Assignment 2, Web app dev

Title of the Assignment: Exploring Django with Docker
Student's Name: Meruyert Taskynbayeva
Date of Submission: 13.10.2024

Table of Contents

Introduction

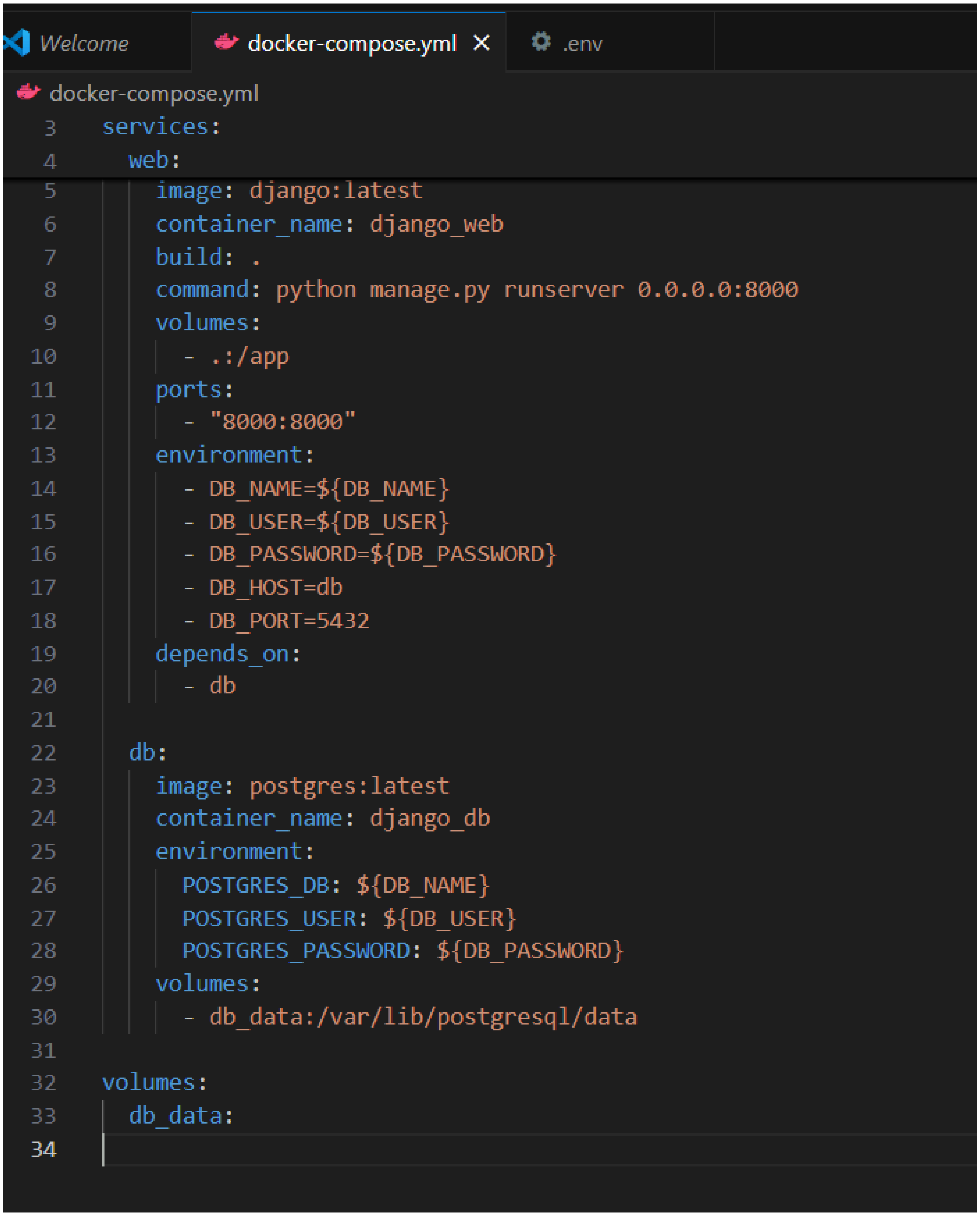
This report provides a comprehensive overview of developing a Django application using Docker. The primary objective is to explore how Docker can streamline the application development process, enhance portability, and simplify deployment.

Docker Compose

Configuration

The `docker-compose.yml` file defines the services required for the application, including the Django app, a PostgreSQL database, and any other necessary components. The configuration specifies the image versions, environment variables, volumes for data persistence, and networking settings, ensuring that all services can communicate effectively.

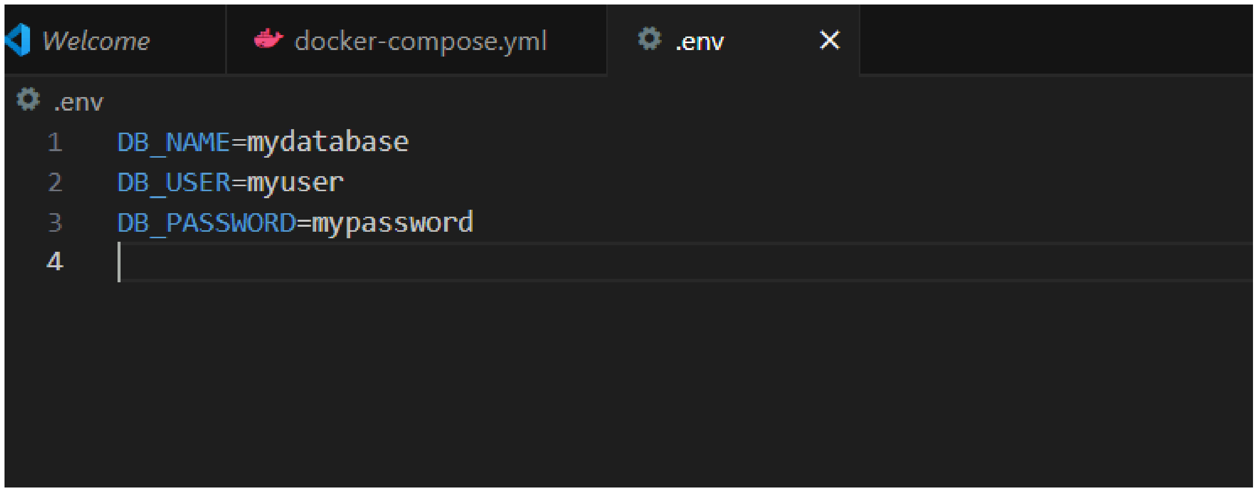
- **Create a Docker Compose File**
    - Create a `docker-compose.yml` file for your Django application.



```yaml
services:
  web:
    image: django:latest
    container_name: django_web
    build: .
    command: python manage.py runserver 0.0.0.0:8000
    volumes:
      - .:/app
    ports:
      - "8000:8000"
    environment:
      - DB_NAME=${DB_NAME}
      - DB_USER=${DB_USER}
      - DB_PASSWORD=${DB_PASSWORD}
      - DB_HOST=db
      - DB_PORT=5432
    depends_on:
      - db

  db:
    image: postgres:latest
    container_name: django_db
    environment:
      POSTGRES_DB: ${DB_NAME}
      POSTGRES_USER: ${DB_USER}
      POSTGRES_PASSWORD: ${DB_PASSWORD}
    volumes:
      - db_data:/var/lib/postgresql/data

volumes:
  db_data:

```

- Include services for:
  - Django web server
  - PostgreSQL database (or another database of your choice)

- **Define Environment Variables**
- Use environment variables for database configuration (e.g., DB_NAME, DB_USER, DB_PASSWORD).



