MINISTRY OF EDUCATION AND SCIENCE OF THE REPUBLIC OF KAZAKHSTAN

KAZAKH-BRITISH TECHNICAL UNIVERSITY

SCHOOL OF INFORMATION TECHNOLOGY AND ENGINEERING

REPORT

Web Application Development

Assignment 2

Presented to Serek A.G.

Done by Kabyl Dauren

Student ID: 23MD0452

Almaty 2024

Content

1. Introduction

2. Docker Compose

    2.1. Configuration

    2.2. Build and Run

3. Docker Networking and Volumes

    3.1. Networking

    3.2. Volumes

    3.3. Findings

4. Django Application Setup

    4.1. Project Structure

    4.2. Database Configuration

    4.3. Findings

5. Conclusion

6. References

Introduction

In this work we would show how to work with docker compose (Sections 2 and 3) and how to create simple application on the django framework (Section 4).

Docker compose is a tool that allows developers to easily run containers with many different services inside [1]. This allows developers to more easily and better monitor running containers.

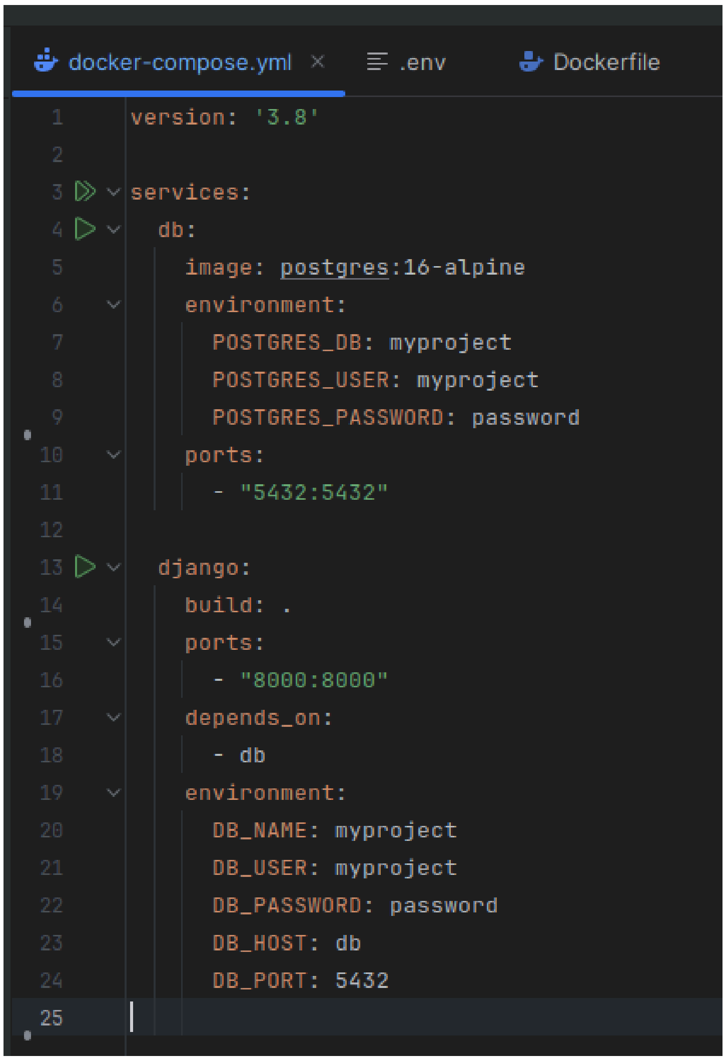
Django is a python framework that makes web development much easier [2]. It has much potential and is easy to understand, so with django you can create different types of applications.

Docker Compose

Configuration

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

As shown in Image 1, in the docker-compose.yml file I wrote setup configuration for Postgresql database and my django project. In the db service I configured the database name, user and password for connecting. In the django service I configured environment variables for connecting to postgres database. Django service depends on this database so if it is not running this service will not run either.



