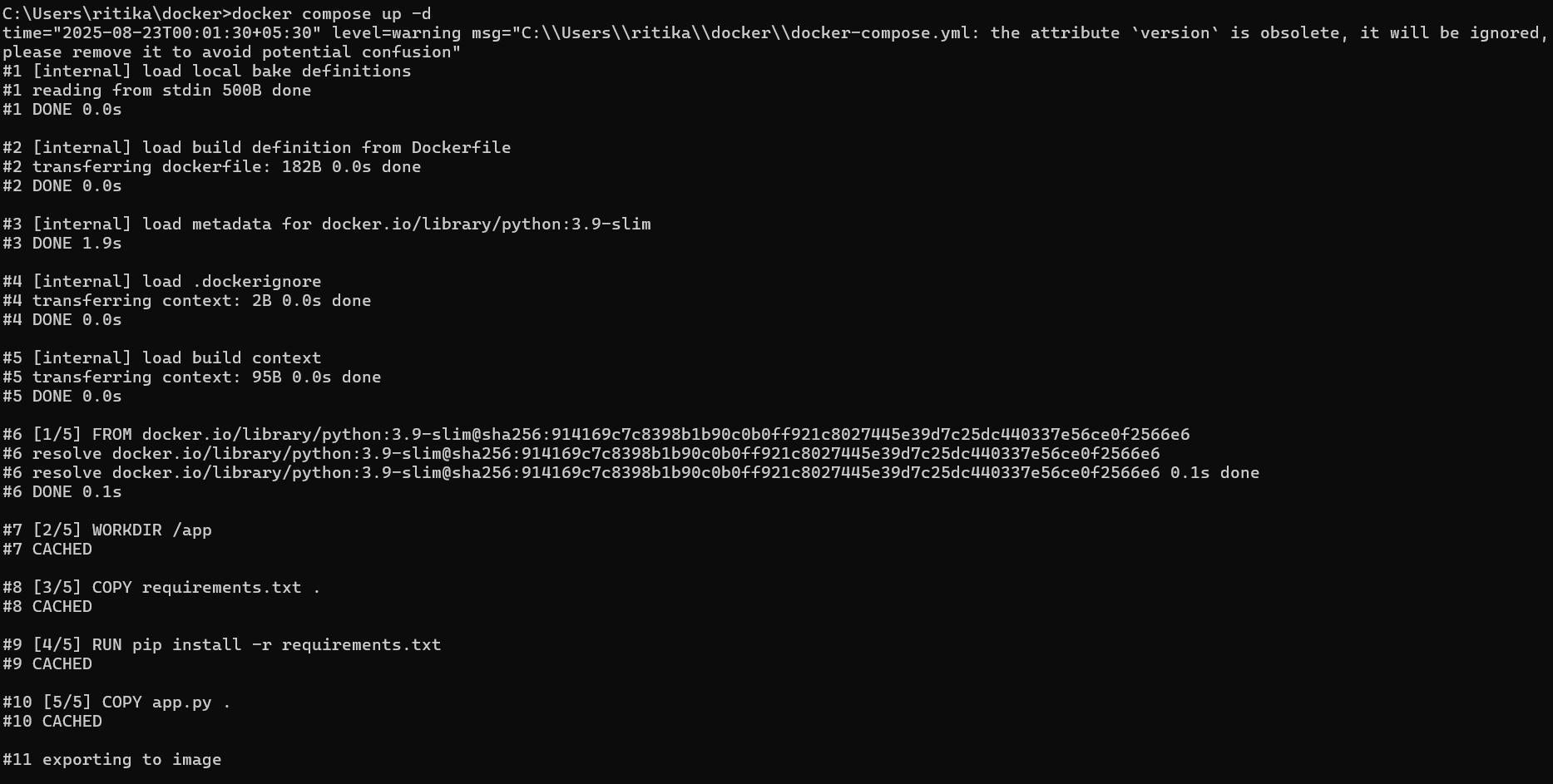
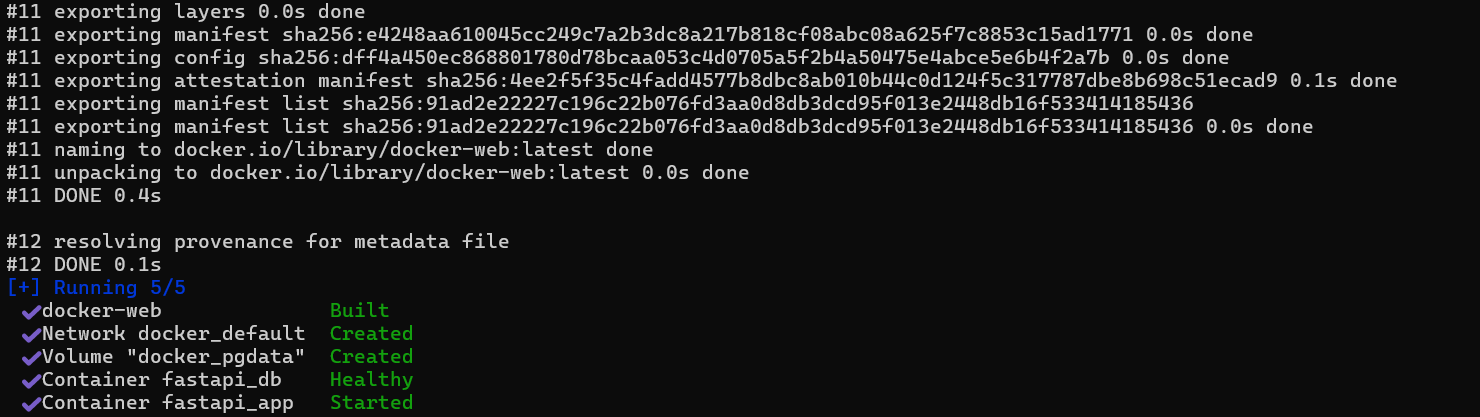
docker compose logs -f

