

Network and System Administration Final Project

Practical Activity - PR



Universitat Oberta
de Catalunya

Nicolas D'Alessandro Calderon

*Bachelor's degree in Techniques for
Software Application Development*

Course instructor

Jaume Jofre Bravo

Date of submission

May 24, 2025

Table of Contents

01 INSTALLING THE OS AND CREATING USERS	1
1.1 Installing the OS on the Virtual Machine	1
1.2 Host-Guest connection verification	1
1.3 Creating Users	1
02 SERVICE STACK (DOCKER COMPOSE)	3
2.1 Postgres Server Configuration	3
2.2 Bun Web Server Configuration	6
2.3 Connection Between Bun Server and Postgres	8
03 REVERSE-PROXY WITH HAProxy	11
3.1 HTTP Proxy	11
3.2 HTTPS Proxy	16
04 KUBERNETES.....	22
05 CONCLUSIONS AND FUTURE WORK	26
APPENDIX CONFIGURATION FILES.....	27

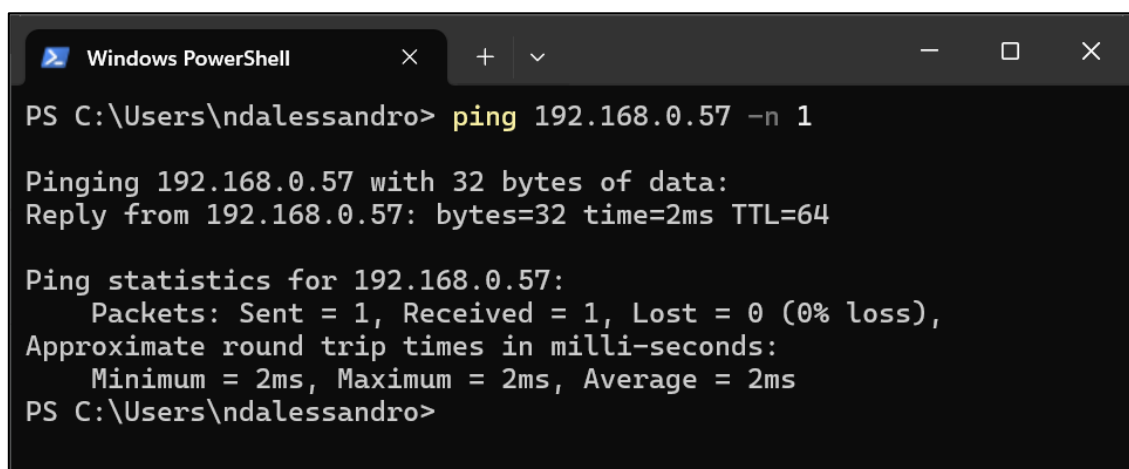
1.1 Installing the OS on the Virtual Machine

Debian 12.10 has been installed on the virtual machine named **ARSO20242**, using VirtualBox as virtualization platform.

The main user of the system has been configured using the campus alias (**ndalessandro**).

1.2 Host-Guest connection verification

To verify the connection between the Host (physical device PC) and the Guest (virtual machine) the following command was executed from PowerShell. As we can see in the image, the response received confirms a successful connection.



```
Windows PowerShell
PS C:\Users\ndalessandro> ping 192.168.0.57 -n 1

Pinging 192.168.0.57 with 32 bytes of data:
Reply from 192.168.0.57: bytes=32 time=2ms TTL=64

Ping statistics for 192.168.0.57:
    Packets: Sent = 1, Received = 1, Lost = 0 (0% loss),
    Approximate round trip times in milli-seconds:
        Minimum = 2ms, Maximum = 2ms, Average = 2ms
PS C:\Users\ndalessandro>
```

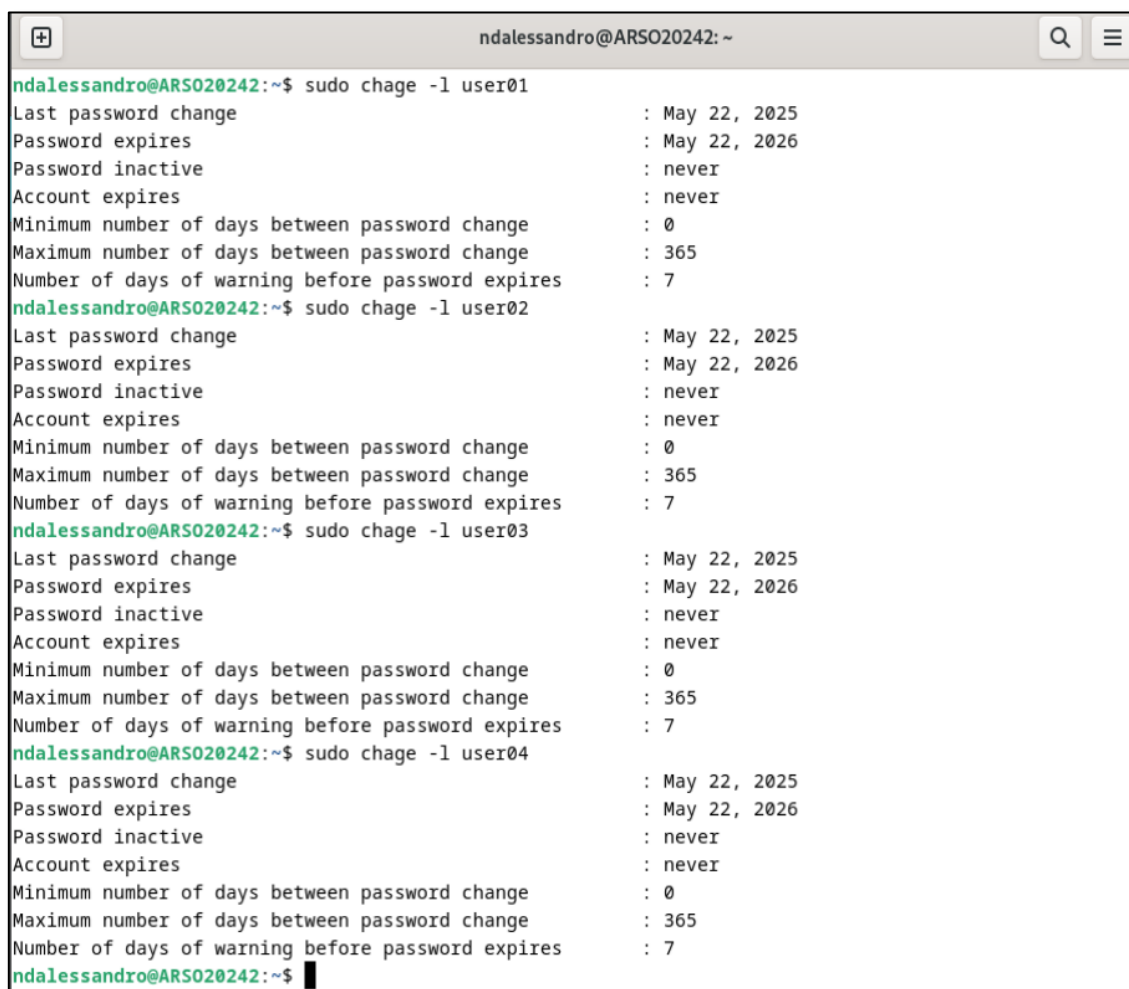
1.3 Creating Users

A set of ten users was created using the prefix user followed by a consecutive number from 01 to 10. Each user was configured so that the password expires after a year (365 days).

The process was automated using the following script executed in the terminal:

```
for i in {01..10}; do  
  
    sudo useradd -m user$i  
  
    sudo passwd user$i  
  
    sudo chage -M 365 user$i  
  
done
```

Finally, I verified that the configuration was correctly applied for each user. The output confirms that every user has the password expiration date set to 365 days:



```
ndalessandro@ARS020242: ~  
ndalessandro@ARS020242:~$ sudo chage -l user01  
Last password change           : May 22, 2025  
Password expires               : May 22, 2026  
Password inactive              : never  
Account expires               : never  
Minimum number of days between password change : 0  
Maximum number of days between password change : 365  
Number of days of warning before password expires : 7  
ndalessandro@ARS020242:~$ sudo chage -l user02  
Last password change           : May 22, 2025  
Password expires               : May 22, 2026  
Password inactive              : never  
Account expires               : never  
Minimum number of days between password change : 0  
Maximum number of days between password change : 365  
Number of days of warning before password expires : 7  
ndalessandro@ARS020242:~$ sudo chage -l user03  
Last password change           : May 22, 2025  
Password expires               : May 22, 2026  
Password inactive              : never  
Account expires               : never  
Minimum number of days between password change : 0  
Maximum number of days between password change : 365  
Number of days of warning before password expires : 7  
ndalessandro@ARS020242:~$ sudo chage -l user04  
Last password change           : May 22, 2025  
Password expires               : May 22, 2026  
Password inactive              : never  
Account expires               : never  
Minimum number of days between password change : 0  
Maximum number of days between password change : 365  
Number of days of warning before password expires : 7  
ndalessandro@ARS020242:~$
```

02 SERVICE STACK (DOCKER COMPOSE)

2.1 Postgres Server Configuration

I started by setting up the database service using Docker. To avoid exposing credentials in the `docker-compose.yml` file I used a `.env` file to store the password. For the data persistence, I mounted a volume call `dbdata` to ensure that the data is not lost every time the container stops.

I have also configured the automatic loading of the initialization script using the folder recognized by Postgres `/docker-entrypoint-initdb.d`. This allows to have the uoc2024 database prepared and ready from the first startup without needing of executing it manually.

