KBTU KAZAKH-BRITISH TECHNICAL UNIVERSITY

Assignment #2

Web Application Development
Exploring Django with Docker

Prepared by:
Seitbekov S.
Checked by:
Serek A.

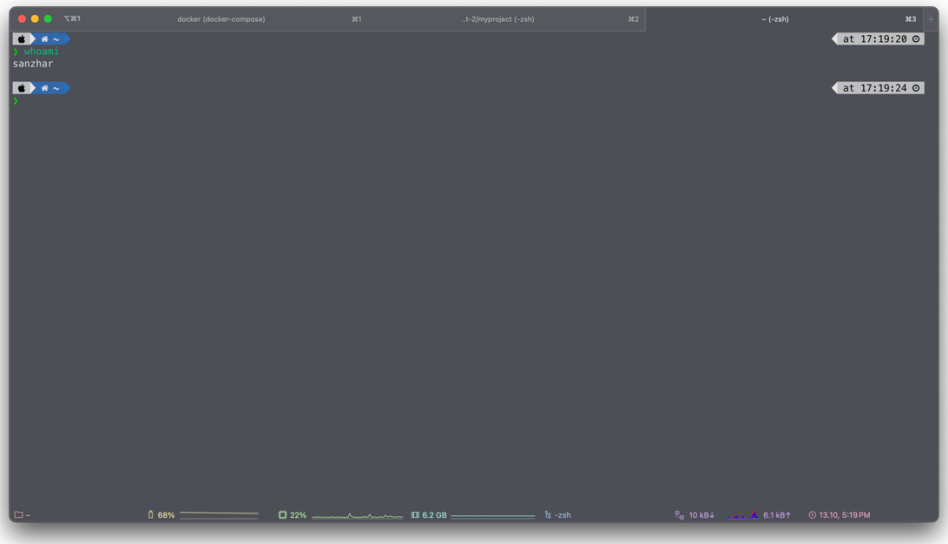
Almaty, 13.10.2024

Table of Contents

Introduction

The following assignment is to create a Django application using Docker. More specifically, it uses Docker Compose for service orchestration, Docker networking for inter-service communication, and Docker volumes for persisting data. Practically, this is about setting up a full-fledged development environment where the Django and PostgreSQL services run seamlessly inside Docker containers. During the laboratory work, I used the `whoami` command to verify my identity (Sanzhar). The source code for this project, including the Docker and Django configurations, is available on my GitHub repository - https://github.com/diable201/WebProgrammingMS

Docker Compose



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)

I created a docker-compose.yml file that defines services for both the Django web server and PostgreSQL database, using the following components:

- **web**: This service is responsible for running the `Django` web server.
- **db**: I used PostgreSQL (`postgres-17:alpine` image) as the database service and included environment variables like `DB_NAME`, `DB_USER`, and `DB_PASSWORD` to configure the database.

```yaml
services:
  web:
    build:
      context: .
      dockerfile: Dockerfile
    command:
gunicorn myproject.wsgi:application --bind 0.0.0.0:8000
    volumes:
      - .:/code
      - static_volume:/code/static
      - media_volume:/code/media
    ports:
      - "8000:8000"
    env_file:
      - .env
    depends_on:
      - db
    networks:
      - app-network

  db:
    image: postgres:17-alpine
    environment:
      - POSTGRES_DB=${DB_NAME}
      - POSTGRES_USER=${DB_USER}
      - POSTGRES_PASSWORD=${DB_PASSWORD}
    volumes:
      - postgres_data:/var/lib/postgresql/data/
    ports:
      - "5432:5432"
    networks:
      - app-network

volumes:
  postgres_data:
  static_volume:
  media_volume:

networks:
  app-network:
```

Define Environment Variables

- Use environment variables for database configuration (e.g., `DB_NAME, DB_USER, DB_PASSWORD`).

Environment Variables: The use of environment variables helps secure sensitive information like database credentials (`password, user, db_name`) making it easy to change them without hardcoding.

```
DB_NAME=assignment-2
DB_USER=admin
DB_PASSWORD=password
```

Build and Run the Containers

- Use `docker-compose up` to build and run the application.

