KAZAKH-BRITISH TECHNICAL UNIVERSITY

Kazakh-British Technical University
School of Information Technologies and Engineering

Assignment #2
Web Application Development
Exploring Django with Docker

Prepared by: Maratuly T.
Checked by: Serek A.

Almaty, 2024

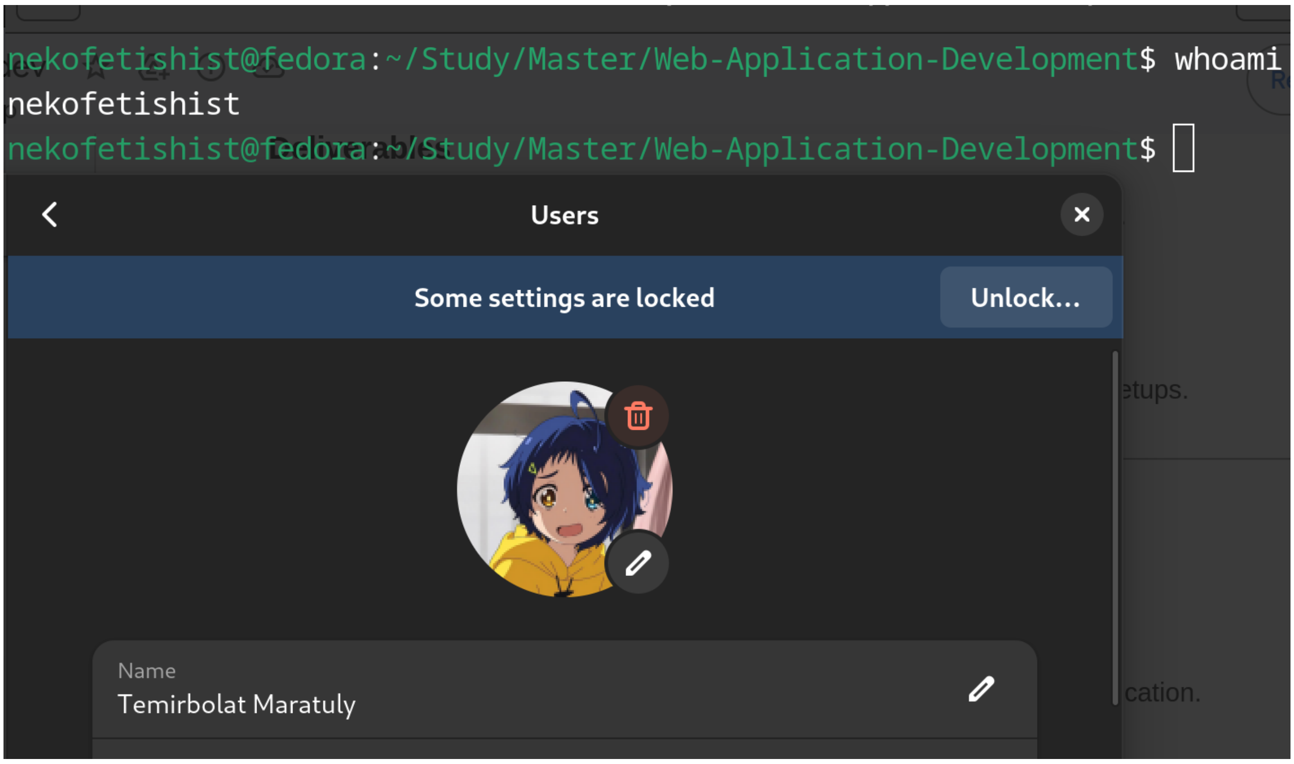
CONTENT

INTRODUCTION

This work is responsible for getting familiar with Django framework fundamentals and setting database for it using docker, docker compose, volumes, networking, etc. Docker is a great tool for developing as only having it we can install and launch all the required containers (with necessary images, run needed command) and easily control them. Fedora was an operational system (OS) which we used to cover this lab. Pay attention to a user’s username (here ‘nekofetishist’ is a username of Maratuly Temirbolat with ID 23MD0409).

1. Docker compose

Before the beginning of the lab's process pay attention that username 'nekofetishist' belongs to Maratuly Temirbolat (ID 23MD0409). Figure 1.1 illustrates who is the owner of the account which will be used to cover the work.



```
nekofetishist@fedora:~/Study/Master/Web-Application-Development$ whoami
nekofetishist
nekofetishist@fedora:~/Study/Master/Web-Application-Development$ 
```