`.env` file

```
ndalessandro@ARSO20242: ~/prac20242-orig
ndalessandro@ARSO20242:~/prac20242-orig$ cat .env
DB_PASSWORD=1234
ndalessandro@ARSO20242:~/prac20242-orig$
```

`docker-compose.yml` file

```
ndalessandro@ARSO20242: ~/prac20242-orig
ndalessandro@ARSO20242:~/prac20242-orig$ cat docker-compose.yml
services:
  db:
    image: postgres:17.4
    container_name: dbhost
    restart: always
    ports:
      - "5432:5432"
    volumes:
      - dbdata:/var/lib/postgresql/data
      - ./dataset:/docker-entrypoint-initdb.d
    environment:
      POSTGRES_DB: uoc2024
      POSTGRES_USER: ndalessandro
      POSTGRES_PASSWORD: ${DB_PASSWORD}
volumes:
  dbdata:
ndalessandro@ARSO20242:~/prac20242-orig$
```

IP_HOST: 192.168.0.6
IP_GUEST: 192.168.0.57

After starting the container with the command `sudo docker compose up --build` I verified that the service was listening on port 5432. I also checked the connection from both the host (Windows) and the VM confirming that I can correctly access with my user `ndalessandro`.

Running container

```
ndalessandro@ARSO20242: ~/prac20242-orig
ndalessandro@ARSO20242: ~/prac20242-orig x ndalessandro@ARSO20242: ~/prac20242-orig x
ndalessandro@ARSO20242:~/prac20242-orig$ sudo docker ps
[sudo] password for ndalessandro:
CONTAINER ID   IMAGE          COMMAND                  CREATED        STATUS        PORTS
3b116082fe57   postgres:17.4 "docker-entrypoint.s..." About a minute ago Up About a minute 0.0.0.0
:5432->5432/tcp, [::]:5432->5432/tcp dbhost
ndalessandro@ARSO20242:~/prac20242-orig$
```

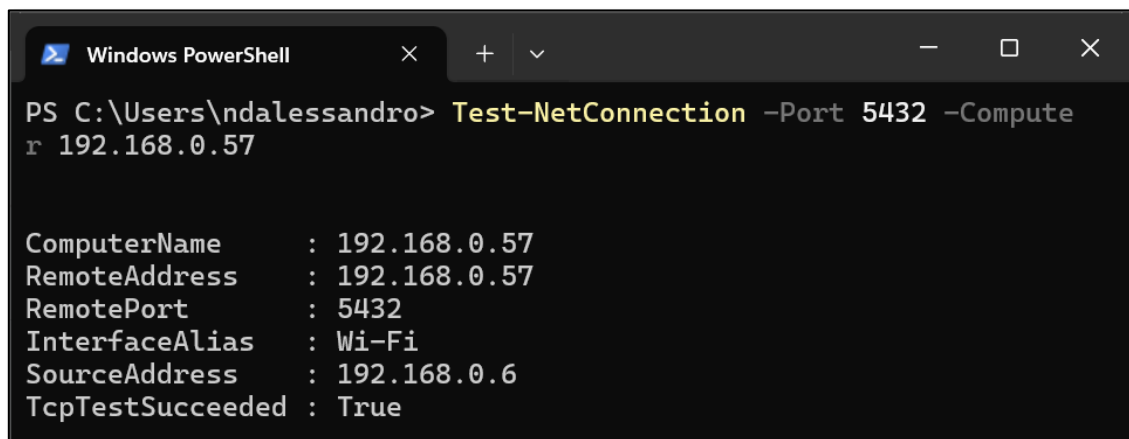
Verification of port 5432 listening from the VM

```
ndalessandro@ARSO20242: ~/prac20242-orig
ndalessandro@ARSO20242: ~/prac20242-orig x ndalessandro@ARSO20242: ~/prac20242-orig x
ndalessandro@ARSO20242:~/prac20242-orig$ sudo docker ps
[sudo] password for ndalessandro:
CONTAINER ID   IMAGE          COMMAND                  CREATED        STATUS        PORTS
3b116082fe57   postgres:17.4 "docker-entrypoint.s..." About a minute ago Up About a minute 0.0.0.0
:5432->5432/tcp, [::]:5432->5432/tcp dbhost
ndalessandro@ARSO20242:~/prac20242-orig$
```

```
ndalessandro@ARSO20242: ~/prac20242-orig
ndalessandro@ARSO20242: ~/prac20242-orig x ndalessandro@ARSO20242: ~/prac20242-orig x
ndalessandro@ARSO20242:~/prac20242-orig$ sudo netstat -an | grep 5432
tcp        0      0 0.0.0.0:5432          0.0.0.0:*           LISTEN
tcp6       0      0 :::5432             :::*                 LISTEN
ndalessandro@ARSO20242:~/prac20242-orig$
```

IP_HOST: 192.168.0.6
IP_GUEST: 192.168.0.57

Verification from the host using PowerShell



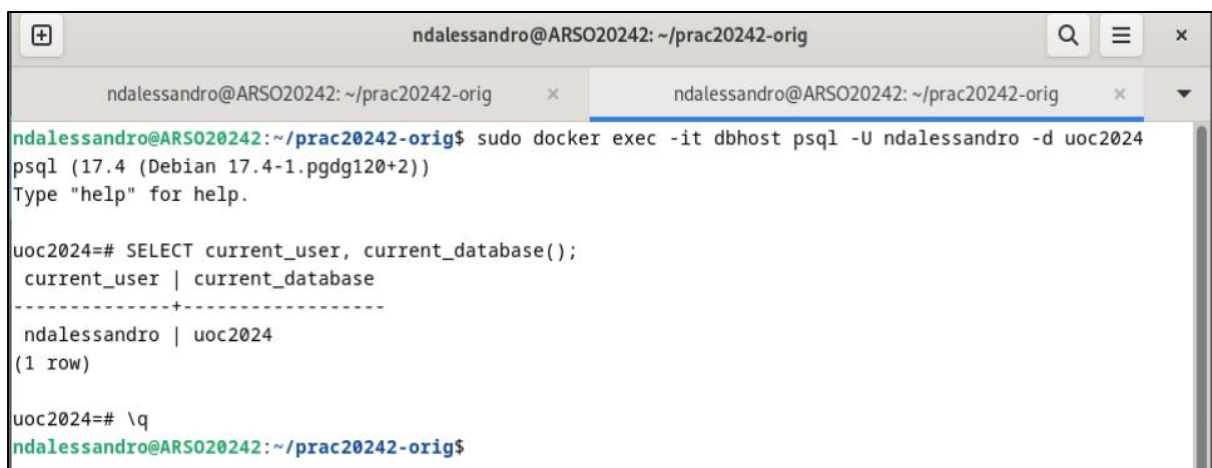
```
Windows PowerShell
PS C:\Users\ndalessandro> Test-NetConnection -Port 5432 -ComputerName 192.168.0.57

ComputerName      : 192.168.0.57
RemoteAddress     : 192.168.0.57
RemotePort        : 5432
InterfaceAlias    : Wi-Fi
SourceAddress     : 192.168.0.6
TcpTestSucceeded  : True
```

The result indicates that the port was accessible from the PC by confirming the service was correctly exposed.

Successful access to the database from the **psql** client:

Once connected, I was confirmed that the user **ndalessandro** could successfully access to the **uoc2024** database:



```
ndalessandro@ARSO20242: ~/prac20242-orig
ndalessandro@ARSO20242:~/prac20242-orig$ sudo docker exec -it dbhost psql -U ndalessandro -d uoc2024
psql (17.4 (Debian 17.4-1.pgdg120+2))
Type "help" for help.

uoc2024=# SELECT current_user, current_database();
 current_user | current_database 
-----+-----
 ndalessandro | uoc2024
(1 row)

uoc2024=# \q
ndalessandro@ARSO20242:~/prac20242-orig$
```

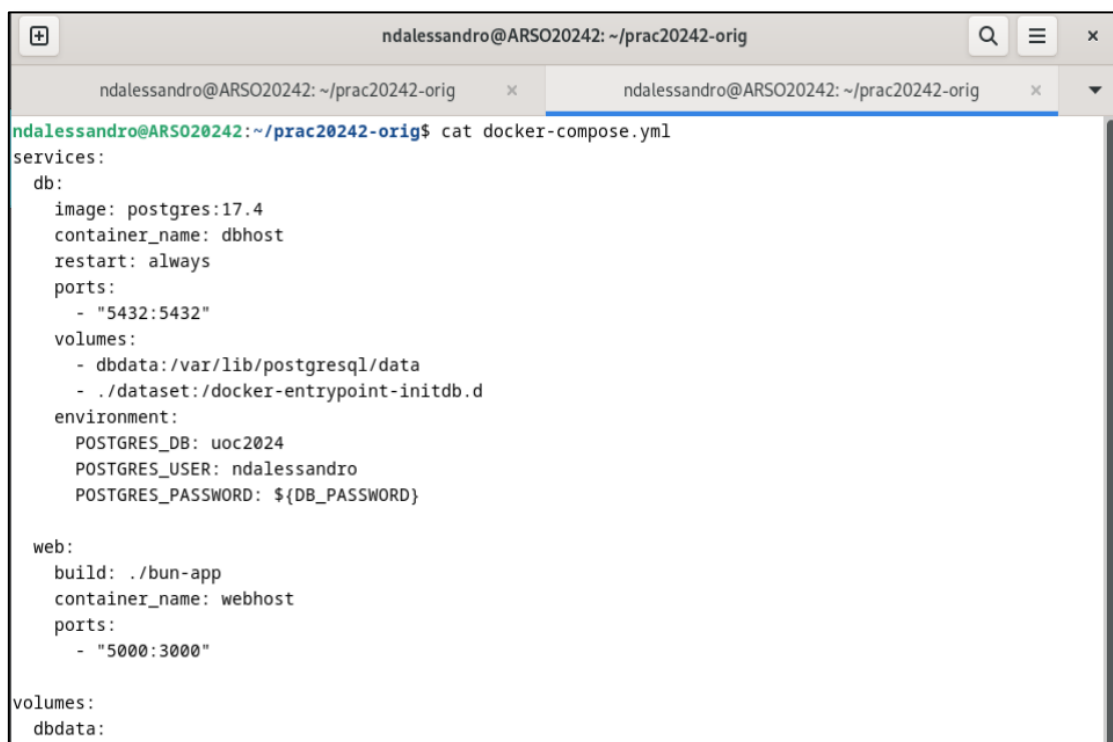
2.2 Bun Web Server Configuration

Once the database service was deployed, a new service named **web** was added to the **docker-compose.yml** file.

