Practical Lab Guide 1: Dockerizing a 3-Tier Web Application

Introduction

This lab is designed to provide you with a clear understanding and practical skills in using Docker to containerize a web application. We will focus on a 3-tier architecture, consisting of a React.js frontend, Django backend, and a MySQL database. By the end of this lab, you will have gained hands-on experience in:

- **Containerizing** each component of the application.
- **Running** the complete setup locally in a networked environment using Docker Compose.
- **Pushing** your Docker images to DockerHub for global accessibility.

Requirements

Before starting, ensure you have the following installed:

- **Operating System**: Windows or Linux
- **Software**:
    - Docker
    - Git
    - Python
    - Node.js
    - django
    - XAMPP

Step-by-Step Instructions

Step 1: Installing Required Software

Docker Installation

- **Windows**:
    1. Download Docker Desktop from the official Docker website : https://docs.docker.com/desktop/install/windows-install/
    2. Make sure to enable the WSL 2 feature on Windows and set it up as the backend for Docker.
    3. Follow the provided instructions during installation.
- **Linux**:
    1. Update your package manager: `sudo apt-get update`
    2. Install Docker: `sudo apt install docker.io`
    3. Start Docker: `sudo systemctl start docker` , and enable it at boot: `sudo systemctl enable docker`

    This will install Docker directly from the default Ubuntu repositories. After the installation, you can verify Docker is working with: **`sudo docker --version`**

4. Add your user to the Docker group to run commands without `sudo`: `sudo usermod -aG docker $USER`. (You will need to log out and back in for this to take effect).

And to test Docker, you can run: **sudo docker run hello-world**

Git Installation

- Download and install Git:
  - **Windows**: From [Git's official site](#).
  - **Linux**: Use `sudo apt-get install git`.

Python Installation

- Download and install Python from the [official Python website](#).
- Make sure Python and Pip are added to your system's PATH.

Node.js Installation

- Download and install Node.js from the [official Node.js website](#).

XAMPP (MySQL) Installation

- Download and install XAMPP from Apache Friends.
- Use XAMPP to manage the MySQL service on your system.

Step 2 : Installing the app

1. Clone the repository

First, clone the repository from GitHub:

```
git clone https://github.com/ReDXTechnologies/Devops-Cloud-TP_ENIS.git
```

Navigate into the cloned repository: `cd Devops-Cloud-TP_ENIS`

2. Get the folder `enis-app-tp`

Inside the repository, you only need the folder called `enis-app-tp`, which contains both the **frontend** and **backend** directories.

You can either manually extract the folder or use the terminal commands to navigate inside the required folder.

3. Running the Frontend

To set up and run the frontend:

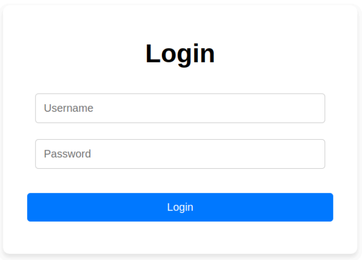
- Open a terminal.
- Navigate to the `frontend` folder: `cd enis-app-tp/frontend`
- Run the following commands to install the dependencies and start the application:
    a. `npm install`
    b. `npm start`

This will install all required npm packages and start the frontend development server.

- The frontend should now be accessible at `http://localhost:5173` by default.

4. Running the Backend

To set up and run the backend:



1. Open a new terminal.
2. Navigate to the `backend` folder: `cd enis-app-tp/backend`
3. Install the required Python packages using `pip`: `pip install -r requirements.txt`
4. Start the Django development server: `python3 manage.py runserver`

The backend should now be accessible at `http://localhost:8000` by default.

5. Setting Up the Database

1. **Start XAMPP**:
    ○ Launch XAMPP and start **Apache** and **MySQL**.
2. **Install django:**
    ○ python3 -m pip install django
3. **Create a Database**:
    ○ Open your browser and go to `http://localhost/phpmyadmin`.
    ○ Create a new database named `enis_tp`:
        ■ In the left panel, click on **New**.
        ■ Enter the database name as `enis_tp` and click **Create**.

6. Configure the Database Connection in Django

1. Open the `settings.py` file, located inside the `backend/backend` folder.
2. Modify the `DATABASES` configuration in the `settings.py` file as follows:

   ```
   DATABASES = {
       'default': {
           'ENGINE': 'django.db.backends.mysql',
           'NAME': 'enis_tp',  # Make sure this matches your database name
           'USER': 'your_username',  # Replace with your MySQL username
           'PASSWORD': 'your_password',  # Replace with your MySQL password
           'HOST': 'localhost',  # If you're using XAMPP, 'localhost' should be fine
           'PORT': '3306',  # Change if you're using a different port for MySQL
       }
   }
   ```

Replace `'your_username'` with your MySQL username (typically `root` for XAMPP).

Replace `'your_password'` with your MySQL password.

After configuring the database in the `settings.py` file, the next steps are to create the database tables using Django's `makemigrations` and `migrate` commands. Here's how you can proceed:

7. Apply Migrations and View Tables

1. **Run `makemigrations`**:
   First, you need to create new migrations based on any changes in your Django models: `python3 manage.py makemigrations`

   This command checks for any changes in the models and prepares them for migration.

2. **Run `migrate`**:
   Now, apply the migrations to create or modify the database tables: `python3 manage.py migrate`

   This command will apply the migration files and create the necessary tables in the MySQL database (`enis_tp`).

3. **View the Generated Tables in phpMyAdmin**:
   - Open your browser and go to http://localhost/phpmyadmin.

- Select the enis_tp database from the left-hand sidebar.
- You should now see the newly created tables, such as those for users, sessions, and any custom tables based on your Django models.

These tables are generated based on the models you defined in your Django app.