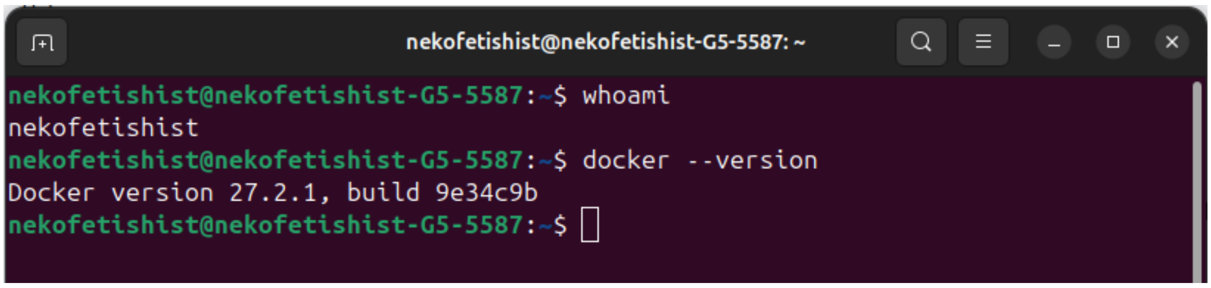
Name: Temirbolat Maratuly

Figure 1.1 - Username and account's owner credentials information

Thus, according to a figure, the command 'whoami' from the Fedora command line shows the current user, moreover, this username is used in the whole path (marked with green letters). The next subsections are responsible for docker compose initialization, its customization, building to images (postgresql and django application).

1.1 Exercise: 1 Docker compose file

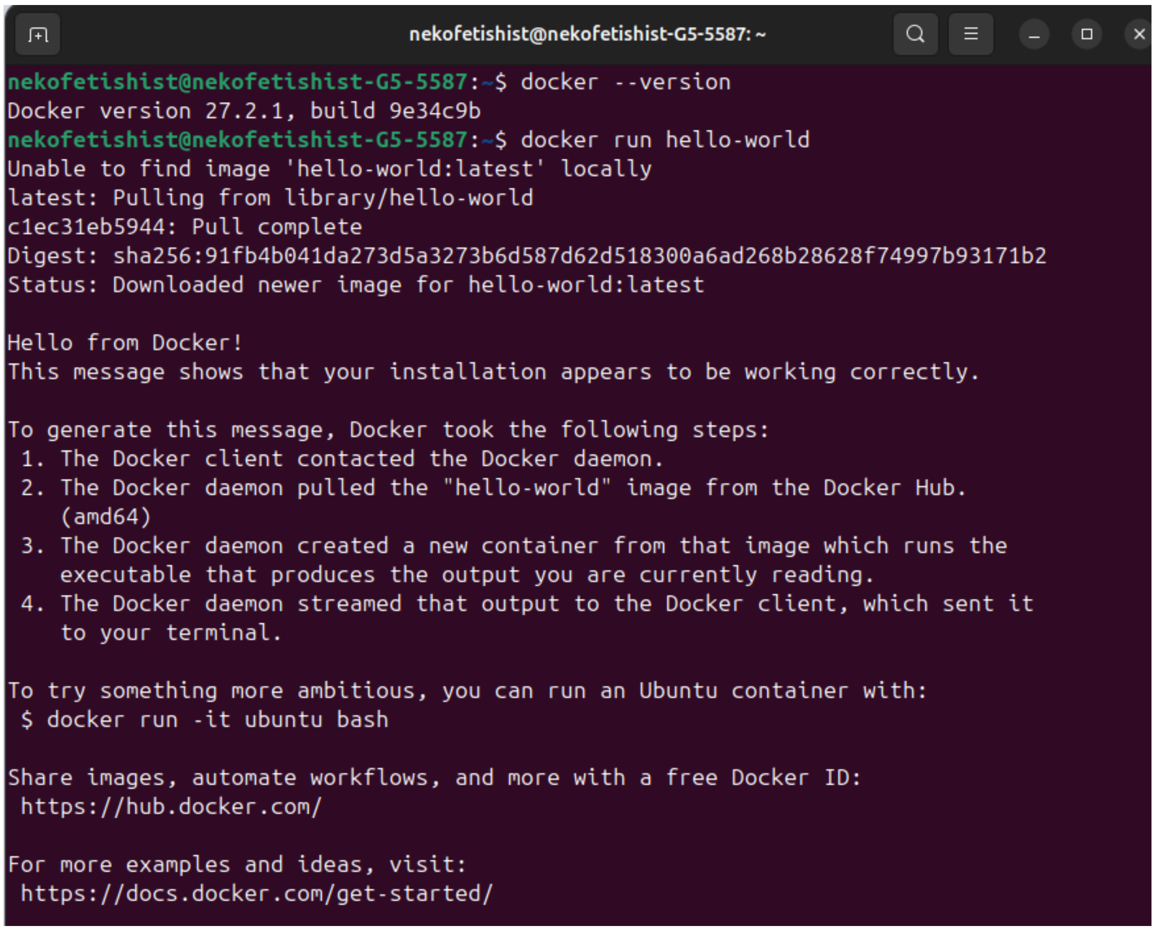
As soon as we verified the account's credentials, we needed to install docker for further work. All the work was done in command interface only. So, there are not any descriptions and explanations about docker desktop interface to work with run containers, related screenshots, etc. As we installed docker via official documentation, we need to make sure that it works. The best way to use 'docker --version' which must show the installed version of a docker. The information about docker's version is illustrated in Figure 1.2.



```
nekofetishist@nekofetishist-G5-5587:~$ whoami
nekofetishist
nekofetishist@nekofetishist-G5-5587:~$ docker --version
Docker version 27.2.1, build 9e34c9b
nekofetishist@nekofetishist-G5-5587:~$ 
```