I wrote Dockerfile for this project. This `Dockerfile` sets up a `python:3.11` image using the slim variant. It unbuffers Python output for real-time logging and sets the working directory to `/code`. The `requirements.txt` is copied to the container, and dependencies are installed with pip without caching. Finally, all project files are copied into the container.

```
FROM python:3.11-slim

ENV PYTHONUNBUFFERED=1

WORKDIR /code

COPY requirements.txt /code/

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

COPY . /code/
```

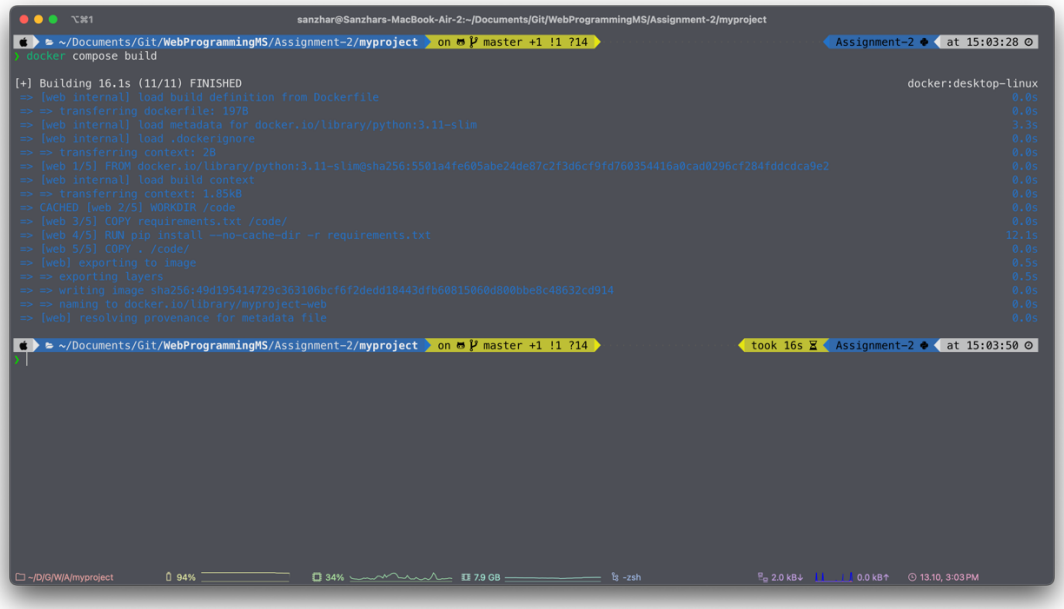
Configuration:

The `docker-compose.yml` file defines **2** services:

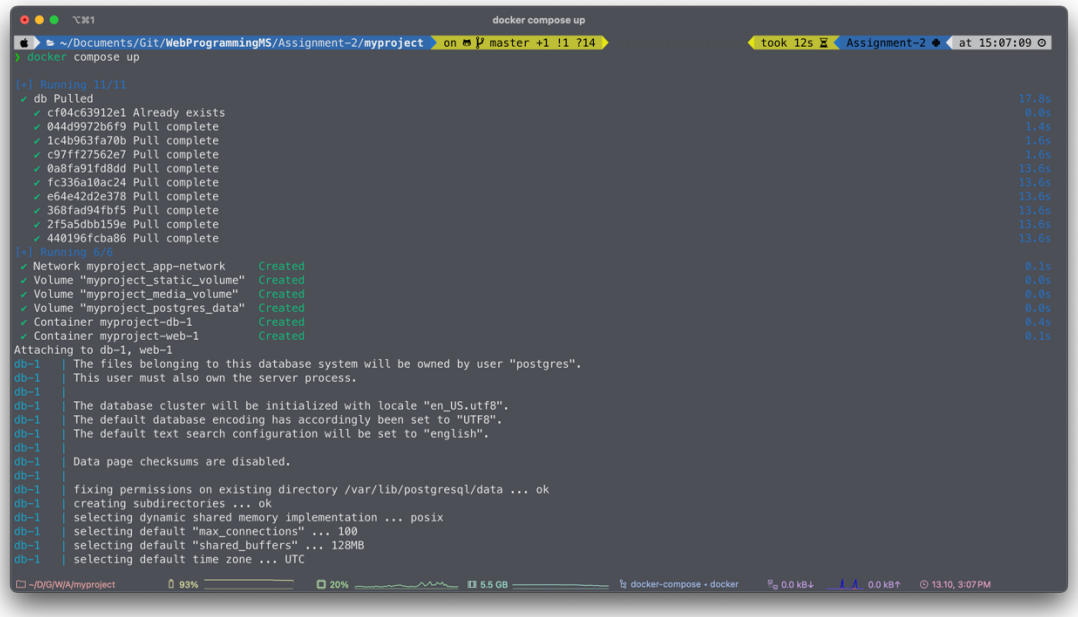
1. **Web (Django)**: This service runs the Django web application using `Gunicorn`. It connects to port `8000` for external access.
2. **PostgreSQL (Database)**: The database is defined using the `postgres:17-alpine` image, with its data persisted via volumes to ensure the database is retained across container restarts. The service exposes port `5432` for database connections.

