# LAB 4: Virtualization with containers (Docker)

Network Architecture

Universitat Pompeu Fabra

Yihan Jin - 253297

Zihao Zhou - 285879

Martí Esquius - 267444

2024-2025

Table of Contents

1. Introduction..... 3

2. Laboratory work..... 4

2.1. Docker Installation Verification..... 4

2.2. Non-root Docker Access..... 4

2.3. Python Package Uninstallation..... 4

2.4. Dockerfile Setup..... 5

2.5. Image Building..... 6

2.6. Network Creation..... 7

2.7. Container Deployment..... 7

2.8. Container Inspection..... 8

2.9. Application Testing..... 10

2.10. & 2.11. Log Management and Real-time Monitoring..... 10

2.12. & 2.13. Container Lifecycle and Log Persistence..... 11

2.14. Persistent Storage Solution..... 11

2.15. DockerHub..... 13

2.16. Cleaning up Docker environment..... 15

3. Conclusion..... 16

# 1. Introduction

This lab session introduced us to Docker, a powerful containerization platform widely used in cloud computing and modern application deployment. Through hands-on exercises, we explored how containers provide isolated, portable environments for running applications, demonstrating their advantages over traditional virtualization.

The lab was structured to progressively build our understanding: starting with basic Docker commands, then moving to container creation, network configuration, and finally deploying a Flask web server in an isolated environment. We also learned critical Docker features like logging, persistent storage, and image sharing via DockerHub.

These exercises not only taught us the technical aspects of containerization but also highlighted its real-world applications in software development.

## 2. Laboratory work

### 2.1. Docker Installation Verification

We began by verifying Docker's installation using docker --version and exploring available commands with docker --help. This confirmed our environment was properly set up for container operations.

```
mininet@arqxxs:~$ sudo docker --version
Docker version 20.10.21, build 20.10.21-0ubuntu1~20.04.2
```

### 2.2. Non-root Docker Access

To improve security, we added our user to the docker group using: *sudo usermod -aG docker $USER*, eliminating the need for sudo with Docker commands and avoiding the potential risk of having several containers running in sudo mode..

```
mininet@arqxxs:~$ sudo usermod -aG docker $USER
mininet@arqxxs:~$ sudo shutdown now
```

Then, after rebooting, we confirmed this worked by running docker run hello-world.

```
mininet@arqxxs:~$ docker -v && docker run hello-world
Docker version 20.10.21, build 20.10.21-0ubuntu1~20.04.2
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
e6590344b1a5: Pull complete
Digest: sha256:c41088499908a59aae84b0a49c70e86f4731e588a737f1637e73c8c09d995654
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
```

### 2.3. Python Package Uninstallation

We intentionally uninstalled *flask* and *jsonify* from the host system (*sudo pip3 uninstall flask jsonify*) to later demonstrate container isolation, where applications can run regardless of host system dependencies.

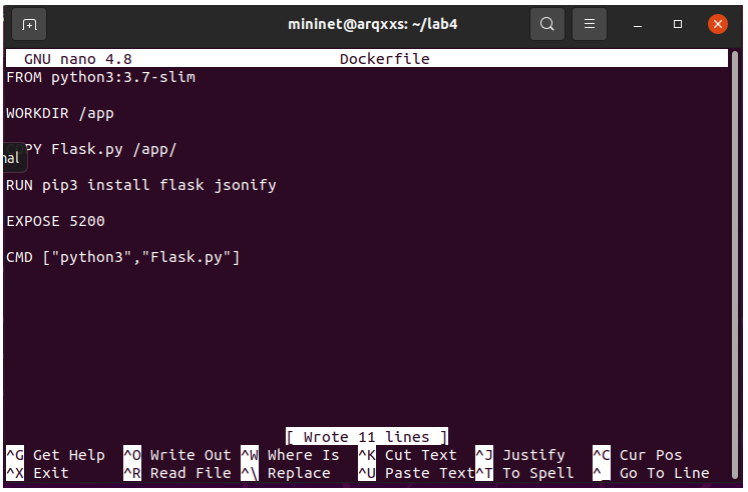
```
mininet@arqxxs:~$ sudo pip3 uninstall flask jsonify
[sudo] password for mininet:
Found existing installation: Flask 2.2.3
Uninstalling Flask-2.2.3:
  Would remove:
    /usr/local/bin/flask
    /usr/local/lib/python3.8/dist-packages/Flask-2.2.3.dist-info/*
    /usr/local/lib/python3.8/dist-packages/flask/*
Proceed (y/n)? y
  Successfully uninstalled Flask-2.2.3
Found existing installation: jsonify 0.5
Uninstalling jsonify-0.5:
  Would remove:
    /usr/local/lib/python3.8/dist-packages/jsonify-0.5.dist-info/*
    /usr/local/lib/python3.8/dist-packages/jsonify/*
Proceed (y/n)? y
  Successfully uninstalled jsonify-0.5
```

## 2.4. Dockerfile Setup

In a dedicated folder, we:

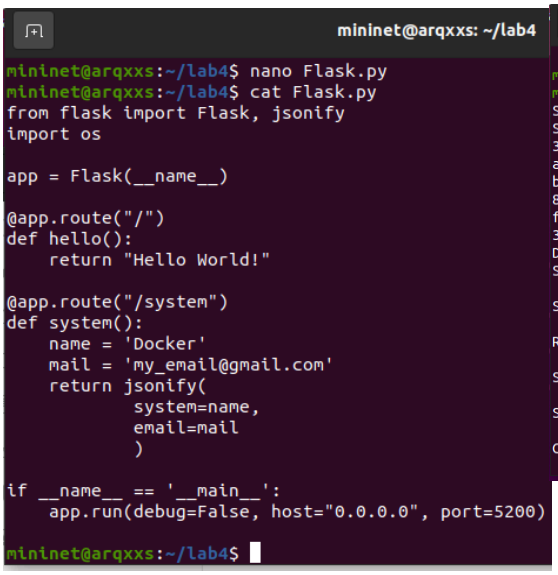
- Copied the Flask web server script (Flask.py).
- Created a *Dockerfile* specifying the environment (Python, dependencies).

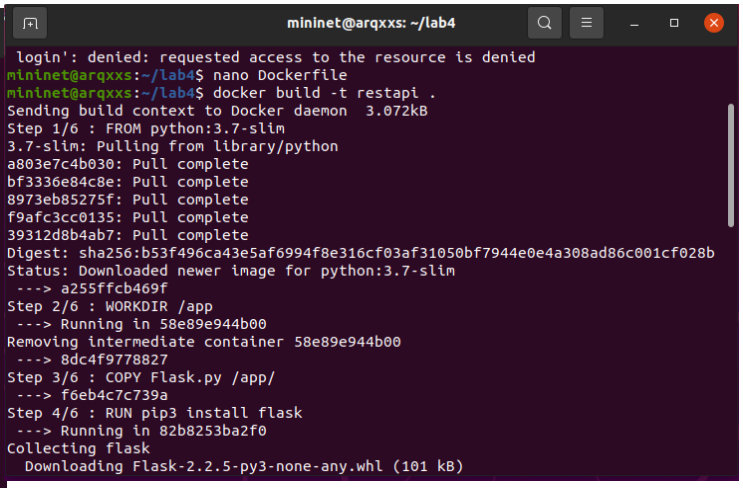
We used *gedit* instead of *nano* for text editing.



## 2.5. Image Building

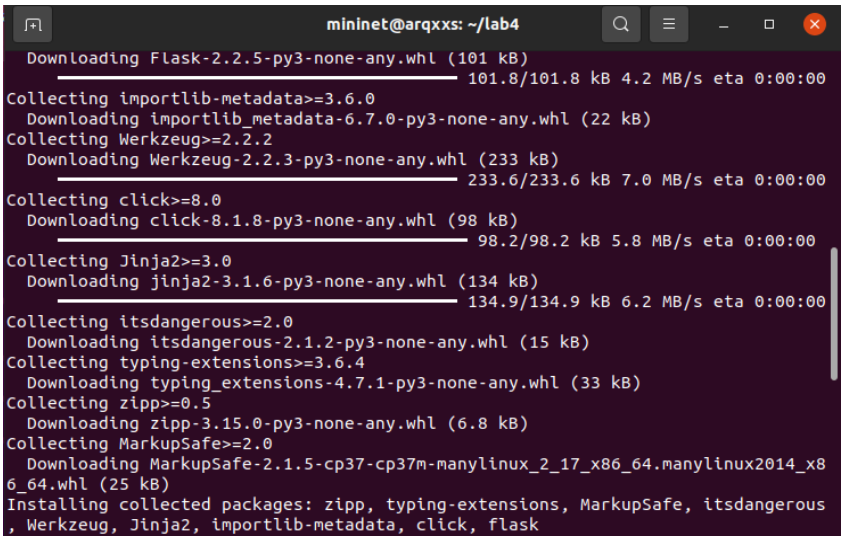
We built our first Docker image with *docker build -t restapi* . Then verified its creation using *docker images*. We observed that image size depends on the base OS and installed packages, finally smaller images are generally preferred for efficiency.



