

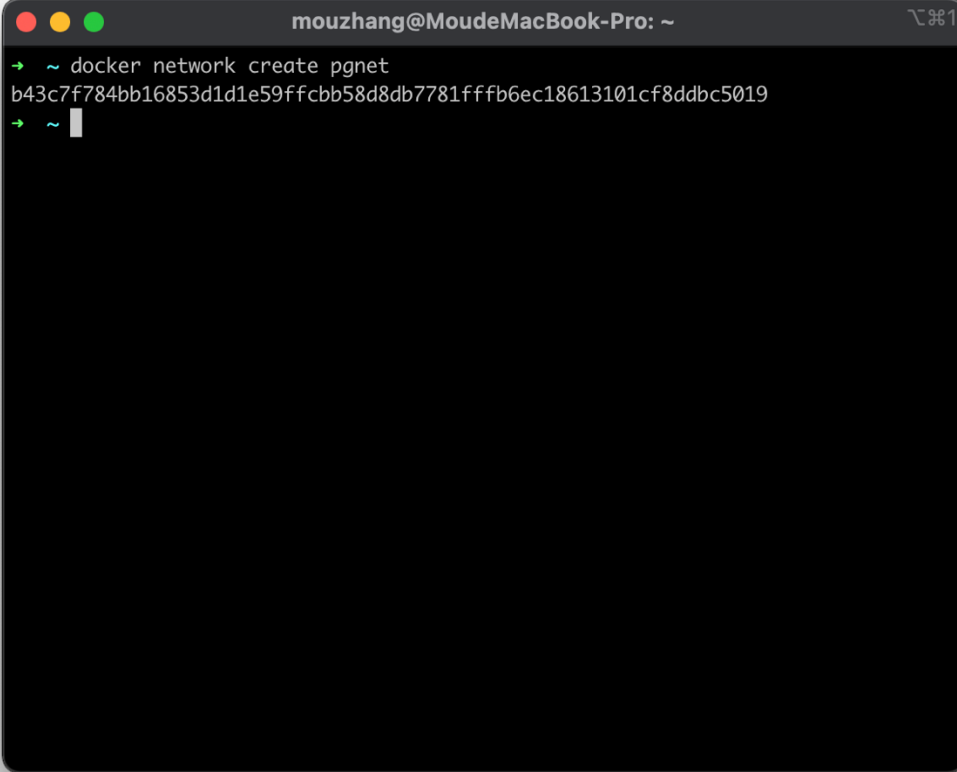
Lab 6: Create a Service Spanning Multiple Containers

Mou Zhang

Section 1 Docker Network

1. Setting up the network pgnet as the doc suggest

```
docker network create pgnet
```

A terminal window titled 'mouzhang@MoudeMacBook-Pro: ~' with standard macOS window controls (red, yellow, green buttons) in the top-left corner. The terminal shows a command prompt '~' followed by the command 'docker network create pgnet'. The output is a long alphanumeric string: 'b43c7f784bb16853d1d1e59ffcbb58d8db7781fffb6ec18613101cf8ddbc5019'. A second prompt '~' is visible on the next line.

```
mouzhang@MoudeMacBook-Pro: ~  
→ ~ docker network create pgnet  
b43c7f784bb16853d1d1e59ffcbb58d8db7781fffb6ec18613101cf8ddbc5019  
→ ~
```