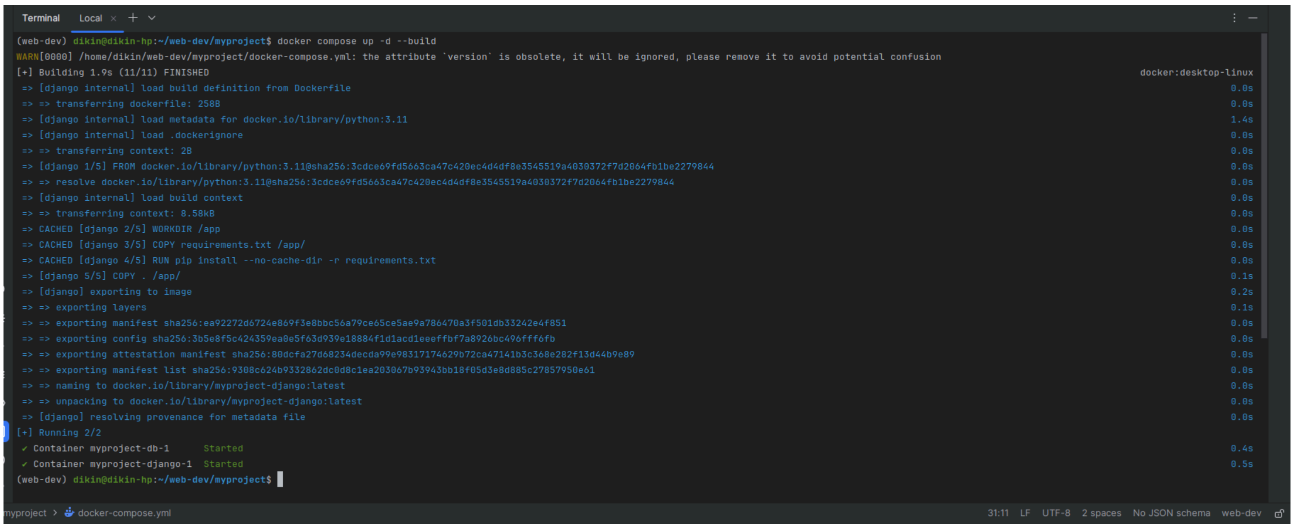
```yaml
version: '3.8'

services:
  db:
    image: postgres:16-alpine
    environment:
      POSTGRES_DB: myproject
      POSTGRES_USER: myproject
      POSTGRES_PASSWORD: password
    ports:
      - "5432:5432"

  django:
    build: .
    ports:
      - "8000:8000"
    depends_on:
      - db
    environment:
      DB_NAME: myproject
      DB_USER: myproject
      DB_PASSWORD: password
      DB_HOST: db
      DB_PORT: 5432