Build and Run:

1. **Build**: The project was built using `docker compose build`, which read the `Dockerfile` to construct the necessary images.



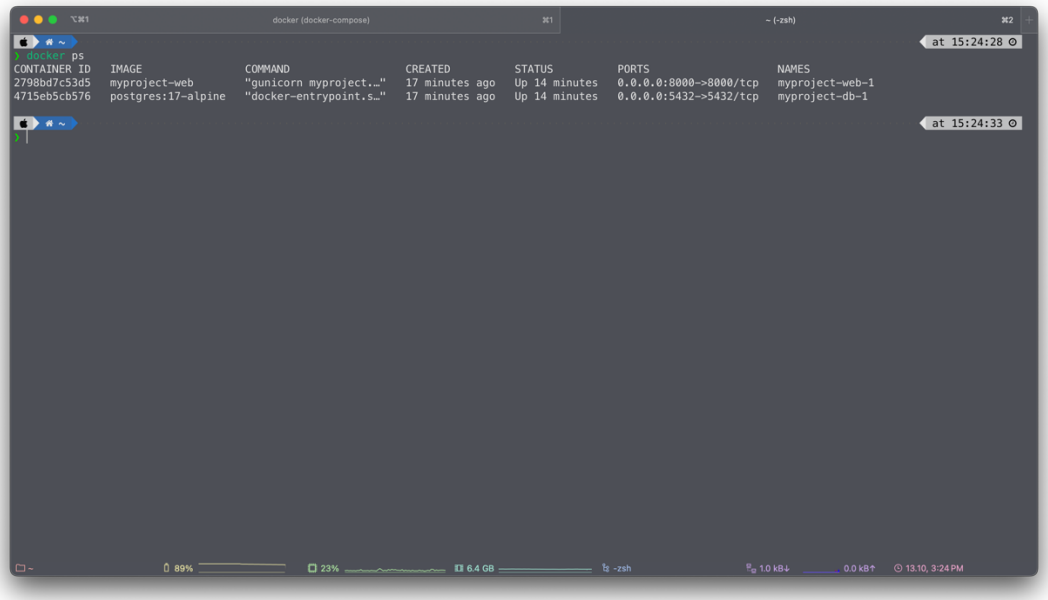
2. **Run**: After building, I used `docker compose up` to start both services. The PostgreSQL database was initialized successfully, and the Django application was able to connect and run without issues. No major challenges were encountered during this process.



- Ensure that the services are running correctly.