Section 2 Postgres Container

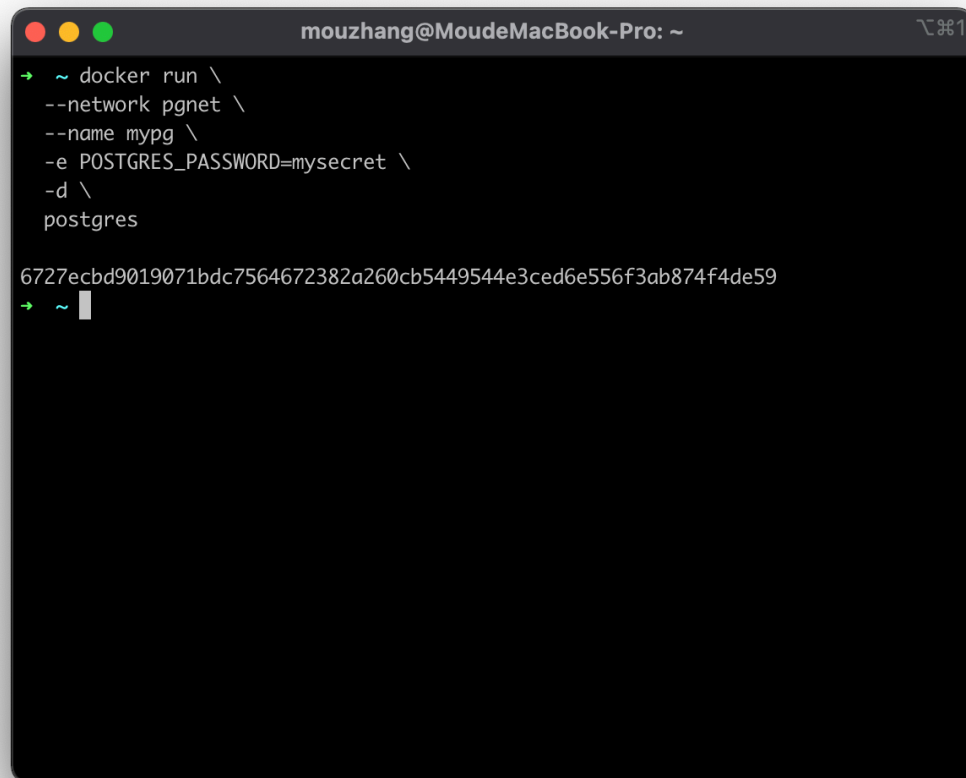
2. Pull the latest postgres docker

```
docker pull postgres
```

```
mouzhang@MoudeMacBook-Pro: ~  
→ ~ docker network create pgnet  
b43c7f784bb16853d1d1e59ffccb58d8db7781fffb6ec18613101cf8ddbc5019  
→ ~ docker pull postgres  
Using default tag: latest  
latest: Pulling from library/postgres  
852e50cd189d: Pull complete  
0269cd569193: Pull complete  
879ff0b54097: Pull complete  
3114d3793d45: Pull complete  
0ee7737137c3: Pull complete  
3fff8dbf0e49: Pull complete  
6b78b1e21dd1: Pull complete  
ddecce665bec: Pull complete  
3fae122f28bd: Pull complete  
69289a4f935b: Pull complete  
56d6c36d6958: Pull complete  
d794b9ea73a0: Pull complete  
1738fda53f17: Pull complete  
bbcdf7c12dbd: Pull complete  
Digest: sha256:839d6212e7aadb9612fd216374279b72f494c9c4ec517b8e98d768ac9dd74a15  
Status: Downloaded newer image for postgres:latest  
docker.io/library/postgres:latest  
→ ~
```

3. Start a init postgres container with no table

```
docker run \  
  --network pgnet \  
  --name mypg \  
  -e POSTGRES_PASSWORD=mysecret \  
  -d \  
  postgres
```

A terminal window titled 'mouzhang@MoudeMacBook-Pro: ~' with standard macOS window controls (red, yellow, green buttons) in the top-left corner. The terminal shows a Docker command being executed: 'docker run --network pgnet --name mypg -e POSTGRES_PASSWORD=mysecret -d postgres'. The output is a long alphanumeric string: '6727ecbd9019071bdc7564672382a260cb5449544e3ced6e556f3ab874f4de59'. The prompt '~' is followed by a cursor.

```
mouzhang@MoudeMacBook-Pro: ~  
→ ~ docker run \  
--network pgnet \  
--name mypg \  
-e POSTGRES_PASSWORD=mysecret \  
-d \  
postgres  
  
6727ecbd9019071bdc7564672382a260cb5449544e3ced6e556f3ab874f4de59  
→ ~
```

