Web Development
Report for Assignment-2

Assignment: Exploring Django with Docker

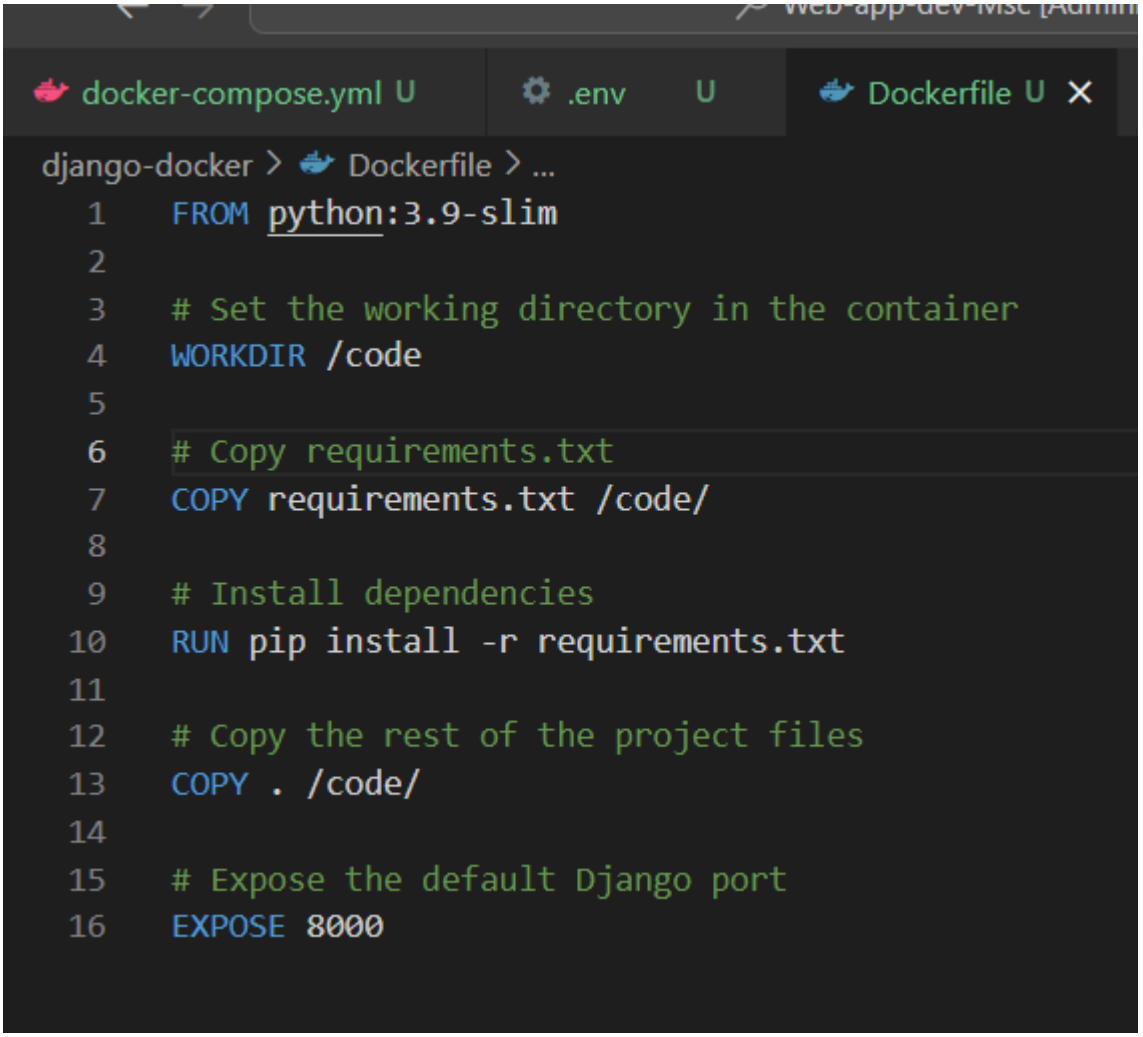
Software Engineering, 2nd year
Nurzhan Momynkul
ID: 23MD0414