Figure 1.2 - Docker's installed version information

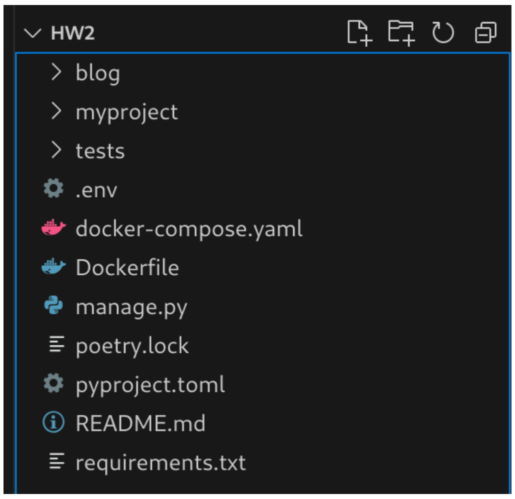
As we installed the docker, we can run 'hello-world' container to see that docker works properly with no issues and errors. Figure 1.3 demonstrates the results of this command. Docker will download 'hello-world' image if you do not have it.



```
nekofetishist@nekofetishist-G5-5587:~$ docker --version
Docker version 27.2.1, build 9e34c9b
nekofetishist@nekofetishist-G5-5587:~$ docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
c1ec31eb5944: Pull complete
Digest: sha256:91fb4b041da273d5a3273b6d587d62d518300a6ad268b28628f74997b93171b2
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://hub.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/get-started/
```

Figure 1.3 - Logs after running 'hello-world' container

As we know that docker works perfectly, we can create our Django application using command 'django-admin startproject myproject'. Then, we can create a simple application 'blog' using command 'python3 manage.py startapp blog' and add all necessary files such as 'Dockerfile', 'docker-compose.yaml', 'requirements.txt', 'pyproject.toml', and so on. It is also necessary to hold the project's packages at specified versions, and we need to have "env" file to hold all the virtual environments, required packages versions (installed from the ‘requirements.txt’ file). Figure 1.4 demonstrates the tree of the project.

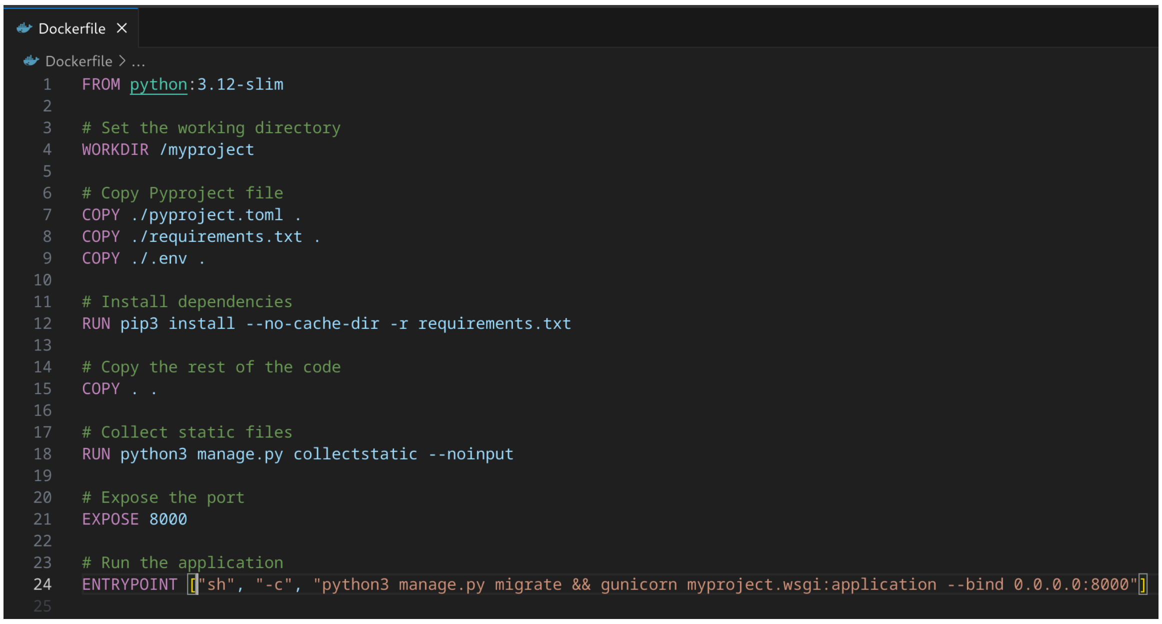


```
HW2
  blog
  myproject
  tests
  .env
  docker-compose.yaml
  Dockerfile
  manage.py
  poetry.lock
  pyproject.toml
  README.md
  requirements.txt
```

Figure 1.4 – Project tree of the project

According to the figure we can see that there are 3 main folders and 8 files. The "blog" directory is a Django app with functionality related to a blog (models, views, etc.), while "myproject" holds the main project configuration files, such as settings and URLs.