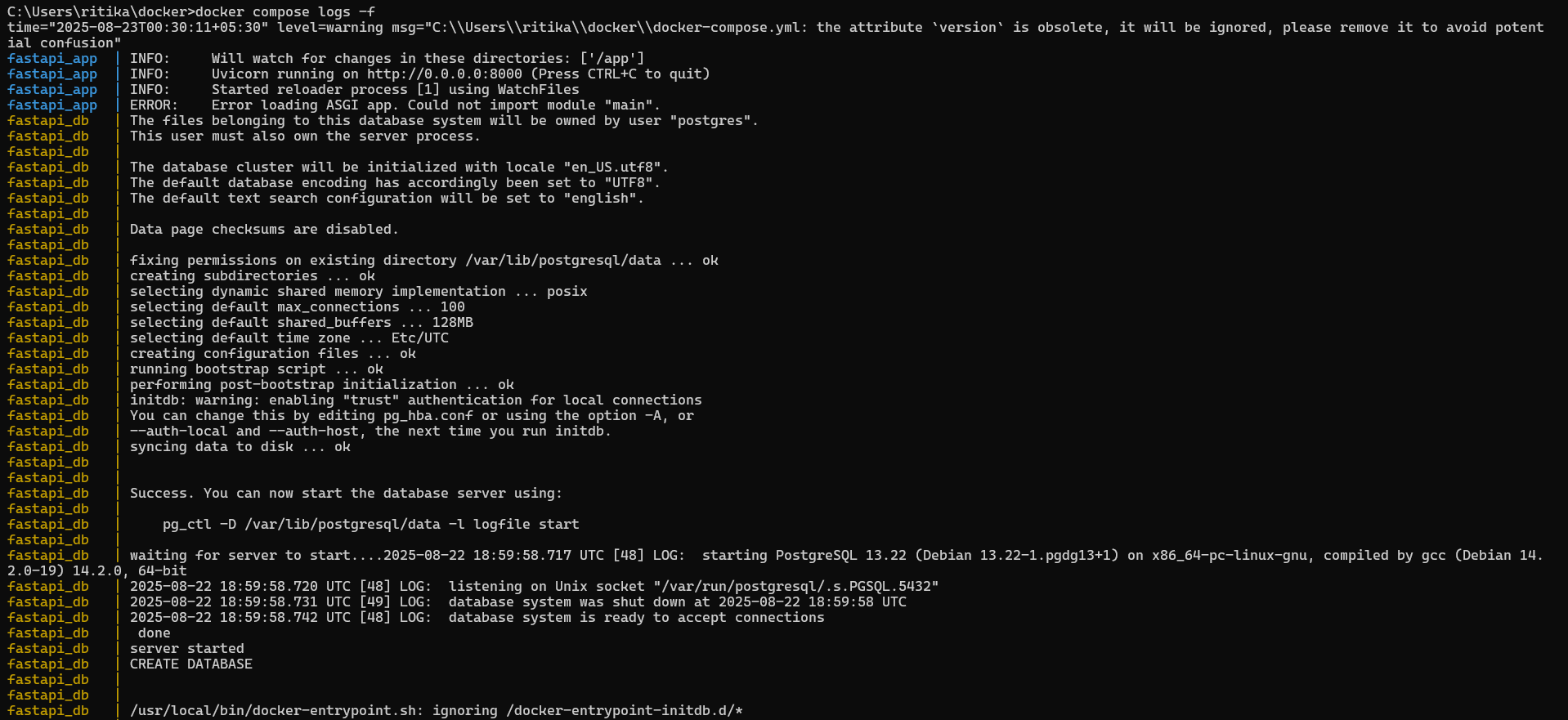
docker compose ps

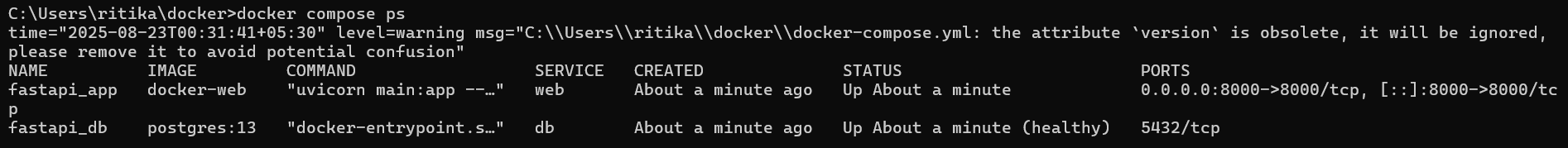
**● Write a docker-compose.yml for FastAPI + Postgres.**