```

Image 1. docker-compose.yml file configuration

Build and Run

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

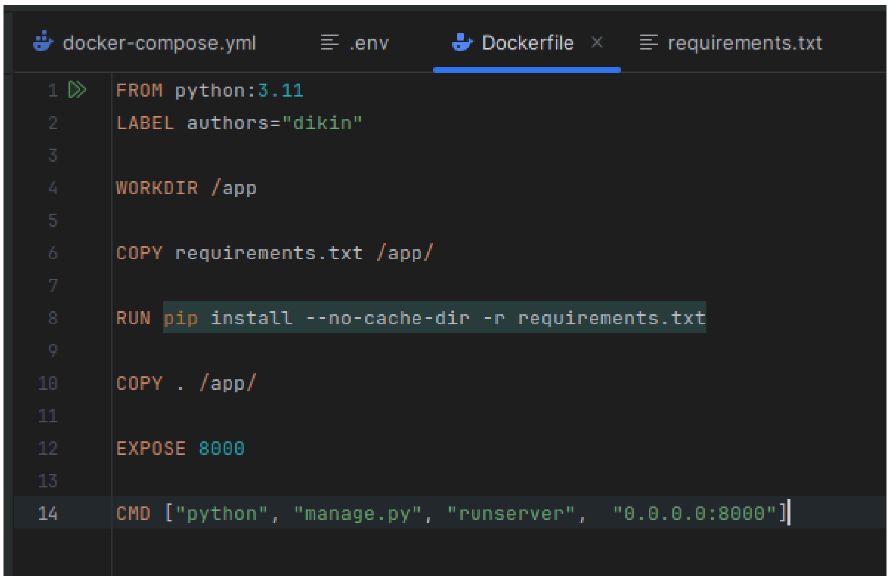
With command **docker compose up -d –build** to build and run services in detach mode I started this services. At first it pulled the required image for postgres and built a local Dockerfile image with a python image. Then it installed required python packages and runned django service on port 8000. That they are correctly running you can see in Image 4.



```
(web-dev) dikin@dikin-hp:~/web-dev/myproject$ docker compose up -d --build
WARN[0000] /home/dikin/web-dev/myproject/docker-compose.yml: the attribute `version` is obsolete, it will be ignored, please remove it to avoid potential confusion
[+] Building 1.9s (11/11) FINISHED                                                                                     docker:desktop-linux
 => [django internal] load build definition from Dockerfile                                                                         0.0s
 => => transferring dockerfile: 258B                                                                                                0.0s
 => [django internal] load metadata for docker.io/library/python:3.11                                                               1.4s
 => [django internal] load .dockerignore                                                                                            0.0s
 => => transferring context: 2B                                                                                                     0.0s
 => [django 1/5] FROM docker.io/library/python:3.11@sha256:3cdce69fd5663ca47c420ec4d4df8e3545519a4030372f7d2064fb1be2279844          0.0s
 => => resolve docker.io/library/python:3.11@sha256:3cdce69fd5663ca47c420ec4d4df8e3545519a4030372f7d2064fb1be2279844                0.0s
 => [django internal] load build context                                                                                            0.0s
 => => transferring context: 8.58kB                                                                                                 0.0s
 => CACHED [django 2/5] WORKDIR /app                                                                                                0.0s
 => CACHED [django 3/5] COPY requirements.txt /app/                                                                                 0.0s
 => CACHED [django 4/5] RUN pip install --no-cache-dir -r requirements.txt                                                          0.0s
 => [django 5/5] COPY . /app/                                                                                                       0.1s
 => [django] exporting to image                                                                                                     0.2s
 => => exporting layers                                                                                                             0.1s
 => => exporting manifest sha256:ea92272d6724e869f3e8bbc56a79ce65ce5ae9a786470a3f501db33242e4f851                                   0.0s
 => => exporting config sha256:3b5e8f5c424359ea0e5f63d939e18884f1d1acd1eeeffbf7a8926bc496fff6fb                                     0.0s
 => => exporting attestation manifest sha256:80dcfa27d68234decda99e98317174629b72ca47141b3c368e282f13d44b9e89                       0.0s
 => => exporting manifest list sha256:9308c624b9332862dc0d8c1ea203067b93943bb18f05d3e8d885c27857950e61                              0.0s
 => => naming to docker.io/library/myproject-django:latest                                                                          0.0s
 => => unpacking to docker.io/library/myproject-django:latest                                                                       0.0s
 => [django] resolving provenance for metadata file                                                                                 0.0s
[+] Running 2/2
 ✔ Container myproject-db-1      Started                                                                                            0.4s
 ✔ Container myproject-django-1  Started                                                                                            0.5s
(web-dev) dikin@dikin-hp:~/web-dev/myproject$
```

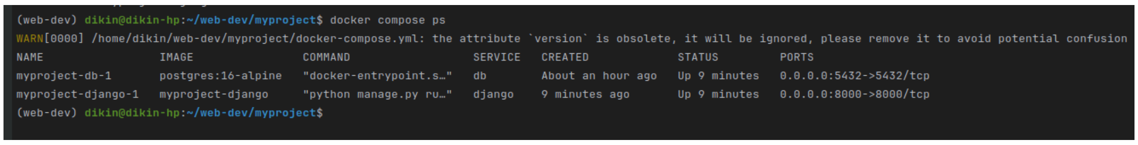
myproject > docker-compose.yml
31:11 LF UTF-8 2 spaces No JSON schema web-dev