Let’s now look at “Dockerfile” to understand the project’s building approach. Figure 1.5 contains information about django’s docker file.



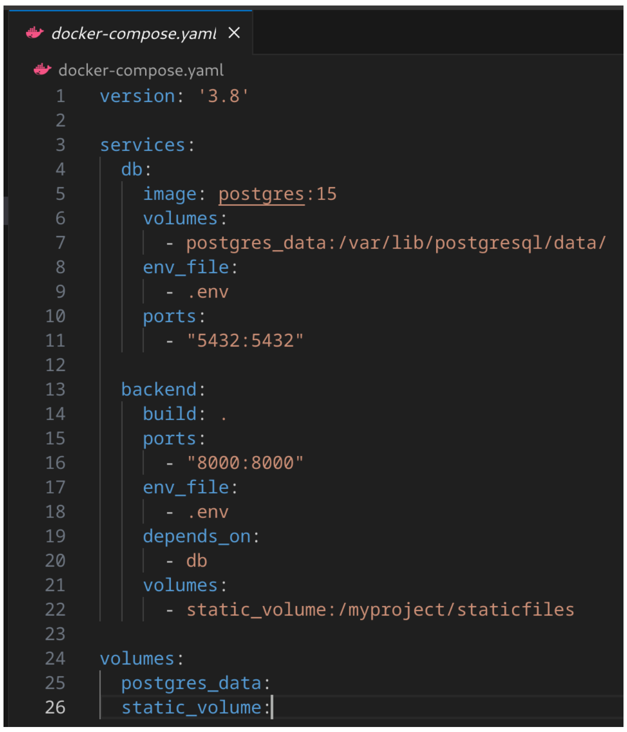
```
FROM python:3.12-slim

# Set the working directory
WORKDIR /myproject

# Copy Pyproject file
COPY ./pyproject.toml .
COPY ./requirements.txt .
COPY ./.env .

# Install dependencies
RUN pip3 install --no-cache-dir -r requirements.txt

# Copy the rest of the code
COPY . .

# Collect static files
RUN python3 manage.py collectstatic --noinput

# Expose the port
EXPOSE 8000

# Run the application
ENTRYPOINT ["sh", "-c", "python3 manage.py migrate && gunicorn myproject.wsgi:application --bind 0.0.0.0:8000"]
```

Figure 1.5 – Dockerfile’s details for project building

The project also includes several important files. The ".env" file stores environment variables (db variables). "docker-compose.yaml" and "Dockerfile" are used for creating and managing Docker containers, which helps with deploying the project in a consistent environment. "manage.py" is the main Django utility tool for running tasks such as the server and migrations. The "poetry.lock" and "pyproject.toml" files manage the project's dependencies, ensuring consistent package versions. Lastly, "README.md" provides documentation for the project, and "requirements.txt" lists the Python dependencies needed to run the project.

The file (Figure 1.5) contains all the comments about each command but emphasizing some notes it can be added that we use python 3.12 slim version and run the application using guvicorn to bind a specified url. Initially, we copy only “pyproject.toml”, “requirements.txt” and “.env” files because we want to cache these steps as we contain there all necessary dependencies. The description of the “Dockerfile” is now over.

Now we can start with “docker-compose.yaml” file. It is used locally to manage all the containers which are described there to make the project work correctly using all the services. For our case, we need to include images of “postgresql” (here it was decided to use 15th version) and locally built “Dockerfile” of the project. The information about “docker-compose” file data is shown in Figure 1.6.



```yaml
version: '3.8'

services:
  db:
    image: postgres:15
    volumes:
      - postgres_data:/var/lib/postgresql/data/
    env_file:
      - .env
    ports:
      - "5432:5432"

  backend:
    build: .
    ports:
      - "8000:8000"
    env_file:
      - .env
    depends_on:
      - db
    volumes:
      - static_volume:/myproject/staticfiles

volumes:
  postgres_data:
  static_volume:
```

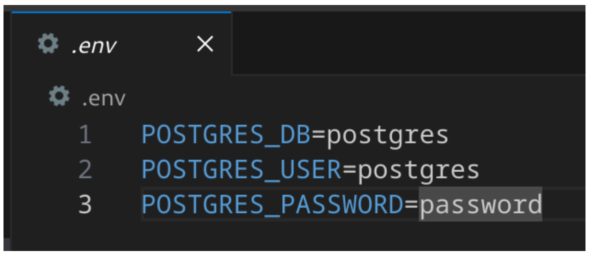
Figure 1.6 – Docker compose file’s configuration

This file defines a multi-service setup with Docker for a project using version '3.8'. It contains two services: "db" and "backend". The "db" service uses the “postgres:15 image”, its data stored in a volume called "postgres_data". It also uses an environment file named ".env" for configuration and exposes port "5432" to communicate with other services or the host.