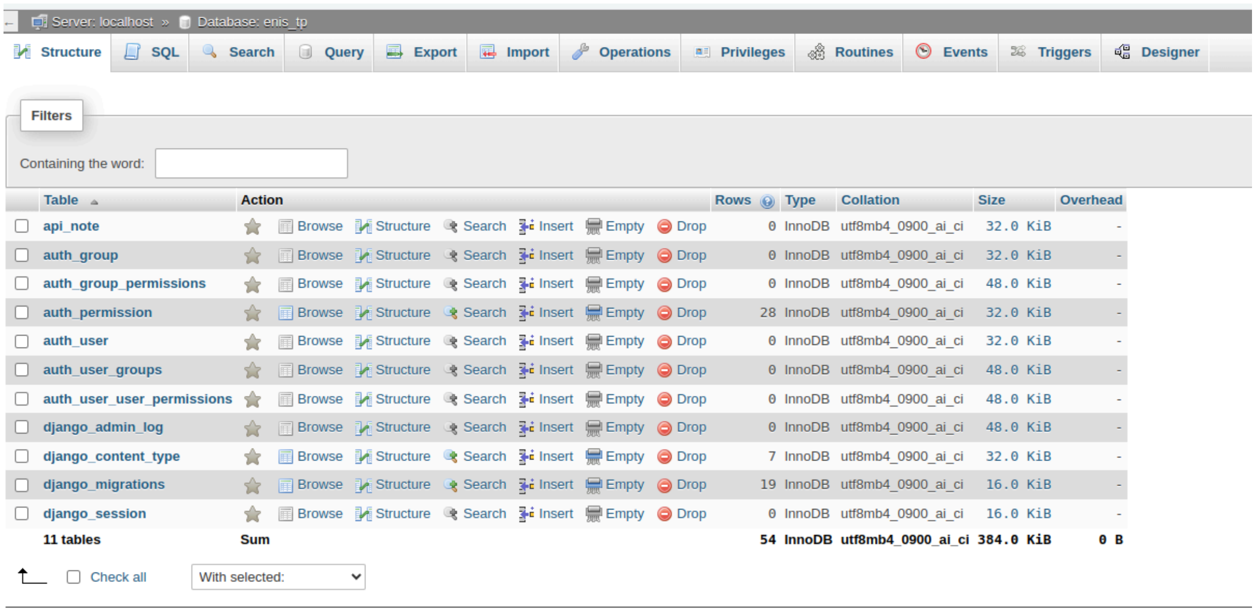
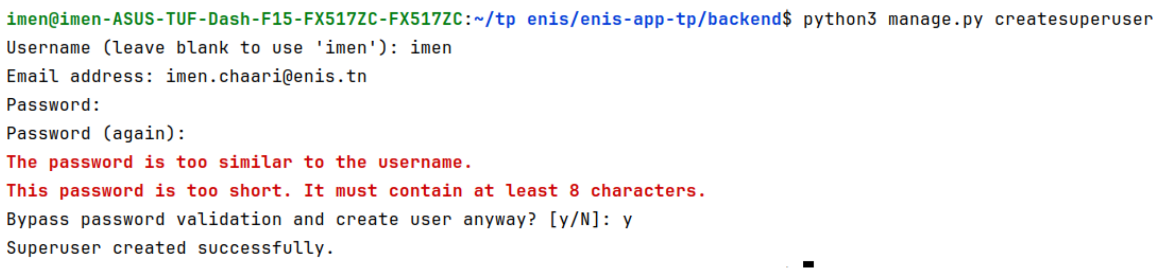


Table	Action	Rows	Type	Collation	Size	Overhead
api_note	Browse Structure Search Insert Empty Drop	0	InnoDB	utf8mb4_0900_ai_ci	32.0 KiB	-
auth_group	Browse Structure Search Insert Empty Drop	0	InnoDB	utf8mb4_0900_ai_ci	32.0 KiB	-
auth_group_permissions	Browse Structure Search Insert Empty Drop	0	InnoDB	utf8mb4_0900_ai_ci	48.0 KiB	-
auth_permission	Browse Structure Search Insert Empty Drop	28	InnoDB	utf8mb4_0900_ai_ci	32.0 KiB	-
auth_user	Browse Structure Search Insert Empty Drop	0	InnoDB	utf8mb4_0900_ai_ci	32.0 KiB	-
auth_user_groups	Browse Structure Search Insert Empty Drop	0	InnoDB	utf8mb4_0900_ai_ci	48.0 KiB	-
auth_user_user_permissions	Browse Structure Search Insert Empty Drop	0	InnoDB	utf8mb4_0900_ai_ci	48.0 KiB	-
django_admin_log	Browse Structure Search Insert Empty Drop	0	InnoDB	utf8mb4_0900_ai_ci	48.0 KiB	-
django_content_type	Browse Structure Search Insert Empty Drop	7	InnoDB	utf8mb4_0900_ai_ci	32.0 KiB	-
django_migrations	Browse Structure Search Insert Empty Drop	19	InnoDB	utf8mb4_0900_ai_ci	16.0 KiB	-
django_session	Browse Structure Search Insert Empty Drop	0	InnoDB	utf8mb4_0900_ai_ci	16.0 KiB	-
11 tables	Sum	54	InnoDB	utf8mb4_0900_ai_ci	384.0 KiB	0 B

9. Verify the Application is Working

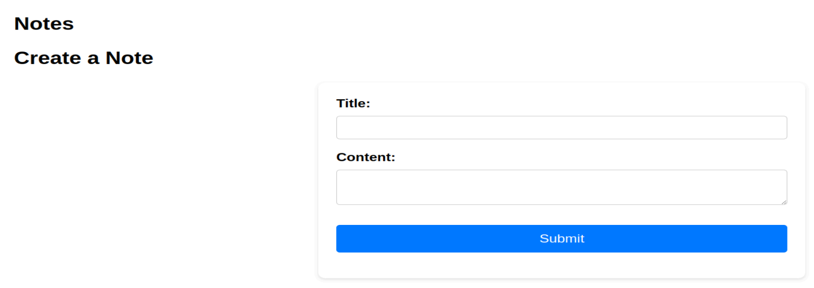
Once the database setup is complete and the tables are visible:

- Make sure both the **frontend** and **backend** are running.
- create a superuser



```
imen@imen-ASUS-TUF-Dash-F15-FX517ZC-FX517ZC:~/tp enis/enis-app-tp/backend$ python3 manage.py createsuperuser
Username (leave blank to use 'imen'): imen
Email address: imen.chaari@enis.tn
Password:
Password (again):
The password is too similar to the username.
This password is too short. It must contain at least 8 characters.
Bypass password validation and create user anyway? [y/N]: y
Superuser created successfully.
```

- You can now test the application by interacting with it from the frontend by logging with the username and password , then create a note


Notes

Create a Note

Step 3 : containerize the frontend

Create a Dockerfile in the frontend directory

a. In the frontend directory, create a new file called Dockerfile.
b. Add the following content to the Dockerfile:

```
FROM node:18-alpine as build

WORKDIR /app

COPY package*.json ./

RUN npm install

COPY . .

RUN npm run build

FROM nginx:alpine

COPY --from=build /app/dist /usr/share/nginx/html

EXPOSE 80

CMD ["nginx", "-g", "daemon off;"]
```

Explanation for Each Line

1. **`FROM node:18-alpine as build`**
   - **Purpose:** This line specifies the base image to use for the build stage. `node:18-alpine` is a lightweight version of the official Node.js image, based on Alpine Linux. The `as build` names this stage 'build', which allows for referencing it later.

- **Why Node.js:** Node.js is a runtime environment that lets you run JavaScript on the server side, which is necessary for building JavaScript-based applications like React or Angular.