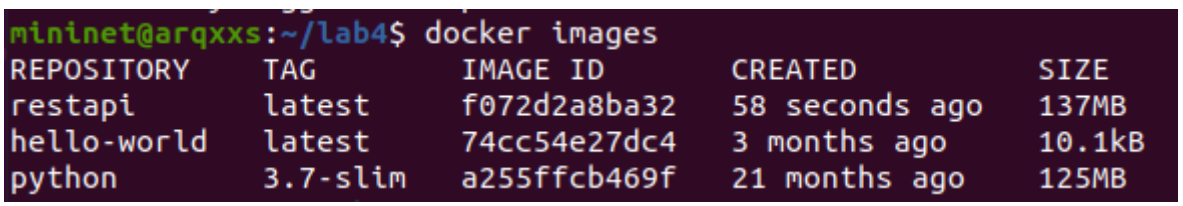




← We checked Flask.py

↑ Build Docker image

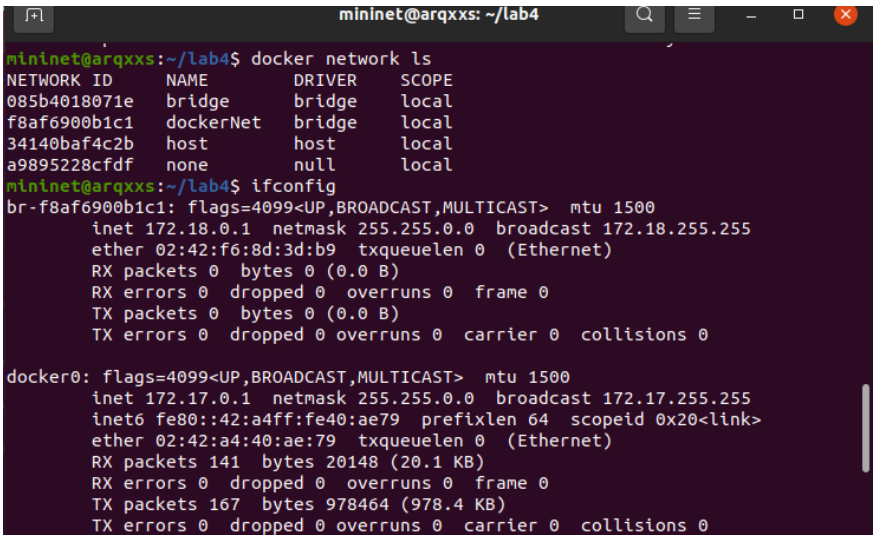


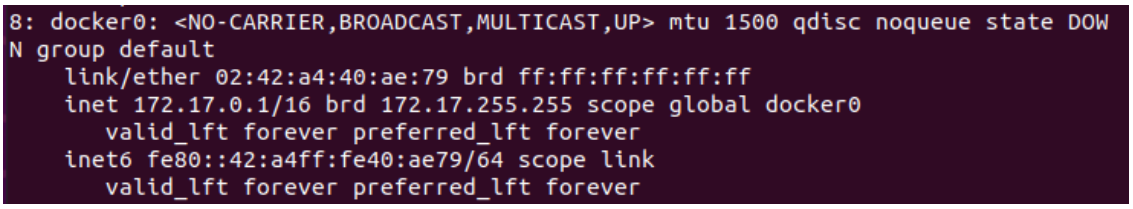

Above, we can see how Flask and other python libraries specified by us in the Dockerfile are being downloaded into the image when compiling. After the process, as we can see in the following image, a 'REPOSITORY' called *restapi* with IMAGE ID = *f072d2a8ba32* has been created.



## 2.6. Network Creation

A custom Docker network (dockerNet) was created with *docker network create --subnet=172.18.0.0/16 dockerNet*. The ifconfig command revealed a new host interface (172.18.0.1) for container communication.

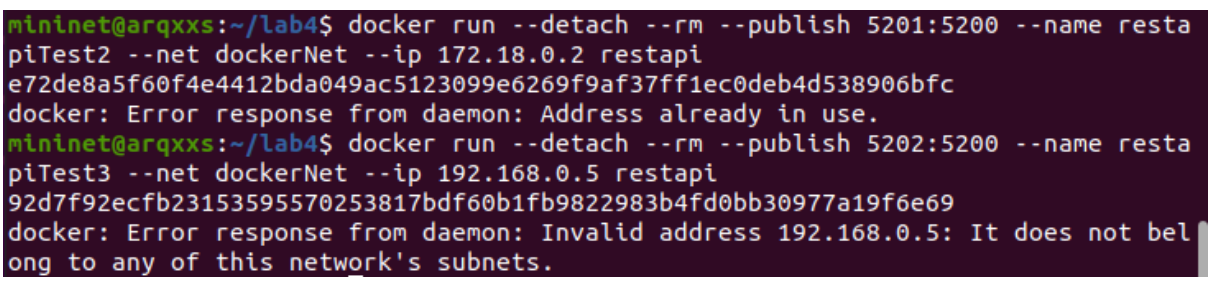



## 2.7. Container Deployment

We launched our Flask container with: *docker run -d --rm -p 5200:5200 --name restapiTest --net dockerNet --ip 172.18.0.2 restapi*. Docker enforced IP/subnet validation, preventing conflicts. After running the command, we can see how docker returns the container id.

We have also tried the case when another container uses the same ip or another container uses another ip outside of the subnet, in this case subnet is /16. We can see the result:



## 2.8. Container Inspection

Using docker exec, we:

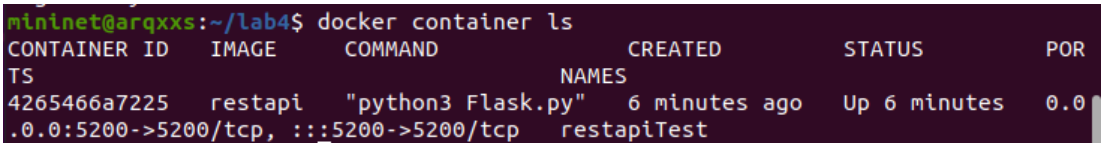
- Ran commands inside the container (ls, mkdir).
- Accessed a shell with docker exec -it restapiTest sh.
- Verified network config differences between host and container.