The "backend" service is set to build from the current directory where the “docker-compose” is located, and it also references the ".env" file for its configuration. It depends on the "db" service, ensuring that the database is ready before the backend starts. The "backend" service is exposed on port "8000" and uses a volume named "static_volume" to store static files at the specified path in the container.

1.2 Exercise 2: Environmental variables

This section is responsible for setting environmental variables for “postgresql”, variables to configure connection of databases in Django. All the environmental variables are located in “.env” file and it is added into “env_file” section to set variables from there instead of manual way. The list of environmental variables is represented in Figure 1.7.



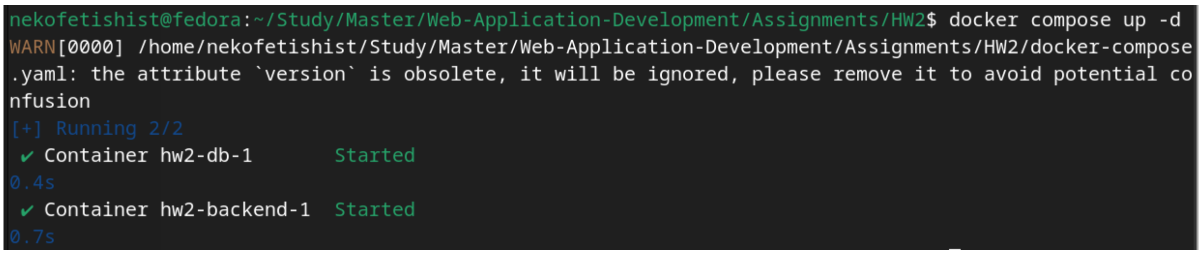
```
.env
POSTGRES_DB=postgres
POSTGRES_USER=postgres
POSTGRES_PASSWORD=password
```

Figure 1.7 – List of all the environmental variables

According to a list we have the following variables: “POSTGRES_DB”, “POSTGRES_USER” and “POSTGRES_PASSWORD”. In the exercise it was suggested to use “DB_NAME”, “DB_USER” and “DB_PASSWORD”, but according to a image documentation of the postgre sql, instead of “DB” prefix we need to use “POSTGRES” as it accepts only them.

1.3 Exercise 3: Container’s building and running

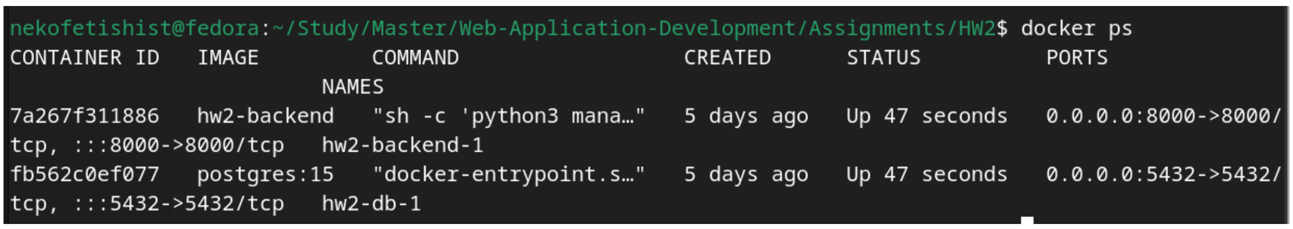
The aim of this subsection is to build configured docker compose file using commands. The command which we used to build and run is “docker compose up -d” meaning that we pull the images if they have not been downloaded yet and run in detached mode (in background). Figure 1.8 illustrates the process of launching and running docker compose file.



```
nekofetishist@fedora:~/Study/Master/Web-Application-Development/Assignments/HW2$ docker compose up -d
WARN[0000] /home/nekofetishist/Study/Master/Web-Application-Development/Assignments/HW2/docker-compose
.yaml: the attribute `version` is obsolete, it will be ignored, please remove it to avoid potential co
nfusion
[+] Running 2/2
 ✔ Container hw2-db-1       Started
0.4s
 ✔ Container hw2-backend-1  Started
0.7s
```

Figure 1.8 - View of a process to build and launch docker compose images

The proof of it shown in Figure 1.9 as both containers are working.



