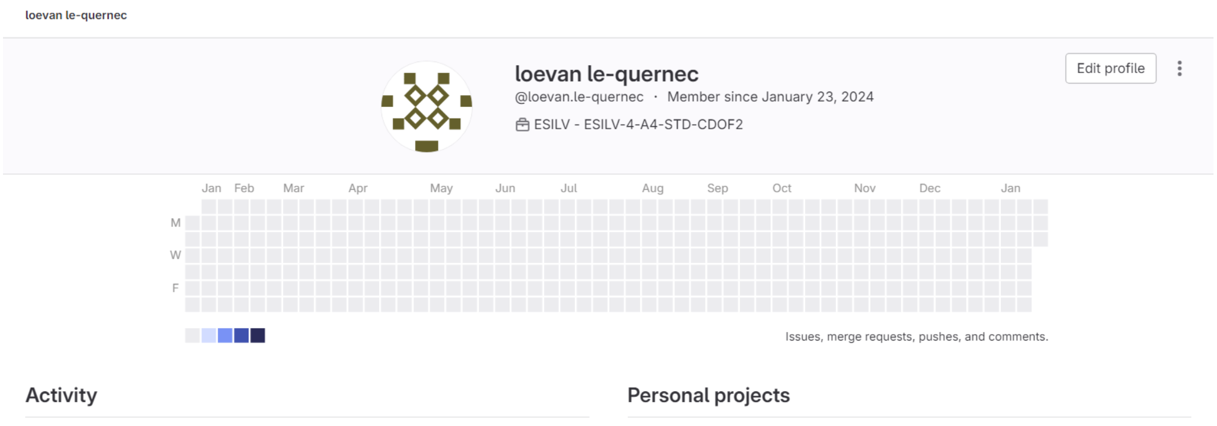
LE QUERNEC Loévan CDOF2

Containerization Technologies – TD 4

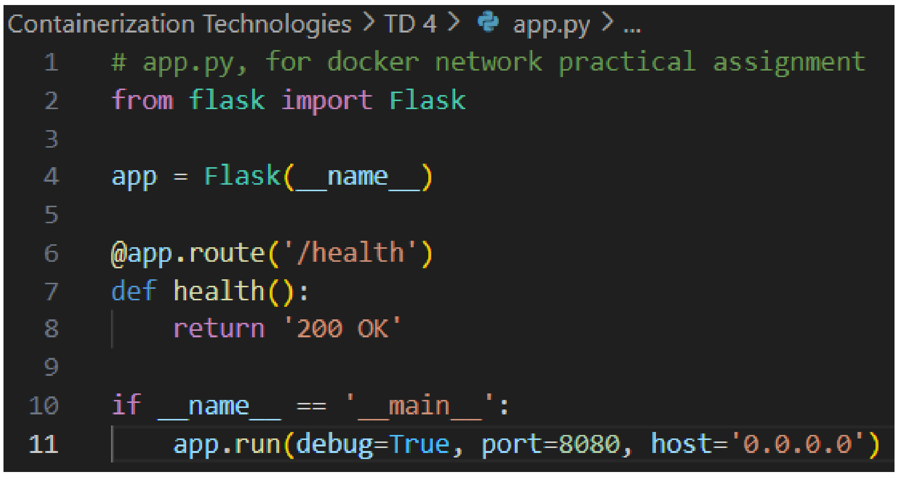
Step 0: GitLab

Account successfully created:


loevan le-quernec
loevan le-quernec
@loevan.le-quernec · Member since January 23, 2024
ESILV - ESILV-4-A4-STD-CDOF2
Edit profile
Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec Jan
M
W
F
Issues, merge requests, pushes, and comments.
Activity
Personal projects

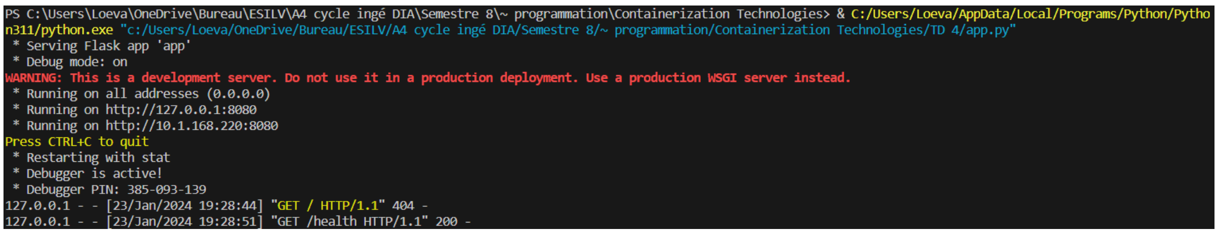
Step 1: Create the application

Here is the Python file where I use Flask to write the API:



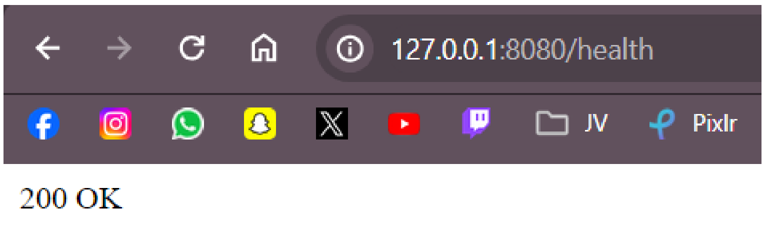
```python
# app.py, for docker network practical assignment
from flask import Flask

app = Flask(__name__)

@app.route('/health')
def health():
    return '200 OK'

if __name__ == '__main__':
    app.run(debug=True, port=8080, host='0.0.0.0')
```

The app is running:



```
PS C:\Users\Loeva\OneDrive\Bureau\ESILV\A4 cycle ingé DIA\Semestre 8\~ programmation\Containerization Technologies> & C:/Users/Loeva/AppData/Local/Programs/Python/Pytho
n311/python.exe "c:/Users/Loeva/OneDrive/Bureau/ESILV/A4 cycle ingé DIA/Semestre 8/~ programmation/Containerization Technologies/TD 4/app.py"
 * Serving Flask app 'app'
 * Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on all addresses (0.0.0.0)
 * Running on http://127.0.0.1:8080
 * Running on http://10.1.168.220:8080
Press CTRL+C to quit
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 385-093-139
127.0.0.1 - - [23/Jan/2024 19:28:44] "GET / HTTP/1.1" 404 -
127.0.0.1 - - [23/Jan/2024 19:28:51] "GET /health HTTP/1.1" 200 -
```

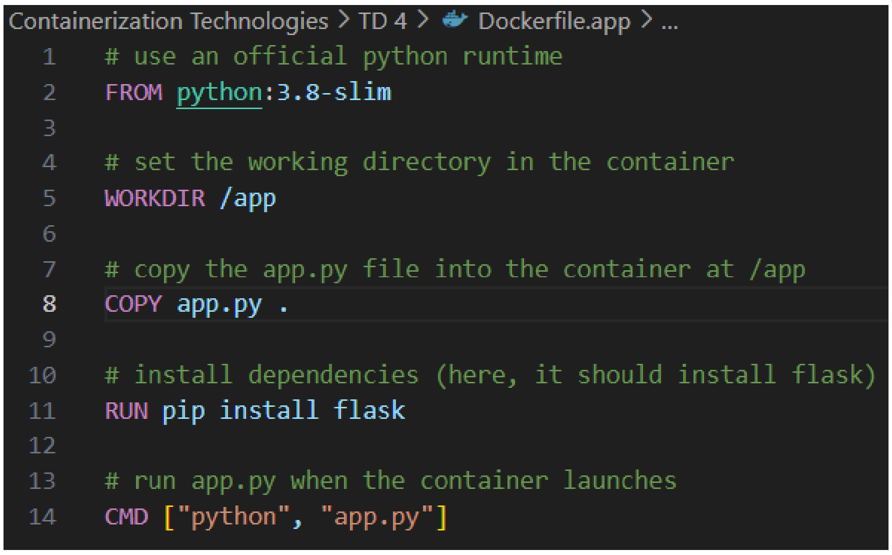
And displaying the message '200 OK':



```
127.0.0.1:8080/health

200 OK
```

Step 2: Create the application Dockerfile

The 'Dockerfile.app' file:

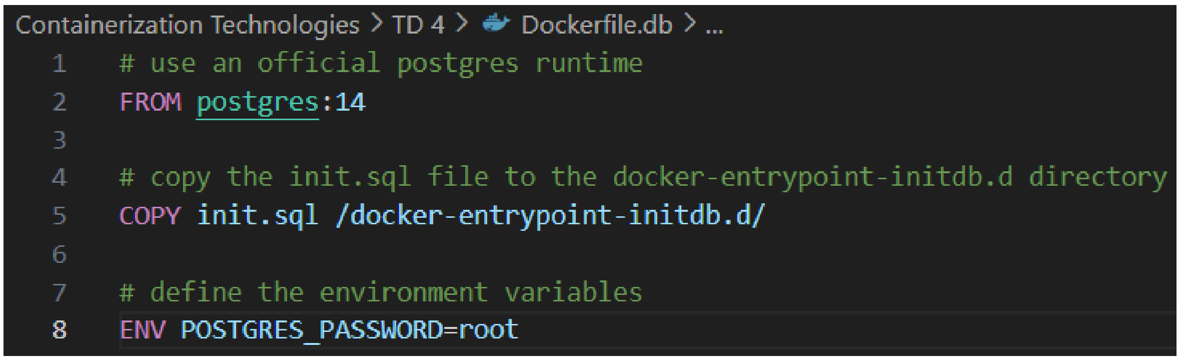


Containerization Technologies > TD 4 > Dockerfile.app > ...

```dockerfile
# use an official python runtime
FROM python:3.8-slim

# set the working directory in the container
WORKDIR /app

# copy the app.py file into the container at /app
COPY app.py .

# install dependencies (here, it should install flask)
RUN pip install flask

# run app.py when the container launches
CMD ["python", "app.py"]
```

Step 3: Create the database Dockerfile

The 'Dockerfile.db' file:



Containerization Technologies > TD 4 > Dockerfile.db > ...

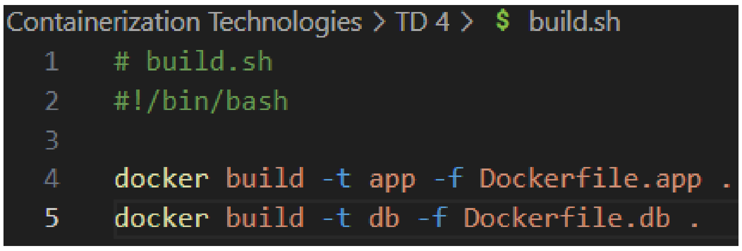
```dockerfile
# use an official postgres runtime
FROM postgres:14

# copy the init.sql file to the docker-entrypoint-initdb.d directory
COPY init.sql /docker-entrypoint-initdb.d/

# define the environment variables
ENV POSTGRES_PASSWORD=root
```

Due to an error in the logs of the 'db' container, indicating that I needed to set a password for the superuser, I added an environment variable for the password directly in the Dockerfile of the database (without this, the container can't run).

Step 4: Build

Here is the script that build both ‘app’ and ‘db’ images using our Dockerfiles created before.

‘-t’ is setting up the name of the image and ‘-f’ indicates first the Dockerfile we use and second the directory where it is located.



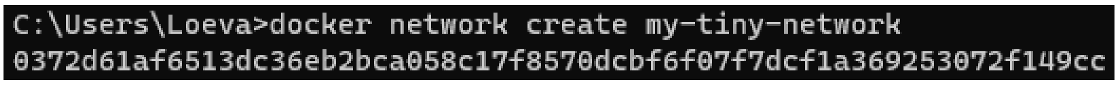
Containerization Technologies > TD 4 > $ build.sh

```
# build.sh
#!/bin/bash

docker build -t app -f Dockerfile.app .
docker build -t db -f Dockerfile.db .
```

Step 5: Network

Now we create a network of type ‘bridge’ (by default).