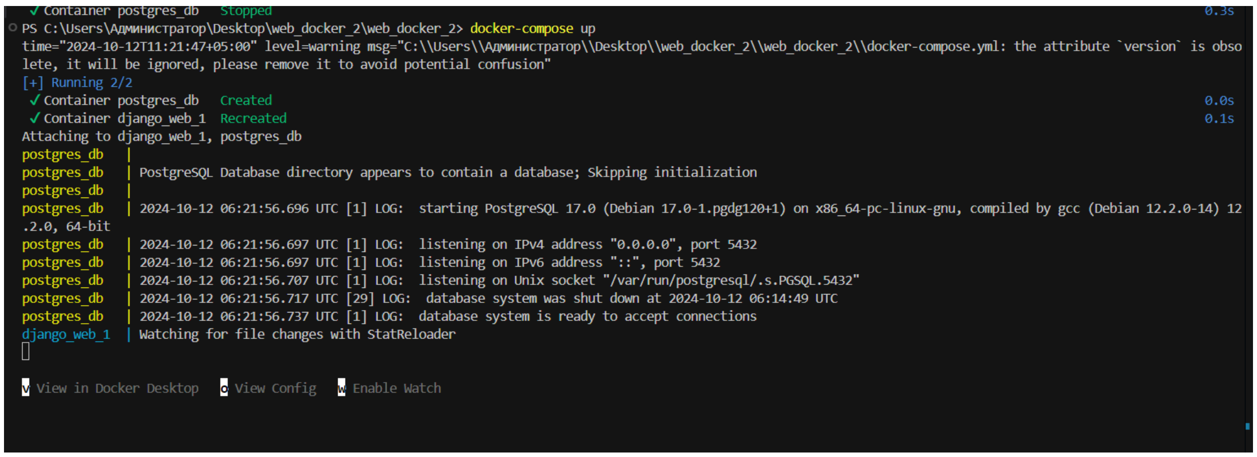
```
DB_NAME=mydatabase
DB_USER=myuser
DB_PASSWORD=mypassword
```

- **Build and Run the Containers**
    - Use `docker-compose up` to build and run the application.
    - Ensure that the services are running correctly.

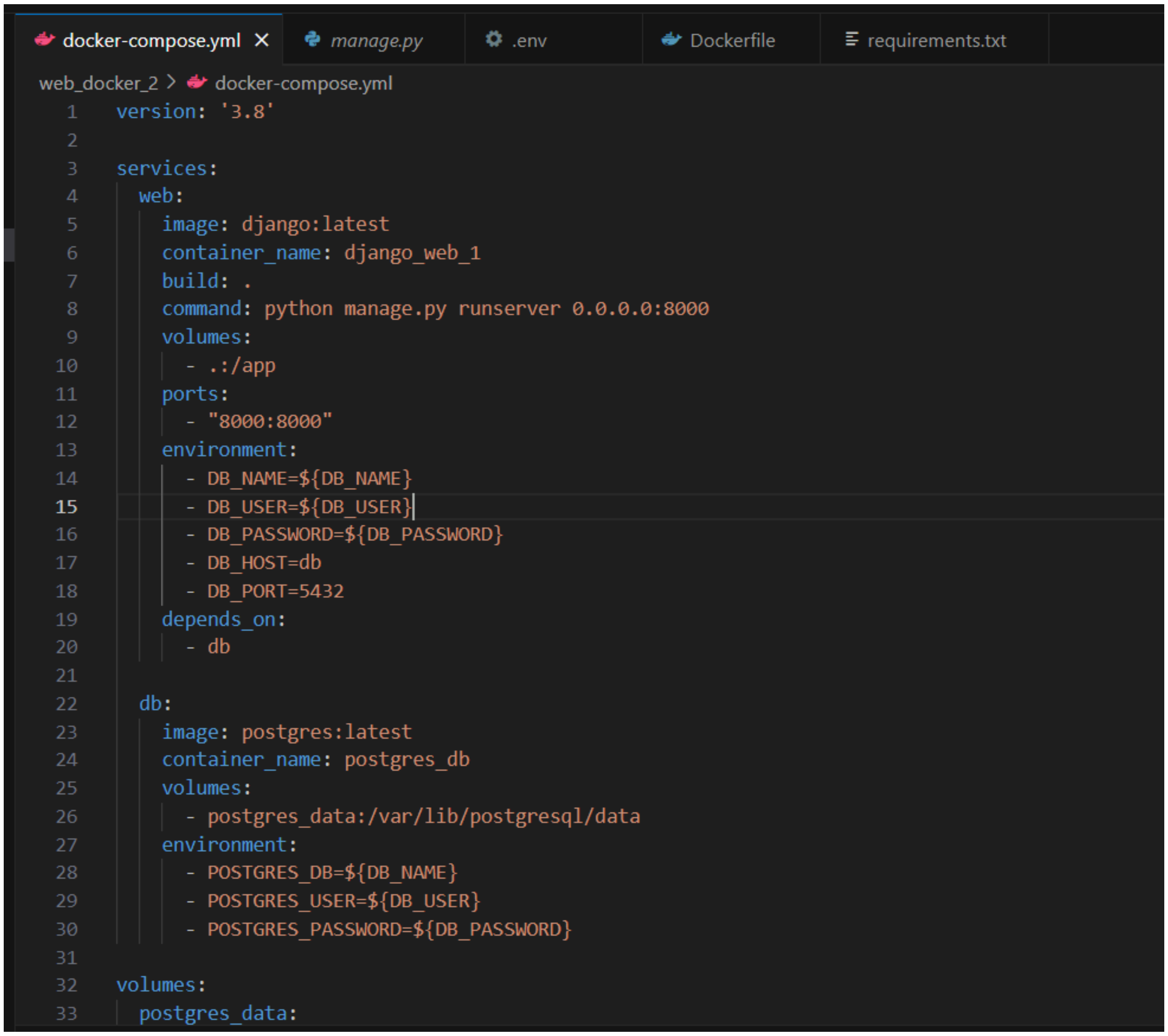
Build and Run

To build and run the containers, the command `docker-compose up --build` is used. This command constructs the images based on the configurations specified in the `docker-compose.yml` file and initiates the services. Challenges encountered included resolving dependency conflicts between packages and ensuring proper environment variable management.