Image 2. Building and Running docker-compose.yml



```
FROM python:3.11
LABEL authors="dikin"

WORKDIR /app

COPY requirements.txt /app/

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

COPY . /app/

EXPOSE 8000

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

Image 3. Dockerfile



```
(web-dev) dikin@dikin-hp:~/web-dev/myproject$ docker compose ps
WARN[0000] /home/dikin/web-dev/myproject/docker-compose.yml: the attribute `version` is obsolete, it will be ignored, please remove it to avoid potential confusion
NAME                 IMAGE                COMMAND                    SERVICE   CREATED              STATUS         PORTS
myproject-db-1       postgres:16-alpine   "docker-entrypoint.s…"     db        About an hour ago    Up 9 minutes   0.0.0.0:5432->5432/tcp
myproject-django-1   myproject-django     "python manage.py ru…"     django    9 minutes ago        Up 9 minutes   0.0.0.0:8000->8000/tcp
(web-dev) dikin@dikin-hp:~/web-dev/myproject$
```

Image 4. Running docker images

Docker Networking and Volumes

Networking

- **Set Up Docker Networking**
    - Define a custom network in your docker-compose.yml file to allow communication between services.
    - Verify that the Django app can connect to the database using the network.

As shown in Image 5, I added network configurations to db. Network **django_network** allows django service to identify the db service host by hostname **db** except IP address of host. So, I can connect to the database like **db:5432** as the address.

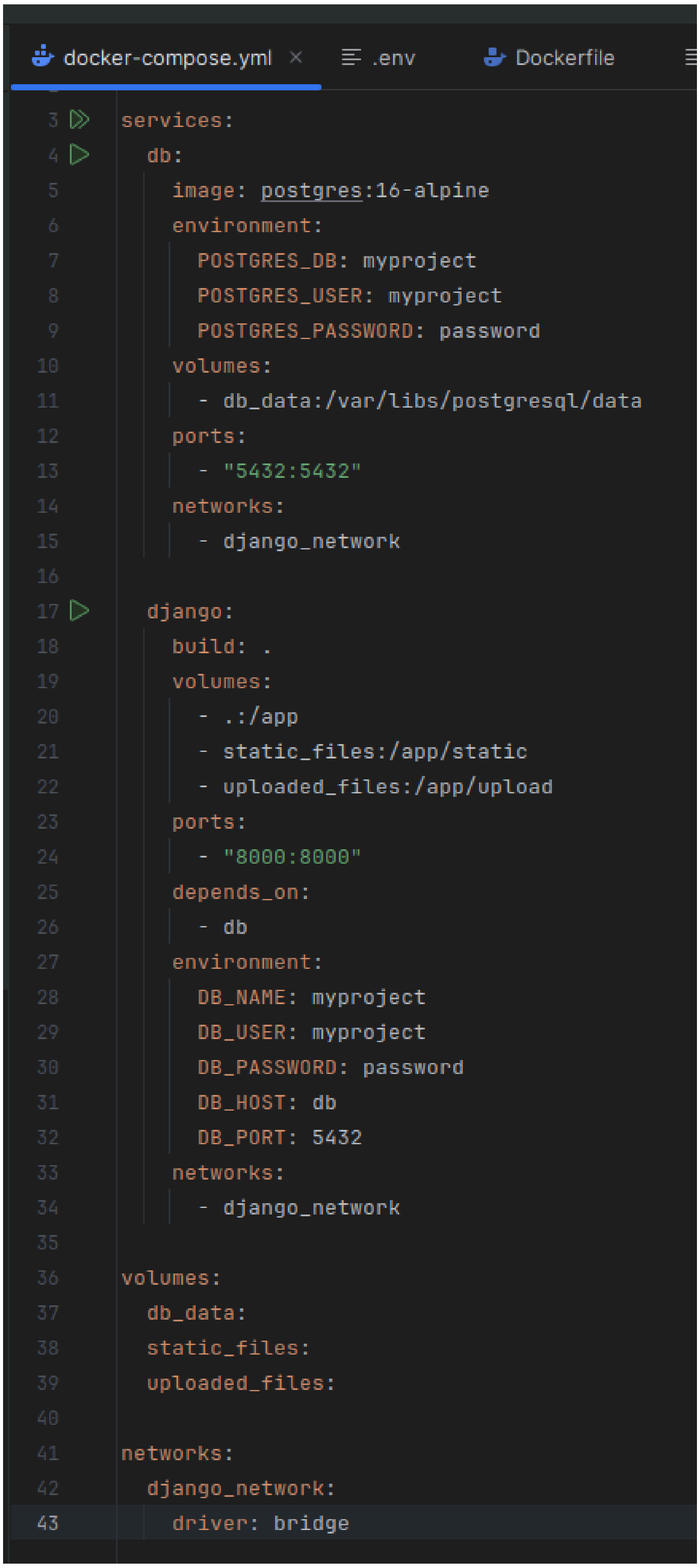
Volumes

- **Implement Docker Volumes**
    - Configure a volume in the `docker-compose.yml` file to persist PostgreSQL data.
    - Add a volume for Django to persist uploaded files and static files.

As shown in Image 5, I added volumes configuration for postgres data, static files and uploaded files to db. Data saved to these volumes will be saved even if the container restarts.

Findings

Networks in docker make it easier to interact between services and volumes that work as storages to all data. They make it easier to understand docker structure and working with real apps.