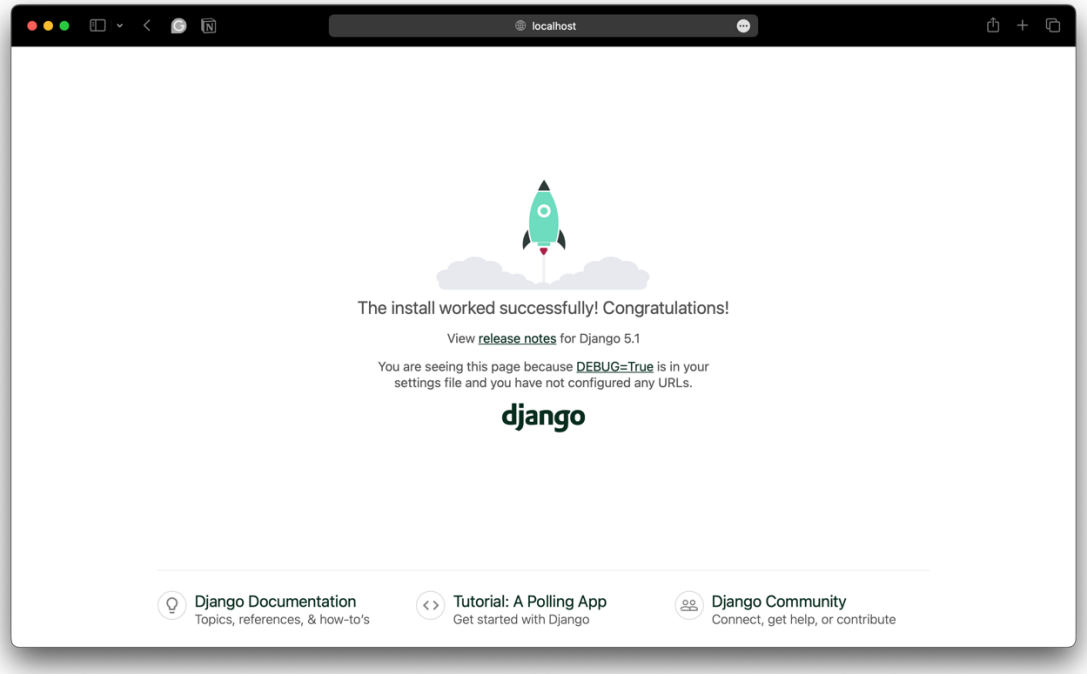
  The `docker ps` command showed that both services were running correctly.

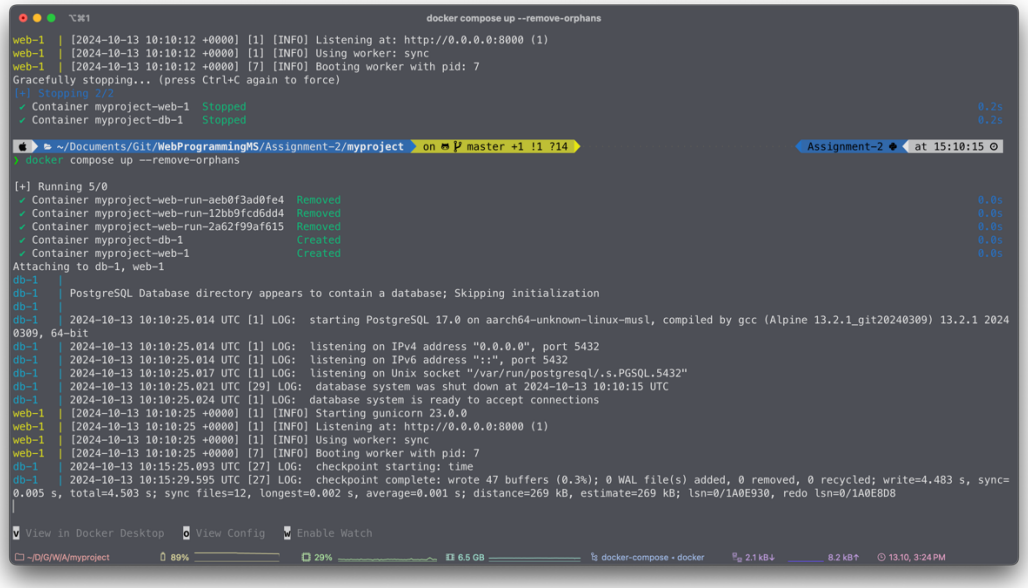
  The `Django` web app was accessible via port `8000`, while the `PostgreSQL` database was accessible internally via port `5432`. The services worked as expected, confirming the correct setup



The successful installation of `Django` is confirmed by the welcome page shown in the screenshot, accessed through localhost. This page indicates that `Django 5.1` is properly set up and running in debug mode. The app is functioning, although no specific URLs have been configured yet.

This confirms that the `Django` web server is working correctly in the `Docker` container, completing the basic setup.





Docker Networking and Volumes

Set Up Docker Networking

- Define a custom network in your `docker-compose.yml` file to allow communication between services.