This service is based on the **oven/bun:latest** image. To run the web content located in the **app/** folder, Bun is used directly with the command **bun src/app.ts**, which launches the application server.

Port exposure is configured from **internal port 3000** to **external port 5000**.

Updated **docker-compose.yml** file

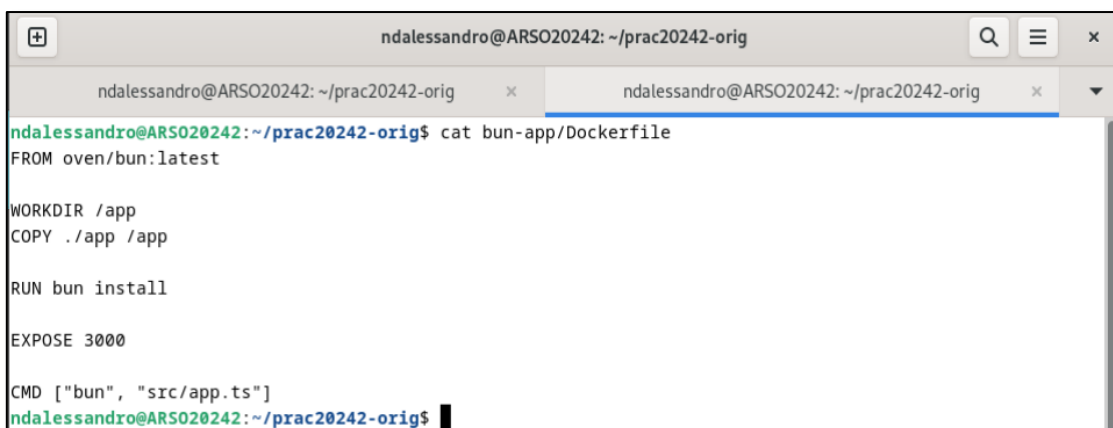


```
ndalessandro@ARSO20242: ~/prac20242-orig
ndalessandro@ARSO20242: ~/prac20242-orig$ cat docker-compose.yml
services:
  db:
    image: postgres:17.4
    container_name: dbhost
    restart: always
    ports:
      - "5432:5432"
    volumes:
      - dbdata:/var/lib/postgresql/data
      - ./dataset:/docker-entrypoint-initdb.d
    environment:
      POSTGRES_DB: uoc2024
      POSTGRES_USER: ndalessandro
      POSTGRES_PASSWORD: ${DB_PASSWORD}

  web:
    build: ./bun-app
    container_name: webhost
    ports:
      - "5000:3000"

volumes:
  dbdata:
```

bun-app/Dockerfile file



```
ndalessandro@ARSO20242: ~/prac20242-orig
ndalessandro@ARSO20242: ~/prac20242-orig$ cat bun-app/Dockerfile
FROM oven/bun:latest

WORKDIR /app
COPY ./app /app

RUN bun install

EXPOSE 3000

CMD ["bun", "src/app.ts"]
ndalessandro@ARSO20242: ~/prac20242-orig$
```


IP_HOST: 192.168.0.6
IP_GUEST: 192.168.0.57

Running containers

```
ndalessandro@ARSO20242: ~/prac20242-orig
ndalessandro@ARSO20242: ~/prac20242-orig
ndalessandro@ARSO20242:~/prac20242-orig$ sudo docker ps
CONTAINER ID   IMAGE                                COMMAND                  CREATED        STATUS        PORTS
bf7844685206   postgres:17.4                       "docker-entrypoint.s..." 10 seconds ago Up 10 seconds  0.0.0.0:54
32->5432/tcp, [::]:5432->5432/tcp dbhost
8892099d2b20   prac20242-orig-web                 "/usr/local/bin/dock..." 10 seconds ago Up 10 seconds  0.0.0.0:50
00->3000/tcp, [::]:5000->3000/tcp webhost
ndalessandro@ARSO20242:~/prac20242-orig$
```

Successful access from Windows browser (Host)



2.3 Connection Between Bun Server and Postgres

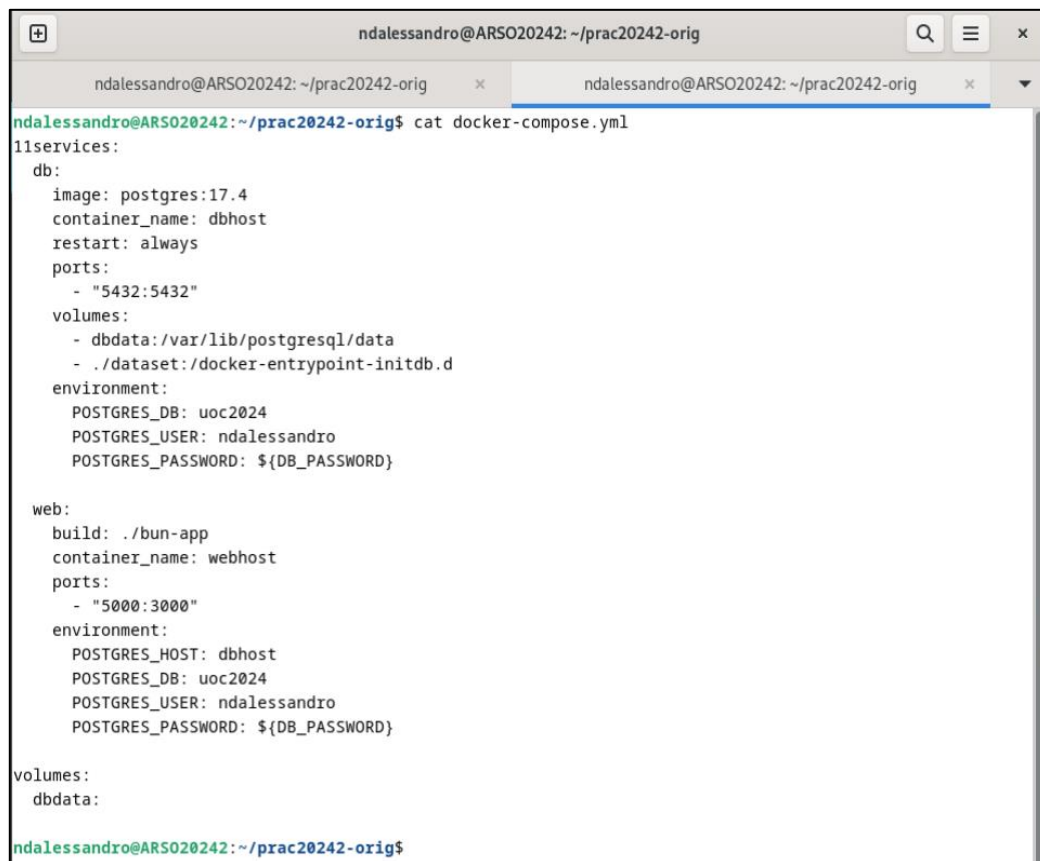
The connection between the web server (**webhost** container) and the PostgreSQL database (**dbhost** container) was established. This integration allows to the frontend to recover the information directly from the database.

In the **src/app.ts** file, the following environment variables were required:

- **POSTGRES_HOST**
- **POSTGRES_DB**
- **POSTGRES_USER**
- **POSTGRES_PASSWORD**

These variables were added to the web service in the **docker-compose.yml**. The value **POSTGRES_HOST** was assigned as **dbhost** corresponding to the database container service within the same Docker Compose stack.

Updated **docker-compose.yml** file:



```
ndalessandro@ARSO20242: ~/prac20242-orig
ndalessandro@ARSO20242:~/prac20242-orig$ cat docker-compose.yml
11services:
  db:
    image: postgres:17.4
    container_name: dbhost
    restart: always
    ports:
      - "5432:5432"
    volumes:
      - dbdata:/var/lib/postgresql/data
      - ./dataset:/docker-entrypoint-initdb.d
    environment:
      POSTGRES_DB: uoc2024
      POSTGRES_USER: ndalessandro
      POSTGRES_PASSWORD: ${DB_PASSWORD}

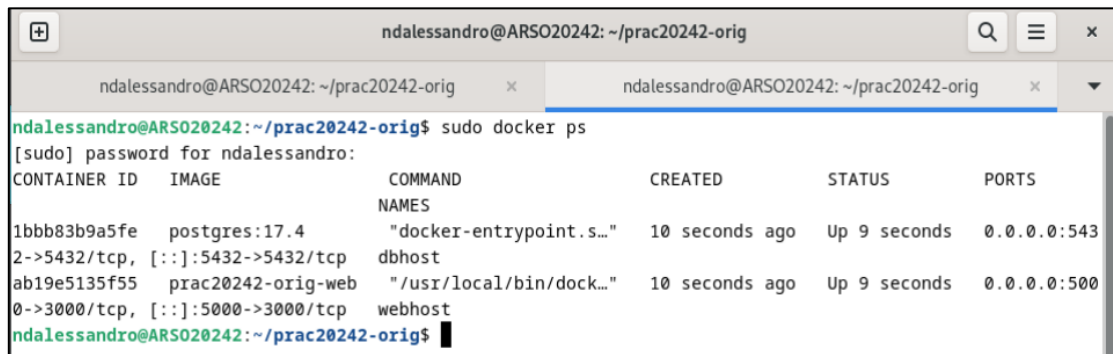
  web:
    build: ./bun-app
    container_name: webhost
    ports:
      - "5000:3000"
    environment:
      POSTGRES_HOST: dbhost
      POSTGRES_DB: uoc2024
      POSTGRES_USER: ndalessandro
      POSTGRES_PASSWORD: ${DB_PASSWORD}

volumes:
  dbdata:
```

IP_HOST: 192.168.0.6
IP_GUEST: 192.168.0.57

After updating the configuration, the stack was restarted with the command **docker compose up --build**.