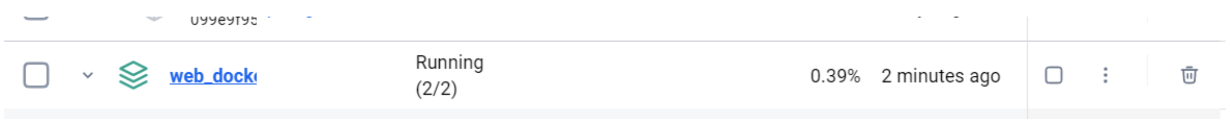
- **Document the Process**
    - Take screenshots of the Docker Compose file and the terminal output during the build and run process.



```
 ✓ Container postgres_db   Stopped                                                                                0.3s
PS C:\Users\Администратор\Desktop\web_docker_2\web_docker_2> docker-compose up
time="2024-10-12T11:21:47+05:00" level=warning msg="C:\\Users\\Администратор\\Desktop\\web_docker_2\\web_docker_2\\docker-compose.yml: the attribute `version` is obso
lete, it will be ignored, please remove it to avoid potential confusion"
[+] Running 2/2
 ✓ Container postgres_db   Created                                                                                0.0s
 ✓ Container django_web_1  Recreated                                                                              0.1s
Attaching to django_web_1, postgres_db
postgres_db   |
postgres_db   | PostgreSQL Database directory appears to contain a database; Skipping initialization
postgres_db   |
postgres_db   | 2024-10-12 06:21:56.696 UTC [1] LOG:  starting PostgreSQL 17.0 (Debian 17.0-1.pgdg120+1) on x86_64-pc-linux-gnu, compiled by gcc (Debian 12.2.0-14) 12
.2.0, 64-bit
postgres_db   | 2024-10-12 06:21:56.697 UTC [1] LOG:  listening on IPv4 address "0.0.0.0", port 5432
postgres_db   | 2024-10-12 06:21:56.697 UTC [1] LOG:  listening on IPv6 address "::", port 5432
postgres_db   | 2024-10-12 06:21:56.707 UTC [1] LOG:  listening on Unix socket "/var/run/postgresql/.s.PGSQL.5432"
postgres_db   | 2024-10-12 06:21:56.717 UTC [29] LOG:  database system was shut down at 2024-10-12 06:14:49 UTC
postgres_db   | 2024-10-12 06:21:56.737 UTC [1] LOG:  database system is ready to accept connections
django_web_1  | Watching for file changes with StatReloader

v View in Docker Desktop   o View Config   w Enable Watch
```



```yaml
version: '3.8'

services:
  web:
    image: django:latest
    container_name: django_web_1
    build: .
    command: python manage.py runserver 0.0.0.0:8000
    volumes:
      - .:/app
    ports:
      - "8000:8000"
    environment:
      - DB_NAME=${DB_NAME}
      - DB_USER=${DB_USER}
      - DB_PASSWORD=${DB_PASSWORD}
      - DB_HOST=db
      - DB_PORT=5432
    depends_on:
      - db

  db:
    image: postgres:latest
    container_name: postgres_db
    volumes:
      - postgres_data:/var/lib/postgresql/data
    environment:
      - POSTGRES_DB=${DB_NAME}
      - POSTGRES_USER=${DB_USER}
      - POSTGRES_PASSWORD=${DB_PASSWORD}

volumes:
  postgres_data:
```


web_dock

Running (2/2)

0.39% 2 minutes ago

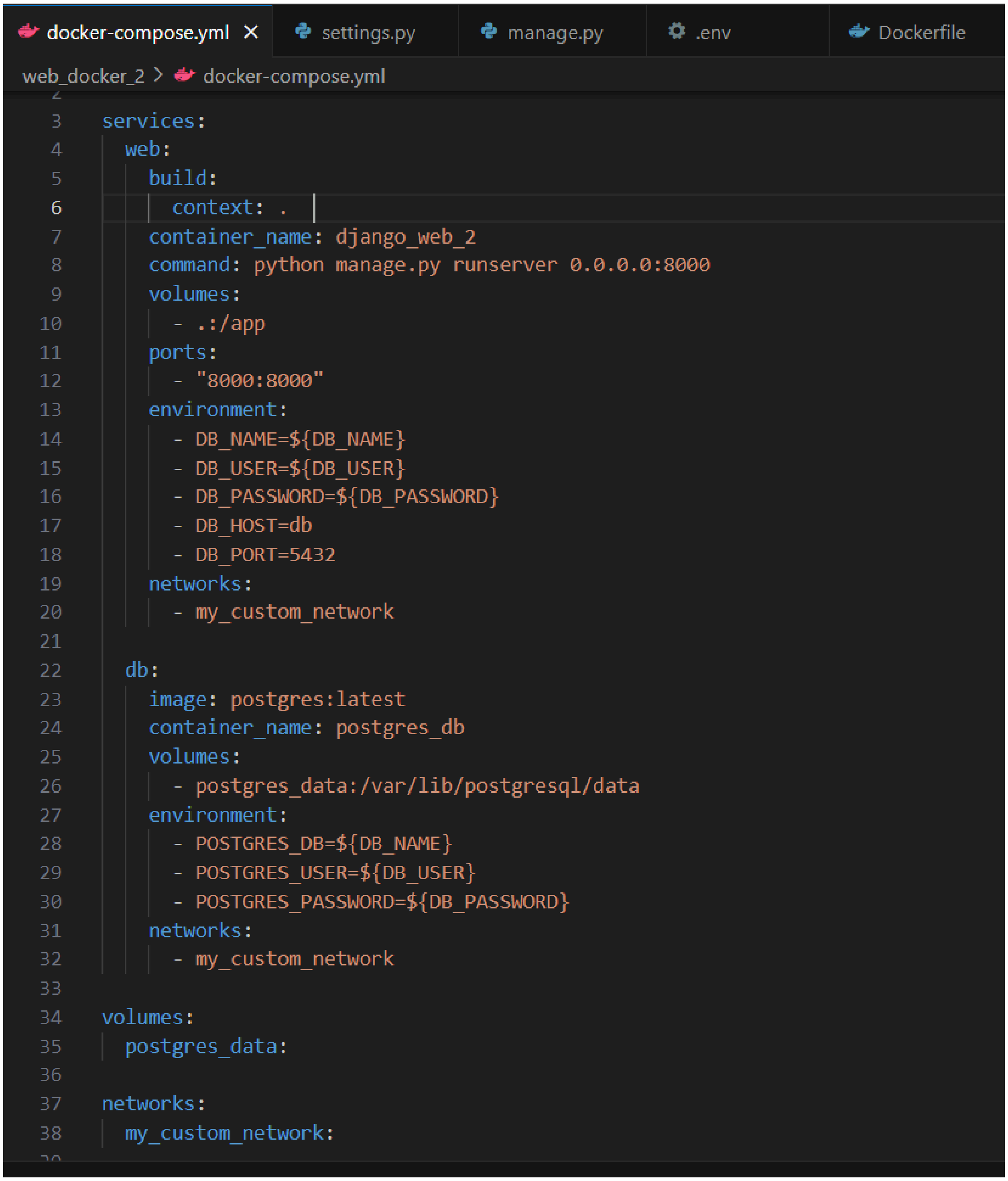
- **Write a brief explanation of the configurations used.**
  The `docker-compose.yml` file sets up a Django application with a PostgreSQL database. It specifies the version and defines two services: one for Django and one for PostgreSQL. The Django service builds the app, maps port 8000 for access, and uses environment variables for database configuration, ensuring sensitive data is not hard-coded. The PostgreSQL service uses the official image, sets up the database with the same environment variables, and includes a volume to persist data. To run the application, we execute `docker-compose up --build`, which builds the images and starts both services, simplifying development by managing them within containers.