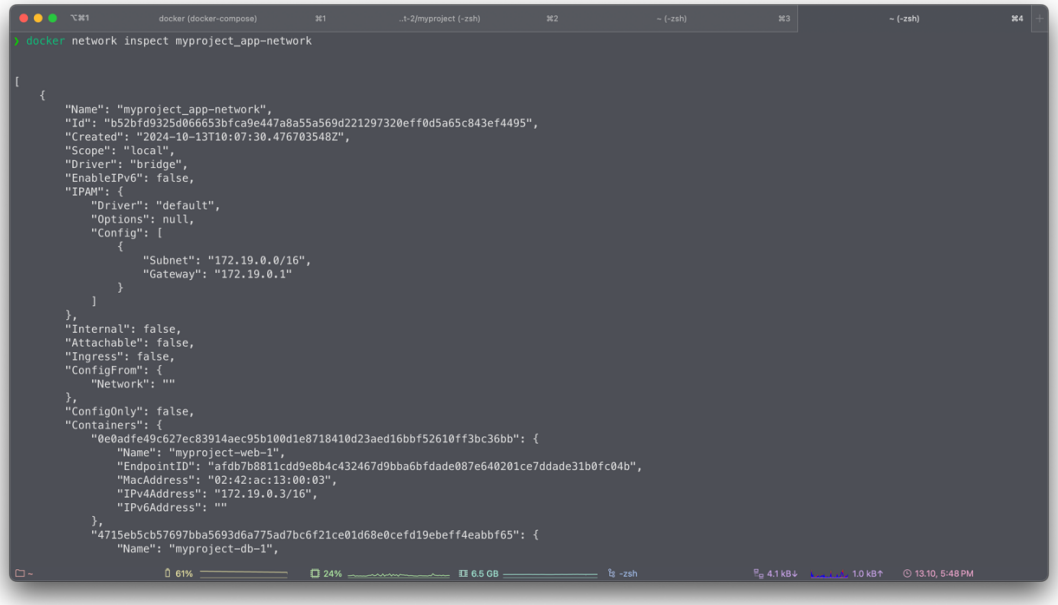
I modified the `docker-compose.yaml` file by adding a **custom network** that would allow the `Django` service to reach the `PostgreSQL` service. This way, two services could **connect** seamlessly within the `Docker`.

Network Setup: I named the custom network `app_network`, and assigned both services to this network. This guarantees that they can interact securely and efficiently within the same environment.

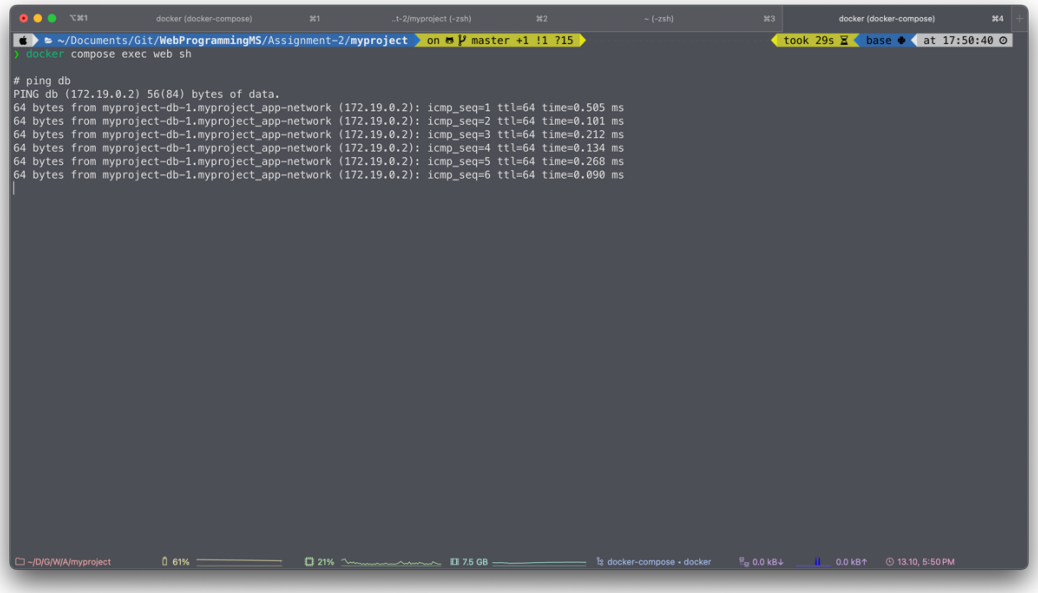
```
networks:
  - app-network
```

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

Verifying Network: The `Django` app successfully connected to `PostgreSQL` over this network, it was verified by running migrations and verifying how the database was reachable. One can see the custom network configuration in the network inspection screenshot. The network `myproject_app-network` was created using the bridge driver. The Docker Compose file has a subnet for `172.19.0.0/16`, which allows communication between containers through their internal IP addresses.