Running containers:



```
ndalessandro@ARSO20242: ~/prac20242-orig
ndalessandro@ARSO20242: ~/prac20242-orig$ sudo docker ps
[sudo] password for ndalessandro:
CONTAINER ID   IMAGE          COMMAND                  CREATED        STATUS        PORTS
1bbb83b9a5fe   postgres:17.4  "docker-entrypoint.s..." 10 seconds ago Up 9 seconds  0.0.0.0:5432->5432/tcp, [::]:5432->5432/tcp dbhost
ab19e5135f55   prac20242-orig-web "/usr/local/bin/dock..." 10 seconds ago Up 9 seconds  0.0.0.0:5000->3000/tcp, [::]:5000->3000/tcp webhost
ndalessandro@ARSO20242: ~/prac20242-orig$
```

Web access via browser:

Accessing to the <http://192.168.0.57:5000> URL from the host browse, we can check that the site successfully displayed data from the database, confirming that the connection was properly established using the configured environment variables.

State of ports exposed from the VM:



```
ndalessandro@ARSO20242: ~/prac20242-orig
ndalessandro@ARSO20242: ~/prac20242-orig$ sudo netstat -tulnp | grep LISTEN
tcp        0      0 0.0.0.0:5432          0.0.0.0:*           LISTEN      4066/docker-proxy
tcp        0      0 0.0.0.0:5000          0.0.0.0:*           LISTEN      4098/docker-proxy
tcp        0      0 127.0.0.1:631         0.0.0.0:*           LISTEN      591/cupsd
tcp6       0      0 :::5432              :::*                LISTEN      4074/docker-proxy
tcp6       0      0 :::5000              :::*                LISTEN      4107/docker-proxy
tcp6       0      0 :::1:631             :::*                LISTEN      591/cupsd
ndalessandro@ARSO20242: ~/prac20242-orig$
```

IP_HOST: 192.168.0.6
IP_GUEST: 192.168.0.57

Verification from the host using PowerShell

```
Windows PowerShell
PS C:\Users\ndalessandro> Test-NetConnection -Port 5000 -ComputerName 192.168.0.57

ComputerName      : 192.168.0.57
RemoteAddress     : 192.168.0.57
RemotePort        : 5000
InterfaceAlias    : Wi-Fi
SourceAddress     : 192.168.0.6
TcpTestSucceeded  : True
```

Successful access to the web



3.1 HTTP Proxy

A reverse proxy based on **HAProxy** was configured. The container was built from scratch with a Dockerfile and a **haproxy.cfg** configuration file, located in the **reverse-proxy/** directory.

I started from scratch by creating a specific container called **proxyhost**, which included its own Dockerfile and a **haproxy.cfg** configuration file, both located in the **reverse-proxy/** folder. Different to other services that are easier to run, in this case it was necessary to fine-tune the routes and ports, as the proxy had to receive HTTP requests on port 80 and forward them to the webhost container, where the Bun server runs.

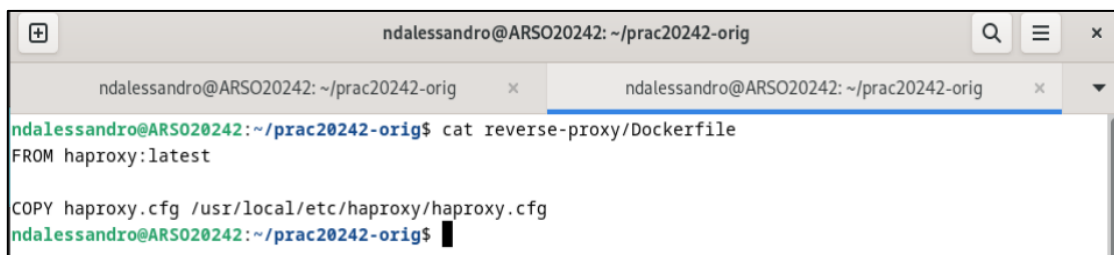
To make all this work I have defined two different networks in the **docker-compose.yml** file:

- **web-network**: connects **proxyhost** and webhost
- **db-network**: connects webhost and dbhost

This allowed for a clear separation between the various services, respecting the principle of minimal exposure: the database container remains inaccessible from the outside, while the frontend is only exposed through the proxy.

After rebuilding the containers and deploying the stack, I verified that port 80 was listening on the VM and that it could be accessed correctly from the host. I also checked that the containers were connected to the appropriate networks, which confirmed that the network rules were implemented correctly. At this part, access from the host browser correctly redirect to the web server using the proxy, achieving the required basic functionality.

Reverse-proxy Dockerfile

A terminal window screenshot showing the content of a Dockerfile. The terminal title is 'ndalessandro@ARSO20242: ~/prac20242-orig'. The command 'cat reverse-proxy/Dockerfile' has been executed, displaying the following content:

```
ndalessandro@ARSO20242:~/prac20242-orig$ cat reverse-proxy/Dockerfile
FROM haproxy:latest

COPY haproxy.cfg /usr/local/etc/haproxy/haproxy.cfg
ndalessandro@ARSO20242:~/prac20242-orig$
```

IP_HOST: 192.168.0.6
IP_GUEST: 192.168.0.57

Updated [docker-compose.yml](#) file, which includes the db, web, and proxy services correctly associated with their networks



```
ndalessandro@ARSO20242: ~/prac20242-orig
ndalessandro@ARSO20242:~/prac20242-orig$ cat docker-compose.yml
services:
  db:
    image: postgres:17.4
    container_name: dbhost
    restart: always
    ports:
      - "5432:5432"
    volumes:
      - dbdata:/var/lib/postgresql/data
      - ./dataset:/docker-entrypoint-initdb.d
    environment:
      POSTGRES_DB: uoc2024
      POSTGRES_USER: ndalessandro
      POSTGRES_PASSWORD: ${DB_PASSWORD}
    networks:
      - db-network

  web:
    build: ./bun-app
    container_name: webhost
    ports:
      - "5000:3000"
    environment:
      POSTGRES_HOST: dbhost
      POSTGRES_DB: uoc2024
      POSTGRES_USER: ndalessandro
      POSTGRES_PASSWORD: ${DB_PASSWORD}
    networks:
      - web-network
      - db-network

  proxy:
    build: ./reverse-proxy
    container_name: proxyhost
    ports:
      - "80:80"
    networks:
      - web-network

volumes:
  dbdata:

networks:
  web-network:
    name: web-network
  db-network:
    name: db-network
ndalessandro@ARSO20242:~/prac20242-orig$
```

IP_HOST: 192.168.0.6
IP_GUEST: 192.168.0.57

Reverse-proxy haproxy.cfg file

```
ndalessandro@ARSO20242: ~/prac20242-orig
ndalessandro@ARSO20242: ~/prac20242-orig$ cat reverse-proxy/haproxy.cfg
global
    log stdout format raw local0

defaults
    log global
    mode http
    timeout connect 5000ms
    timeout client 50000ms
    timeout server 50000ms

frontend http_front
    bind *:80
    default_backend http_back

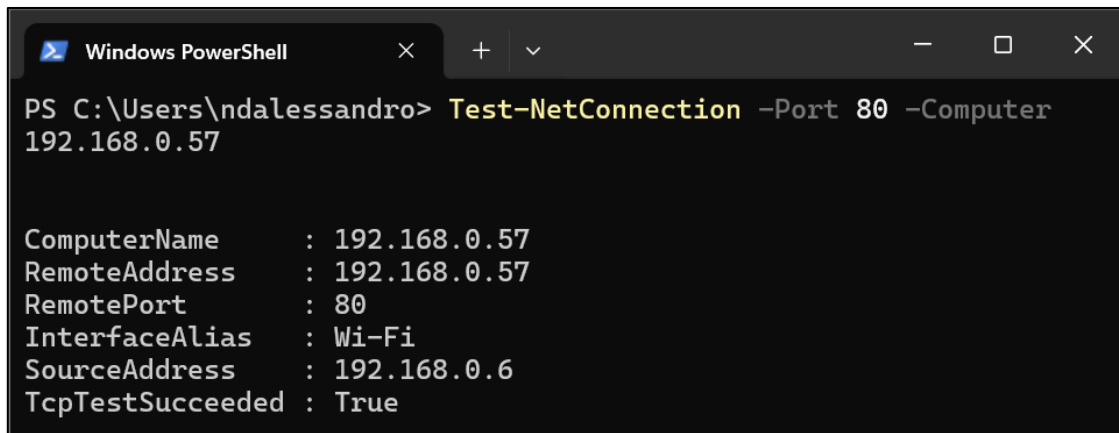
backend http_back
    server websrv webhost:3000 check
ndalessandro@ARSO20242: ~/prac20242-orig$
```

Verify that port 80 is listening on the VM

```
ndalessandro@ARSO20242: ~/prac20242-orig
ndalessandro@ARSO20242: ~/prac20242-orig$ sudo netstat -a | grep http
[sudo] password for ndalessandro:
tcp        0      0 0.0.0.0:http          0.0.0.0:*             LISTEN
tcp        0      0 ARSO20242:46820      ec2-3-94-224-37.c:https ESTABLISHED
tcp6       0      0 [::]:http            [::]:*                LISTEN
ndalessandro@ARSO20242: ~/prac20242-orig$
```