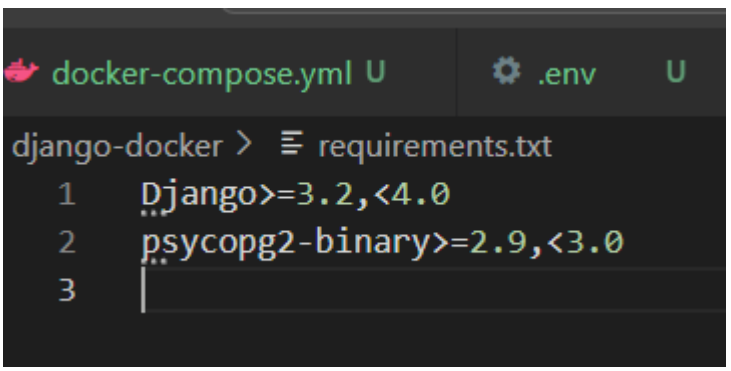
13.10.2024

1. Docker Compose

- **Configuration**

Components of the Project:
Dockerfile - Python dependency file
requirements.txt - file of dependencies to be used in Dockerfile
docker-compose.yml - file describing the build services of a multi-container application, in our case: Web (Django) and db (Postgre)





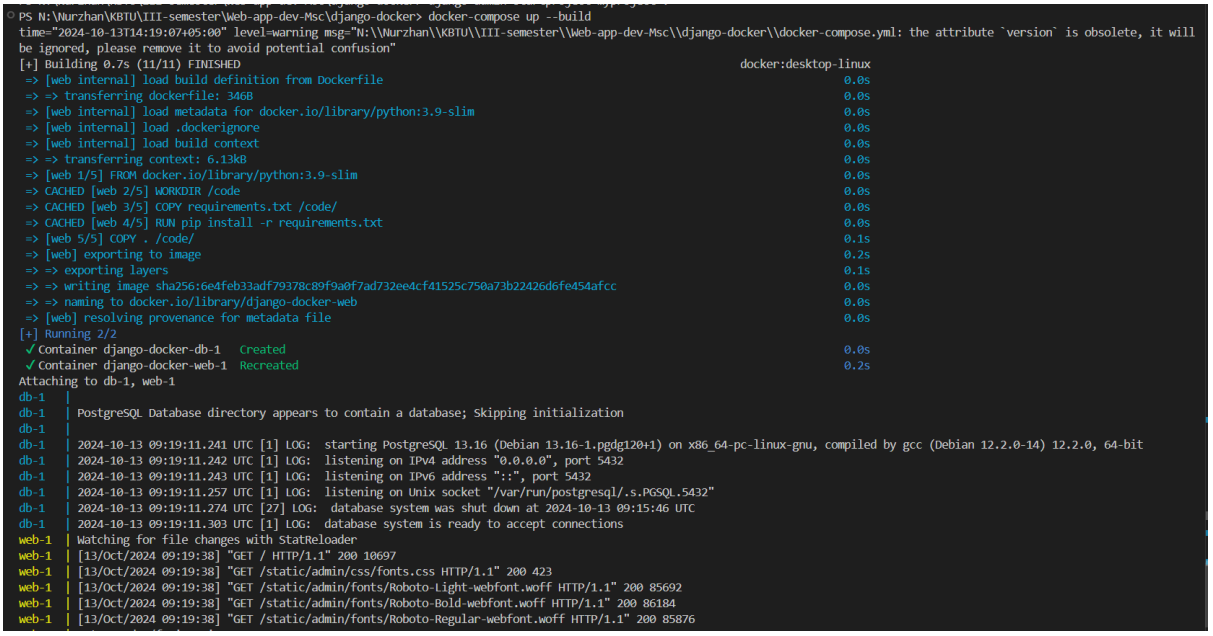

```
version: '3'
```

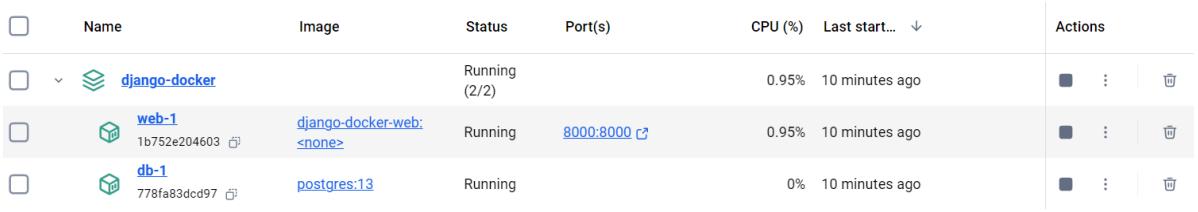
```yaml
services:
  db:
    image: postgres:13
    environment:
      POSTGRES_DB: ${DB_NAME}
      POSTGRES_USER: ${DB_USER}
      POSTGRES_PASSWORD: ${DB_PASSWORD}
    volumes:
      - postgres_data:/var/lib/postgresql/data
    networks:
      - app-network

  web:
    build: .
    command: python manage.py runserver 0.0.0.0:8000
    volumes:
      - .:/code
      - static_volume:/code/static
      - media_volume:/code/media
    ports:
      - "8000:8000"
    depends_on:
      - db
    environment:
      DB_NAME: ${DB_NAME}
      DB_USER: ${DB_USER}
      DB_PASSWORD: ${DB_PASSWORD}
      DB_HOST: db
      DB_PORT: 5432
    networks:
      - app-network

volumes:
  postgres_data:
  static_volume:
  media_volume:

networks:
  app-network:
```

- **Docker-compose.yml, brief explanation:**

The file describes two services db and web that create the application. These names then can be used in place of IP in container configs. The services are a web server and a database. The interconnection of these services, their volumes, ports, variables are described. Volumes are needed to save the results of work after containers are stopped. Ports describe the port forwarding from the local machine to the container. In variables we specify login, password, database from .env file.

- **Build and Run, The output in the terminal:**





2. Docker Networking and Volumes

- **Set Up Docker Networking**

```
db:
  image: postgres:13
  environment:
    POSTGRES_DB: ${DB_NAME}
    POSTGRES_USER: ${DB_USER}
```

```
    POSTGRES_PASSWORD: ${DB_PASSWORD}
  volumes:
    - postgres_data:/var/lib/postgresql/data
  networks:
    - app-network
web:
  build: .
  command: python manage.py runserver 0.0.0.0:8000
  volumes:
    - .:/code
    - static_volume:/code/static
    - media_volume:/code/media
  ports:
    - "8000:8000"
  depends_on:
    - db
  environment:
    DB_NAME: ${DB_NAME}
    DB_USER: ${DB_USER}
    DB_PASSWORD: ${DB_PASSWORD}
    DB_HOST: db
    DB_PORT: 5432
  networks:
    - app-network

networks:
 app-network:
```