Lastly, a `ping` test from the Django container to your PostgreSQL database was done, and it had successful internal communications inside the network, with responses from `myproject-db-1` with `IP` address `172.19.0.2`. This would also mean the Django app and PostgreSQL are properly wired through the Docker network, and that should make both the web and database services seamless.



Implement Docker Volumes

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

For data persistence, I configured two volumes:

- One for `PostgreSQL`, to store database files so that on container restart, the data remains intact.

```
volumes:
  - postgres_data:/var/lib/postgresql/data/
```

Another for Django, to store `static files` and `media uploads` files since these are critical to any web application.

```
volumes:
  - .:/code
  - static_volume:/code/static
  - media_volume:/code/media
```

```
volumes:
  postgres_data:
  static_volume:
  media_volume:
```

Benefits of Networking and Volumes: `Docker networking` granted safe internal communication of services, while `volumes` allowed persistence both for the database and for static files, to ensure durability of data. In what follows you may find the complete `docker-compose.yaml` file.

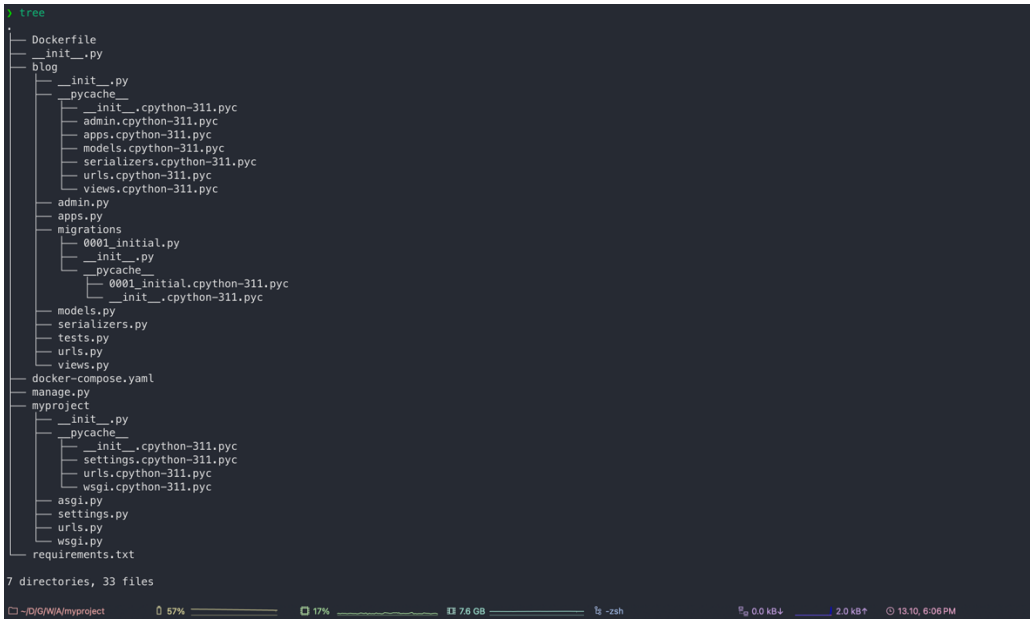
```yaml
services:
  web:
    build:
      context: .
      dockerfile: Dockerfile
    command:
gunicorn myproject.wsgi:application --bind 0.0.0.0:8000
    volumes:
      - .:/code
      - static_volume:/code/static
      - media_volume:/code/media
    ports:
      - "8000:8000"
    env_file:
      - .env
    depends_on:
      - db
    networks:
      - app-network

  db:
    image: postgres:17-alpine
    environment:
      - POSTGRES_DB=${DB_NAME}
      - POSTGRES_USER=${DB_USER}
      - POSTGRES_PASSWORD=${DB_PASSWORD}
    volumes:
      - postgres_data:/var/lib/postgresql/data/
    ports:
      - "5432:5432"
    networks:
      - app-network

volumes:
  postgres_data:
  static_volume:
  media_volume:

networks:
  app-network:
```

Django Application Setup

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.