Docker Networking and Volumes

Networking

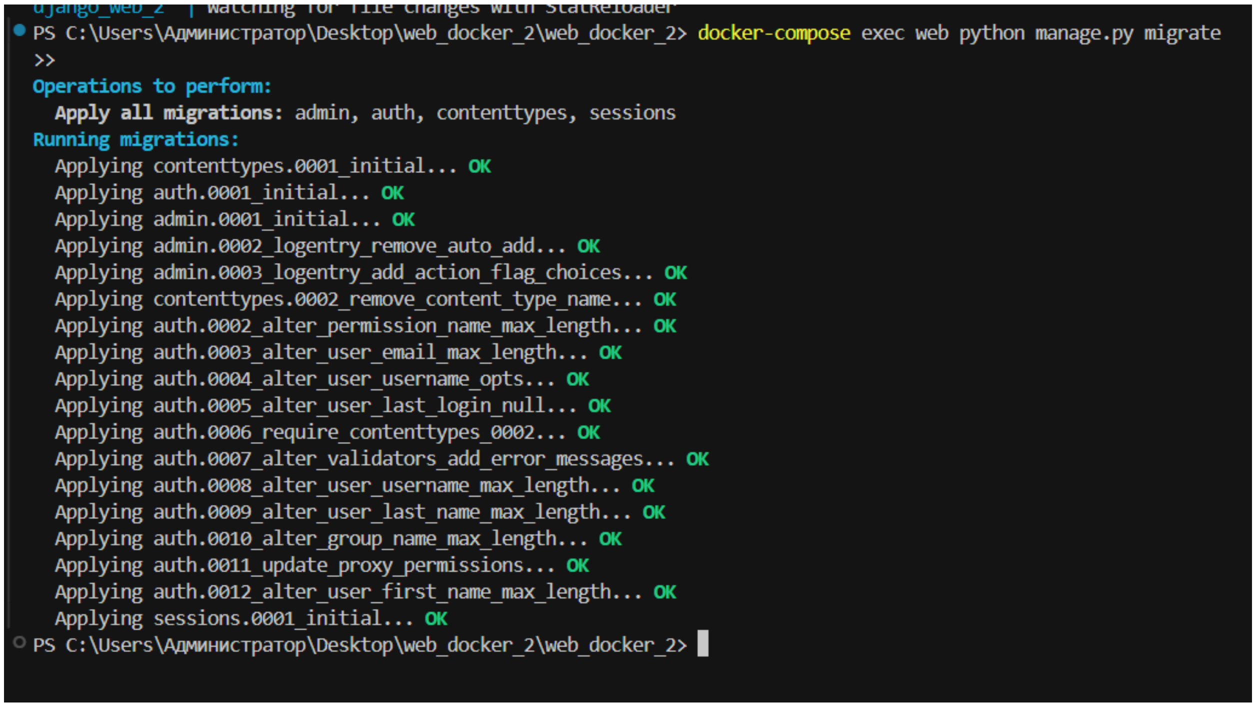
A custom network is created to facilitate communication between the containers. This setup isolates the application from the host machine, enhancing security and preventing port conflicts. The benefits include simplified service discovery and improved performance.

- **Set Up Docker Networking**
  - Define a custom network in your `docker-compose.yml` file to allow communication between services.



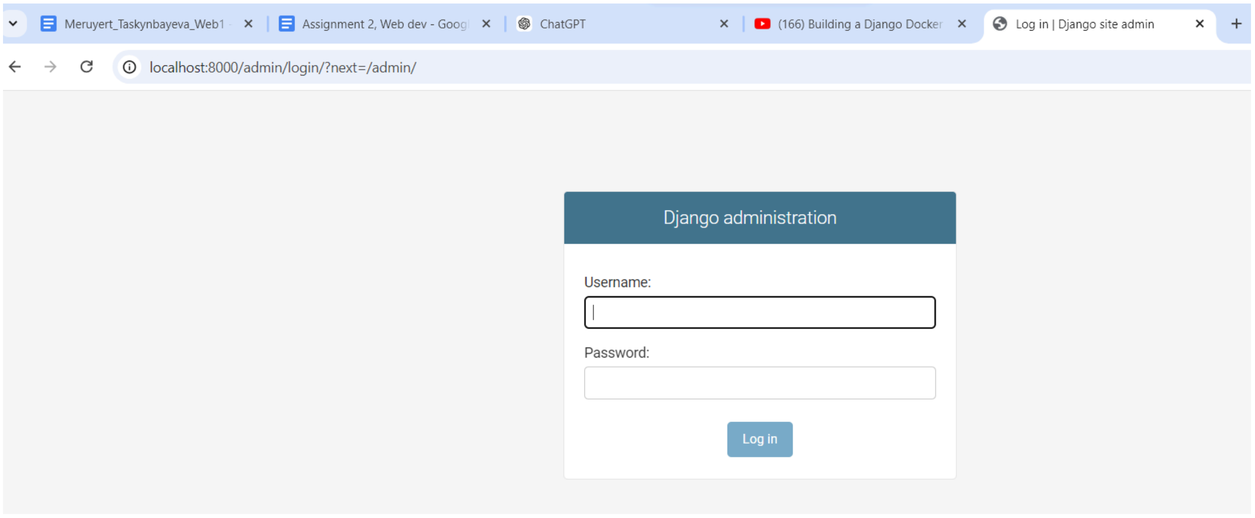
```yaml
services:
  web:
    build:
      context: .
    container_name: django_web_2
    command: python manage.py runserver 0.0.0.0:8000
    volumes:
      - .:/app
    ports:
      - "8000:8000"
    environment:
      - DB_NAME=${DB_NAME}
      - DB_USER=${DB_USER}
      - DB_PASSWORD=${DB_PASSWORD}
      - DB_HOST=db
      - DB_PORT=5432
    networks:
      - my_custom_network

  db:
    image: postgres:latest
    container_name: postgres_db
    volumes:
      - postgres_data:/var/lib/postgresql/data
    environment:
      - POSTGRES_DB=${DB_NAME}
      - POSTGRES_USER=${DB_USER}
      - POSTGRES_PASSWORD=${DB_PASSWORD}
    networks:
      - my_custom_network

volumes:
  postgres_data:

networks:
  my_custom_network:
```

- Verify that the Django app can connect to the database using the network.