2. **`WORKDIR /app`**
    - **Purpose:** Sets the working directory in the Docker container. All subsequent commands will be executed in this directory.
    - **Why `/app`:** It's a conventional directory where the application code resides inside the container, keeping the structure organized.
3. **`COPY package*.json ./`**
    - **Purpose:** Copies the `package.json` and `package-lock.json` (or `yarn.lock`) files into the container. These files define the project dependencies.
    - **Why Copy First:** By copying these files and running `npm install` before copying the rest of the application, Docker can cache installed dependencies. This avoids reinstalling all dependencies every build, as long as these files don't change.
4. **`RUN npm install`**
    - **Purpose:** Installs the project dependencies specified in `package.json` and `package-lock.json`.
    - **Why NPM Install:** This ensures all necessary libraries and frameworks are downloaded and available to the application within the container.
5. **`COPY . .`**
    - **Purpose:** Copies the remaining files and directories from the project directory on the host into the working directory `/app` in the container.
    - **Why Copy Everything:** Ensures all the source code, including scripts, stylesheets, and other assets needed for the build step, are inside the container.
6. **`RUN npm run build`**
    - **Purpose:** Executes the build script defined in `package.json`, which compiles the application into static files.
    - **Why Build:** Generates a production-ready version of the application with all assets optimized and bundled.
7. **`FROM nginx:alpine`**
    - **Purpose:** Starts a new stage from the `nginx:alpine` base image, which is a minimal distribution of Nginx on Alpine Linux.
    - **Why Nginx:** Nginx is used as a web server to serve the static files generated by the build process because of its efficiency and performance.
8. **`COPY --from=build /app/dist /usr/share/nginx/html`**
    - **Purpose:** Copies the static files from the `dist` directory in the build stage to the default directory where Nginx serves static files.
    - **Why Use Multi-Stage:** This keeps the production image size small, as only the compiled app files are copied over, not the entire Node.js environment.
9. **`EXPOSE 80`**
    - **Purpose:** Informs Docker that the container listens on port 80.

   - **Why Expose Port:** This allows you to map this port on the container to a port on the host, making the application accessible.

10. `CMD ["nginx", "-g", "daemon off;"]`
    - **Purpose:** Specifies the command that will be executed when the container starts.
    - **Why This Command:** Runs Nginx in the foreground. Docker containers stop when the main process ends, so keeping Nginx in the foreground prevents the container from exiting.

This Dockerfile effectively sets up a multi-stage build to optimize the final image size and performance, providing an efficient and organized way to build and serve a frontend application in production.

Now that you have your **Dockerfile** for the frontend ready, let's go through the steps to:

1. **Build the Docker image**
   a. Open your terminal.
   b. Navigate to the directory where your `Dockerfile` is located (in the `frontend` folder).
   c. Build the Docker image. You can give it a custom tag, like `frontend-app`.

```
docker build -t frontend-app .
```

Explanation:

- `docker build` tells Docker to build an image.
- `-t frontend-app` gives the image a name (tag), so you can refer to it later.
- `.` specifies the current directory (where the Dockerfile is located).

2. **Run the container and expose it on port 81**

```
docker run -d -p 81:80 frontend-app
```

`-d`: Detached Mode

- **Explanation**: The `-d` flag stands for **detached mode**. This means that the Docker container will run in the background, and you won't see the logs or output directly in your terminal.
- **Why Use It**: When you run a container in detached mode, you can continue to use your terminal for other commands, and the container will keep running in the background. This is useful for long-running services like web servers, databases, or backend applications.

`-p`: Port Mapping

- **Explanation**: The `-p` flag is used for **port mapping**. It maps a port on your **host machine** (your computer) to a port on the **Docker container**.
  The format for the `-p` flag is: -p <host_port>:<container_port>

  - **Host Port (left side)**: The port on your local machine that you want to use to access the application.
  - **Container Port (right side)**: The port inside the Docker container where the application is listening.

3. **Visit the application:**
   a. Open your browser and go to http://localhost:81

Step 4 : containerize the backend

Create a Dockerfile in the backend directory

c. In the backend directory, create a new file called Dockerfile.
d. Add the following content to the Dockerfile:

```
FROM python:3.10-buster

WORKDIR /app

COPY requirements.txt ./

RUN pip install --no-cache-dir -r requirements.txt

COPY . .

EXPOSE 8000

CMD ["python3", "manage.py", "runserver", "0.0.0.0:8000"]
```

Explanation for Each Line

1. **`FROM python:3.10-buster`**
   - **Purpose: This sets the base image for the container.**
     `python:3.10-buster` is a version of the official Python image that is based on Debian Buster, a stable and full-featured Linux distribution.
   - **Why Python 3.10: Python 3.10 is a more recent version of Python that includes important new features and performance improvements.**

Django, being a Python web framework, requires a Python runtime, and Python 3.10 ensures compatibility with modern libraries and Django features.

2. `WORKDIR /app`
   - Purpose: This command sets the working directory to `/app` inside the container. All subsequent commands will be executed within this directory.
   - Why Set a Working Directory: It keeps the file structure organized inside the container.
3. `COPY requirements.txt ./`
   - Purpose: This copies the `requirements.txt` file (which lists the dependencies) from your local machine into the container's working directory.
   - Why Only Copy `requirements.txt` First: By copying the `requirements.txt` file separately, Docker can cache the installed dependencies and avoid re-installing them unless the `requirements.txt` file changes.
4. `RUN pip install --no-cache-dir -r requirements.txt`
   - Purpose: This command installs all Python dependencies listed in `requirements.txt`.
   - Why `pip install`: Django and its dependencies need to be installed inside the container for the application to run.
   - Why Use `--no-cache-dir`: This option prevents `pip` from storing unnecessary cache, which reduces the size of the final Docker image.
5. `COPY . .`
   - Purpose: This copies the rest of your Django application's code into the container. This includes your Python files, Django settings, static files, etc.
   - Why Copy All Code: The entire application code is needed to run the Django app.
6. `EXPOSE 8000`
   - Purpose: This tells Docker that the container will listen on port 8000, which is Django's default port when using the development server.
   - Why Expose the Port: This allows you to access the application through port 8000 on your machine or the mapped port you define when running the container.
7. `CMD ["python", "manage.py", "runserver", "0.0.0.0:8000"]`
   - Purpose: This command tells Docker to start the Django development server. The `runserver` command will start the server and bind it to all network interfaces (`0.0.0.0`), making it accessible externally via port 8000.
   - Why Use `runserver`: This command starts the Django application in development mode, which is useful for local testing. For production, you'd use something like Gunicorn instead.