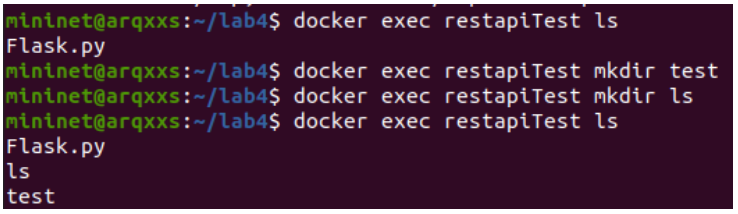
When accessing the container using docker exec -it restapiTest sh, the flags serve specific purposes:

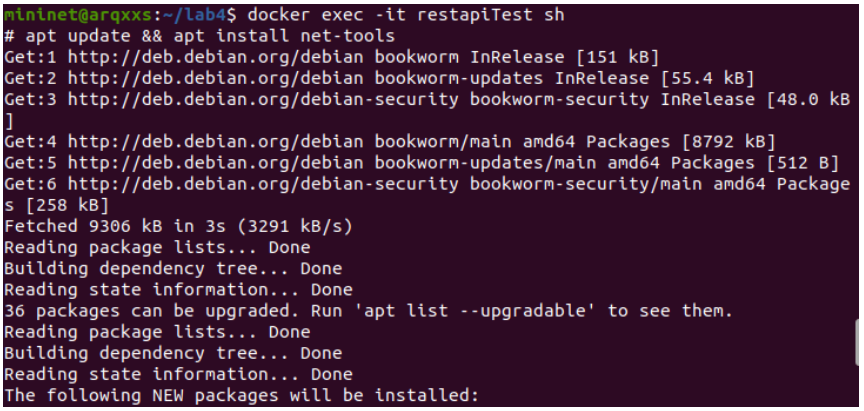
*-i (interactive)*: Keeps STDIN open, allowing commands to be sent to the container.

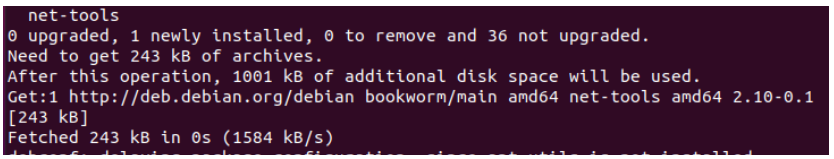
*-t (tty)*: Allocates a pseudo-terminal, enabling interactive shell behavior (e.g., proper prompt display).

When running ifconfig on the host versus inside the container, the outputs differ because the host shows all interfaces (eth0, docker0), while the container only displays its assigned IP (172.18.0.2). This demonstrates how containers maintain independent network stacks despite sharing the host's kernel, a key feature of lightweight virtualization that keeps environments separate yet efficient.



*Ifconfig* Docker output:

```
# ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 172.18.0.2  netmask 255.255.0.0  broadcast 172.18.255.255
        ether 02:42:ac:12:00:02  txqueuelen 0  (Ethernet)
        RX packets 465  bytes 9582963 (9.1 MiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 341  bytes 19808 (19.3 KiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 8  bytes 772 (772.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 8  bytes 772 (772.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

*Ifconfig* output for Host:

```
br-f8af6900b1c1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 172.18.0.1  netmask 255.255.0.0  broadcast 172.18.255.255
        inet6 fe80::42:f6ff:fe8d:3db9  prefixlen 64  scopeid 0x20<link>
        ether 02:42:f6:8d:3d:b9  txqueuelen 0  (Ethernet)
        RX packets 341  bytes 15034 (15.0 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 435  bytes 9579558 (9.5 MB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

docker0: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500
        inet 172.17.0.1  netmask 255.255.0.0  broadcast 172.17.255.255
        inet6 fe80::42:a4ff:fe40:ae79  prefixlen 64  scopeid 0x20<link>
        ether 02:42:a4:40:ae:79  txqueuelen 0  (Ethernet)
        RX packets 141  bytes 20148 (20.1 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 167  bytes 978464 (978.4 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

enp0s3: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.0.2.15  netmask 255.255.255.0  broadcast 10.0.2.255
        inet6 fe80::ff55:d222:dab1:85c8  prefixlen 64  scopeid 0x20<link>
        ether 08:00:27:df:09:05  txqueuelen 1000  (Ethernet)
        RX packets 40582  bytes 60154501 (60.1 MB)
        RX errors 0  dropped 0  overruns 0  frame 0
```

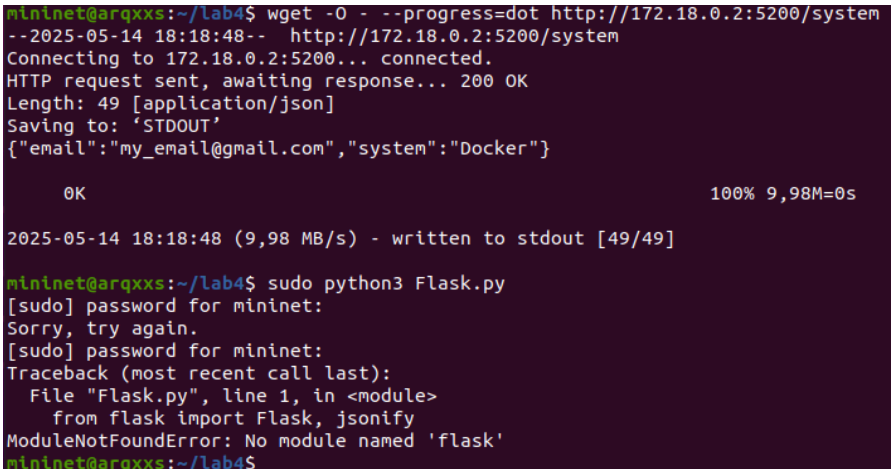
```
lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 2324  bytes 130258 (130.2 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 2324  bytes 130258 (130.2 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