```yaml
services:
  db:
    image: postgres:16-alpine
    environment:
      POSTGRES_DB: myproject
      POSTGRES_USER: myproject
      POSTGRES_PASSWORD: password
    volumes:
      - db_data:/var/libs/postgresql/data
    ports:
      - "5432:5432"
    networks:
      - django_network

  django:
    build: .
    volumes:
      - .:/app
      - static_files:/app/static
      - uploaded_files:/app/upload
    ports:
      - "8000:8000"
    depends_on:
      - db
    environment:
      DB_NAME: myproject
      DB_USER: myproject
      DB_PASSWORD: password
      DB_HOST: db
      DB_PORT: 5432
    networks:
      - django_network

volumes:
  db_data:
  static_files:
  uploaded_files:

networks:
  django_network:
    driver: bridge
```

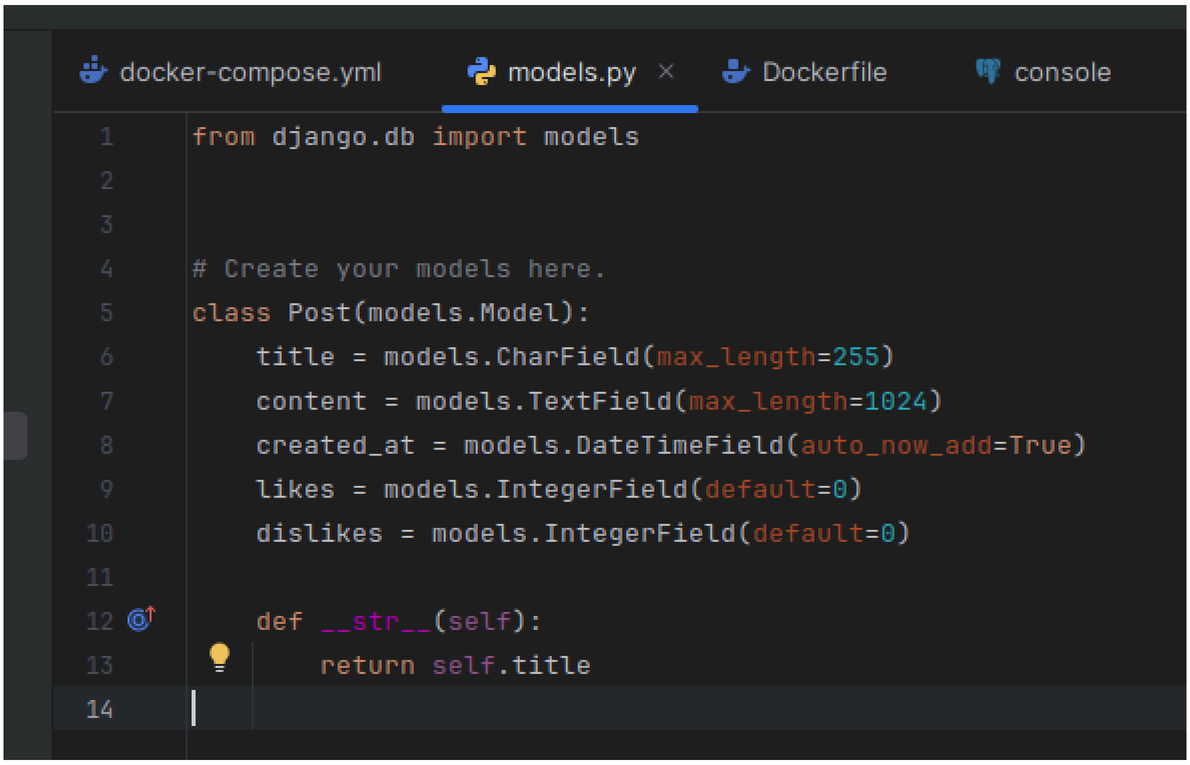
Image 5. Volumes and Networks

Django Application Setup

Project Structure

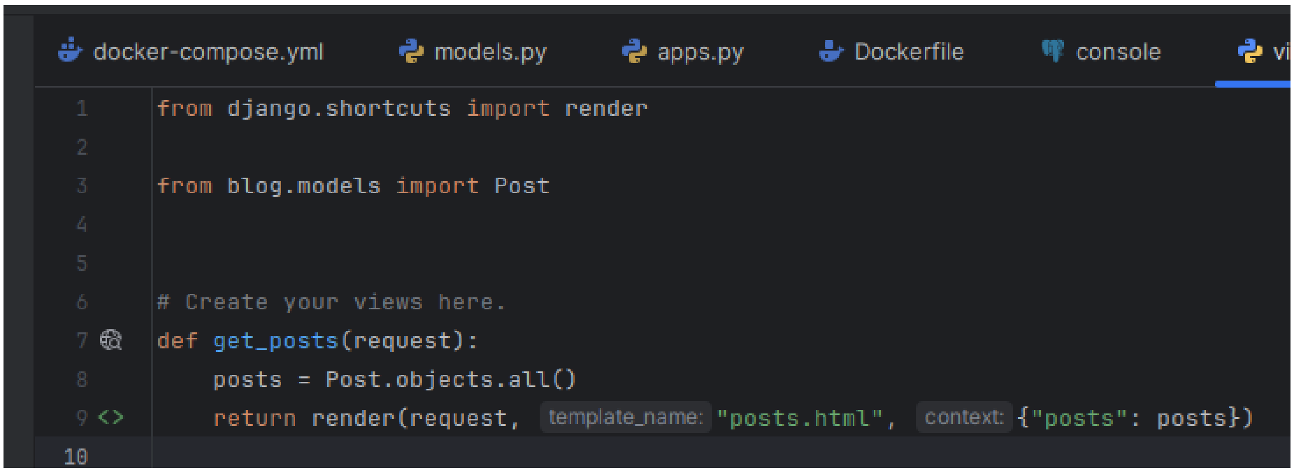
- **Create a Django Project**
  - Inside the Django service container, create a new Django project using the command django-admin startproject myproject.
  - Create a simple app (e.g., blog) with at least one model and a corresponding view.

As shown in Image 6 and Image 7 I created a blog app in my django project. This app includes a model Post that saves blog posts of blog in database and posts.html view that shows these blogs.