IP_HOST: 192.168.0.6
IP_GUEST: 192.168.0.57

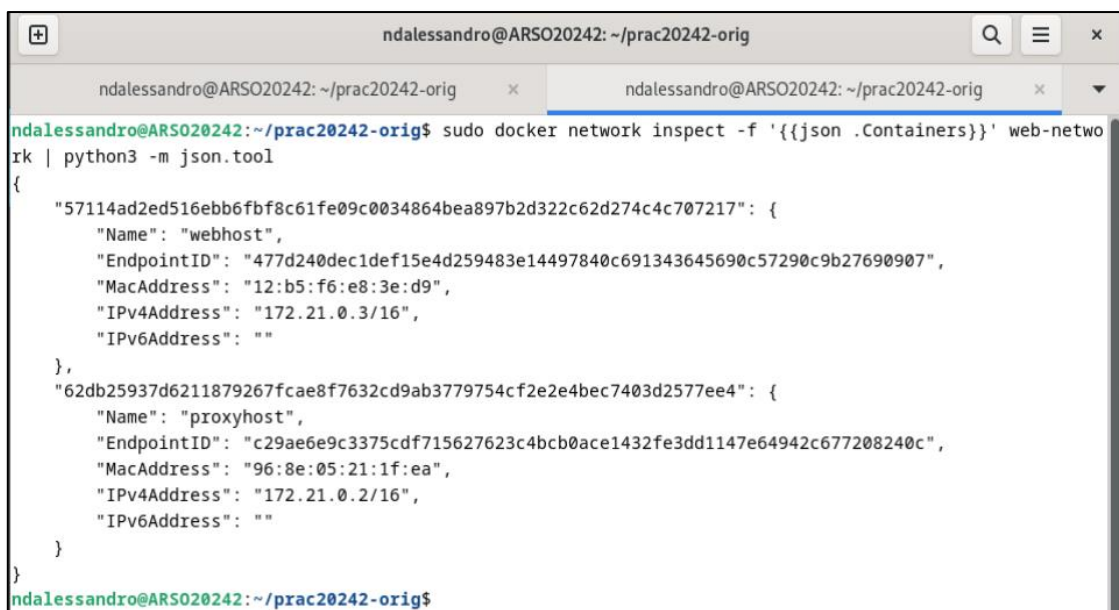
Accessibility is verified from windows PowerShell (host)



```
Windows PowerShell
PS C:\Users\ndalessandro> Test-NetConnection -Port 80 -Computer 192.168.0.57

ComputerName      : 192.168.0.57
RemoteAddress     : 192.168.0.57
RemotePort        : 80
InterfaceAlias    : Wi-Fi
SourceAddress     : 192.168.0.6
TcpTestSucceeded  : True
```

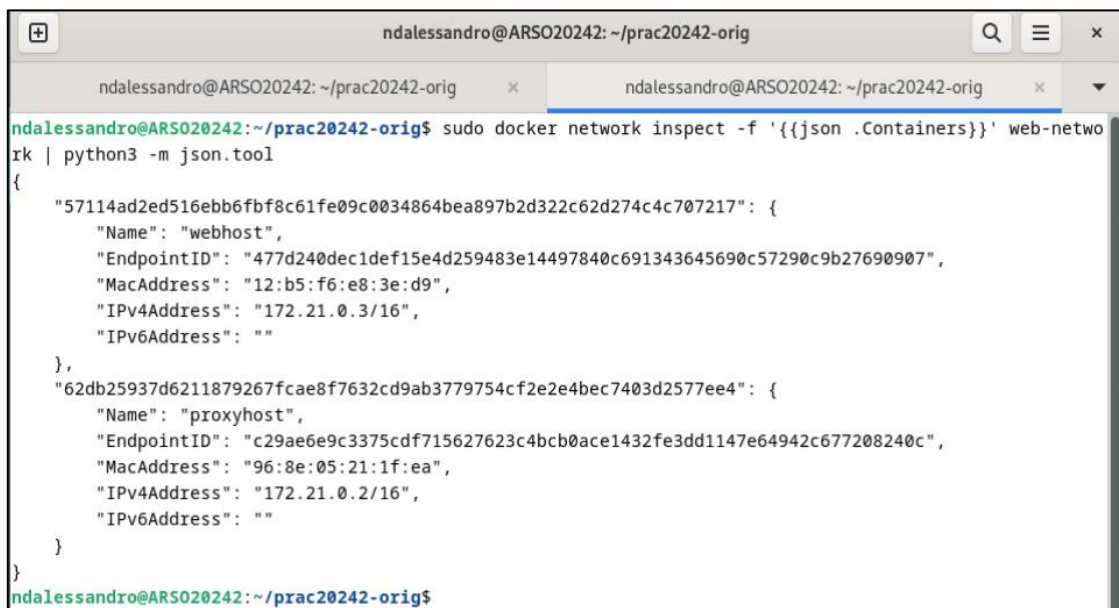
Containers connected to the web-network are verified



```
ndalessandro@ARSO20242: ~/prac20242-orig
ndalessandro@ARSO20242: ~/prac20242-orig$ sudo docker network inspect -f '{{json .Containers}}' web-netwo
rk | python3 -m json.tool
{
  "57114ad2ed516ebb6fbf8c61fe09c0034864bea897b2d322c62d274c4c707217": {
    "Name": "webhost",
    "EndpointID": "477d240dec1def15e4d259483e14497840c691343645690c57290c9b27690907",
    "MacAddress": "12:b5:f6:e8:3e:d9",
    "IPv4Address": "172.21.0.3/16",
    "IPv6Address": ""
  },
  "62db25937d6211879267fcae8f7632cd9ab3779754cf2e2e4bec7403d2577ee4": {
    "Name": "proxyhost",
    "EndpointID": "c29ae6e9c3375cdf715627623c4bcb0ace1432fe3dd1147e64942c677208240c",
    "MacAddress": "96:8e:05:21:1f:ea",
    "IPv4Address": "172.21.0.2/16",
    "IPv6Address": ""
  }
}
```


IP_HOST: 192.168.0.6
IP_GUEST: 192.168.0.57

Check containers connected to db-network



```
ndalessandro@ARSO20242: ~/prac20242-orig
ndalessandro@ARSO20242: ~/prac20242-orig$ sudo docker network inspect -f '{{json .Containers}}' web-network
[{"57114ad2ed516ebb6fbf8c61fe09c0034864bea897b2d322c62d274c4c707217": {
  "Name": "webhost",
  "EndpointID": "477d240dec1def15e4d259483e14497840c691343645690c57290c9b27690907",
  "MacAddress": "12:b5:f6:e8:3e:d9",
  "IPv4Address": "172.21.0.3/16",
  "IPv6Address": ""
},
{"62db25937d6211879267fcae8f7632cd9ab3779754cf2e2e4bec7403d2577ee4": {
  "Name": "proxyhost",
  "EndpointID": "c29ae6e9c3375cdf715627623c4bcb0ace1432fe3dd1147e64942c677208240c",
  "MacAddress": "96:8e:05:21:1f:ea",
  "IPv4Address": "172.21.0.2/16",
  "IPv6Address": ""
}
}]
ndalessandro@ARSO20242: ~/prac20242-orig$
```

As a final result, the proxy has been successfully deployed. It is verified that it responds on port 80 from the host and routes requests to the web server.

3.2 HTTPS Proxy

The next step was to add HTTPS support, which is essential to comply with minimum security practices in any web environment. To do this, I generated a self-signed certificate with OpenSSL, including my UOC email as part of the certificate subject. I then combined the private key and certificate into a **.pem** file, which was copied to the proxy container.

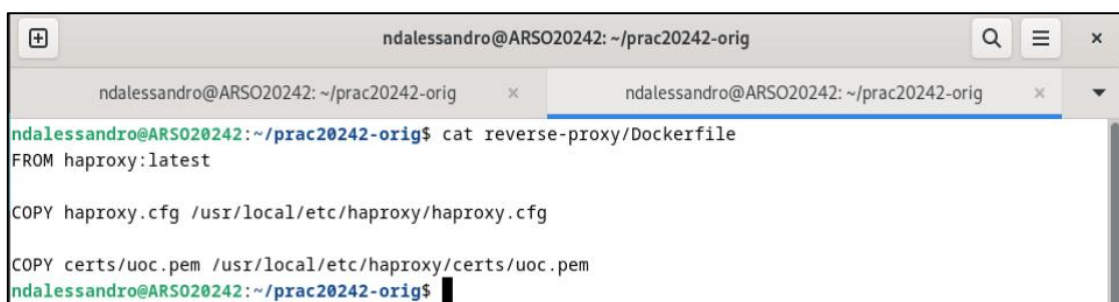
```
mkdir -p reverse-proxy/certs
openssl req -x509 -nodes -days 365 -newkey rsa:2048 \
  -keyout certs/uoc.key -out certs/uoc.crt \
  -subj "/CN=localhost/emailAddress=ndalessandro@uoc.edu"

cat certs/uoc.crt certs/uoc.key > certs/uoc.pem
```

I updated both the **Dockerfile** and **haproxy.cfg** so that the container listens on port 443 and correctly handles secure connections.

I also added automatic **HTTP** to **HTTPS** redirection logic, so that any connection attempts on port 80 are forwarded to 443.