Section 3 Init Container

4. Get the ip address for the postgres container in pgnet

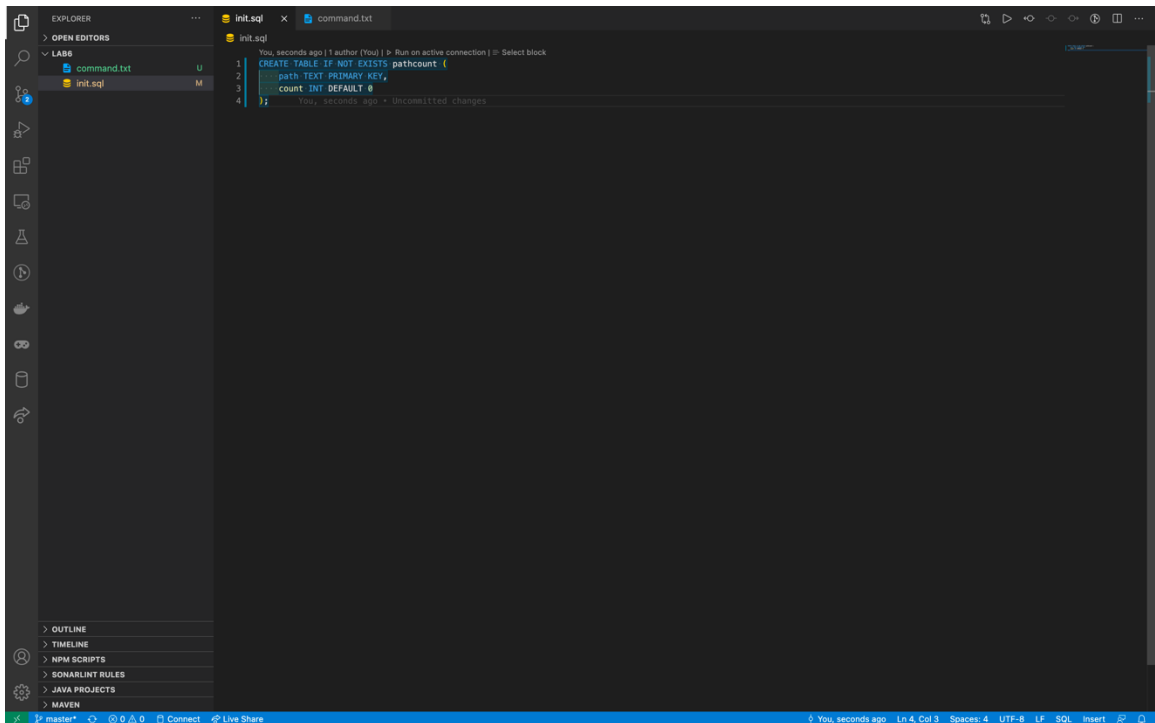
```
docker container inspect mypg \  
-f '{{.NetworkSettings.Networks.pgnet.IPAddress}}'
```

```
mouzhang@MoudeMacBook-Pro: ~  
→ ~ docker run \  
  --network pgnet \  
  --name mypg \  
  -e POSTGRES_PASSWORD=mysecret \  
  -d \  
  postgres  
  
6727ecbd9019071bdc7564672382a260cb5449544e3ced6e556f3ab874f4de59  
→ ~ docker container inspect mypg \  
  -f '{{.NetworkSettings.Networks.pgnet.IPAddress}}'  
172.18.0.2  
→ ~
```

We can find that the ip address to connect to postgres is 172.18.0.2

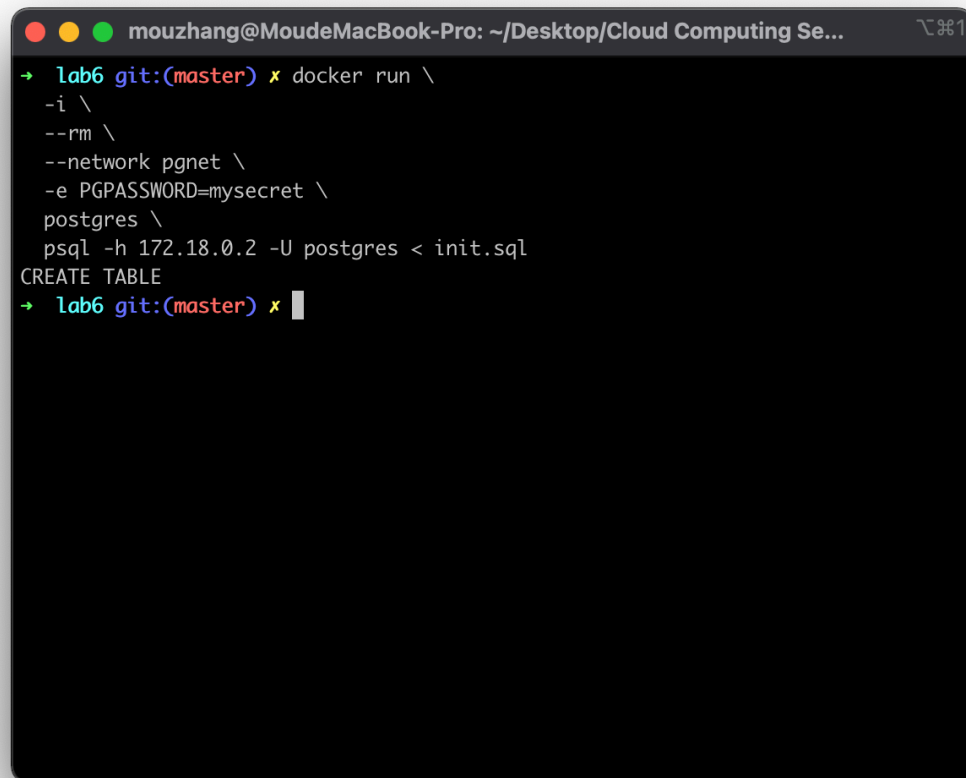
5. Prepare the init.sql to initialize the pathcount database we need

```
CREATE TABLE IF NOT EXISTS pathcount (  
  path TEXT PRIMARY KEY,  
  count INT DEFAULT 0  
);
```