Inside the Django container, I created a new project using the command `django-admin startproject myproject`. I then developed a simple app named blog using command `django-admin startapp blog`, which includes a basic model and corresponding view to display content. You can see below the structure of project.



Configure the Database

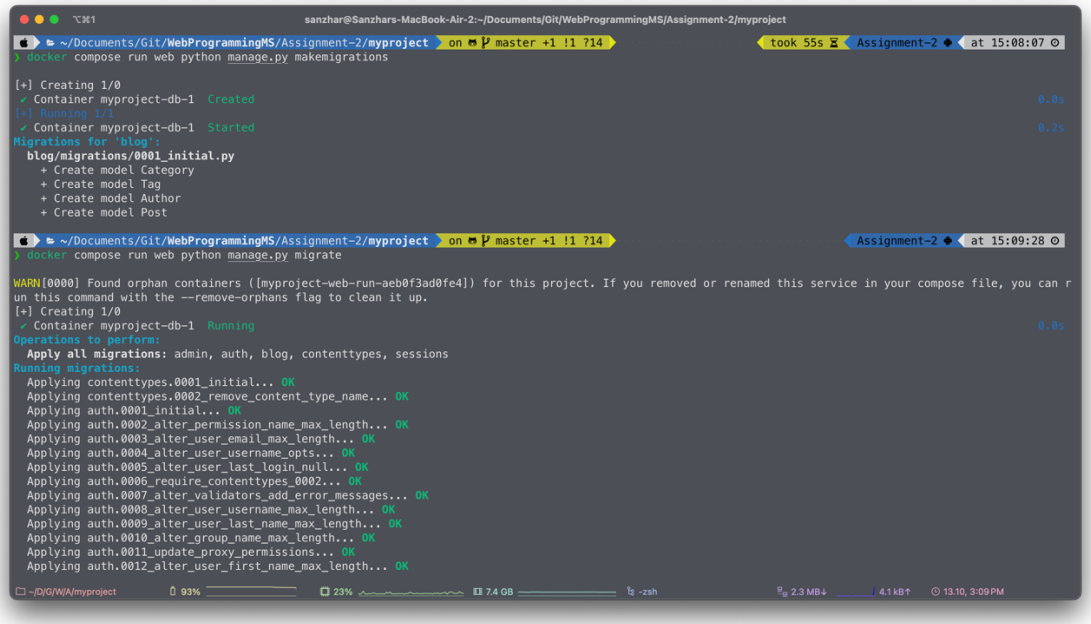
- Update the Django settings to use the PostgreSQL database configured in your Docker Compose setup.

I updated the `settings.py` file within the `Django` project to point to the `PostgreSQL` service by using the environment variables for `DB_NAME`, `DB_USER`, `DB_PASSWORD`, and other database configurations like `ENGINE`, `HOST` and `PORT`.

```
DATABASES = {
    "default": {
        "ENGINE": "django.db.backends.postgresql",
        "NAME": os.getenv("DB_NAME", "postgres"),
        "USER": os.getenv("DB_USER", "postgres"),
        "PASSWORD": os.getenv("DB_PASSWORD",
"postgres"),
        "HOST": "db",
        "PORT": 5432,
    }
}
```

- Run migrations to set up the database schema.

After configuring the database, I ran migrations using the command `python manage.py makemigrations` and `python manage.py migrate`, which successfully set up the database schema as defined in the Django models.



Model Definition: The `Post` model represents a blog post with fields like `title`, `content`, `author`, `status`, and date fields such as `created_at` and `published_at`. It supports relationships via `ForeignKey` for author and `ManyToManyFields` for categories and tags. The model's default ordering is by `published_at`, and status can be either "draft" or "published."

```python
class Post(models.Model):
    STATUS_CHOICES = (
        ("draft", "Draft"),
        ("published", "Published"),
    )

    title = models.CharField(max_length=200, unique=True)
    author = models.ForeignKey(
        Author, on_delete=models.CASCADE, related_name="blog_posts"
    )
    content = models.TextField()
    published_at = models.DateTimeField(null=True, blank=True)
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)
    status = models.CharField(max_length=10, choices=STATUS_CHOICES, default="draft")
    categories = models.ManyToManyField(Category, related_name="blog_posts", blank=True)
    tags = models.ManyToManyField("Tag", related_name="posts", blank=True)

    class Meta:
        ordering = ("-published_at",)
        verbose_name = "Post"
        verbose_name_plural = "Posts"

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

Migrations: The migration file defines the creation of the `Post` model, reflecting the fields and relationships from the model. It ensures that the database schema matches the model definitions.