Updated reverse-proxy Dockerfile



```
ndalessandro@ARSO20242: ~/prac20242-orig
ndalessandro@ARSO20242: ~/prac20242-orig$ cat reverse-proxy/Dockerfile
FROM haproxy:latest

COPY haproxy.cfg /usr/local/etc/haproxy/haproxy.cfg

COPY certs/uoc.pem /usr/local/etc/haproxy/certs/uoc.pem
ndalessandro@ARSO20242: ~/prac20242-orig$
```

Updated reverse-proxy haproxy.cfg file

```
ndalessandro@ARSO20242: ~/prac20242-orig
ndalessandro@ARSO20242: ~/prac20242-orig$ cat reverse-proxy/haproxy.cfg
global
    log stdout format raw local0

defaults
    log     global
    mode    http
    timeout connect 5000ms
    timeout client 50000ms
    timeout server 50000ms

frontend http_front
    bind *:80
    redirect scheme https code 302 if !{ ssl_fc }

frontend https_front
    bind *:443 ssl crt /usr/local/etc/haproxy/certs/uoc.pem
    default_backend http_back

backend http_back
    server webserv webhost:3000 check
ndalessandro@ARSO20242: ~/prac20242-orig$
```

Running containers

```
ndalessandro@ARSO20242: ~/prac20242-orig
ndalessandro@ARSO20242: ~/prac20242-orig$ sudo docker ps
[sudo] password for ndalessandro:
CONTAINER ID   IMAGE                                COMMAND                                  CREATED        STATUS        PORTS
NAMES
1ee9b7c0d638   postgres:17.4                      "docker-entrypoint.s..."  48 seconds ago Up 46 seconds 0.0.0.0:
5432->5432/tcp, [::]:5432->5432/tcp  dbhost
a6d1fc0defe7   prac20242-orig-web                 "/usr/local/bin/dock..."  48 seconds ago Up 46 seconds 0.0.0.0:
5000->3000/tcp, [::]:5000->3000/tcp  webhost
11a5367209ca   prac20242-orig-proxy               "docker-entrypoint.s..."  48 seconds ago Up 46 seconds 0.0.0.0:
80->80/tcp, [::]:80->80/tcp, 0.0.0.0:443->443/tcp, [::]:443->443/tcp proxyhost
ndalessandro@ARSO20242: ~/prac20242-orig$
```

IP_HOST: 192.168.0.6
IP_GUEST: 192.168.0.57

Updated [docker-compose.yml](#) file, including ports 80 and 443



```
ndalessandro@ARSO20242: ~/prac20242-orig
ndalessandro@ARSO20242:~/prac20242-orig$ cat docker-compose.yml
services:
  db:
    image: postgres:17.4
    container_name: dbhost
    restart: always
    ports:
      - "5432:5432"
    volumes:
      - dbdata:/var/lib/postgresql/data
      - ./dataset:/docker-entrypoint-initdb.d
    environment:
      POSTGRES_DB: uoc2024
      POSTGRES_USER: ndalessandro
      POSTGRES_PASSWORD: ${DB_PASSWORD}
    networks:
      - db-network

  web:
    build: ./bun-app
    container_name: webhost
    ports:
      - "5000:3000"
    environment:
      POSTGRES_HOST: dbhost
      POSTGRES_DB: uoc2024
      POSTGRES_USER: ndalessandro
      POSTGRES_PASSWORD: ${DB_PASSWORD}
    networks:
      - web-network
      - db-network

  proxy:
    build: ./reverse-proxy
    container_name: proxyhost
    ports:
      - "80:80"
      - "443:443"
    networks:
      - web-network

volumes:
  dbdata:

networks:
  web-network:
    name: web-network
  db-network:
    name: db-network
ndalessandro@ARSO20242:~/prac20242-orig$
```

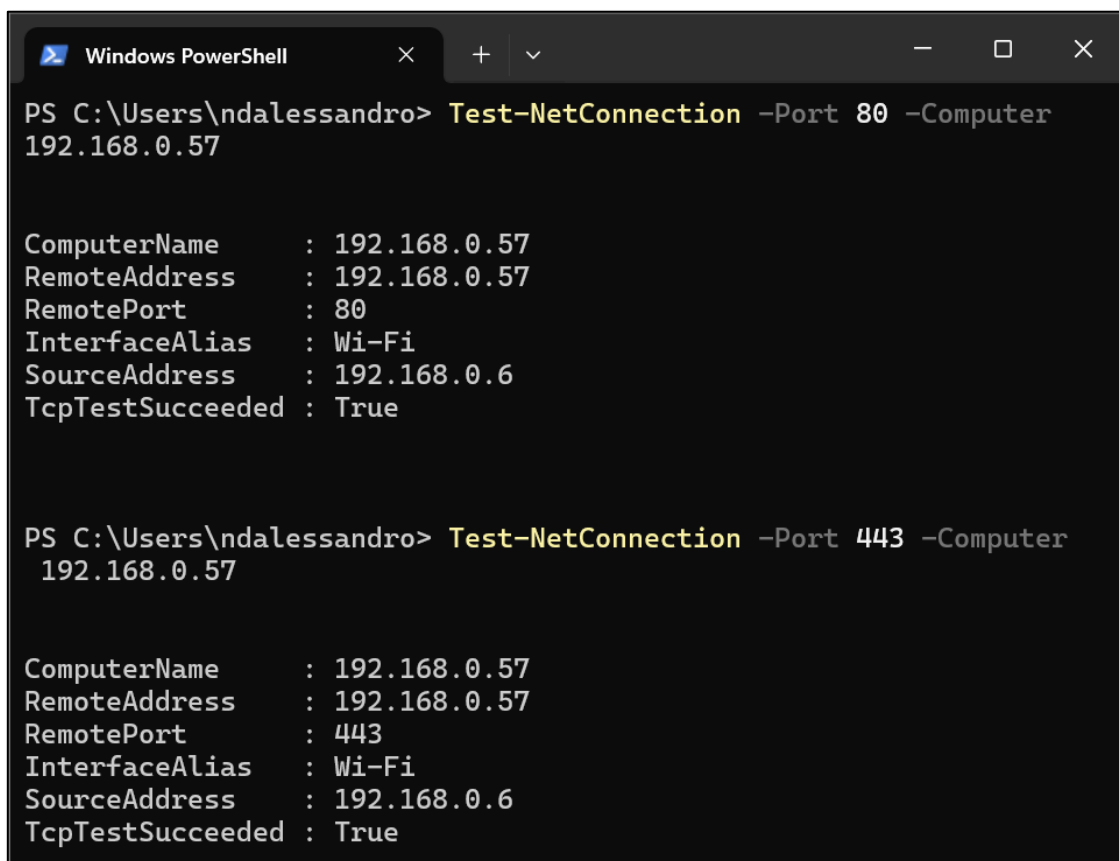
IP_HOST: 192.168.0.6
IP_GUEST: 192.168.0.57

Verify the operation of ports 80 and 443 listening on the VM



```
ndalessandro@ARSO20242: ~/prac20242-orig
ndalessandro@ARSO20242: ~/prac20242-orig$ sudo netstat -a | grep http
tcp        0      0 0.0.0.0:https        0.0.0.0:*           LISTEN
tcp        0      0 0.0.0.0:http         0.0.0.0:*           LISTEN
tcp6       0      0 :::https            :::*                 LISTEN
tcp6       0      0 :::http             :::*                 LISTEN
ndalessandro@ARSO20242: ~/prac20242-orig$
```

Accessibility is verified from the Host on both ports



```
Windows PowerShell
PS C:\Users\ndalessandro> Test-NetConnection -Port 80 -Computer 192.168.0.57

ComputerName      : 192.168.0.57
RemoteAddress     : 192.168.0.57
RemotePort        : 80
InterfaceAlias    : Wi-Fi
SourceAddress     : 192.168.0.6
TcpTestSucceeded  : True

PS C:\Users\ndalessandro> Test-NetConnection -Port 443 -Computer 192.168.0.57

ComputerName      : 192.168.0.57
RemoteAddress     : 192.168.0.57
RemotePort        : 443
InterfaceAlias    : Wi-Fi
SourceAddress     : 192.168.0.6
TcpTestSucceeded  : True
```

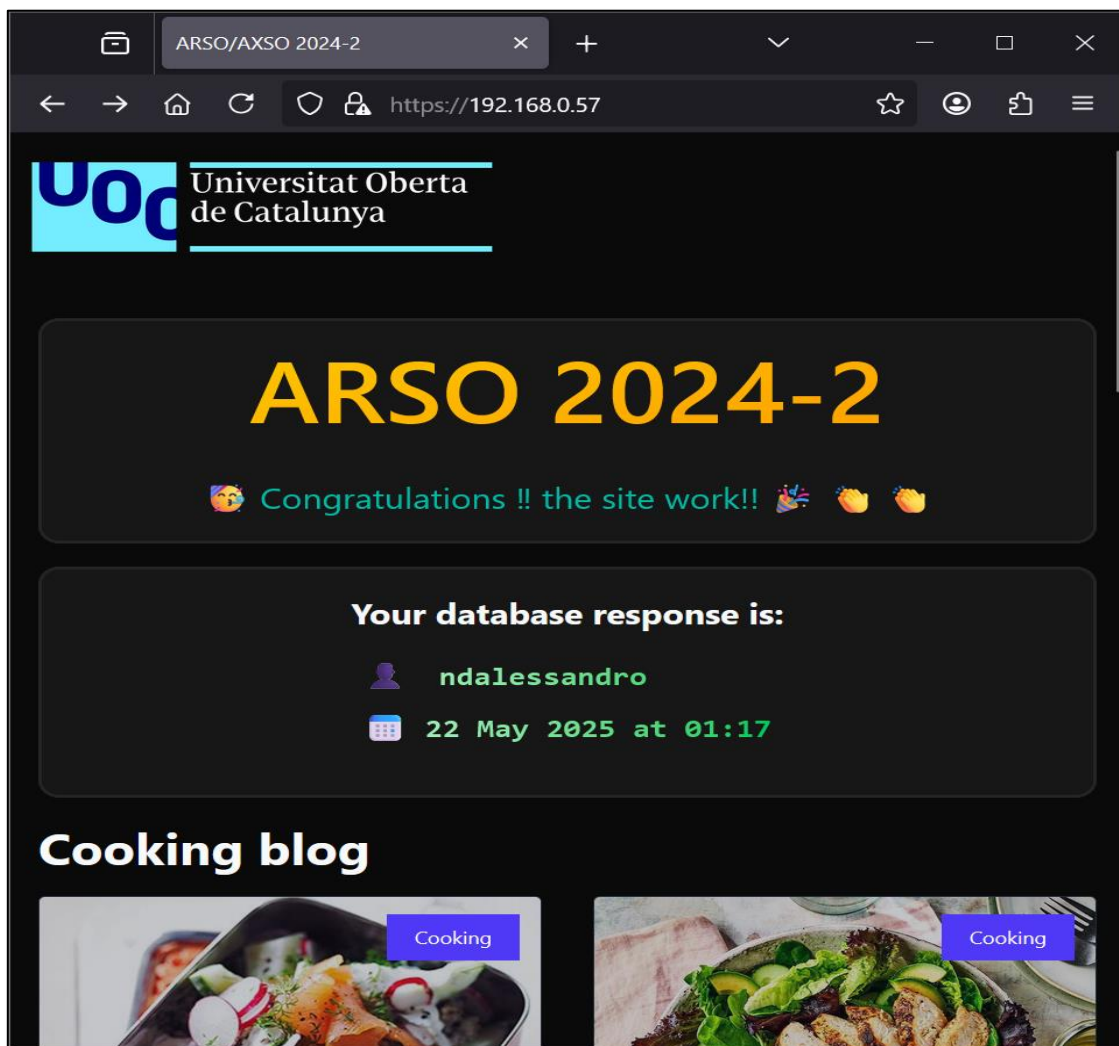
IP_HOST: 192.168.0.6
IP_GUEST: 192.168.0.57