Now that you have your **Dockerfile** for the backend ready, let's go through the steps to:

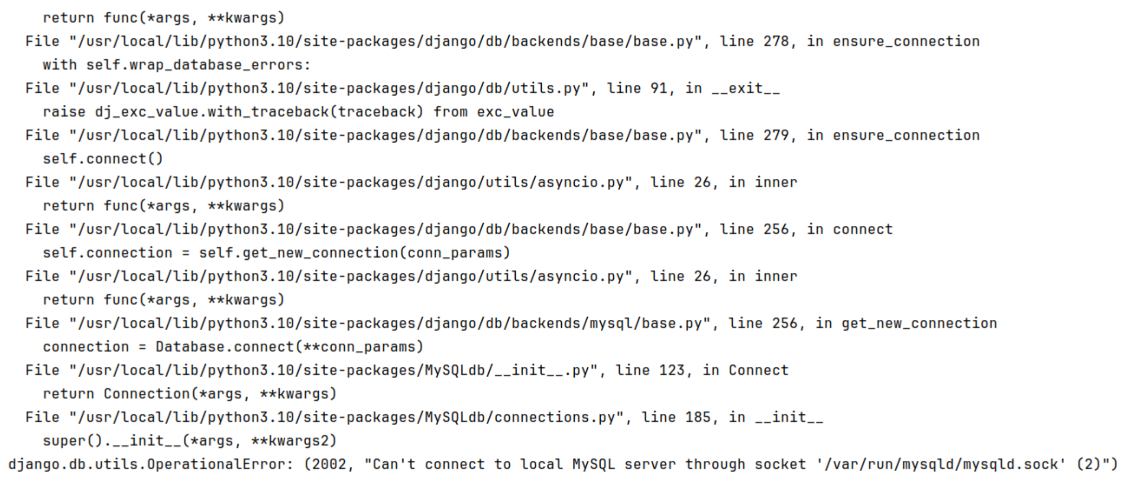
4. **Build the Docker image**
   a. Open your terminal.
   b. Navigate to the directory where your `Dockerfile` is located (in the `backend` folder).
   c. Build the Docker image. You can give it a custom tag, like `backend-app`.

   ```
   docker build -t backend-app .
   ```

5. **Run the container and expose it on port 8001 and let's see the logs**

   ```
   docker run -p 9000:8000 backend-app
   ```

you will likely face this error



```
    return func(*args, **kwargs)
  File "/usr/local/lib/python3.10/site-packages/django/db/backends/base/base.py", line 278, in ensure_connection
    with self.wrap_database_errors:
  File "/usr/local/lib/python3.10/site-packages/django/db/utils.py", line 91, in __exit__
    raise dj_exc_value.with_traceback(traceback) from exc_value
  File "/usr/local/lib/python3.10/site-packages/django/db/backends/base/base.py", line 279, in ensure_connection
    self.connect()
  File "/usr/local/lib/python3.10/site-packages/django/utils/asyncio.py", line 26, in inner
    return func(*args, **kwargs)
  File "/usr/local/lib/python3.10/site-packages/django/db/backends/base/base.py", line 256, in connect
    self.connection = self.get_new_connection(conn_params)
  File "/usr/local/lib/python3.10/site-packages/django/utils/asyncio.py", line 26, in inner
    return func(*args, **kwargs)
  File "/usr/local/lib/python3.10/site-packages/django/db/backends/mysql/base.py", line 256, in get_new_connection
    connection = Database.connect(**conn_params)
  File "/usr/local/lib/python3.10/site-packages/MySQLdb/__init__.py", line 123, in Connect
    return Connection(*args, **kwargs)
  File "/usr/local/lib/python3.10/site-packages/MySQLdb/connections.py", line 185, in __init__
    super().__init__(*args, **kwargs2)
django.db.utils.OperationalError: (2002, "Can't connect to local MySQL server through socket '/var/run/mysqld/mysqld.sock' (2)")
```

The error occurred because Django was trying to connect to MySQL **locally** (on the same container) using the Unix socket, but there was no MySQL server running in that container. Docker containers are isolated environments, so Django cannot automatically see other services like MySQL unless you specifically connect the containers via a network and update your `settings.py` to point to the MySQL container.

you have to :

- Run MySQL in its own container,
- Connect the containers via a bridge network,
- Update `settings.py` to point to the MySQL container,

Step 5 : Pull MySQL Database from Docker Hub

Since the error indicates that Django cannot connect to the MySQL database, we'll run a MySQL container that Django can connect to.

1. **Pull MySQL Docker Image**: In your terminal, run the following command to pull the official MySQL image: `docker pull mysql:8.0`

This will pull MySQL 8.0, which is a stable version compatible with Django and `mysqlclient`.

2. **Create a Docker Network (Bridge) :** We need to create a custom Docker network to allow the Django container and the MySQL container to communicate.

   ```
   docker network create my_bridge
   ```

3. **Run MySQL Container:** Now, run the MySQL container and connect it to the network.

   ```
   docker run --name mysql-db --network my_bridge -e
   MYSQL_ROOT_PASSWORD=yourpassword -e MYSQL_DATABASE=enis_tp -p
   3307:3306 -d mysql:8.0
   ```

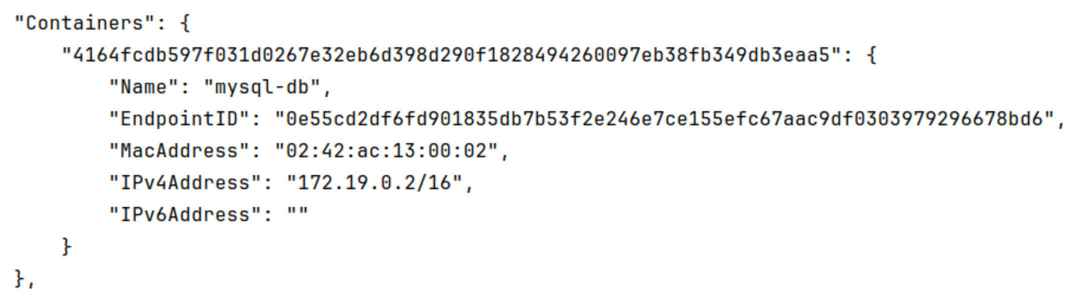
- **`--name mysql-db`: This sets the container name as `mysql-db`.**
- **`--network my_bridge`: Connects the container to the custom network.**
- **`-e MYSQL_ROOT_PASSWORD=yourpassword`: Sets the MySQL root password. Replace `yourpassword` with a secure password.**
- **`-e MYSQL_DATABASE=enis_tp`: Automatically creates the database `enis_tp`.**
- **`-p 3307:3306`: Exposes MySQL's default port (3306).**
- **`-d`: Runs MySQL in detached mode.**

To check the created database inside the container and view the containers in your Docker network, follow these steps:

1. View Containers in a Specific Network

To see the list of containers connected to the `my_bridge` network:

```
docker network inspect my_bridge
```



```
"Containers": {
    "4164fcdb597f031d0267e32eb6d398d290f1828494260097eb38fb349db3eaa5": {
        "Name": "mysql-db",
        "EndpointID": "0e55cd2df6fd901835db7b53f2e246e7ce155efc67aac9df030397929667 8bd6",
        "MacAddress": "02:42:ac:13:00:02",
        "IPv4Address": "172.19.0.2/16",
        "IPv6Address": ""
    }
},
```

2. Access the MySQL Database in the Container

To access the MySQL database inside the `mysql-db` container, you can connect to the MySQL instance running inside the container.