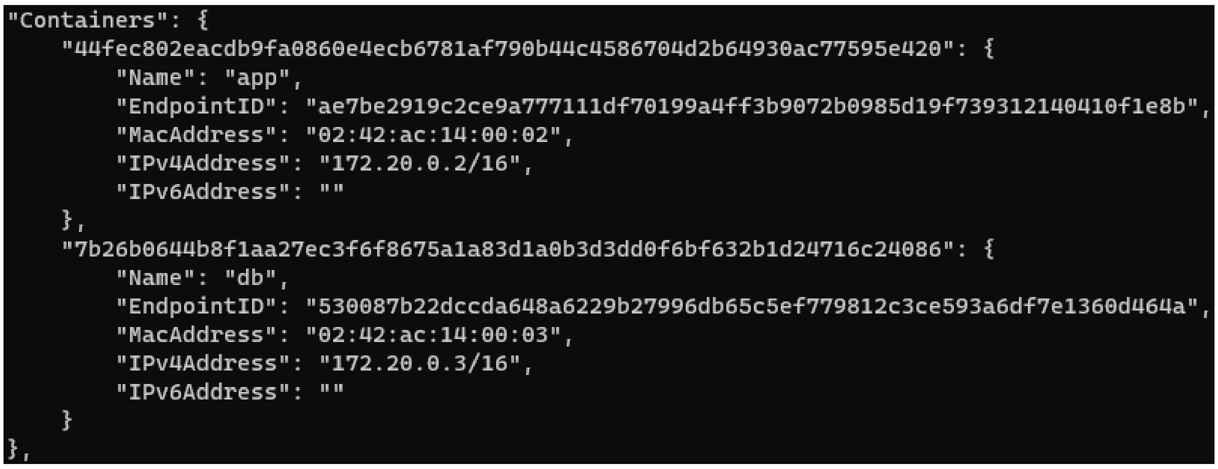
The command: ‘**docker network create my-tiny-network**’.



```
C:\Users\Loeva>docker network create my-tiny-network
0372d61af6513dc36eb2bca058c17f8570dcbf6f07f7dcf1a369253072f149cc
```

Now we can check that our two containers are on the same network.

The command: ‘**docker network inspect my-tiny-network**’.



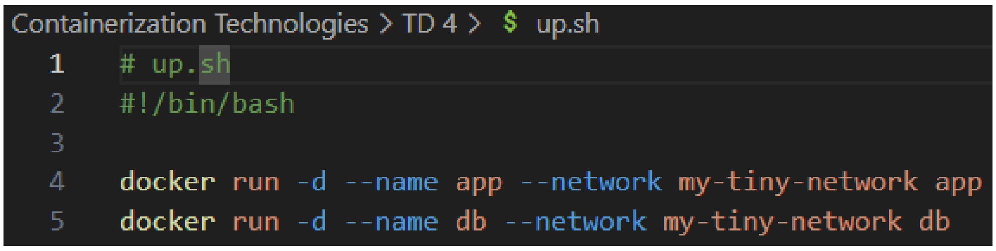
```
"Containers": {
    "44fec802eacdb9fa0860e4ecb6781af790b44c4586704d2b64930ac77595e420": {
        "Name": "app",
        "EndpointID": "ae7be2919c2ce9a777111df70199a4ff3b9072b0985d19f739312140410f1e8b",
        "MacAddress": "02:42:ac:14:00:02",
        "IPv4Address": "172.20.0.2/16",
        "IPv6Address": ""
    },
    "7b26b0644b8f1aa27ec3f6f8675a1a83d1a0b3d3dd0f6bf632b1d24716c24086": {
        "Name": "db",
        "EndpointID": "530087b22dccda648a6229b27996db65c5ef779812c3ce593a6df7e1360d464a",
        "MacAddress": "02:42:ac:14:00:03",
        "IPv4Address": "172.20.0.3/16",
        "IPv6Address": ""
    }
},
```

Step 6: Run

Here is the script that run both ‘app’ and ‘db’ containers using our images created before.

‘-d’ allows the container to run in the background, where ‘d’ stands for the first letter of the word ‘detached’. The container is essentially autonomous.

The other flags have explicit meanings.



Containerization Technologies > TD 4 > $ up.sh

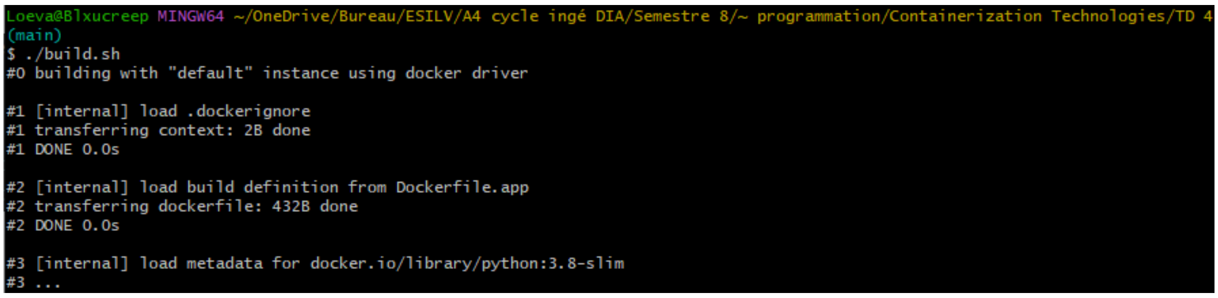
```
# up.sh
#!/bin/bash

docker run -d --name app --network my-tiny-network app
docker run -d --name db --network my-tiny-network db
```