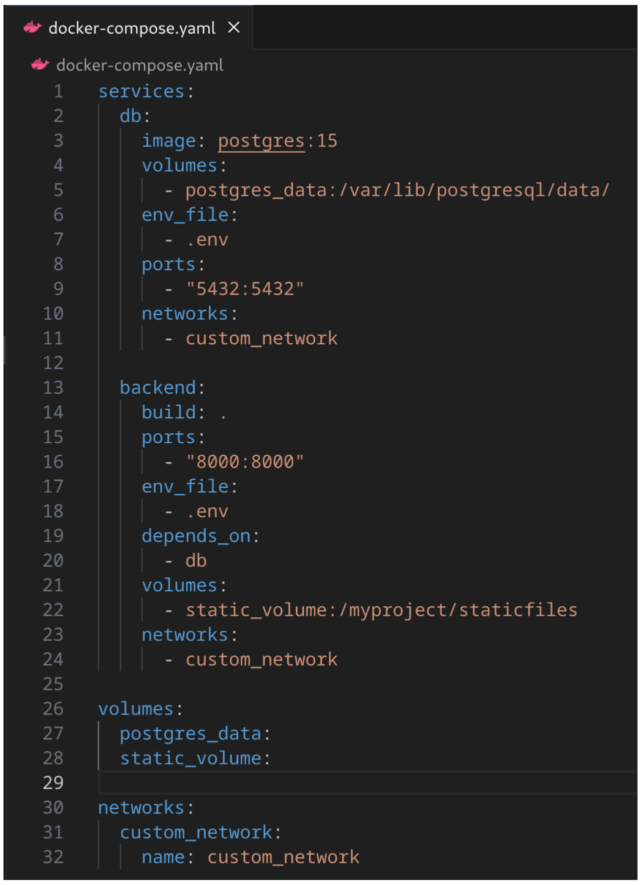
```
nekofetishist@fedora:~/Study/Master/Web-Application-Development/Assignments/HW2$ docker ps
CONTAINER ID   IMAGE         COMMAND                  CREATED      STATUS          PORTS
                                      NAMES
7a267f311886   hw2-backend   "sh -c 'python3 mana…"   5 days ago   Up 47 seconds   0.0.0.0:8000->8000/
tcp, :::8000->8000/tcp   hw2-backend-1
fb562c0ef077   postgres:15   "docker-entrypoint.s…"   5 days ago   Up 47 seconds   0.0.0.0:5432->5432/
tcp, :::5432->5432/tcp   hw2-db-1
```

Figure 1.9 – Command to show all the active containers

It can be clearly seen that two services ("db" and "backend") were built and launched successfully.

2. Docker networking and volumes

This section is aimed to edit current docker compose file to include own custom network and customize the services to work inside of this isolated network. To achieve this result we need to customize a bit our "docker-compose.yaml" file. Figure 1.10 holds the updated version of a file.



```yaml
docker-compose.yaml
services:
  db:
    image: postgres:15
    volumes:
      - postgres_data:/var/lib/postgresql/data/
    env_file:
      - .env
    ports:
      - "5432:5432"
    networks:
      - custom_network

  backend:
    build: .
    ports:
      - "8000:8000"
    env_file:
      - .env
    depends_on:
      - db
    volumes:
      - static_volume:/myproject/staticfiles
    networks:
      - custom_network

volumes:
  postgres_data:
  static_volume:

networks:
  custom_network:
    name: custom_network
```

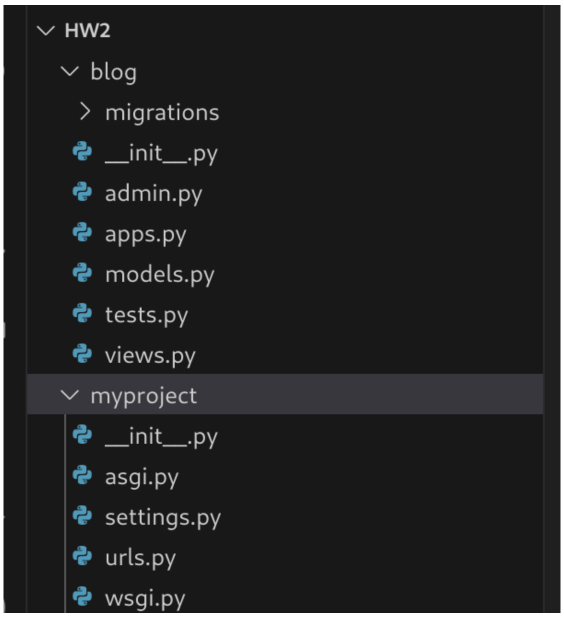
Figure 1.10 – Updated docker compose file with custom networks

Just by adding new section with networks we can specify where they will work. It is necessary to specify the networks at each service. It can be additionally seen that each service has extra row about networks where “custom_network” name is specified. By default, the driver “bridge” is used there and containers can see.

As for the volumes, there are 2 unique volumes “postgres_data”, “static_volume”. The first one is used to store tables, and, overall, database data in “data” folder for the “db” service in order not to lose data after stopping. The next volume is “static_volume” which is used in order not to lose all the django’s static files and hold them in “staticfiles” of the project.