The HTTP to HTTPS redirection (302) is verified with curl.exe in PowerShell

```
Windows PowerShell
PS C:\Users\ndalessandro> curl.exe -I http://192.168.0.57/
HTTP/1.1 302 Found
content-length: 0
location: https://192.168.0.57/
cache-control: no-cache

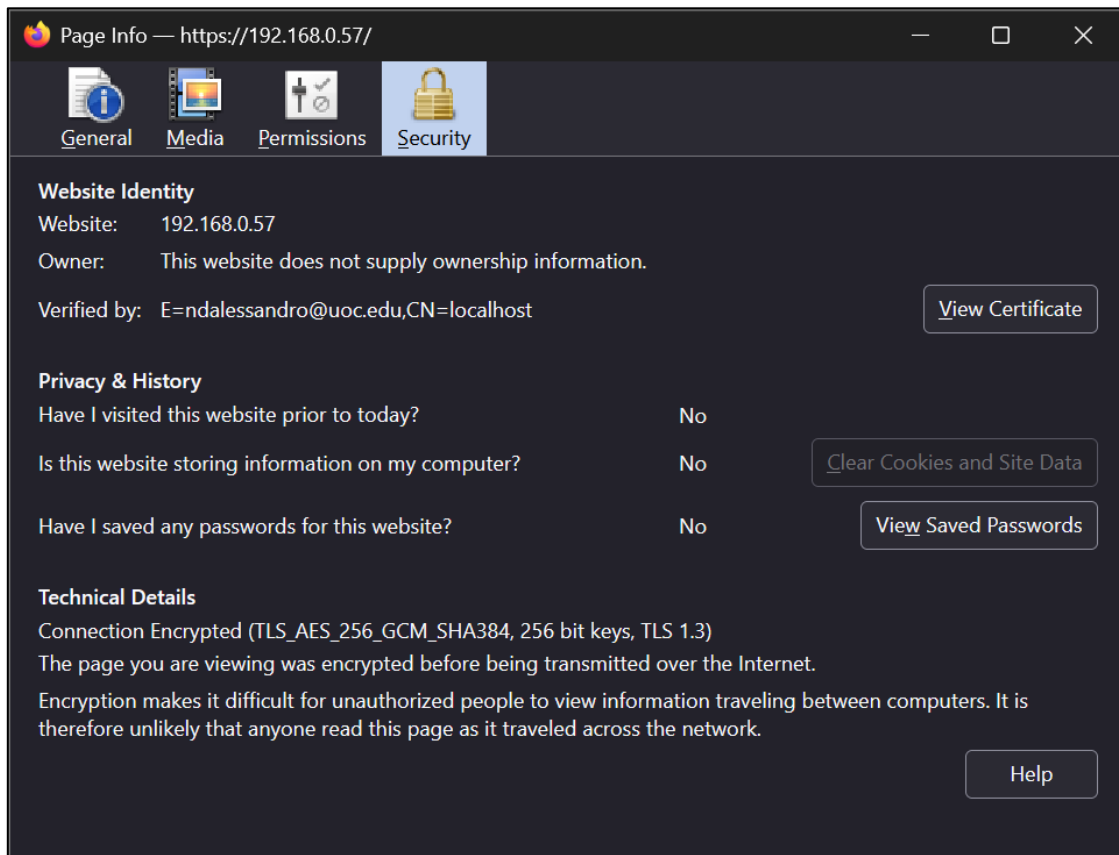
PS C:\Users\ndalessandro> |
```

Access to the website via HTTPS is verified from the Host's browser



IP_HOST: 192.168.0.6
IP_GUEST: 192.168.0.57

Certificate details are verified in the browser with email present



Everything worked as expected, and the environment became much more complete. External access is now limited to the proxy, which is responsible for routing, securing, and controlling traffic to internal services.

For this exercise, the goal was to deploy a multi-layer application in a Kubernetes environment.

I first installed **Minikube** on the virtual machine, choosing Docker as the driver to take advantage of the already configured environment. After adding the user to the Docker group and rebooting, I was able to start the cluster without any issues with:

```
minikube start --driver=docker
```

Then I accessed the Kubernetes dashboard:

```
minikube dashboard
```

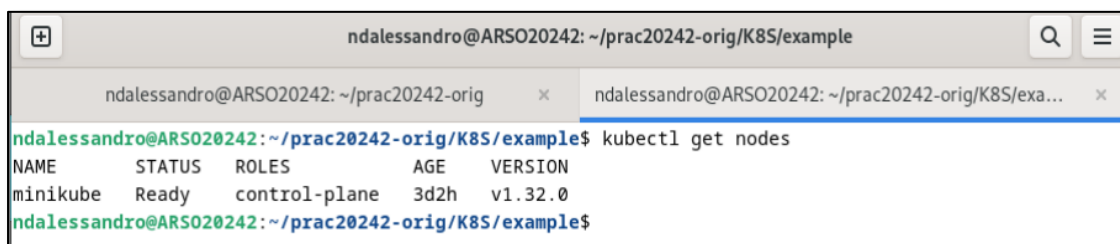
From the **K8S/example** directory, I deployed all the resources with:

```
kubectl apply -f .
```

The application includes several pods: **vote**, **result**, **db**, **redis**, and **worker**. I checked their status with:

```
kubectl get pods
```

With this I confirmed that they were all in **Running** status, indicating that the deployment was successful.



```
ndalessandro@ARSO20242: ~/prac20242-orig/K8S/example
ndalessandro@ARSO20242: ~/prac20242-orig/K8S/example$ kubectl get nodes
```

NAME	STATUS	ROLES	AGE	VERSION
minikube	Ready	control-plane	3d2h	v1.32.0

```
ndalessandro@ARSO20242: ~/prac20242-orig/K8S/example$
```



```
ndalessandro@ARSO20242: ~/prac20242-orig/K8S/example
ndalessandro@ARSO20242: ~/prac20242-orig x ndalessandro@ARSO20242: ~/prac20242-orig/K8S/exa... x
ndalessandro@ARSO20242:~/prac20242-orig/K8S/example$ kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
db-74574d66dd-k2rkh                 1/1     Running   1 (10m ago) 3d2h
redis-6c5fb9c4b7-fgvz8              1/1     Running   1 (10m ago) 3d2h
result-5f99548f7c-lhh2w             1/1     Running   1 (10m ago) 3d2h
vote-5d74dcd7c7-bzpdz               1/1     Running   1 (10m ago) 3d2h
worker-6f5f6cdd56-7wpqc             1/1     Running   1 (10m ago) 3d2h
ndalessandro@ARSO20242:~/prac20242-orig/K8S/example$ kubectl get svc
NAME            TYPE        CLUSTER-IP    EXTERNAL-IP  PORT(S)          AGE
db              ClusterIP   10.110.66.230 <none>        5432/TCP         3d2h
kubernetes      ClusterIP   10.96.0.1     <none>        443/TCP          3d2h
redis           ClusterIP   10.109.213.48 <none>        6379/TCP         3d2h
result          NodePort    10.99.200.141 <none>        8081:31001/TCP   3d2h
vote            NodePort    10.105.180.183 <none>        8080:31000/TCP   3d2h
ndalessandro@ARSO20242:~/prac20242-orig/K8S/example$ minikube service vote --url
http://192.168.49.2:31000
ndalessandro@ARSO20242:~/prac20242-orig/K8S/example$ minikube service result --url
http://192.168.49.2:31001
ndalessandro@ARSO20242:~/prac20242-orig/K8S/example$
```

Comments on the dashboard sections

- **Cluster:** Represents the entire Kubernetes environments such as nodes, pods, services, networks, etc. It is the complete infrastructure that manages the apps.
- **Cluster Role Binding:** Assigns global permissions to users or service accounts. It is used when something needs access to all namespaces.
- **Namespace:** Allows to organize resources into isolated groups within the cluster. Very useful for separating environments or projects.
- **Network Policies:** Rules that control traffic between pods. They are used to define who can communicate with whom within the cluster.
- **Nodes:** These are the machines (physical or virtual) where the containers run. A cluster cannot exist without at least one node.
- **Persistent Volumes:** Storage space that retains data even if pods are deleted. Ideal for databases and important files.
- **Roles:** Define permissions within a specific namespace. They limit what a user or service can do in that space.
- **Role Binding:** Connects a role to a user or service account. This is what actually applies permissions.

- **Service Accounts:** Internal accounts that pods use to authenticate and communicate with the Kubernetes API or other services.

YAML Resource Analysis

The **K8S/example/** folder contains all the YAML files needed to deploy the various application components within the Kubernetes cluster. There are **Deployment** files, which indicate how the containers should be deployed, and **Service** files, which expose those containers so they can communicate with each other or, in some cases, be accessible from outside the cluster. To achieve this, I analyzed the deployment and service files for the result component, which is part of the app's backend.