Step 6bis: Execute scripts

To be able to execute my bash scripts, with the .sh extension, I used Git Bash, already installed on my computer. I just have to navigate through the right directory, and then execute the two commands for the two scripts.

Below just a preview of the output for the build (too long to screen it).

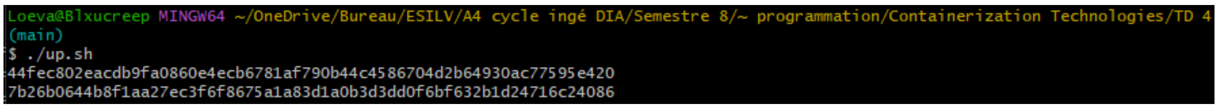
The command: ‘**./build.sh**’.



```
Loeva@Blxucreep MINGW64 ~/OneDrive/Bureau/ESILV/A4 cycle ingé DIA/Semestre 8/~ programmation/Containerization Technologies/TD 4
(main)
$ ./build.sh
#0 building with "default" instance using docker driver

#1 [internal] load .dockerignore
#1 transferring context: 2B done
#1 DONE 0.0s

#2 [internal] load build definition from Dockerfile.app
#2 transferring dockerfile: 432B done
#2 DONE 0.0s

#3 [internal] load metadata for docker.io/library/python:3.8-slim
#3 ...
```

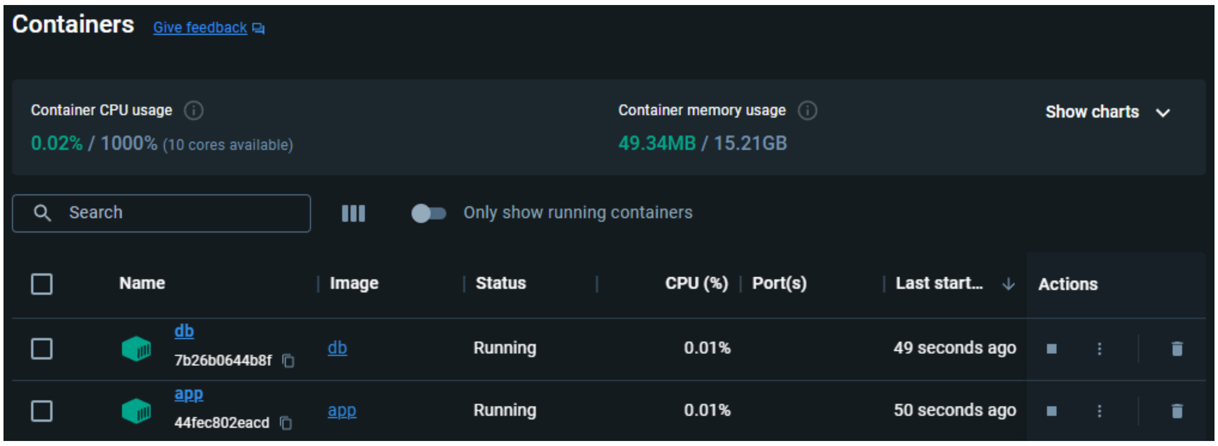
And below the output for the run.