3. Django application setup

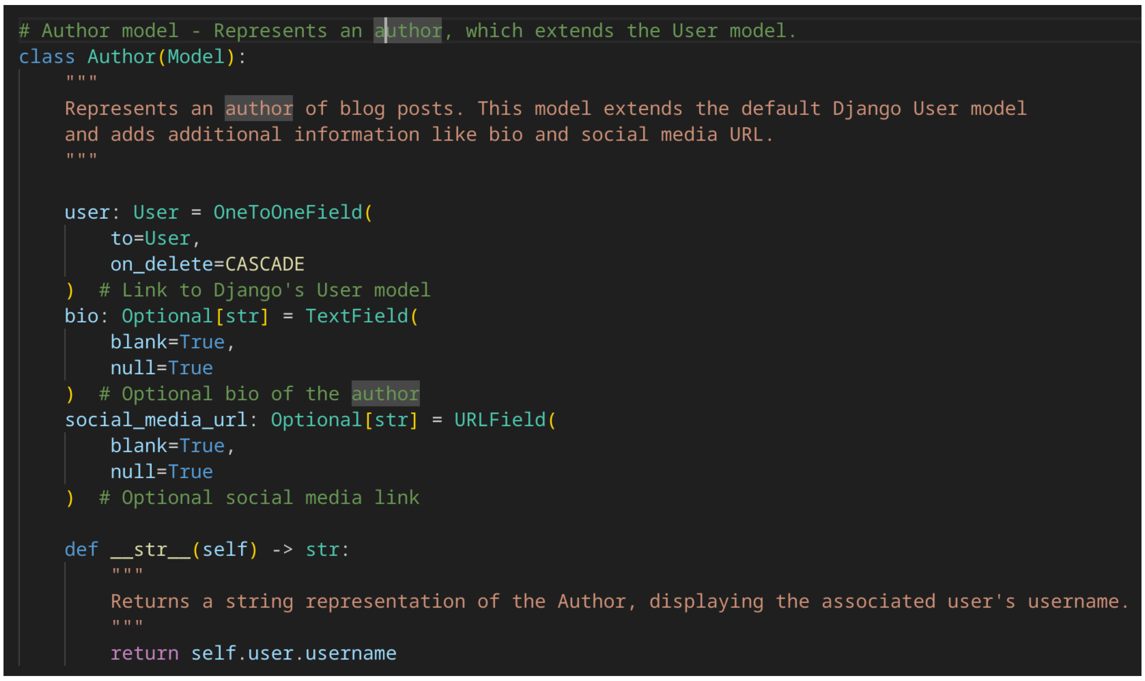
Django is a python based framework that is used to create web applications, basically, for MVP (minimum value product) projects where the load is not high. By default, Django uses MVC (model view controller) architecture approach, meaning that you combine frontend and backend parts together. For the current assignment we needed to create a sample application with 1 model. To create a new project we need to setup it using “django-admin startproject <project_name>” where we used “myproject” as a name. Django basically has applications which store models, necessary views, etc. Figure 1.11 illustrates unwrapped information about django’s tree project.



```
HW2
  blog
    migrations
    __init__.py
    admin.py
    apps.py
    models.py
    tests.py
    views.py
  myproject
    __init__.py
    asgi.py
    settings.py
    urls.py
    wsgi.py
```

Figure 1.11 – Django project’s tree project

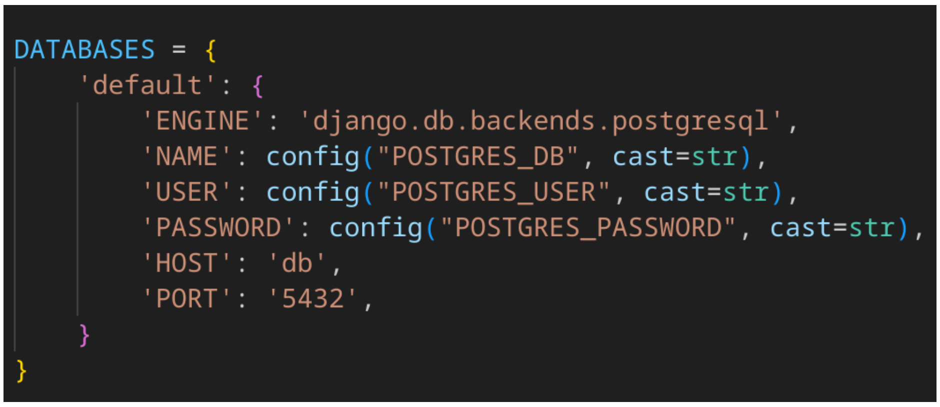
Firstly, it is necessary to add that this is a default project’s configuration, there are no any custom changes which usually take place in real production. So, according to a figure the “blog” application folder or any application folder has: “admin.py” to install the created models on administration site and configure their view, “apps.py” is applied to configure the application behavior e.g. its verbose name, etc., “models.py” is a file where all our models (db tables) are stored and configured related only to the application's scope, "tests.py" file is used to create all the tests, "views.py" in case of default Django (MVC approach) is used to create functions or methods to work as a provided between a model and a template. The "myproject" folder contains project's settings and commonly named as the whole project name, however, in real production the name is replaced with just "settings". The models related to a blog scope are stored in "models.py" file and shown in Figure 1.12 (here represented only 1 model due to the sizes).



```python
# Author model - Represents an author, which extends the User model.
class Author(Model):
    """
    Represents an author of blog posts. This model extends the default Django User model
    and adds additional information like bio and social media URL.
    """

    user: User = OneToOneField(
        to=User,
        on_delete=CASCADE
    )  # Link to Django's User model
    bio: Optional[str] = TextField(
        blank=True,
        null=True
    )  # Optional bio of the author
    social_media_url: Optional[str] = URLField(
        blank=True,
        null=True
    )  # Optional social media link

    def __str__(self) -> str:
        """
        Returns a string representation of the Author, displaying the associated user's username.
        """
        return self.user.username
```