**● Use docker compose up, inspect logs and containers.**





**Exercise 9: Healthchecks and Best Practices**

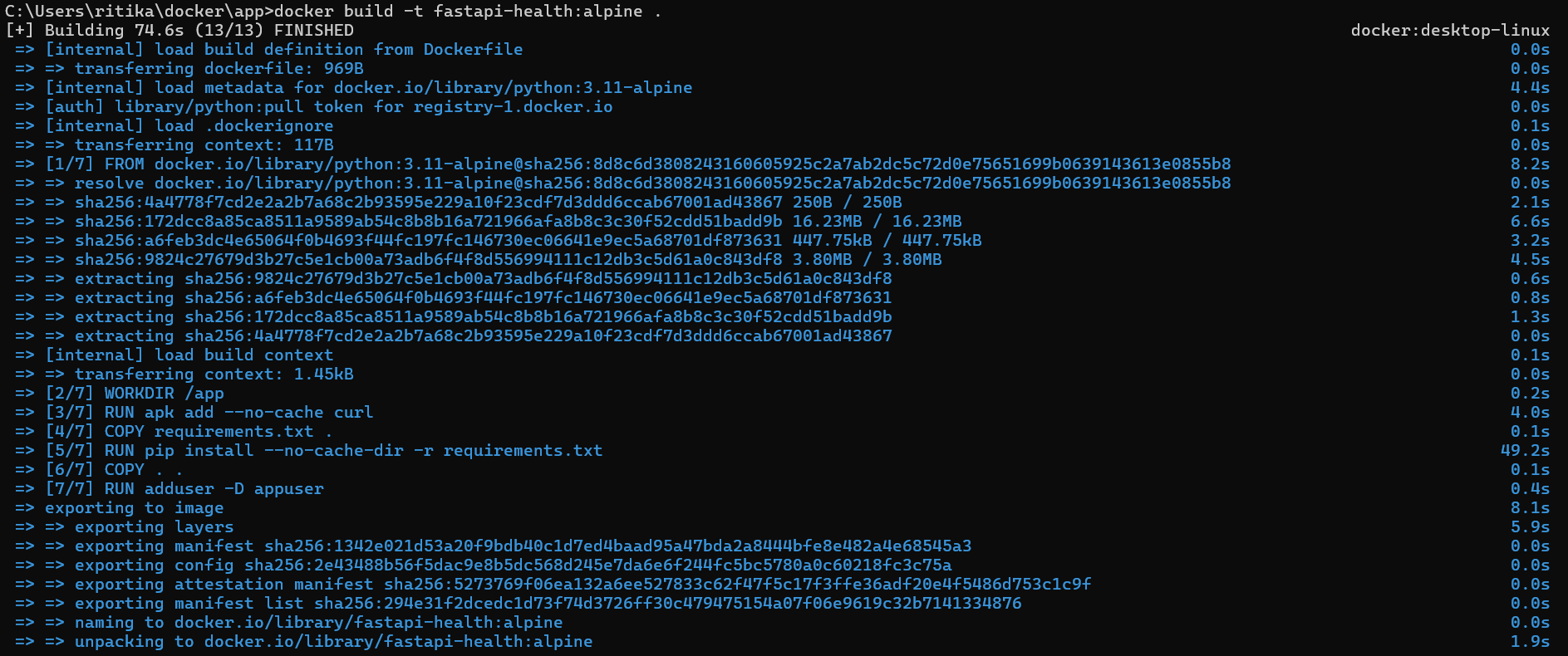
**Commands Used:**

docker build -t fastapi-health:alpine .