```python
from django.db import models


# Create your models here.
class Post(models.Model):
    title = models.CharField(max_length=255)
    content = models.TextField(max_length=1024)
    created_at = models.DateTimeField(auto_now_add=True)
    likes = models.IntegerField(default=0)
    dislikes = models.IntegerField(default=0)

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

Image 6. Posts model



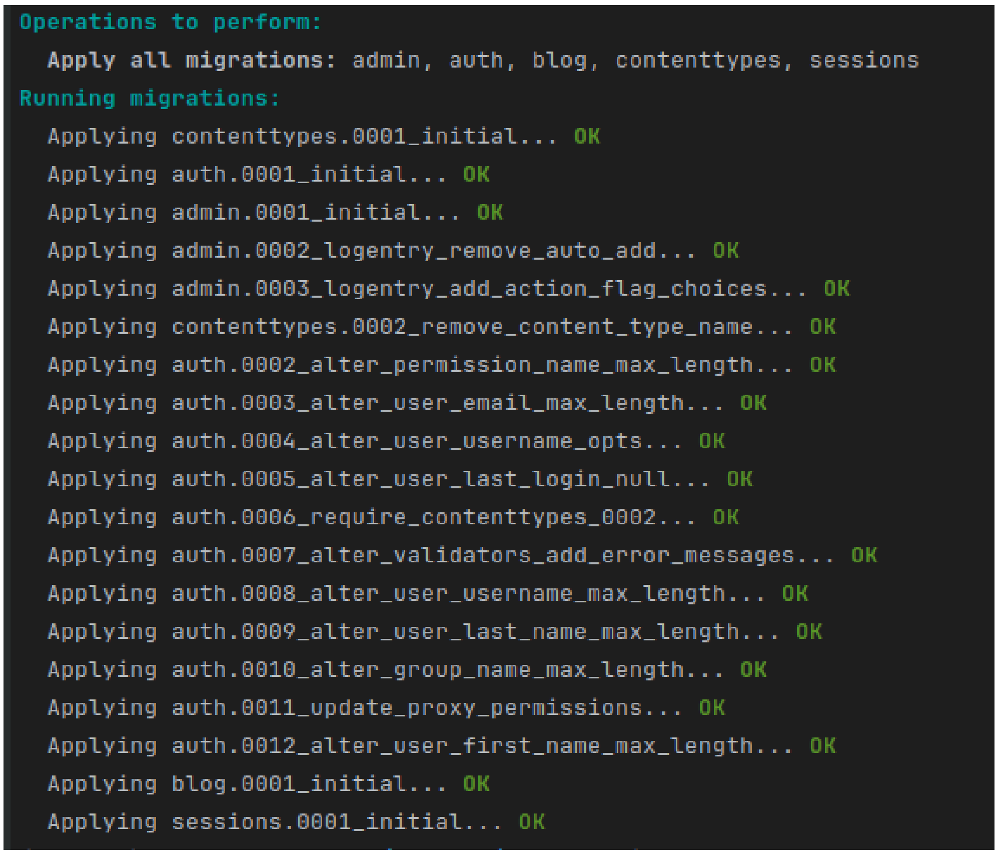
```python
from django.shortcuts import render

from blog.models import Post


# Create your views here.
def get_posts(request):
    posts = Post.objects.all()
    return render(request, template_name: "posts.html", context: {"posts": posts})
```

Image 7. Posts view

Database Configuration

- **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.

As shown in Image 8 I migrated my Posts to db with **python manage.py makemigrations** and **python manage.py migrate** commands.



```
Operations to perform:
  Apply all migrations: admin, auth, blog, 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 blog.0001_initial... OK
  Applying sessions.0001_initial... OK
```

Image 8. Database Migrations

Findings

In Django we can easily create an app with endpoints. It uses MVT (Model-View-Template) architecture to easily show models in views. Django ORM automatically creates migration classes according to our models, which makes working with database much easier.

Conclusion

In this work we discussed Docker Compose and Django. We saw how to work with docker containers using Docker Compose and how to create docker networks and volumes. Also, we created a simple app blog that shows blog posts with Django and created migrations using Django Models and Django ORM. It was the first step to developing a fully functional web application using Django, so our future work will be targeted to learn Django much better.

References

1. Docker documentation: https://docs.docker.com/manuals/

2. PostgreSql documentation: https://www.postgresql.org/docs/

3. Django documentation: https://docs.djangoproject.com/en/5.1/