6. Setting up pathcount table in the postgres docker using another temporary docker with init.sql

```
docker run \  
-i \  
--rm \  
--network pgnet \  
-e PGPASSWORD=mysecret \  
postgres \  
psql -h 172.18.0.2 -U postgres < init.sql
```

A terminal window on a Mac with the title bar 'mouzhang@MoudeMacBook-Pro: ~/Desktop/Cloud Computing Se...'. The prompt is 'lab6 git:(master) x'. The command entered is 'docker run \', followed by several options: '-i \', '--rm \', '--network pgnet \', '-e PGPASSWORD=mysecret \', 'postgres \', and 'psql -h 172.18.0.2 -U postgres < init.sql'. The output shows 'CREATE TABLE' and then the prompt returns to 'lab6 git:(master) x' with a cursor.

```
→ lab6 git:(master) x docker run \  
-i \  
--rm \  
--network pgnet \  
-e PGPASSWORD=mysecret \  
postgres \  
psql -h 172.18.0.2 -U postgres < init.sql  
CREATE TABLE  
→ lab6 git:(master) x
```

As you can see from the picture, the table is successfully created in the postgres container.

Section 4 Service Container

7. Setting up the dockerfile and environment variable for service container

The **Dockerfile** is shown as below:

```
FROM python:3.6  
RUN mkdir /app  
ADD . /app  
WORKDIR /app  
RUN pip install flask \  
    Psycopg2  
EXPOSE 8080  
ENV DB_NAME="postgres"  
ENV DB_USER="postgres"  
ENV DB_HOST="172.18.0.2"  
ENV DB_PASSWORD="mysecret"
```

```
ENV FLASK_APP=main.py
CMD flask run --host=0.0.0.0 --port=8080
```

As you can see, It is using python 3.6 docker and using flask and psycopg2 to set up the service and gain access to the postgres docker. The postgres username, database, password, and hostname are configured in the environment variables.

8. Setting up the frontend and backend of the service

The *main.py* is the backend service. The backend service is using flask to get configurations from environment and connect to the database and add count/return count to the frontend with render_templates using jinja2 format.

```
# Copyright 2018 Google LLC
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

# [START gae_python38_app]
from flask import Flask, render_template
import psycopg2
import os

app = Flask(__name__)

dbname = os.getenv('DB_NAME')
dbuser = os.getenv('DB_USER')
# "localhost" # "pgnet" # "172.18.0.2" # os.getenv('DB_HOST')
dbhost = os.getenv('DB_HOST')
dbpasswd = os.getenv('DB_PASSWORD')

@app.route('/', defaults={'u_path': ''}, methods=['GET'])
@app.route('/<path:u_path>', methods=['GET'])
def root(u_path):
```

```

count_paths(str(u_path))
return show_path()

def count_paths(u_path):
    sql = """INSERT INTO pathcount (path, count)
            VALUES (%s, 1)
            ON CONFLICT (path) DO UPDATE
            SET count = pathcount.count + 1
            RETURNING count;"""

    print(u_path)
    try:
        conn = psycopg2.connect(
            database=dbname, user=dbuser, host=dbhost, password=dbpasswd)
        cur = conn.cursor()
        cur.execute(sql, (u_path, ))
        conn.commit()
        cur.close()
        conn.close()
    except psycopg2.DatabaseError as e:
        print(e)
        print("I am unable to connect to the database.")

def show_path():
    sql = """SELECT path, count FROM pathcount ORDER BY path;"""
    data_return = None

    try:
        conn = psycopg2.connect(
            database=dbname, user=dbuser, host=dbhost, password=dbpasswd)
        cur = conn.cursor()
        cur.execute(sql)
        data_return = cur.fetchall()
        conn.commit()
        cur.close()
        conn.close()
    except psycopg2.DatabaseError as e:
        print(e)
        print("I am unable to connect to the database.")