docker run -d --name fastapi9v2 -p 8001:8000 fastapi-health:alpine

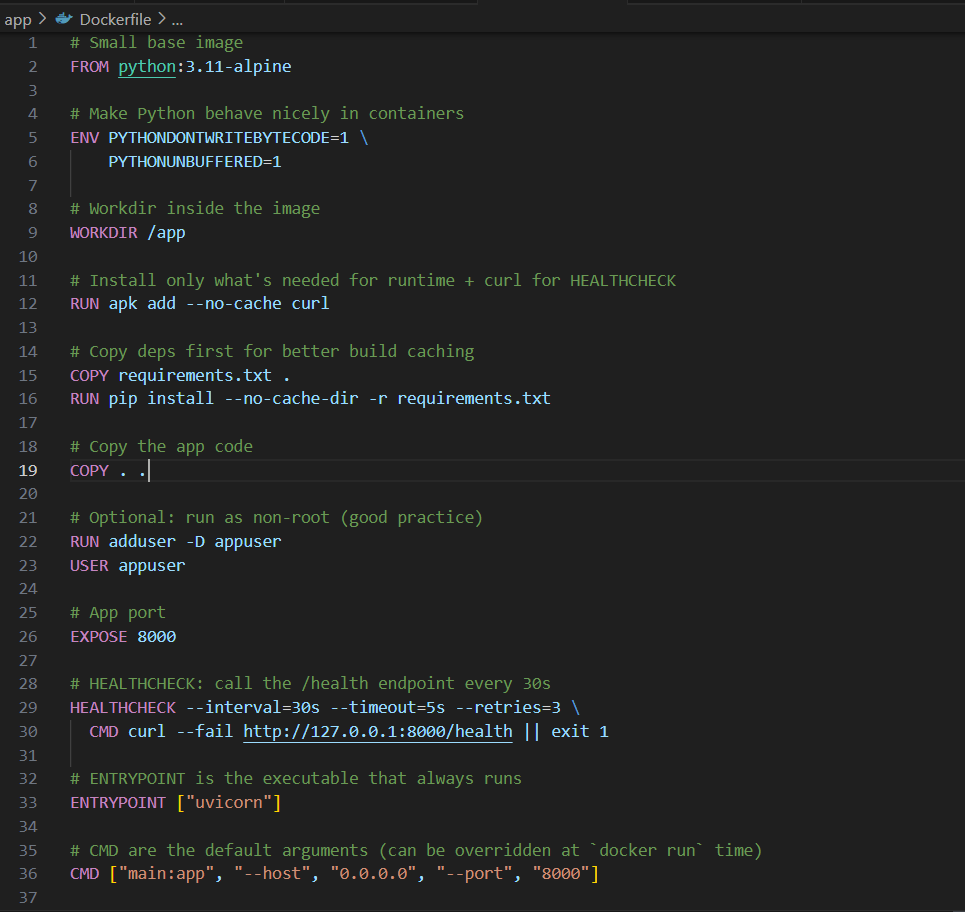
docker inspect --format="{{.State.Health.Status}}" fastapi9v2