1. **Run the following command to enter the container's shell**:

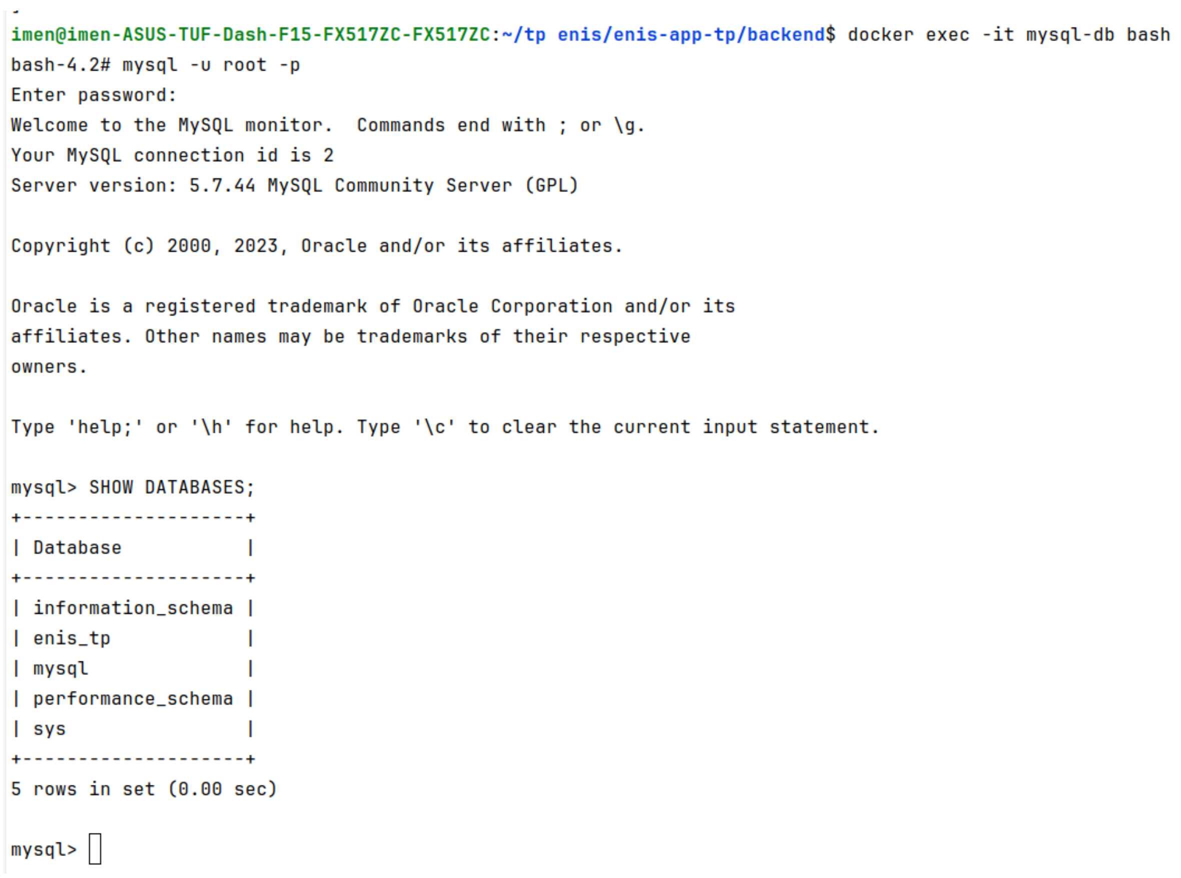
   ```
   docker exec -it mysql-db bash
   ```

2. **Once inside the container, log into MySQL**:

   ```
   mysql -u root -p
   ```

3. **Check the existing databases**:

   Once logged into MySQL, you can list the databases with the following SQL command: `SHOW DATABASES;`



```
imen@imen-ASUS-TUF-Dash-F15-FX517ZC-FX517ZC:~/tp enis/enis-app-tp/backend$ docker exec -it mysql-db bash
bash-4.2# mysql -u root -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 2
Server version: 5.7.44 MySQL Community Server (GPL)

Copyright (c) 2000, 2023, Oracle and/or its affiliates.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> SHOW DATABASES;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| enis_tp            |
| mysql              |
| performance_schema |
| sys                |
+--------------------+
5 rows in set (0.00 sec)

mysql> 
```

Update Django `settings.py` to Connect to MySQL on Port 3307

1. **Open `settings.py`**:
   Locate the `DATABASES` configuration in your `settings.py` file and modify it to match the port exposed by the MySQL container:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
```

```
        'NAME': 'enis_tp',  # Database name created in MySQL container
        'USER': 'root',  # MySQL root user
        'PASSWORD': 'yourpassword',  # The MySQL root password you set when starting the MySQL container
        'HOST': 'mysql-db',  # The name of the MySQL container
        'PORT': '3306',  # Port of the MySQL container
    }
}
```

Explanation:

- **`'HOST': 'mysql-db'`**: This is the container name for MySQL (`mysql-db`). Docker will resolve this to the correct IP address inside the Docker network.
- **`'PORT': '3307'`**: The port number 3307 matches the port that was exposed by the MySQL container on the host (as shown by your `docker ps` command).

Rebuild the Backend Docker Image

Once the `settings.py` file is updated, rebuild your backend Docker image to ensure the updated configuration is used.

1. **Rebuild the Docker image**:

```
docker build -t backend-app .
```

Run the Backend Container in the Same Network as MySQL

Now that you've rebuilt the backend, you need to run the backend container inside the same Docker network (`my_bridge`) as the MySQL container to enable communication.

1. **Run the Backend Container**:

```
docker run --name backend-app --network my_bridge -d -p 9000:8000
backend-app
```

Verify Database Connection

Enter the `backend-app` container to check if the Django app can connect to the MySQL database : `docker exec -it backend-app ping mysql-db`

Apply Migrations and Create the Superuser

After running the backend container, enter the container to apply the migrations and create a superuser.

Get the Backend Container ID:
First, list all running containers to get the container ID for the backend container:

```
docker ps
```

Enter the Backend Container:

Use the container ID to enter the backend container:

```
docker exec -it backend-app /bin/bash
```

Run Migrations:

Inside the container, run the following commands to apply the Django database migrations:

```
python3 manage.py makemigrations
```

```
python3 manage.py migrate
```

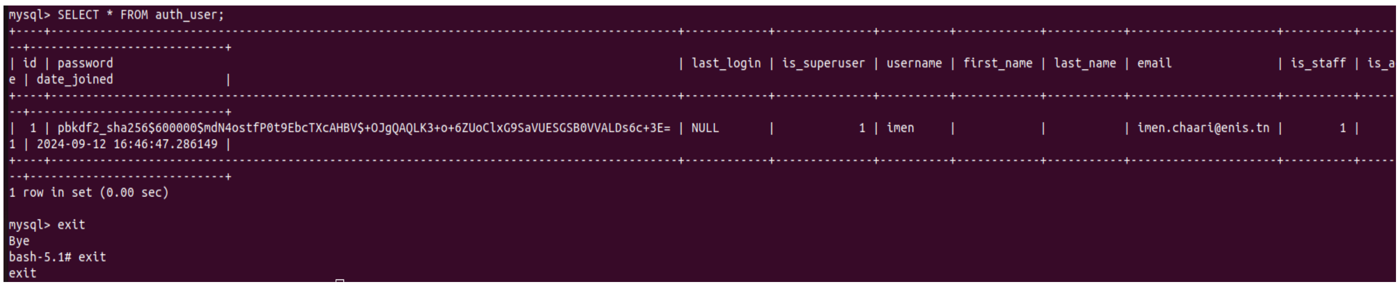
Create a Superuser:

To create a superuser for accessing the Django admin panel, run:

```
python manage.py createsuperuser
```

To see the created user inside the mysql container :

- **docker exec -it mysql-db bash**
- **mysql -u root -p (give password when you are prompted)**
- **USE enis_tp;**
- **SELECT * FROM auth_user;**



```
mysql> SELECT * FROM auth_user;
+----+------------------------------------------------------------------------------------------+------------+--------------+----------+------------+-----------+---------------------+----------+-----
--+----------------------------+
| id | password                                                                                 | last_login | is_superuser | username | first_name | last_name | email               | is_staff | is_a
e | date_joined                |
+----+------------------------------------------------------------------------------------------+------------+--------------+----------+------------+-----------+---------------------+----------+-----
--+----------------------------+
|  1 | pbkdf2_sha256$600000$mdN4ostfP0t9EbcTXcAHBV$+OJgQAQLK3+o+6ZUoClxG9SaVUESGSB0VVALDs6c+3E= | NULL       |            1 | imen     |            |           | imen.chaari@enis.tn |        1 |
1 | 2024-09-12 16:46:47.286149 |
+----+------------------------------------------------------------------------------------------+------------+--------------+----------+------------+-----------+---------------------+----------+-----
--+----------------------------+
1 row in set (0.00 sec)

mysql> exit
Bye
bash-5.1# exit
exit
```

Step 6 : Connect all containers in same my_bridge network

Now that the backend and MySQL are correctly set up and connected, we will connect the **frontend** to the **backend**.

We’ll modify the API configuration in the frontend code to point to the backend container by its Docker container name (instead of `localhost`), rebuild the frontend, and run the container in the same network as the backend and database.

Open the Frontend Code:

- Locate the `frontend/src/api.js` file.
- In this file, you will find the `axios` configuration that handles API requests.

Modify the `baseURL`:

- Change the base URL from `http://localhost:8000` to the port of the backend container (`9000`)

Update the `api.js` file:

```
const api = axios.create({
  baseURL: "http://localhost:9000"
});
```

Explanation:

External Access to Backend:

- **Port Mapping:** Ensure the backend container's port is mapped to a port on the host machine. This allows external access (from the browser) to the backend via the host machine's IP address or `localhost` (if accessing locally).
- **Update Axios Configuration for Browser Access:** The frontend's axios configuration should point to the host's address that is accessible from the browser. This is typically `localhost` with the mapped port if you’re working on your local machine.

Rebuild the Frontend Docker Image

Since we’ve made changes to the frontend source code, you need to rebuild the Docker image.

Navigate to the Frontend Directory: `docker build -t frontend-app .`

Stop and Remove the Previous Frontend Container

If you have a previous frontend container running, stop and remove it before starting a new one.

Stop the Running Frontend Container:
First, get the container ID by listing all running containers: `docker ps`

Then stop the frontend container: `docker stop <frontend-container-id>`

Remove the Frontend Container:

After stopping the container, remove it: `docker rm <frontend-container-id>`

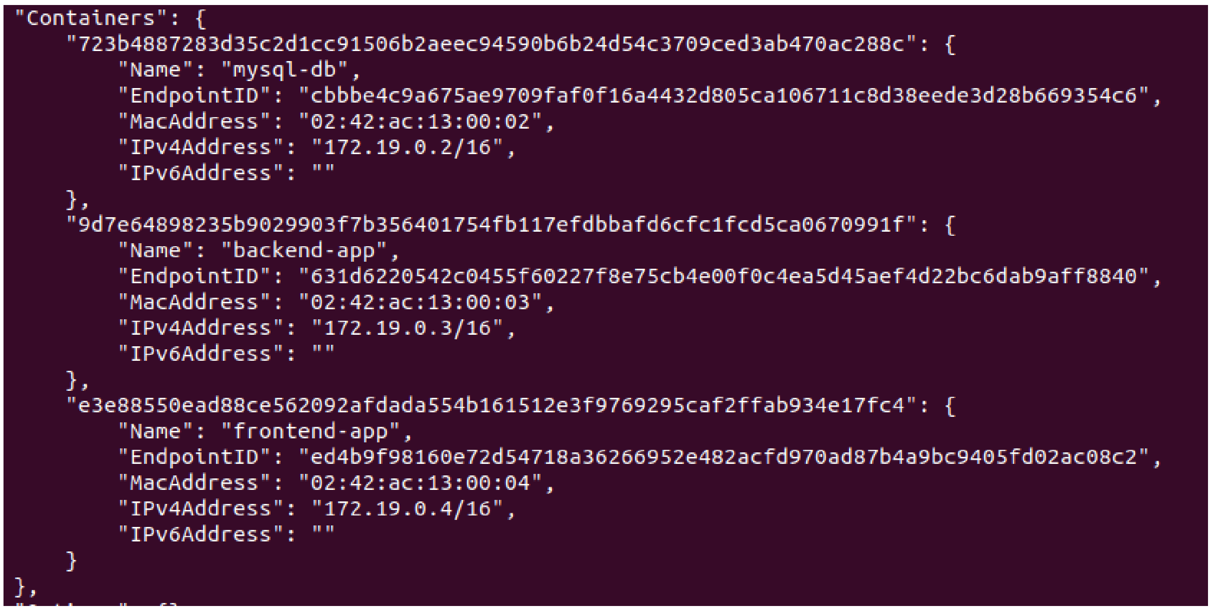
Run the New Frontend Container on the Same Network

Now, run the newly built frontend container and make sure it is connected to the same `my_bridge` network as the backend and MySQL containers.

1. **Run the Frontend Container:10***