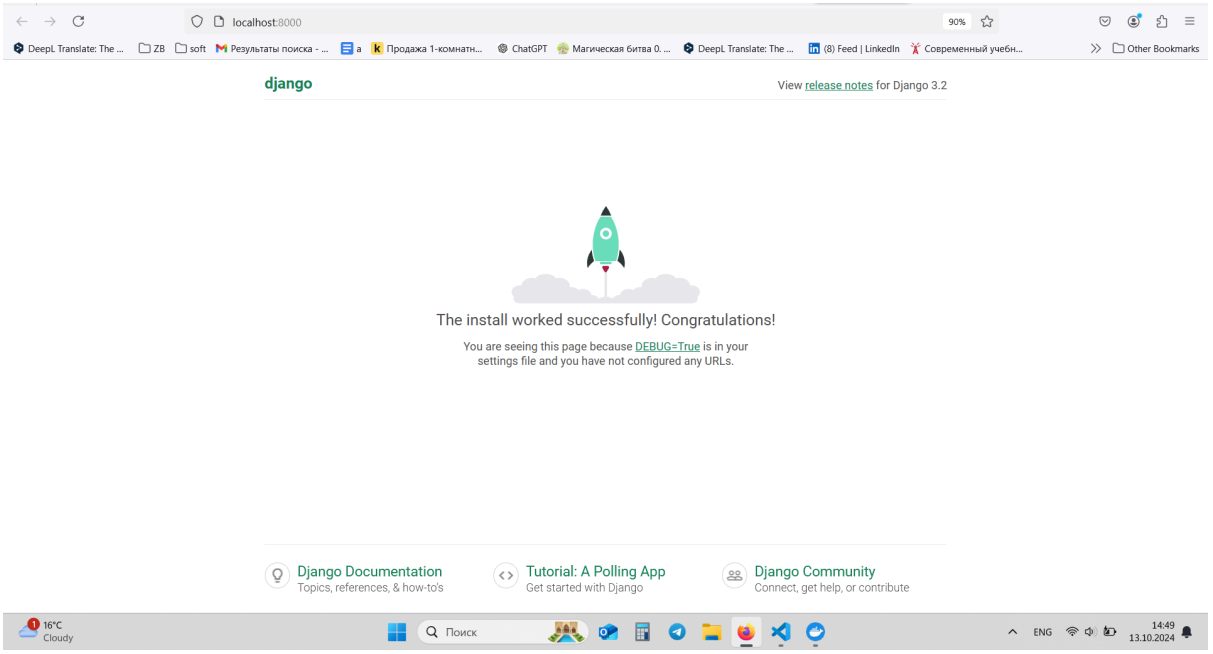
By specifying **networks: app-network**, we created a custom Docker network. Both the web (Django) and db (Postgre) services are connected to this network.

Benefits

Containers on the same network can communicate with each other by their names.

Isolation from other Docker containers outside the network. Easy scaling in the future, as all services communicate over a custom network.

Django connects to the PostgreSQL database. We can confirm this by visiting http://localhost:8000 once the server is running.



- **Implement Docker Volumes**

Volumes are crucial for persisting data, there are risks that data would be lost if a container is stopped or removed. That's why we can specify volumes too:

PostgreSQL Data Volume: Persist the PostgreSQL database data using a volume so that the database retains its state even when the container is recreated.

Static and Media Files Volumes: Set up volumes for static and media files in Django, ensuring that these files persist across container restarts.

```
volumes:
  postgres_data:  # Volume for persisting PostgreSQL data
  static_volume:  # Volume for persisting static files
  media_volume:   # Volume for persisting media files
```