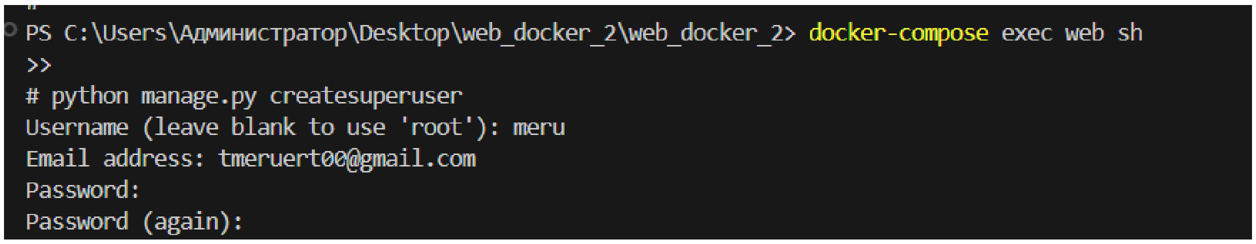


```
django_web_2  | Watching for file changes with StatReloader
PS C:\Users\Администратор\Desktop\web_docker_2\web_docker_2> docker-compose exec web python manage.py migrate
>>
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, sessions
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... OK
  Applying admin.0003_logentry_add_action_flag_choices... OK
  Applying contenttypes.0002_remove_content_type_name... OK
  Applying auth.0002_alter_permission_name_max_length... OK
  Applying auth.0003_alter_user_email_max_length... OK
  Applying auth.0004_alter_user_username_opts... OK
  Applying auth.0005_alter_user_last_login_null... OK
  Applying auth.0006_require_contenttypes_0002... OK
  Applying auth.0007_alter_validators_add_error_messages... OK
  Applying auth.0008_alter_user_username_max_length... OK
  Applying auth.0009_alter_user_last_name_max_length... OK
  Applying auth.0010_alter_group_name_max_length... OK
  Applying auth.0011_update_proxy_permissions... OK
  Applying auth.0012_alter_user_first_name_max_length... OK
  Applying sessions.0001_initial... OK
PS C:\Users\Администратор\Desktop\web_docker_2\web_docker_2>
```

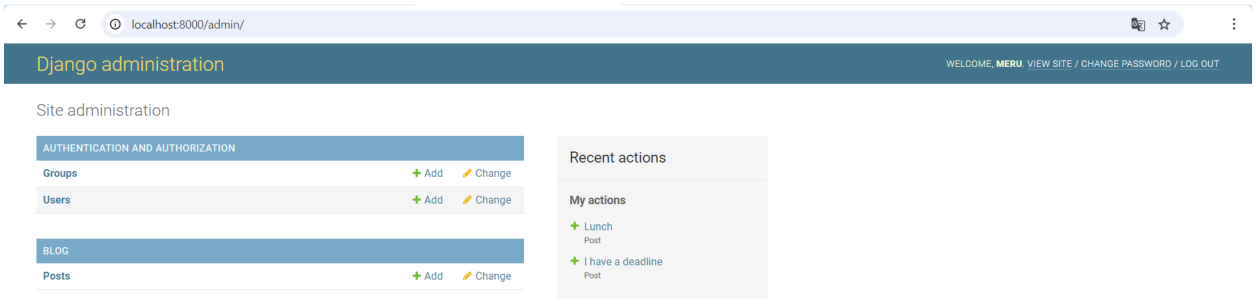
It confirms that Django can connect to PostgreSql
http://localhost:8000/admin/



Meruyert_Taskynbayeva_Web1
Assignment 2, Web dev - Goog
ChatGPT
(166) Building a Django Docker
Log in | Django site admin
localhost:8000/admin/login/?next=/admin/
Django administration
Username:
Password:
Log in



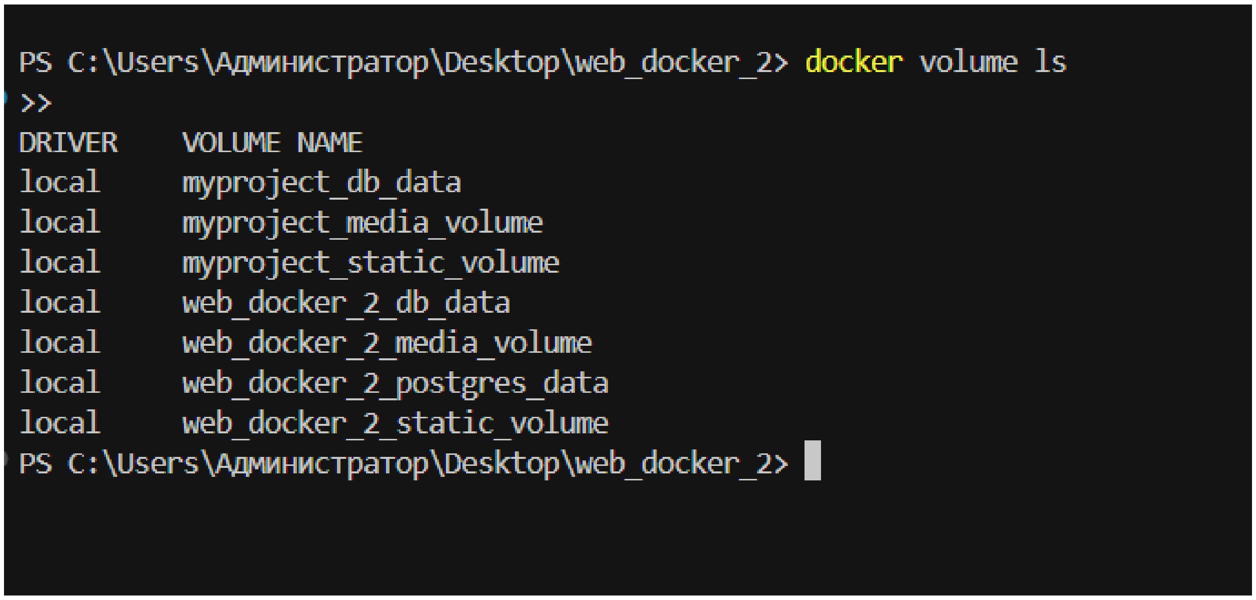
```
PS C:\Users\Администратор\Desktop\web_docker_2\web_docker_2> docker-compose exec web sh
>>
# python manage.py createsuperuser
Username (leave blank to use 'root'): meru
Email address: tmeruert00@gmail.com
Password:
Password (again):
```



Volumes

Volumes are utilized for data persistence, particularly for the PostgreSQL database. This approach ensures that data remains intact across container restarts and rebuilds. The volume mapping in the `docker-compose.yml` file allows the database to store data on the host filesystem.

- **Implement Docker Volumes**
  - Configure a volume in the `docker-compose.yml` file to persist PostgreSQL data.