```



```

print(data_return)
return render_template('index.html', data=data_return)

if __name__ == '__main__':
    # This is used when running locally only. When deploying to Google App
    # Engine, a webserver process such as Gunicorn will serve the app. This
    # can be configured by adding an `entrypoint` to app.yaml.
    app.run(host='127.0.0.1', port=8080, debug=True)
# [END gae_python38_app]

```

9. Setting up the frontend part with *index.html*. Using jinja2 to get and split data, then show the data on the web page in a html table.

```

<!DOCTYPE>
<html>
<title>
    Mou's Count Path
</title>
<table border = 1>
    <tr><th>Path</th><th>Count</th></tr>
    {%for i in data%}
    <tr> <td>{{i[0]}}</td> <td>{{i[1]}}</td> </tr>
    {%endfor%}
</table>
</html>

```

10. Build the service docker. Naming it as lab6-img

```
docker build -t lab6-img .
```

- 11.

```
..Security/lab6 (-zsh) 0%1
→ lab6 git:(master) x docker build -t lab6-img .
[+] Building 12.6s (10/10) FINISHED
=> [internal] load .dockerignore 0.0s
=> => transferring context: 2B 0.0s
=> [internal] load build definition from Dockerfile 0.0s
=> => transferring dockerfile: 303B 0.0s
=> [internal] load metadata for docker.io/library/python:3.6 0.5s
=> [1/5] FROM docker.io/library/python:3.6@sha256:84c13b6992eba6731074b8 0.0s
=> [internal] load build context 0.0s
=> => transferring context: 1.13kB 0.0s
=> CACHED [2/5] RUN mkdir /app 0.0s
=> [3/5] ADD . /app 0.0s
=> [4/5] WORKDIR /app 0.0s
=> [5/5] RUN pip install flask      Psycopg2 11.8s
=> exporting to image 0.2s
=> => exporting layers 0.2s
=> => writing image sha256:b4873ef7408a7dd317690d18544dab866923978e8aa16 0.0s
=> => naming to docker.io/library/lab6-img 0.0s
→ lab6 git:(master) x
```

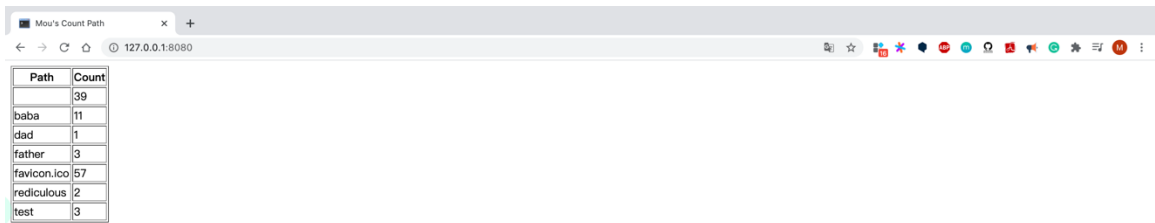
11. Build the service container with lab6-img and connect it to localhost:8080

```
docker run \
  -it \
  --name=path_count \
  --network pgnet\
  -p 8080:8080 \
  lab6-img
```

```
docker (com.docker.cli) 1
=> => transferring context: 751B 0.0s
=> [1/5] FROM docker.io/library/python:3.6@sha256:84c13b6992eba6731074b8 0.0s
=> CACHED [2/5] RUN mkdir /app 0.0s
=> [3/5] ADD . /app 0.0s
=> [4/5] WORKDIR /app 0.0s
=> [5/5] RUN pip install flask Psycopg2 11.6s
=> exporting to image 0.2s
=> => exporting layers 0.2s
=> => writing image sha256:9afe40a0330abf3bf89b50b6d8556f63c0c6244a4a777 0.0s
=> => naming to docker.io/library/lab6-img 0.0s
→ lab6 git:(master) x docker run \
  -it \
  --name=path_count \
  --network pgnet\
  -p 8080:8080 \
  lab6-img

* Serving Flask app "main.py"
* Environment: production
  WARNING: This is a development server. Do not use it in a production deployment.
  Use a production WSGI server instead.
* Debug mode: off
* Running on http://0.0.0.0:8080/ (Press CTRL+C to quit)
```