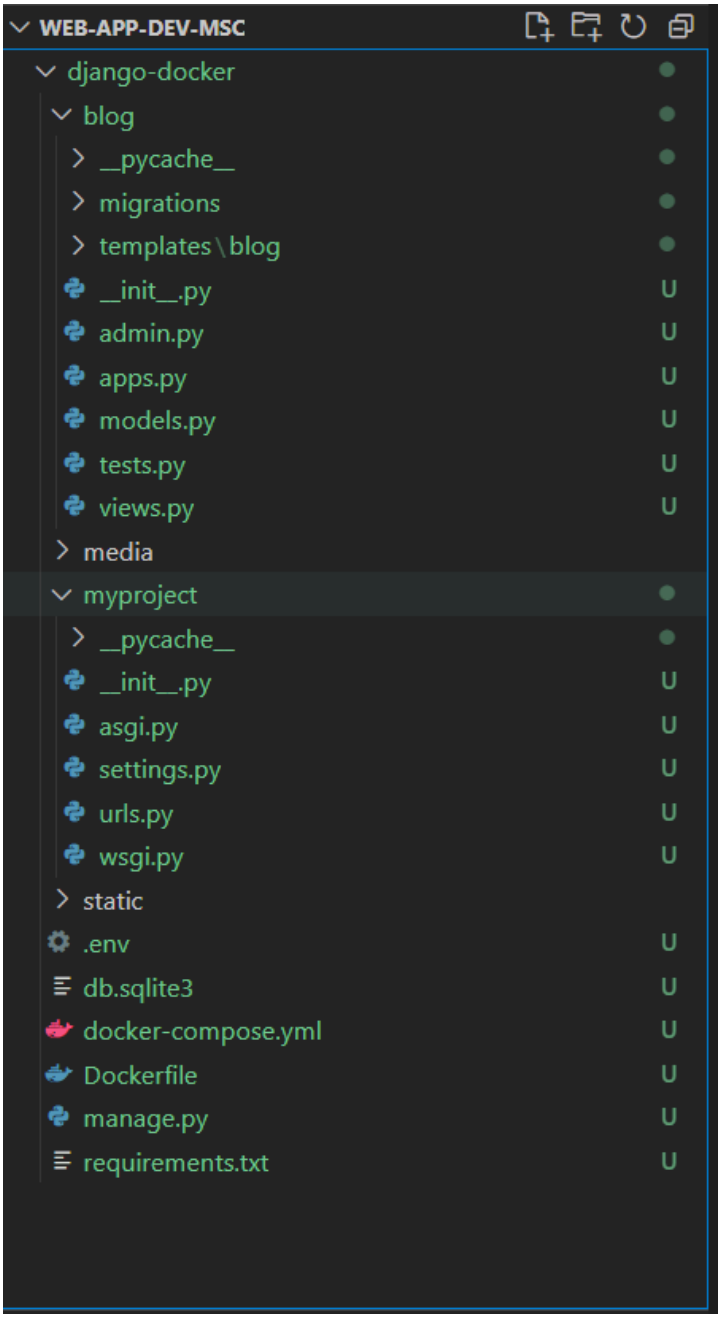
- **Findings**

Networking: The custom network (app-network) ensures that both the web (Django) and db (PostgreSQL) services can easily communicate using the service names (e.g., db as the hostname for PostgreSQL).This setup isolates the services from other Docker containers that may be running, reducing potential conflicts.

Volumes:Using volumes for Postgre, static, and media files ensures that data persists across container restarts. This is essential for database-driven applications and for handling user uploads or static assets. Even if the db container is removed or restarted, the database data is preserved due to the mounted volume. And any user-uploaded files or static assets are not lost between restarts, ensuring a stable environment.

3. Django Application Setup

- **Project Structure**



The Post model defined in models.py represents a blog post with a title, content, and timestamp.

```python
from django.db import models

class Post(models.Model):
    title = models.CharField(max_length=100)
    content = models.TextField()

    def __str__(self):
        return self.title
```

Then migrations created with makemigrations and migrate commands. It will set up the database schema for the Post model.

- **Database Configuration**

The PostgreSQL database is configured in settings.py to connect to the database container. The HOST value is set to the service name (db), which allows Django to connect to the PostgreSQL database running in Docker.

- **Findings**

The ability to easily configure and connect to a PostgreSQL database generates productivity. I found out that Docker containers isolate the application and its dependencies, reducing compatibility issues. Using Docker Compose simplifies multi-container applications, making it easier to manage services like the web server and database. However, setting up the database connection correctly took some time due to potential misconfigurations in environment variables or service names. Navigating the command line interface within the Docker container was initially confusing but became easier with practice.

Conclusion

In this assignment, I learned how to set up a Django application using Docker, create a PostgreSQL database, and run migrations. Docker provided a consistent environment, making it easier to manage dependencies and isolate services. The use of Docker simplified the development process and ensured that the application was portable and scalable. Overall, Docker significantly simplified the integration of Django and PostgreSQL for web development.