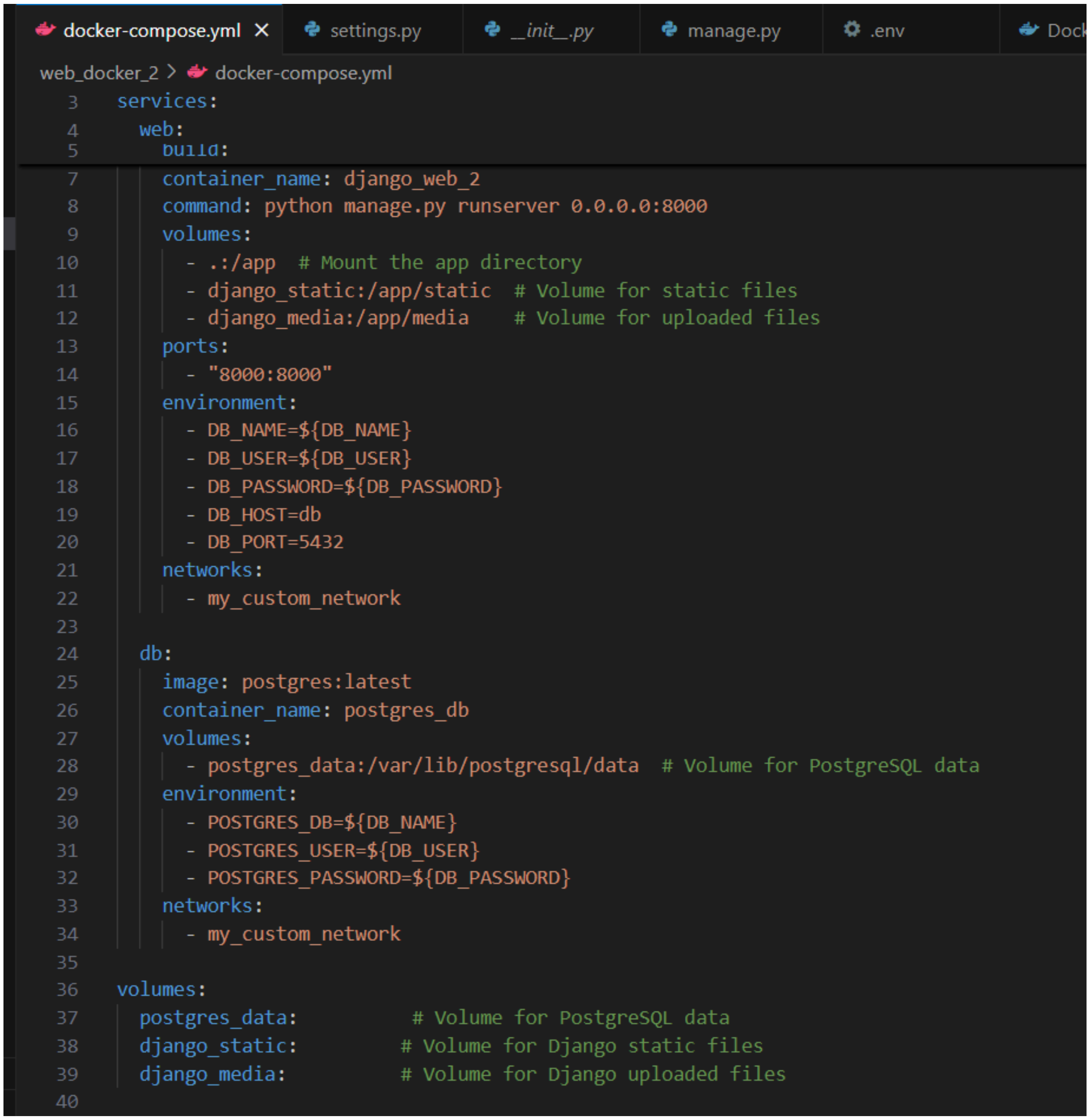


```
PS C:\Users\Администратор\Desktop\web_docker_2> docker volume ls
>>
DRIVER    VOLUME NAME
local     myproject_db_data
local     myproject_media_volume
local     myproject_static_volume
local     web_docker_2_db_data
local     web_docker_2_media_volume
local     web_docker_2_postgres_data
local     web_docker_2_static_volume
PS C:\Users\Администратор\Desktop\web_docker_2>
```

  - Add a volume for Django to persist uploaded files and static files.
- **Document the Process**
  - Take screenshots of the updated `docker-compose.yml` file, and **explain how networking and volumes enhance your application.**

Networking and volumes enhance our application by improving communication and data management. Defining a custom network in the `docker-compose.yml` allows the Django app and PostgreSQL database to communicate securely and efficiently, using service names instead of hard-coded IP addresses. This simplifies configuration and enhances scalability.

Using Docker volumes ensures data persistence. The volume for PostgreSQL keeps the database data intact even when containers are restarted, preventing data loss. Additionally, a volume for the Django app stores uploaded and static files, allowing users to manage files without losing them during container restarts. Overall, these features lead to a more reliable and maintainable application.



```yaml
services:
  web:
    build:
    container_name: django_web_2
    command: python manage.py runserver 0.0.0.0:8000
    volumes:
      - .:/app  # Mount the app directory
      - django_static:/app/static  # Volume for static files
      - django_media:/app/media    # Volume for uploaded files
    ports:
      - "8000:8000"
    environment:
      - DB_NAME=${DB_NAME}
      - DB_USER=${DB_USER}
      - DB_PASSWORD=${DB_PASSWORD}
      - DB_HOST=db
      - DB_PORT=5432
    networks:
      - my_custom_network

  db:
    image: postgres:latest
    container_name: postgres_db
    volumes:
      - postgres_data:/var/lib/postgresql/data  # Volume for PostgreSQL data
    environment:
      - POSTGRES_DB=${DB_NAME}
      - POSTGRES_USER=${DB_USER}
      - POSTGRES_PASSWORD=${DB_PASSWORD}
    networks:
      - my_custom_network

volumes:
  postgres_data:          # Volume for PostgreSQL data
  django_static:         # Volume for Django static files
  django_media:          # Volume for Django uploaded files

```

Findings

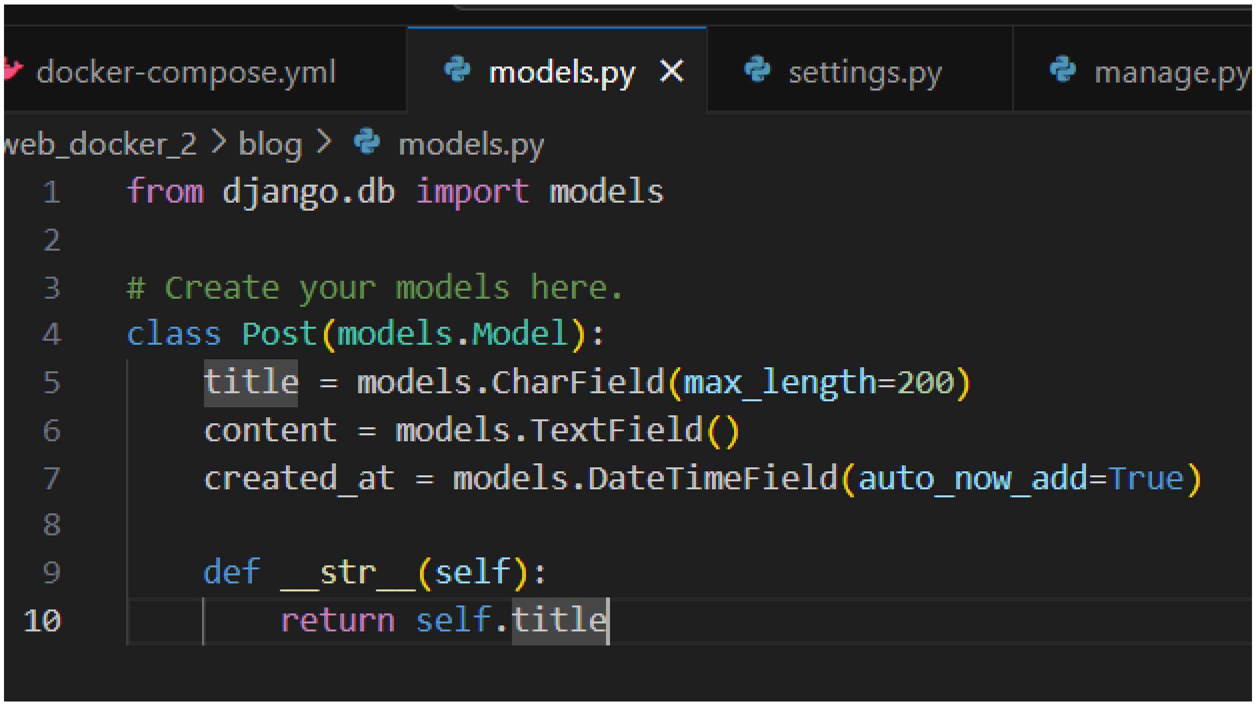
Utilizing Docker networking and volumes significantly enhances application reliability and scalability. Custom networks simplify interactions between services, while volumes ensure data durability, making the development process more efficient.