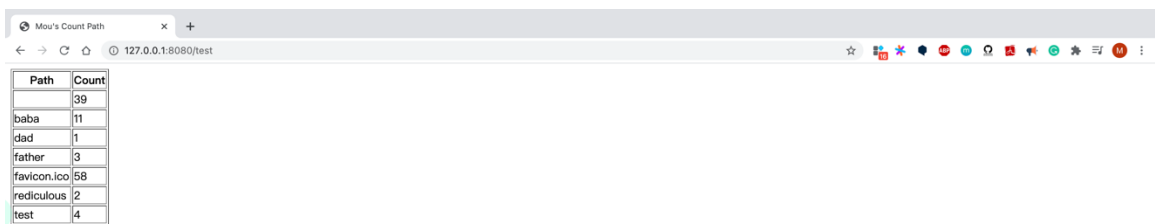
11. Then you can access the service with localhost:8080. The table is shown as expected, with all path and count sorted and printed.



A screenshot of a web browser window titled "Mou's Count Path". The address bar shows "127.0.0.1:8080". The browser's toolbar includes back, forward, refresh, and search icons, along with various extension icons. The main content area displays a table with two columns: "Path" and "Count".

Path	Count
	39
baba	11
dad	1
father	3
favicon.ico	57
rediculous	2
test	3

If you access a path already in the database, like 'localhost:8080/test' , then the count for path 'test' add 1.



A screenshot of a web browser window titled "Mou's Count Path". The address bar shows "127.0.0.1:8080/test". The browser's toolbar is the same as the previous screenshot. The main content area displays an updated table with two columns: "Path" and "Count".

Path	Count
	39
baba	11
dad	1
father	3
favicon.ico	58
rediculous	2
test	4

If you access a path that is not in the database, like 'localhost:8080/newtest' , then that path is add to database and the count is set to one.



Path	Count
	39
baba	11
dad	1
father	3
favicon.ico	59
newtest	1
rediculous	2
test	4

As you can see in the pictures, all functions are implemented correctly. The lab is completed.

Section 5 Answer to some security questions

- **Why did we create a special network instead of exposing the host network?**
Because this prevents us from exposing the database to the public network, which is dangerous and easy to hack. This network provides an isolation between the container and the host network.
- **Why didn' t we use exposed ports everywhere (that they exist)?**
Because it' s not secure. If we exposed ports everywhere, then these ports are easily reached from the host network, which gives hackers chances to hack into them and get data. By using another network, we get an isolation and protection from these hackers and malwares.
- **What could happen if you didn' t use SQL parameters, but relied on string formatting for setting the path in your queries?**
Then it is easy to get SQL injection attack.
- **Why is that particularly important in this setup? What makes those parameters potentially dangerous?**
Because this is helping prevent from the risky parameters. These parameters still have potential problems if not carefully checked.
- **The bridge network we define only works on a single host. What would you have to do to make these containers talk to each other if they were running on *different host machines*?**

One possible way is to set up a bridge network similar to this network and try to communicate through this network. Another possible way is to use more advanced structure like docker compose and Kubernetes.

- **What parts of this did you wish were simpler? Which parts seemed unnecessarily difficult?**

I think the part that should be simpler is the part with writing service docker and connect with postgres docker. Since this is a lab for security not backend developing, I think we shall get more hints on the service container building, including the suggested packages.

The part seems to be unnecessarily difficult is the conflict with postgres docker and pgadmin on my local macbook. When I was trying to connect to the postgres docker using port 5432, this port is occupied on my local machine and there is no reminder on what is the reason. don't find the specific reason for the conflict, but it is really frustrating since I tried so many times of connection and don't find the reason for failure. This problem only be solved after I completely removed all my postgres on the macbook and it takes a lot of time to realize the real problem.