```
docker run --name frontend-app --network my_bridge -d -p 81:80
frontend-app
```

Now when you inspect the network you should see all containers are connected



```
"Containers": {
    "723b4887283d35c2d1cc91506b2aeec94590b6b24d54c3709ced3ab470ac288c": {
        "Name": "mysql-db",
        "EndpointID": "cbbbe4c9a675ae9709faf0f16a4432d805ca106711c8d38eede3d28b669354c6",
        "MacAddress": "02:42:ac:13:00:02",
        "IPv4Address": "172.19.0.2/16",
        "IPv6Address": ""
    },
    "9d7e64898235b9029903f7b356401754fb117efdbbafd6cfc1fcd5ca0670991f": {
        "Name": "backend-app",
        "EndpointID": "631d6220542c0455f60227f8e75cb4e00f0c4ea5d45aef4d22bc6dab9aff8840",
        "MacAddress": "02:42:ac:13:00:03",
        "IPv4Address": "172.19.0.3/16",
        "IPv6Address": ""
    },
    "e3e88550ead88ce562092afdada554b161512e3f9769295caf2ffab934e17fc4": {
        "Name": "frontend-app",
        "EndpointID": "ed4b9f98160e72d54718a36266952e482acfd970ad87b4a9bc9405fd02ac08c2",
        "MacAddress": "02:42:ac:13:00:04",
        "IPv4Address": "172.19.0.4/16",
        "IPv6Address": ""
    }
},
```

let's check the connectivity : http://localhost:81/login and try to connect with the created user , once log in , Congratulations it is all set up and all containers are connecting to each other , enjoy creating your notes

Internal access to Backend:

docker exec -it **frontend-app** curl -X POST http://**backend-app:8000**/api/token/   -H "Content-Type: application/json"   -d '{"username": "imen", "password": "imen123"}'

docker exec -it **frontend-app** curl -X POST http://**backend-app:8000**/api/token/   -H "Content-Type: application/json"   -d '{"username": "nessrine", "password": "nessrine123"}'

{"refresh":"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ0b2tlbl90eXBlIjoicmVmcmVzaCIsImV4cCI6MTcyNjI1NTU1NCwiaWF0IjoxNzI2MTY5MTU0LCJqdGkiOiIwNjU4ZTgzMjA0NGY0NTE4YTc5MTYwZWMyOTlhMjNhYyIsInVzZXJfaWQiOjF9.Ctu0Jq7yvAsPquyBjzvqk9AadIGxr

SmmHm3XAzTL8X0","access":"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ0b2tlbl90eXBlI joiYWNjZXNzIiwiZXhwIjoxNzI2MTcwOTU0LCJpYXQiOjE3MjYxNjkxNTQsImp0aSI6IjJhMTN mMzVkZjE1YTQ0N2ZhMGM2YTNlNDBiNzJmMGNlIiwidXNlcl9pZCI6MX0.GAqbtXgej1S4v9 FsPZRzP3F1RuW_4Xh7cWAW_4tEt4w"}

Step 7 :Docker Compose Setup and Deployment

After running individual containers and verifying that they are connected to the same network we will move to **Docker Compose** to automate the process of starting all containers together in the same network, managing dependencies between services, and maintaining the MySQL database without losing data by preserving its volume.

Why Docker Compose?

Docker Compose simplifies managing multi-container Docker applications by allowing you to:

- Bring all containers up with a single command.
- Ensure that services are connected to the same network.
- Restart containers if they crash.
- Ensure that dependencies like MySQL are healthy before dependent services like the backend and frontend start.
- Bring everything down with a single command.

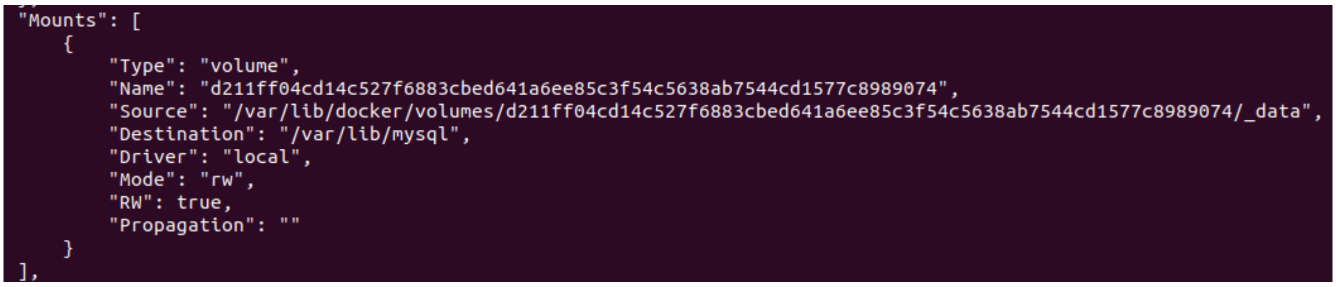
Preserving MySQL Data with Volumes

To ensure the MySQL database data is persistent, you need to **inspect the container mounts** and keep the volume linked to the MySQL database.

1. **Inspect the running MySQL container**:

```
docker inspect mysql-db
```

In the **"Mounts"** section, you will see something like:



```
"Mounts": [
    {
        "Type": "volume",
        "Name": "d211ff04cd14c527f6883cbed641a6ee85c3f54c5638ab7544cd1577c8989074",
        "Source": "/var/lib/docker/volumes/d211ff04cd14c527f6883cbed641a6ee85c3f54c5638ab7544cd1577c8989074/_data",
        "Destination": "/var/lib/mysql",
        "Driver": "local",
        "Mode": "rw",
        "RW": true,
        "Propagation": ""
    }
],
```

Take note of the **volume name**, in this case, `d211ff04cd14c527f6883cbed641a6ee85c3f54c5638ab7544cd1577c8989074`, which stores MySQL data. We will use this volume in our `docker-compose.yml` file to ensure the database persists even when the container is recreated.

Docker Compose File

Here is the full **docker-compose.yml** file that uses the pre-built images and attaches the MySQL database to its preserved volume:

```
version: '3.8'
services:
  mysql:
    image: mysql:8.0
    container_name: mysql-db
    restart: always
    environment:
      MYSQL_ROOT_PASSWORD: Imen_Chaari2023
      MYSQL_DATABASE: enis_tp
    ports:
      - "3307:3306"
    volumes:
      - mysql_data:/var/lib/mysql  # Use the preserved volume to keep data
    networks:
      - my_bridge
    healthcheck:
      test: ["CMD", "mysqladmin", "ping", "-h", "localhost"]
      interval: 30s
      timeout: 10s
      retries: 5
  backend-app:
    image: backend-app:latest  # Use the already built image
    container_name: backend-app
    restart: always
    ports:
      - "8001:8000"
    depends_on:
      mysql:
        condition: service_healthy
    networks:
      - my_bridge
    healthcheck:
      test: ["CMD-SHELL", "curl -f http://localhost:8000/admin/login/?next=/admin/ || exit 1"]
      interval: 30s
      timeout: 10s
      retries: 5

  frontend-app:
    image: frontend-app:latest  # Use the already built image
    container_name: frontend-app
    restart: always
    ports:
      - "81:80"
    depends_on:
      mysql:
        condition: service_healthy
      backend-app:
        condition: service_healthy
    networks:
      - my_bridge
    healthcheck:
      test: ["CMD-SHELL", "curl -f http://localhost || exit 1"]
      interval: 30s
      timeout: 10s
      retries: 3
networks:
  my_bridge:
    external: true
volumes:
  mysql_data:
    external:
      name: bce59e31de70940e568e74f8f29369f5c591a6651d869e8c17c03354e6419380
```

Explanation of `docker-compose.yml`

- **MySQL Service**: The `mysql` service uses the pre-built MySQL 8.0 image, connects it to the preserved `mysql_data` volume, and exposes it on port `3307`. It includes a health check to ensure MySQL is up before starting dependent services.

- **Backend Service**: The `backend-app` uses the pre-built backend image, starts after MySQL is healthy, and runs on port `8001`. The backend also has a health check to ensure it is up before starting the frontend.
- **Frontend Service**: The `frontend-app` uses the pre-built frontend image, starts after both MySQL and the backend are healthy, and is exposed on port `81`.
- **Health Checks**: These are critical to ensure that dependent services wait until their dependencies (like MySQL and backend) are up and running before they start.
- **Networks**: The `my_bridge` network ensures that all containers can communicate with each other.
- **Volumes**: The `mysql_data` volume ensures that the MySQL data is stored outside the container, providing persistence across container restarts or recreations.

Steps to Deploy with Docker Compose

1. **Stop and delete the currently running containers**: To ensure Docker Compose manages everything, we will first stop and remove the individual containers:

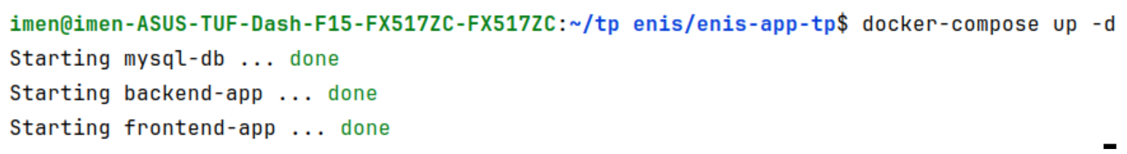
   ```
   docker stop backend-app frontend-app mysql-db

   docker rm backend-app frontend-app mysql-db
   ```

Bring up all services with Docker Compose: Use the following command to bring up all services (MySQL, backend, frontend) at once and ensure they are connected to the same network:

```
docker-compose up -d
```

This command will:

- Start the MySQL container, ensuring the data is preserved in the `mysql_data` volume.
- Start the backend and frontend containers, ensuring they depend on the health of the MySQL container.

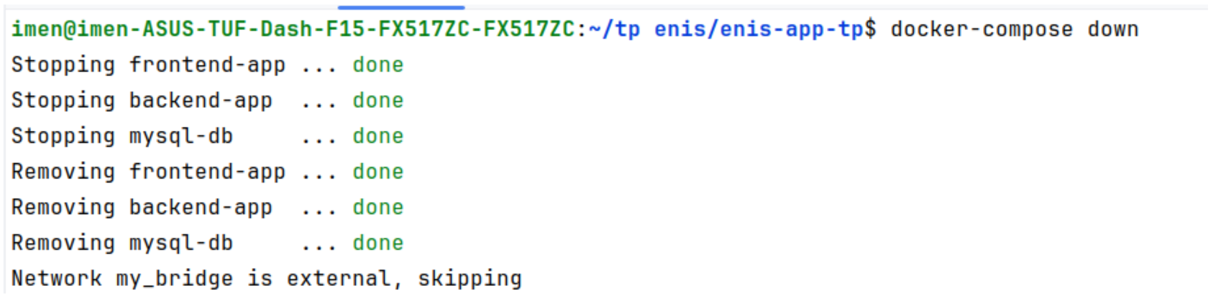


```
imen@imen-ASUS-TUF-Dash-F15-FX517ZC-FX517ZC:~/tp enis/enis-app-tp$ docker-compose up -d
Starting mysql-db ... done
Starting backend-app ... done
Starting frontend-app ... done
```

Test the connection: Open a browser and go to: http://localhost:81/login

Try to connect with the credentials you created. If the login works, congratulations! All containers are up and running, connected through the same network.

Bring down all containers: If you want to stop all containers and bring them down, you can simply run: `docker-compose down`



```
imen@imen-ASUS-TUF-Dash-F15-FX517ZC-FX517ZC:~/tp enis/enis-app-tp$ docker-compose down
Stopping frontend-app ... done
Stopping backend-app  ... done
Stopping mysql-db     ... done
Removing frontend-app ... done
Removing backend-app  ... done
Removing mysql-db     ... done
Network my_bridge is external, skipping
```

Step 8 :Pushing Docker Images to Docker Hub

In this final step, we will push your already built Docker images (frontend-app, backend-app, mysql-db) to Docker Hub. This allows you to store and share your images remotely, making them accessible from any machine.**Prerequisites:**

1. **Docker Hub Account**: Ensure you have a Docker Hub account. If you don't have one, you can create it at https://hub.docker.com/.
2. **Docker Login**: Log into Docker Hub from your terminal. Run: `docker login`

1. Tag Your Images

Each image must be tagged to associate it with your Docker Hub account. The naming convention is:`<docker-hub-username>/<image-name>:<tag>`

For example, if your Docker Hub username is `imenchaari` and the image is `backend-app`, you can tag it like this:

- `docker tag backend-app imenchaari/backend-app:latest`
- `docker tag frontend-app imenchaari/frontend-app:latest`

2. Push the Tagged Images

Once the images are tagged, you can push them to Docker Hub.

- `docker push imenchaari/backend-app:latest`
- `docker push imenchaari/frontend-app:latest`

Verify on Docker Hub

After pushing the images, go to https://hub.docker.com/ and log in. Navigate to your **Repositories** tab, and you should see the backend-app and frontend-app images listed.

Conclusion

You have now successfully pushed your images to Docker Hub, making them available to pull and run from anywhere. This is a crucial step in deploying your Dockerized applications to any environment.