Figure 1.12 – Author database model of blog application

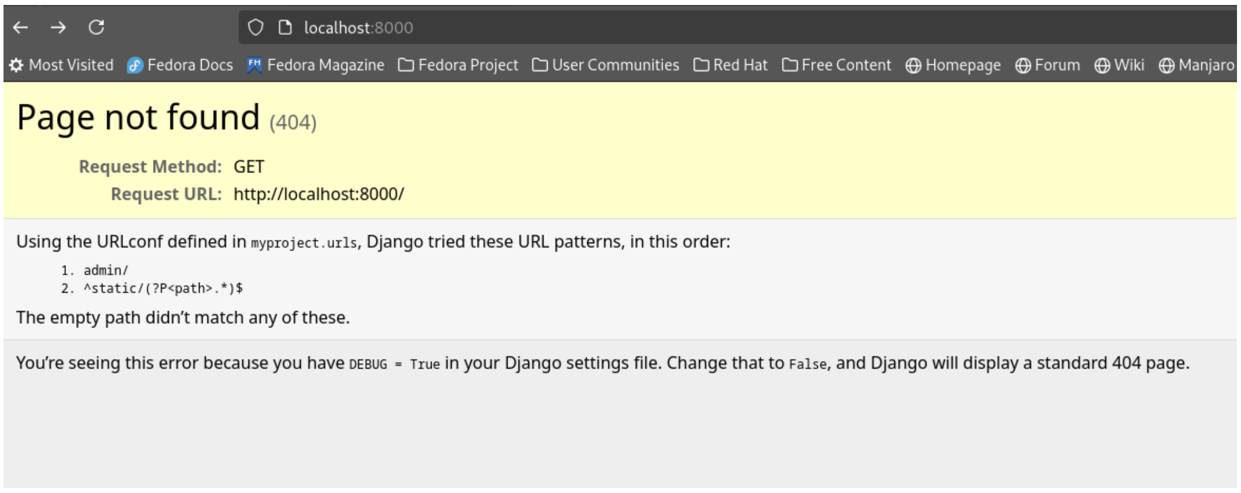
For this assignment the number of models and their content are not mandatory and do not play any role since we don't configure backend endpoints, etc. No need to cover all the fields since the comments about each of the step is left. Let's move to database configuration. The configuration of the database is illustrated in Figure 1.13.



```python
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': config("POSTGRES_DB", cast=str),
        'USER': config("POSTGRES_USER", cast=str),
        'PASSWORD': config("POSTGRES_PASSWORD", cast=str),
        'HOST': 'db',
        'PORT': '5432',
    }
}
```

Figure 1.13 – Database configuration in Django

By default, database configuration is hold in “settings.py” file of the “myproject” folder. In our case we need to estimate database configuration, but instead of “sqlite” we need to provide the configuration for “postgres”. Here, we replaced all the parameters such as engine (driver) is used from django package, while name, user, password are taken from environment variables (using “decouple” lib). The port is 5432 as we set in docker compose, and “db” is written as we had set the name of a service in docker compose as well. The Figure 1.14 represents that our application works perfectly as we started it via docker compose.



Page not found (404)

Request Method: GET
Request URL: http://localhost:8000/

Using the URLconf defined in `myproject.urls`, Django tried these URL patterns, in this order:

```
1. admin/
2. ^static/(?P<path>.*)$
```

The empty path didn’t match any of these.

You’re seeing this error because you have `DEBUG = True` in your Django settings file. Change that to `False`, and Django will display a standard 404 page.

Figure 1.14 – Working Backend application view started via docker compose

So, we can connect to a Django project via 8000 and through 5432 to a database as we configured it.

CONCLUSION

In conclusion, we managed to build our backend dockerfile and launched the project using docker compose, connecting it to a postgresql container. Moreover, we used environmental variables to provide them for database container and backend one at the same time.