**Build the image and run the container**

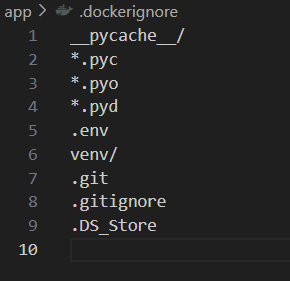




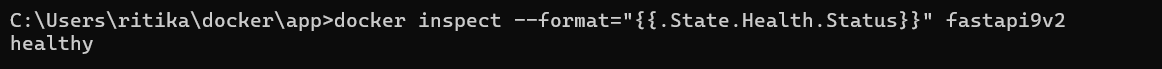
● **Add HEALTHCHECK instruction to your Dockerfile.**



**● Minimize layers and image size**.dockerignore (keeps the image small & clean)

****

**● Inspect container health via docker inspect.**

****

**Exercise 10: Debugging, Cleanup & Troubleshooting**

**Commands Used:**

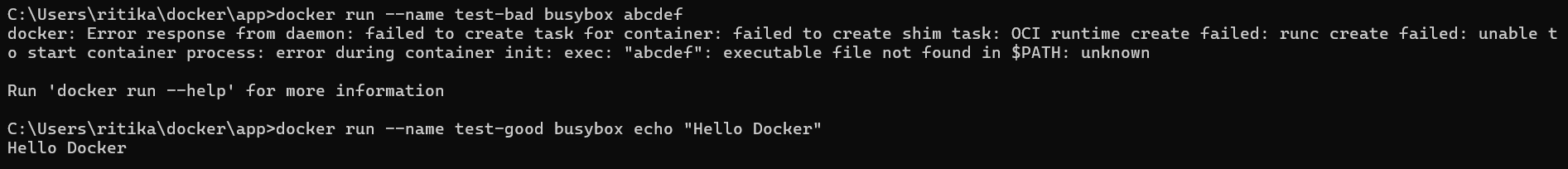
docker run --name test-bad busybox abcdef

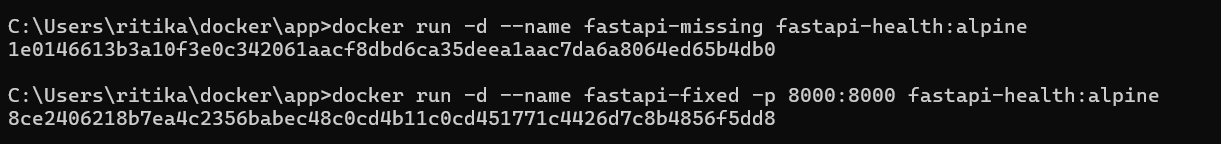
docker run --name test-good busybox echo "Hello Docker"

docker container prune

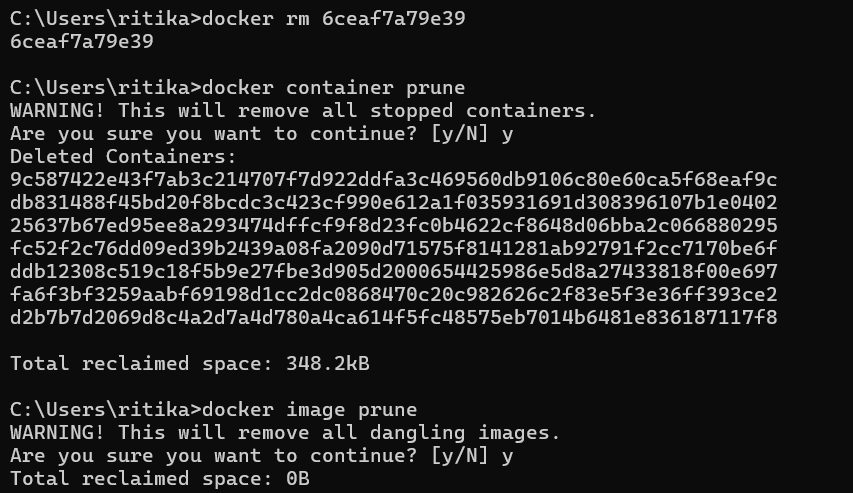
docker image prune

**● Run containers with bad commands or missing ports.**

****

****

**● Clean up unused images, containers, volumes with:**

****