Deployment

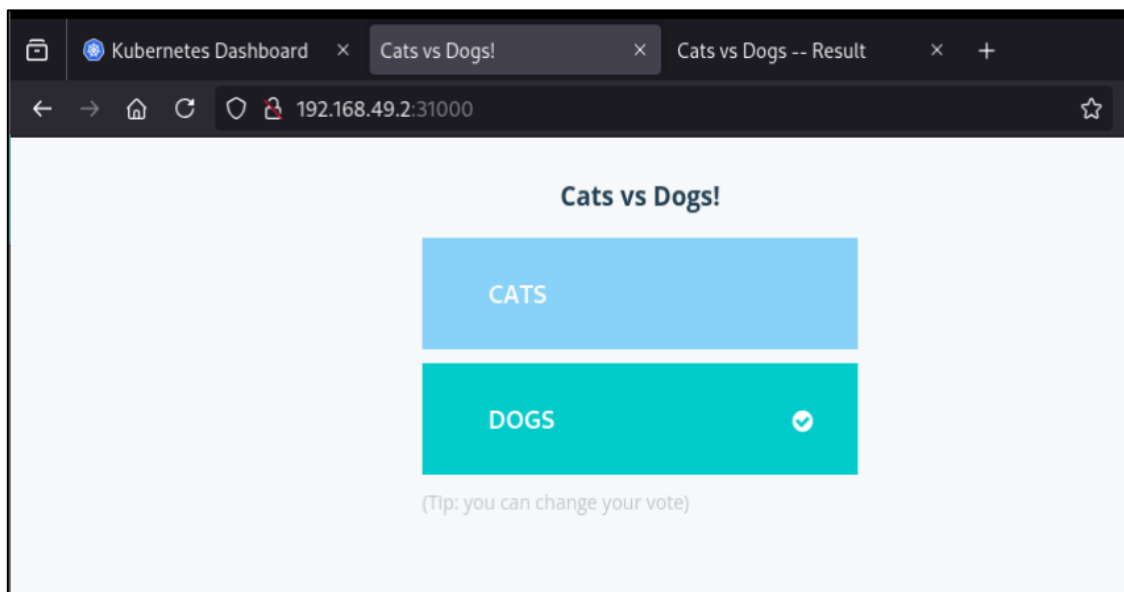
The **result-deployment.yaml** file defines a deployment named **result**, which launches a container based on the **dockersamples/examplevotingapp_result** image. It is configured to always have a replica running, and if the pod fails, Kubernetes automatically restarts it. **Tags (app: result)** are used in both the **Deployment** and the **pod** and are important for linking it to the service. The container is named **"result"** and port 80 is exposed, which will later be used by the service to route traffic correctly.

Service

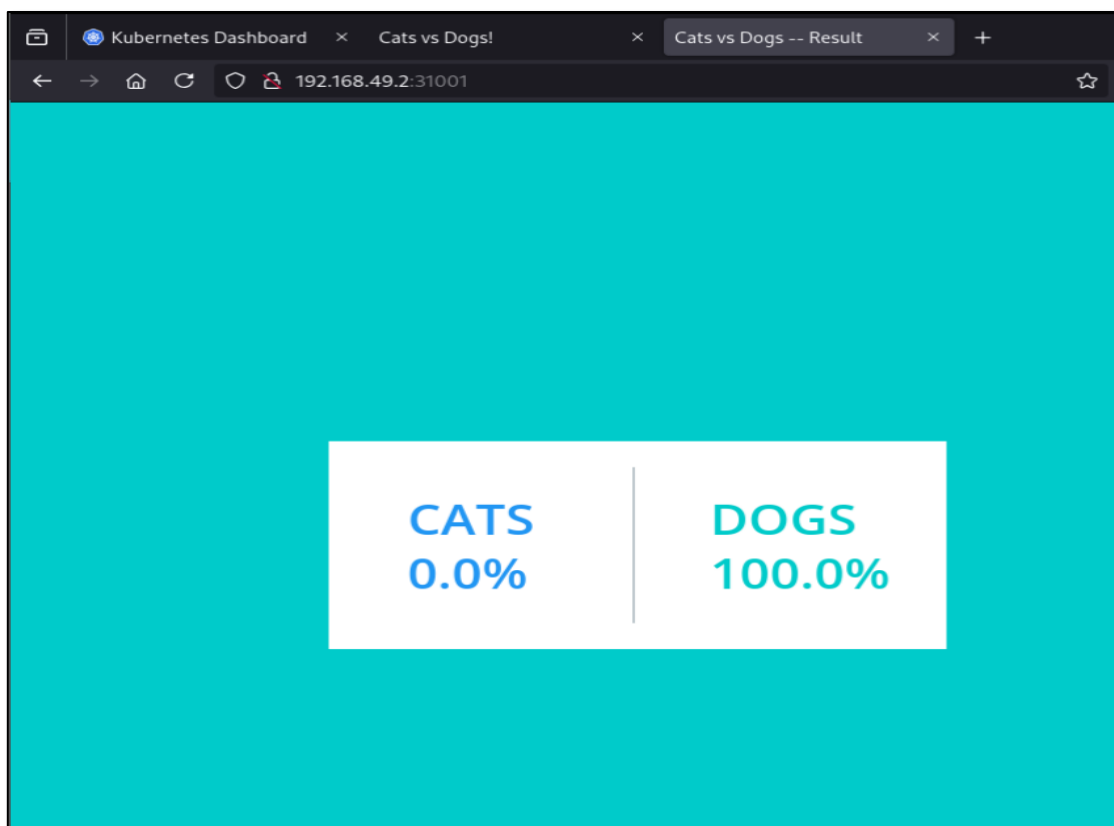
The **result-service.yaml** file defines a **Service** of type **NodePort**, which means it exposes the container to the outside of the cluster through a node port. In this case, it receives connections on port 8081 and redirects them to container port 80, while the assigned external port is 31001. This allows the **Service** to be accessed from outside the cluster using **http://192.168.49.2:31001**. The app: **"result"** selector ensures that traffic is directed to the correct pod. This configuration is useful for development environments like **Minikube**, where there is no load balancer, but we still need to test services from the browser or tools like curl.

IP_HOST: 192.168.0.6
IP_GUEST: 192.168.0.57

The “vote” URL is verified, which shows the voting interface working



The “result” URL is verified, which displays the results interface showing data



CONCLUSIONS AND FUTURE WORK

After concluding this practice, I can say that overall, it has been very valuable for me regarding my professional development. Not only because of the technical tools, since I already use many of them at work, but because of the complete process. I had to build everything from scratch, without using shortcuts or environments predefined or templates, so this made me focus on simple details that I usually don't notice when working because other teams take care of this for me and already solve any issue.

I know Docker and Kubernetes, but I usually use systems on top of these tools that are already setup. As I already mentioned, the fact that I had to do everything myself (from installing the operating system, to make the web services work secure) really help me to learn more about these tools, and the magnitude that probably entails configuring these for a whole company.

For example, the part of creating my own security certificate or setting up a reverse proxy, are a clear example of simple process that I know from the theory, but I never had to do it by myself until this practice. These part that helped me the most not just because of the technical steps but because I understood better what could happens when a secure connection fails or when there is a network problem. I really learned a lot by creating the certificate with OpenSSL, putting it in the container and configuring HAProxy correctly.

The Kubernetes part was also very helpful. Minikube, even if a simple tool for working with these container orchestrator process, really helped to focus on the basics and it was ideal to understand better how the different parts of these process (like deployment and services) work together and what could happen if something does not work right.

I also want to highlight how important it is to document the process. At work, documentation is sometimes missing, and by doing this report where I had to clearly explain what I did, why I did it, and how I verified it, this will probably help others better understand the process, but it will also help me revisit and remember parts of the work that I might forget after some time.

I can finally conclude that this practice was a very valuable experience, and it will definitely help me better understand the entire process flow if the infrastructure, both in my current organization and in any future job.

APPENDIX CONFIGURATION FILES

These are the configuration files that were created or modified during the development of the practice. The final version of each file is included, as it was used for the correct execution of the services.

bun-app/Dockerfile

```
FROM oven/bun:ltest

WORKDIR /app
COPY ./app /app

RUN bun install

EXPOSE 3000

CMD ["bun", "src/app.ts"]
```

reverse-proxy/Dockerfile

```
FROM haproxy:latest

COPY haproxy.cfg /usr/local/etc/haproxy/haproxy.cfg

COPY certs/uoc.pem /usr/local/etc/haproxy/certs/uoc.pem
```

.env

```
POSTGRES_DB=1234
```

reverse-proxy/haproxy.cfg

```
global
    log stdout format raw local0

defaults
    log      global
    mode     http
    timeout connect 5000ms
    timeout client 50000ms
    timeout server 50000ms

frontend http_front
    bind *:80
    redirect scheme https code 302 if !{ ssl_fc }

frontend https_front
    bind *:443 ssl crt /usr/local/etc/haproxy/certs/uoc.pem
    default_backend http_back

backend http_back
    server webserv webhost:3000 check
```

Certificado TLS (reverse-proxy/certs/uoc.pem)

This file was generated with OpenSSL using the following command:

```
mkdir -p reverse-proxy/certs
openssl req -x509 -nodes -days 365 -newkey rsa:2048 \
    -keyout certs/uoc.key -out certs/uoc.crt \
    -subj "/CN=localhost/emailAddress=ndalessandro@uoc.edu"

cat certs/uoc.crt certs/uoc.key > certs/uoc.pem
```

The **uoc.pem** certificate was used in the HAProxy configuration to enable HTTPS access.

Docker-compose.yml

```
services:
  db:
    image: postgres:17.4
    container_name: dbhost
    restart: always
    ports:
      - "5432:5432"
    volumes:
      - dbdata:/var/lib/postgresql/data
      - ./dataset:/docker-entrypoint-initdb.d
    environment:
      POSTGRES_DB: uoc2024
      POSTGRES_USER: ndalessandro
      POSTGRES_PASSWORD: ${DB_PASSWORD}
    networks:
      - db-network
  web:
    build: ./bun-app
    container_name: webhost
    ports:
      - "5000:3000"
    environment:
      POSTGRES_HOST: dbhost
      POSTGRES_DB: uoc2024
      POSTGRES_USER: ndalessandro
      POSTGRES_PASSWORD: ${DB_PASSWORD}
    networks:
      - web-network
      - db-network
  proxy:
    build: ./reverse-proxy
    container_name: proxyhost
    ports:
      - "80:80"
      - "443:443"
    networks:
      - web-network
volumes:
  dbdata:
networks:
  web-network:
    name: web-network
  db-network:
    name: db-network
```