vetha0732db: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet6 fe80::3cec:f7ff:fe94:ad3e  prefixlen 64  scopeid 0x20<link>
        ether 3e:ec:f7:94:ad:3e  txqueuelen 0  (Ethernet)
        RX packets 341  bytes 19808 (19.8 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 468  bytes 9583247 (9.5 MB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

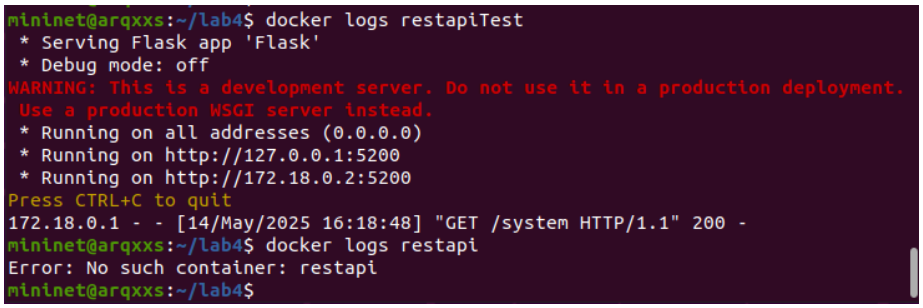
## 2.9. Application Testing

We thoroughly tested our containerized Flask application using *wget -O - --progress=dot http://172.18.0.2:5200/system* The --progress=dot flag provided visible download progress, confirming successful HTTP communication, which worked despite having uninstalled Flask and jsonify on the host. This confirmed Docker's dependency isolation: containers bundle their own software environment. When attempting to run the server directly on the host (sudo python3 Flask.py), it failed due to missing packages.

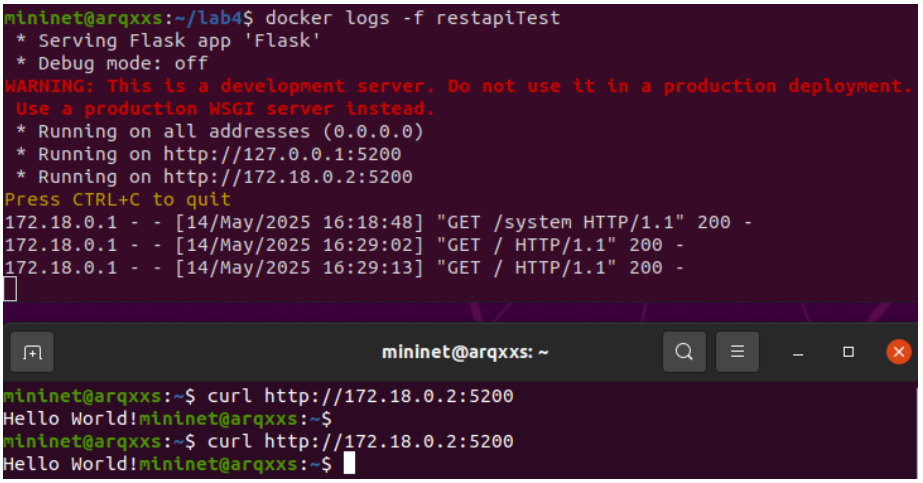
This behavior is expected because containers include all specified dependencies in their images, independent of the host system.




## 2.10. & 2.11. Log Management and Real-time Monitoring




We captured container logs with docker logs restapiTest and added the *-f* flag for real-time monitoring during *wget/curl* tests, showing how Docker provides application visibility.



## 2.12. & 2.13. Container Lifecycle and Log Persistence

When we stopped the container using docker *stop restapiTest*, it was automatically deleted because of the *--rm* flag we used earlier. All the logs disappeared with it, showing that containers don't save data by default. As we can see in the following images.

```
mininet@arqxxs:~/lab4$ docker container stop restapiTest
restapiTest
```

```
mininet@arqxxs:~/lab4$ docker ps -a
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS     NAMES
```

```
mininet@arqxxs:~/lab4$ docker logs restapiTest
 * Serving Flask app 'Flask'
 * Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use a produc
tion WSGI server instead.
 * Running on all addresses (0.0.0.0)
 * Running on http://127.0.0.1:5200
 * Running on http://172.18.0.2:5200
Press CTRL+C to quit
mininet@arqxxs:~/lab4$ docker logs -f restapiTest
```

## 2.14. Persistent Storage Solution

We modified our Docker setup to permanently save logs by:

- Updating the Dockerfile to store logs in /app/logs/flask.log
- Using -v /tmp:/app/logs to create a host-container log directory link

```
1 FROM python:3.7-slim
2
3 WORKDIR /app
4 COPY Flask.py /app/
5 RUN pip3 install flask
6 RUN mkdir /app/logs
7 EXPOSE 5200
8 CMD ["sh", "-c", "python3 Flask.py > /app/logs/flask.log 2>&1"]
```