Django Application Setup

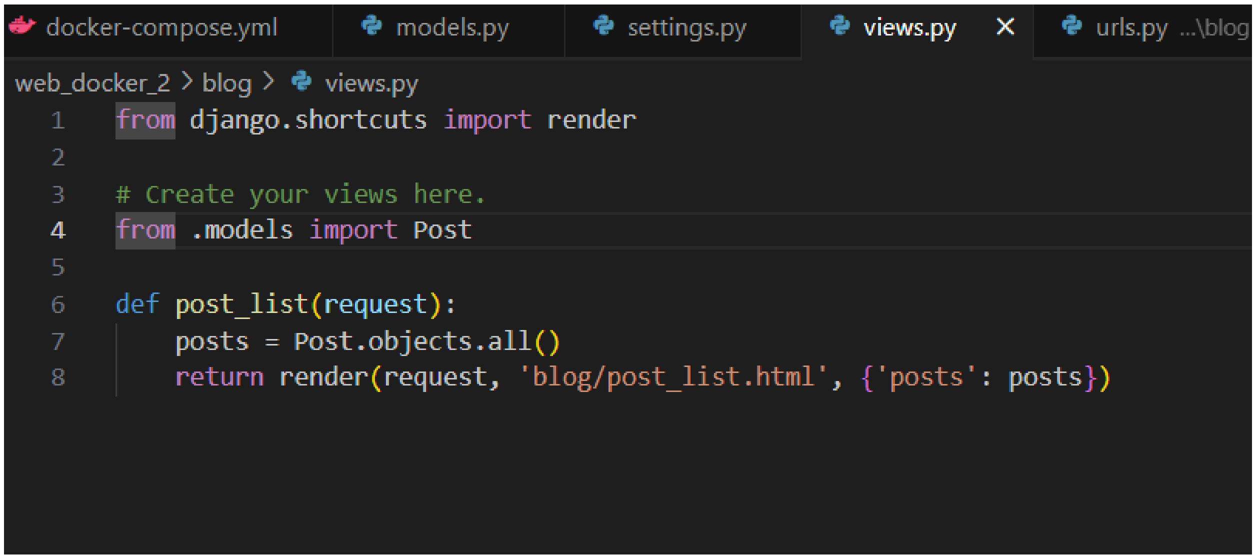
Project Structure

The Django project follows a standard structure, including directories for settings, templates, static files, and applications. Each app is designed to handle specific functionalities, promoting modular development.

- **Create a Django Project**
  - Inside the Django service container, create a new Django project using the command `django-admin startproject myproject`.
- Create a simple app (e.g., `blog`) with at least one model and a corresponding view.



```python
from django.db import models

# Create your models here.
class Post(models.Model):
    title = models.CharField(max_length=200)
    content = models.TextField()
    created_at = models.DateTimeField(auto_now_add=True)

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



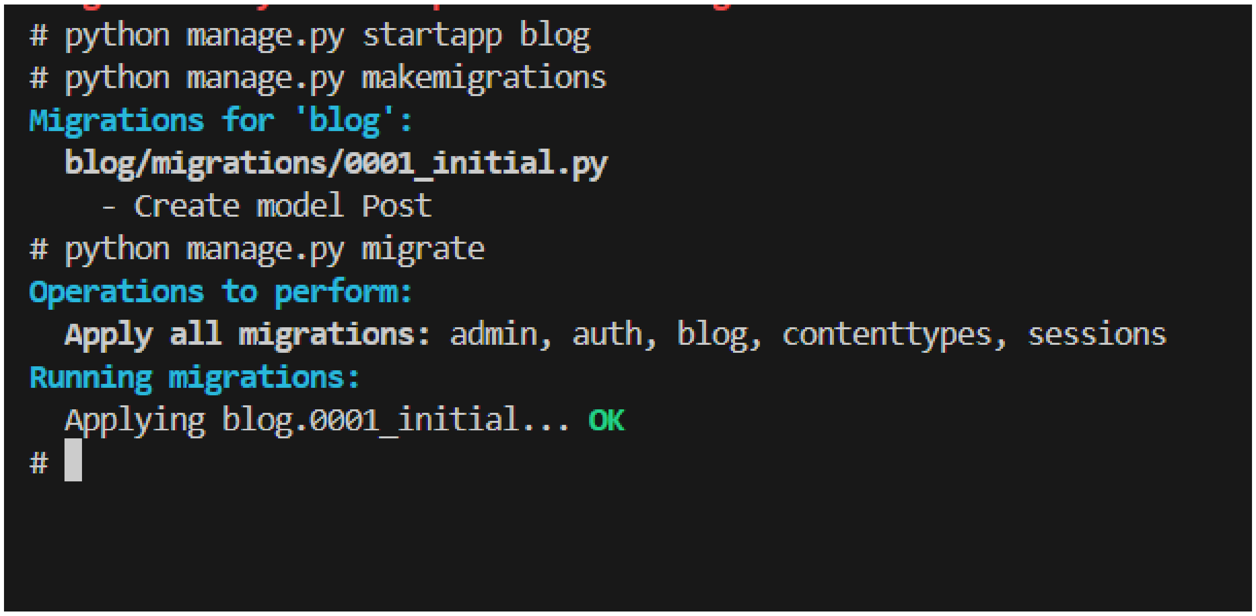
```python
from django.shortcuts import render