The command: ‘**./up.sh**’.



```
Loeva@Blxucreep MINGW64 ~/OneDrive/Bureau/ESILV/A4 cycle ingé DIA/Semestre 8/~ programmation/Containerization Technologies/TD 4
(main)
$ ./up.sh
44fec802eacdb9fa0860e4ecb6781af790b44c4586704d2b64930ac77595e420
7b26b0644b8f1aa27ec3f6f8675a1a83d1a0b3d3dd0f6bf632b1d24716c24086
```

Now we can verify that our two containers are running:



Containers Give feedback

Container CPU usage 0.02% / 1000% (10 cores available)

Container memory usage 49.34MB / 15.21GB

Name	Image	Status	CPU (%)	Port(s)	Last start...	Actions
db 7b26b0644b8f	db	Running	0.01%		49 seconds ago	
app 44fec802eacd	app	Running	0.01%		50 seconds ago	

Step 7: Manual insert

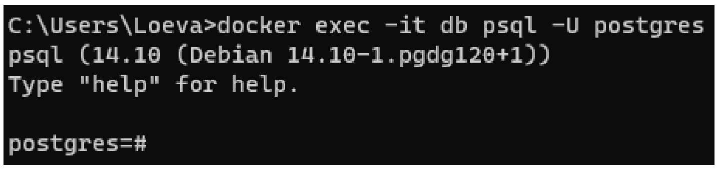
Before doing any insert, let’s use ‘docker exec’ to get a shell (terminal) in the database container.

‘-it’ is for an interactive pseudo-terminal.

‘psql’ is the utility tool to interact with a PostgreSQL database.

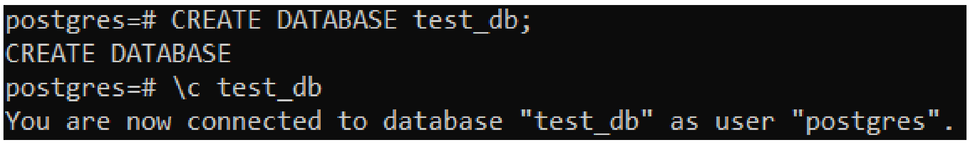
‘-U postgres’ is indicating with wich user we should connect, here it’s the one named ‘postgres’.

The command: ‘**docker exec -it db psql -U postgres**’.



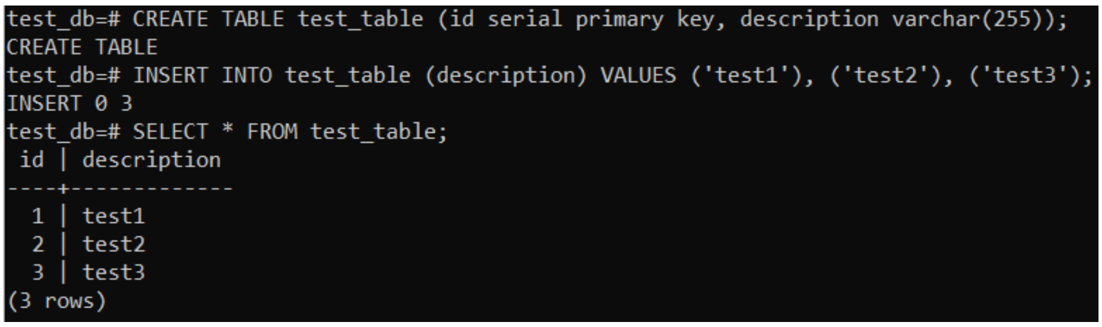
```
C:\Users\Loeva>docker exec -it db psql -U postgres
psql (14.10 (Debian 14.10-1.pgdg120+1))
Type "help" for help.

postgres=#
```

First, we create a new database called ‘test_db’, and we connect to it:



```
postgres=# CREATE DATABASE test_db;
CREATE DATABASE
postgres=# \c test_db
You are now connected to database "test_db" as user "postgres".
```

Finally we can try to do some inserts:

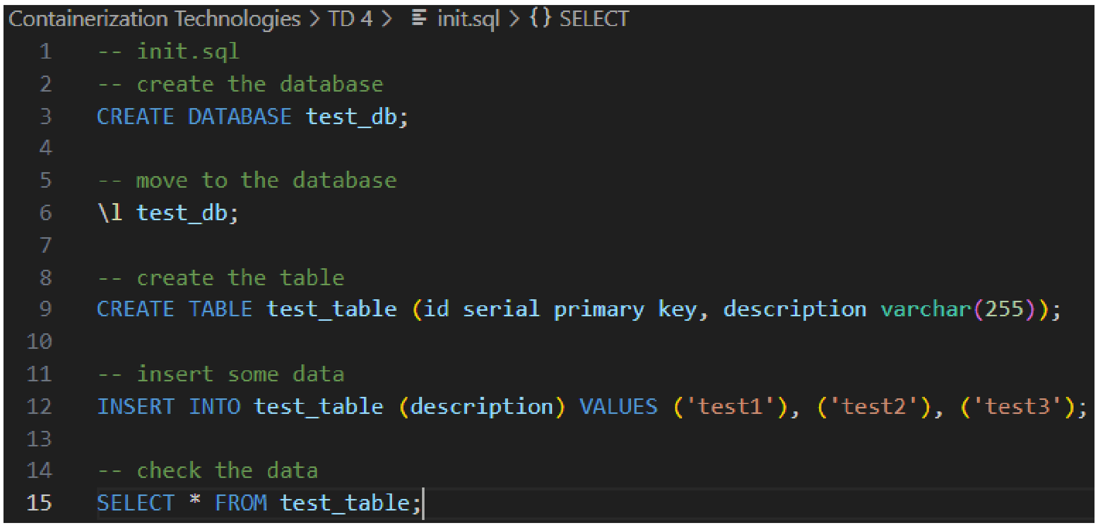


```
test_db=# CREATE TABLE test_table (id serial primary key, description varchar(255));
CREATE TABLE
test_db=# INSERT INTO test_table (description) VALUES ('test1'), ('test2'), ('test3');
INSERT 0 3
test_db=# SELECT * FROM test_table;
 id | description
----+-------------
  1 | test1
  2 | test2
  3 | test3
(3 rows)
```

Step 7bis: Automatic insert

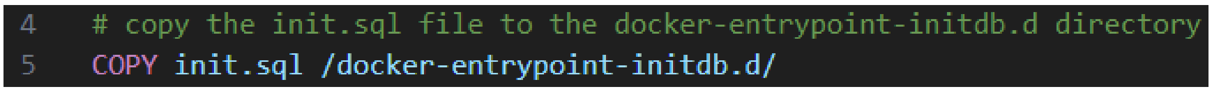
Everything that follows from now is things that I wanted to add (I thought we needed to continue by ourselves and test the communication between the two containers) and you said that we'll see that during the next practical assignment, but I guess I can keep it.

Above, we can see that our inserts are ok. Now, let's do it automatically. I created a file named 'init.sql' that does exactly the same thing than before, and it is executed when the container is launching (thanks to the docker-entrypoint-initdb.d directory in the database container). First, the sql file, named 'init.sql':



```sql
-- init.sql
-- create the database
CREATE DATABASE test_db;

-- move to the database
\l test_db;

-- create the table
CREATE TABLE test_table (id serial primary key, description varchar(255));

-- insert some data
INSERT INTO test_table (description) VALUES ('test1'), ('test2'), ('test3');

-- check the data
SELECT * FROM test_table;
```

Then, the line I added in the 'Dockerfile.db' file (the one for the database container):

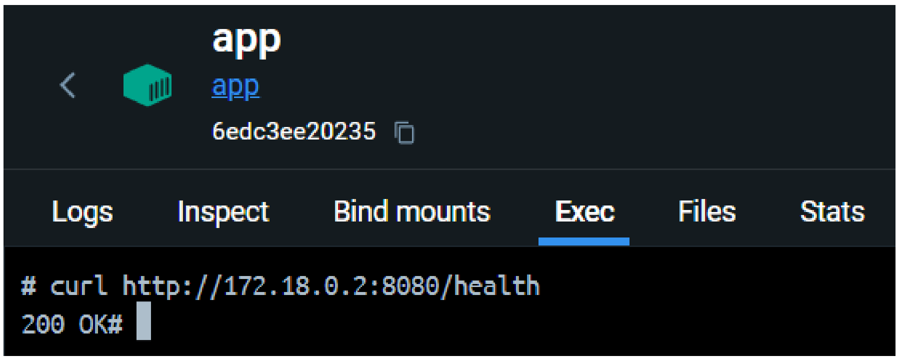


```dockerfile
# copy the init.sql file to the docker-entrypoint-initdb.d directory
COPY init.sql /docker-entrypoint-initdb.d/
```

Step 8: Test the health route

So, now let's test our health, and we should be good!

Here, our health route is working (I installed curl before):



```
# curl http://172.18.0.2:8080/health
200 OK#
```

And we can see that our app container is working properly!