Then, we created the image:

```
mininet@arqxxs:~/lab4$ docker build -t restapi-modified .
Sending build context to Docker daemon  3.072kB
Step 1/7 : FROM python:3.7-slim
 ---> a255ffcb469f
Step 2/7 : WORKDIR /app
 ---> Using cache
 ---> 8dc4f9778827
Step 3/7 : COPY Flask.py /app/
 ---> Using cache
 ---> f6eb4c7c739a
Step 4/7 : RUN pip3 install flask
 ---> Using cache
 ---> 7772d7f980ec
Step 5/7 : RUN mkdir /app/logs
 ---> Running in 5e974ca3ae20
Removing intermediate container 5e974ca3ae20
 ---> 9add84a75795
Step 6/7 : EXPOSE 5200
 ---> Running in 5b001d7a5b05
Removing intermediate container 5b001d7a5b05
 ---> ae7e553e5b38
Step 7/7 : CMD ["sh", "-c", "python3 Flask.py > /app/logs/flask.log 2>&1"]
 ---> Running in c88a4c4c85e6
Removing intermediate container c88a4c4c85e6
 ---> 3cf84a1bc712
Successfully built 3cf84a1bc712
Successfully tagged restapi-modified:latest
```

*restapi-modified* can be found in *docker images* from now on:

```
mininet@arqxxs:~/lab4$ docker run --detach --rm --publish 5201:5200 --name restapiTest-modifi
ed --net dockerNet --ip 172.18.0.3 -v /tmp:/app/logs restapi-modified
bc2467e40118cafea95e7433f9f6adef21dca465a6830008720521388c0b09a5
mininet@arqxxs:~/lab4$ docker images
REPOSITORY          TAG         IMAGE ID       CREATED         SIZE
restapi-modified    latest      9b2a9f55bc14   4 minutes ago   137MB
restapi             latest      f072d2a8ba32   3 hours ago     137MB
hello-world         latest      74cc54e27dc4   3 months ago    10.1kB
python              3.7-slim    a255ffcb469f   21 months ago   125MB
mininet@arqxxs:~/lab4$
```

This ensured logs survived container restarts and remained accessible in /tmp/flask.log, solving the ephemeral logging problem for production environments. Testing,

```
mininet@arqxxs:~/lab4$ curl http://localhost:5201
Hello World!mininet@arqxat /tmp/flask.log  # Logs should appear here
 * Serving Flask app 'Flask'
 * Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use a produc
tion WSGI server instead.
 * Running on all addresses (0.0.0.0)
 * Running on http://127.0.0.1:5200
 * Running on http://172.18.0.3:5200
Press CTRL+C to quit
172.18.0.1 - - [14/May/2025 17:25:40] "GET / HTTP/1.1" 200 -
```

Help

,

confirmed logs persisted through multiple container lifecycles. As we can see in the following image:

```
mininet@arqxxs:~/lab4$ docker stop restapiTest-modified
restapiTest-modified
mininet@arqxxs:~/lab4$ cat /tmp/flask.log
 * Serving Flask app 'Flask'
 * Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use a produc
tion WSGI server instead.
 * Running on all addresses (0.0.0.0)
 * Running on http://127.0.0.1:5200
 * Running on http://172.18.0.3:5200
Press CTRL+C to quit
172.18.0.1 - - [14/May/2025 17:25:40] "GET / HTTP/1.1" 200 -
```

## 2.15. DockerHub

From DockerHub, we followed the following steps:

**Access token description**
read-write-delete-token

**Expires on**
Jun 13, 2025 at 23:59:59

**Access permissions**
Read, Write, Delete

**To use the access token from your Docker CLI client:**

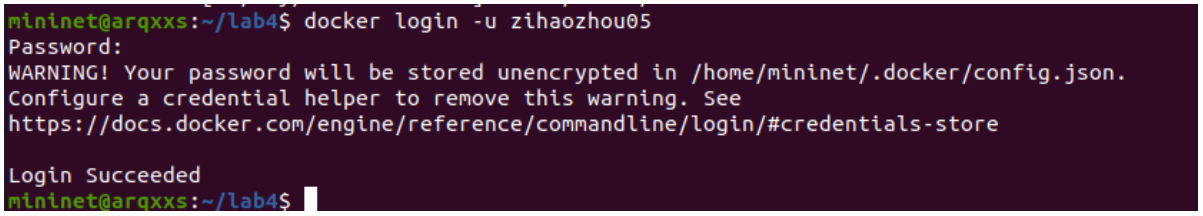
1. Run

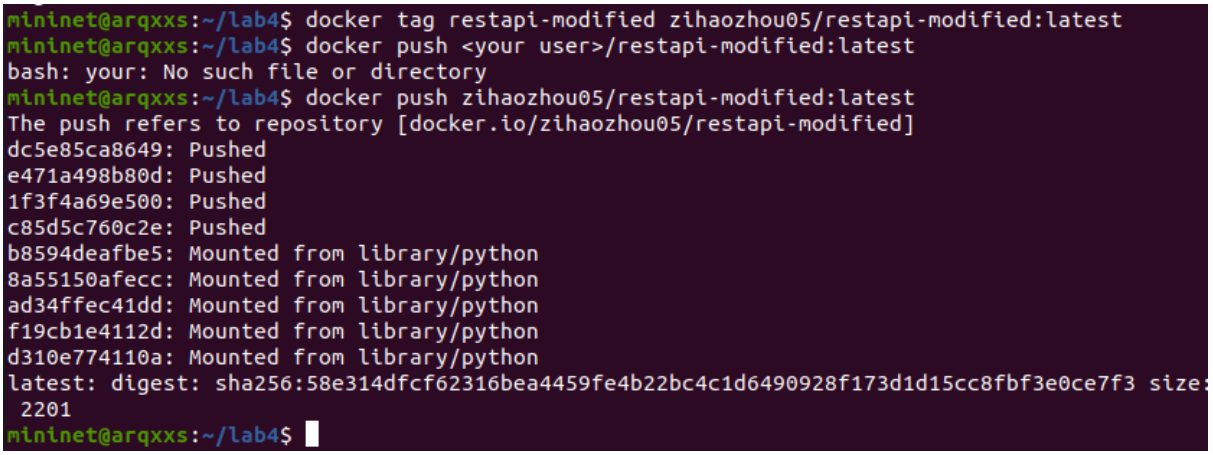
```
$ docker login -u zihaozhou05
```

Copy

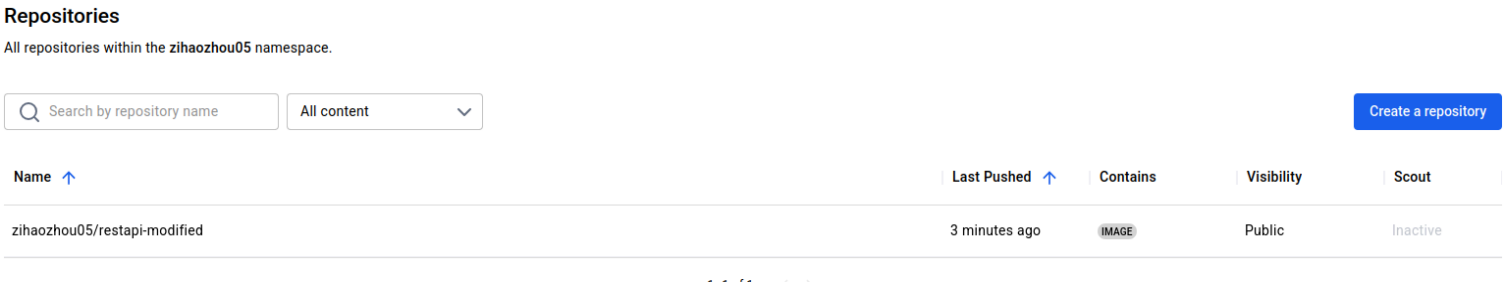
2. At the password prompt, enter the personal access token.

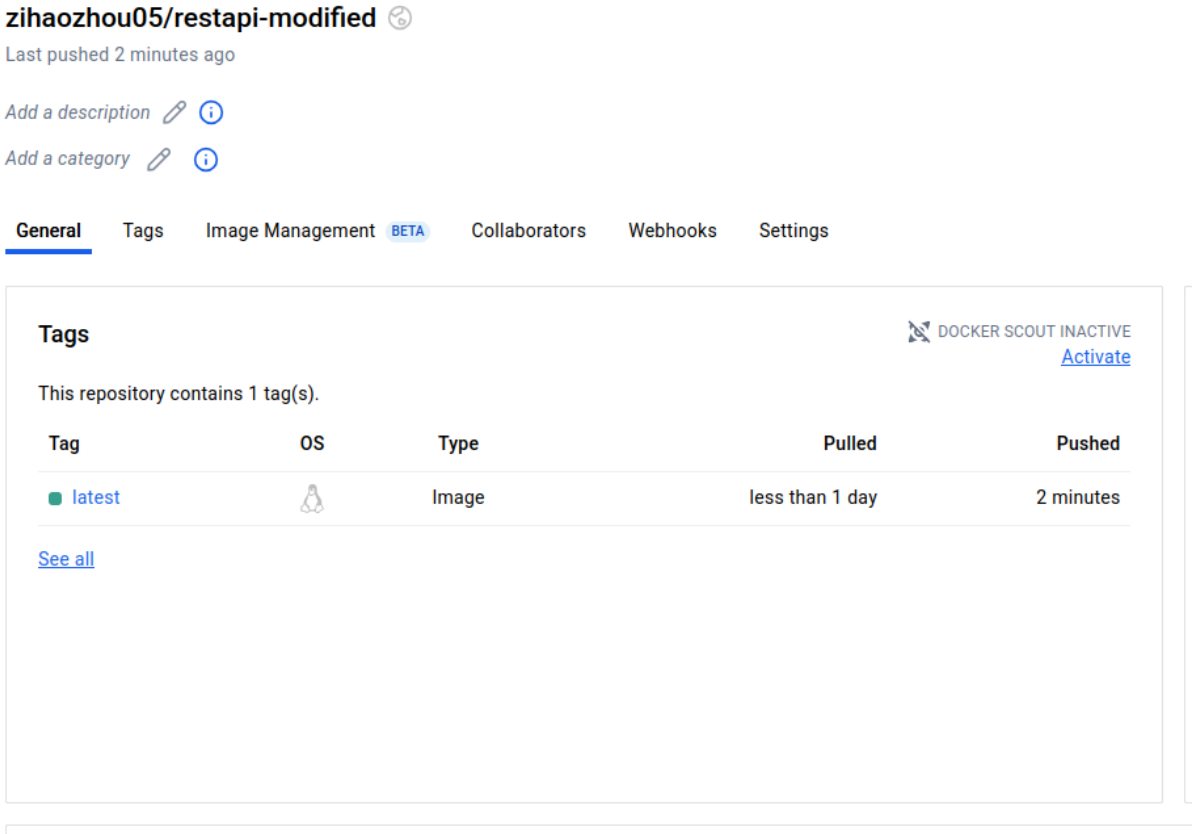
As we can see in the following images:



Then, we realised that we had created a repository, as we can see in the following screenshots extracted from DockerHub:




.

## 2.16. Cleaning up Docker environment

Finally, we cleaned up the whole docker environment, which includes deleting networks, containers and images. The main goal was to free up disk space. We used *docker system prune -a*, as we can see in the following screenshot:

```
mininet@arqxxs:~/lab4$ docker system prune -a
WARNING! This will remove:
  - all stopped containers
  - all networks not used by at least one container
  - all images without at least one container associated to them
  - all build cache

Are you sure you want to continue? [y/N] y
Deleted Images:
untagged: python:3.7-slim
untagged: python@sha256:b53f496ca43e5af6994f8e316cf03af31050bf7944e0e4a308ad86c001cf028b
untagged: restapi:latest
deleted: sha256:f072d2a8ba325ef8ba6273dbc629b16b2f6012f1aa622f248aa2d5d9b7218807
deleted: sha256:5124f7f33d29398199a637e583a62962f9bdaaffa1e3e41bcf5e2177dba212f6
untagged: restapi-modified:latest
untagged: zihaozhou05/restapi-modified:latest
untagged: zihaozhou05/restapi-modified@sha256:58e314dfcf62316bea4459fe4b22bc4c1d6490928f173d1
d15cc8fbf3e0ce7f3
deleted: sha256:3cf84a1bc712de44396f3de73bbaaf5865d37d3d117e28916bdfdd98a4de1c02
deleted: sha256:ae7e553e5b38ec3ab9fc9744febcc31f25ba235b489eb3ad64be0124b53b8d0a
deleted: sha256:9add84a757953d692aeabb4a3ac135a0a0facc2a4b4d2e4fae1896b1f81b5f0e
deleted: sha256:78d7933862b5c18e81d25d67f8f44314af94b31ed737826ec76a62202f2c9b32
deleted: sha256:7772d7f980ec685ddc682a357871d1e31f726d558d94f5820f269a430f801652
deleted: sha256:dae919d5bb7560325b0b1164be87161824955742f604e975d1be2d944b39ddb2
deleted: sha256:f6eb4c7c739abaf73b1d189f653f327ef8a31d3a198d2f230c3a63a14419d6c1
deleted: sha256:873032280473d0acdf5151b5b504dfae9214769c11c978450d0440bfe0d02218
deleted: sha256:8dc4f9778827b1208600829375f6c8ad99eb8aec4fb79162a7f56e0e89f006df
deleted: sha256:8171fddd638f69e79bc97772941e4d4aa59fe24ac84215411e9a7d6654d3f328
deleted: sha256:a255ffcb469f2ec40f2958a76beb0c2bbebfe92ce9af67a9b48d84b4cb695ac8
deleted: sha256:5c4b4ba839cbedd3555359d155b28fb80fed820fc9aaca66469a4aa01787d565
deleted: sha256:c0f3c039e8d337801f8652e704b9b2dd492125bb26ff6e0bf7398cb48d9b8311
deleted: sha256:0a4e1d554b6f6dd9e6b808c070835aba7e7cbaa6c4a574b6aa0f9b527176e20c
deleted: sha256:44ebfbc5084593b644e94db3f6980c0aee62c0ce5db58ac89867ad8f21032955
deleted: sha256:d310e774110ab038b30c6a5f7b7f7dd527dbe527854496bd30194b9ee6ea496e

Total reclaimed space: 136.7MB
mininet@arqxxs:~/lab4$ docker images
REPOSITORY   TAG       IMAGE ID   CREATED   SIZE
mininet@arqxxs:~/lab4$
```

Specifically, in this exercise we need to specify the feature of docker, the shared layer, that is why the reclaimed space is less than the sum of them.

## 3. Conclusion

This lab provided invaluable practical experience with Docker containerization. Key takeaways include:

Isolation: Containers package applications with their dependencies, unaffected by host system configuration (demonstrated by running Flask without host Python packages).

Portability: The DockerHub exercise showed how containerized apps can be easily shared and deployed across different environments

Resource Management: Features like custom networks, bind mounts, and log monitoring gave us insight into production-grade container administration.

Most importantly, we saw firsthand why containers have become fundamental to modern cloud infrastructure, offering lightweight, consistent environments that simplify development and deployment workflows.