# Create your views here.
from .models import Post

def post_list(request):
    posts = Post.objects.all()
    return render(request, 'blog/post_list.html', {'posts': posts})
```

- **Configure the Database**
  - Update the Django settings to use the PostgreSQL database configured in your Docker Compose setup.
  - Run migrations to set up the database schema.

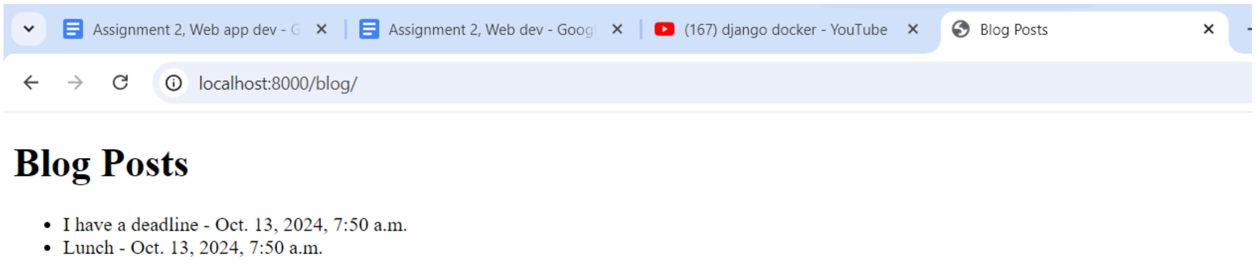
The PostgreSQL database is configured in the `settings.py` file of the Django project. The connection parameters, such as the database name, user, password, and host, are managed through environment variables for security and flexibility.

- **Document the Process**

- Take screenshots of the project structure, model definition, and any migrations.



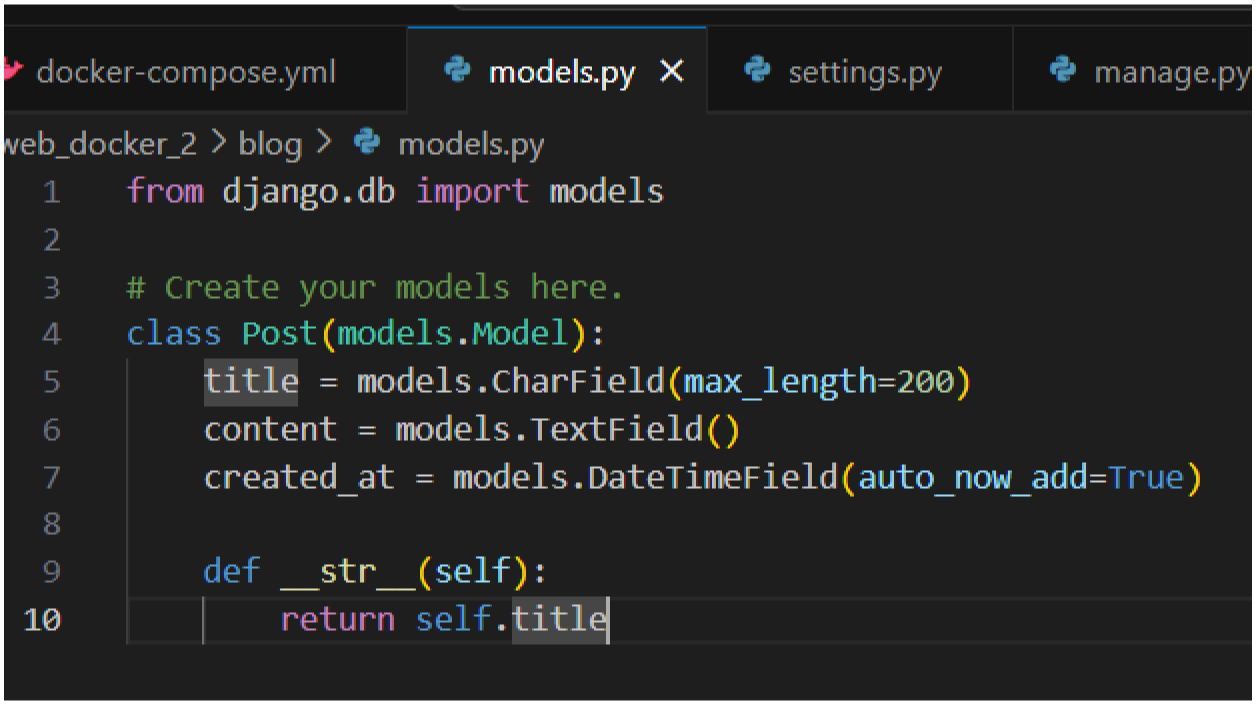
```
# python manage.py startapp blog
# python manage.py makemigrations
Migrations for 'blog':
  blog/migrations/0001_initial.py
    - Create model Post
# python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, blog, contenttypes, sessions
Running migrations:
  Applying blog.0001_initial... OK
# 
```



Assignment 2, Web app dev - G
Assignment 2, Web dev - Goog
(167) django docker - YouTube
Blog Posts
localhost:8000/blog/

Blog Posts

- I have a deadline - Oct. 13, 2024, 7:50 a.m.
- Lunch - Oct. 13, 2024, 7:50 a.m.

- 

docker-compose.yml | models.py | settings.py | manage.py

web_docker_2 > blog > models.py

```python
from django.db import models

# Create your models here.
class Post(models.Model):
    title = models.CharField(max_length=200)
    content = models.TextField()
    created_at = models.DateTimeField(auto_now_add=True)

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

Explain how the Django application is structured and how it interacts with Docker

Project Structure

1. **Project Directory**: Created with the command `django-admin startproject myproject`, this is the main folder that holds all the settings and configurations for the Django app.
2. **App Directory**: Inside the project, we can create apps (like a blog). Each app has its own models, views, templates, and static files, making it easy to manage and reuse.
3. **Models**: These define the data structure, representing tables in the PostgreSQL database. Each model corresponds to a table.
4. **Views**: Views contain the logic for handling requests and responses, often interacting with models to get or save data.
5. **Migrations**: These are files generated when we create or change models, allowing us to update the database schema.

Interaction with Docker

The Django app interacts with Docker through the `docker-compose.yml` file, which defines how everything is set up:

- **Service Definition**: The Django web server is listed as a service in the Docker Compose file, specifying how to build it and which ports to use.

- **Database Configuration**: We update Django's settings to connect to the PostgreSQL database using environment variables for the database name, user, and password. This keeps sensitive information out of the code.
- **Networking**: A custom network allows the Django app to communicate with the PostgreSQL database securely, without exposing extra ports.
- **Volumes**: We use Docker volumes to save PostgreSQL data and any uploaded or static files from Django, ensuring they persist even when containers restart.

Findings

1. **Simplified Setup**: Docker made setting up a consistent development environment much easier, ensuring that all services (Django, database, etc.) worked seamlessly across different machines.
2. **Isolation**: Using Docker containers isolated services like Django and the database, allowing easier management and debugging without conflicts between components.
3. **Database Integration**: Docker simplified database setup, but I had to ensure Django waited for the database service to be ready. This was solved using the `depends_on` directive and a wait-for-it script.
4. **Persistent Data**: Docker volumes helped retain data between container restarts, which was useful during development.
5. **Dependency Management**: Ensuring all Python dependencies were properly installed within the Docker container required some troubleshooting but was resolved through the `requirements.txt` file.
6. **Portability**: Docker allowed for easy portability between development and production environments with minimal changes.

Conclusion

In summary, this assignment highlighted the significant advantages of using Docker alongside Django for application development. Key learnings include the importance of containerization for consistent environments, the benefits of networking for service communication, and the role of volumes in maintaining data integrity. Overall, Docker proves to be an invaluable tool for modern application development.

References

- Tutorial Building a Django Docker Container
- https://stackoverflow.com/questions/55301756/docker-daemon-is-not-running for troubleshooting the issue with Docker
- Docker Documentation: https://docs.docker.com
- Django Documentation: https://docs.djangoproject.com