```python
# Generated by Django 5.1.2 on 2024-10-13 10:09

import django.contrib.auth.models
import django.contrib.auth.validators
import django.db.models.deletion
import django.utils.timezone
from django.conf import settings
from django.db import migrations, models


class Migration(migrations.Migration):

    initial = True

    dependencies = [
        ('auth', '0012_alter_user_first_name_max_length'),
    ]

    operations = [
        migrations.CreateModel(
            name='Post',
            fields=[
                ('id', models.BigAutoField(auto_created=True, primary_key=True,
serialize=False, verbose_name='ID')),
                ('title', models.CharField(max_length=200, unique=True)),
                ('content', models.TextField()),
                ('published_at', models.DateTimeField(blank=True, null=True)),
                ('created_at', models.DateTimeField(auto_now_add=True)),
                ('updated_at', models.DateTimeField(auto_now=True)),
                ('status', models.CharField(choices=[('draft', 'Draft'), ('published'
, 'Published')], default='draft', max_length=10)),
                ('author', models.ForeignKey(on_delete=django.db.models.deletion.
CASCADE, related_name='blog_posts', to=settings.AUTH_USER_MODEL)),
                ('categories', models.ManyToManyField(blank=True, related_name=
'blog_posts', to='blog.category')),
                ('tags', models.ManyToManyField(blank=True, related_name='posts', to=
'blog.tag')),
            ],
            options={
                'verbose_name': 'Post',
                'verbose_name_plural': 'Posts',
                'ordering': ('-published_at',),
            },
        ),
    ]
```

Serialization: The `PostSerializer` prepares the `Post` data for the API, including fields like `id`, `title`, `author`, `status`, `content`, and relationships

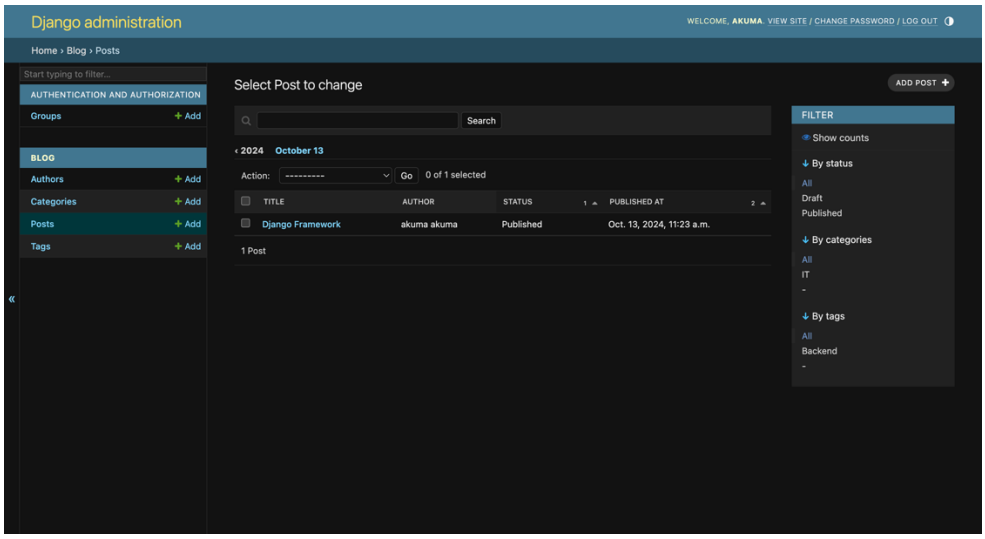
such as `categories` and `tags`. Read-only fields are `id`, `author`, and `updated_at`.

```
class PostSerializer(serializers.ModelSerializer):
    author = AuthorSerializer(read_only=True)
    categories = CategorySerializer(many=True, required=False)
    tags = TagSerializer(many=True, required=False)

    class Meta:
        model = Post
        fields = [
            "id",
            "title",
            "author",
            "content",
            "published_at",
            "created_at",
            "updated_at",
            "status",
            "categories",
            "tags",
        ]
        read_only_fields = ["id", "author", "published_at", "created_at",
"updated_at"]
```

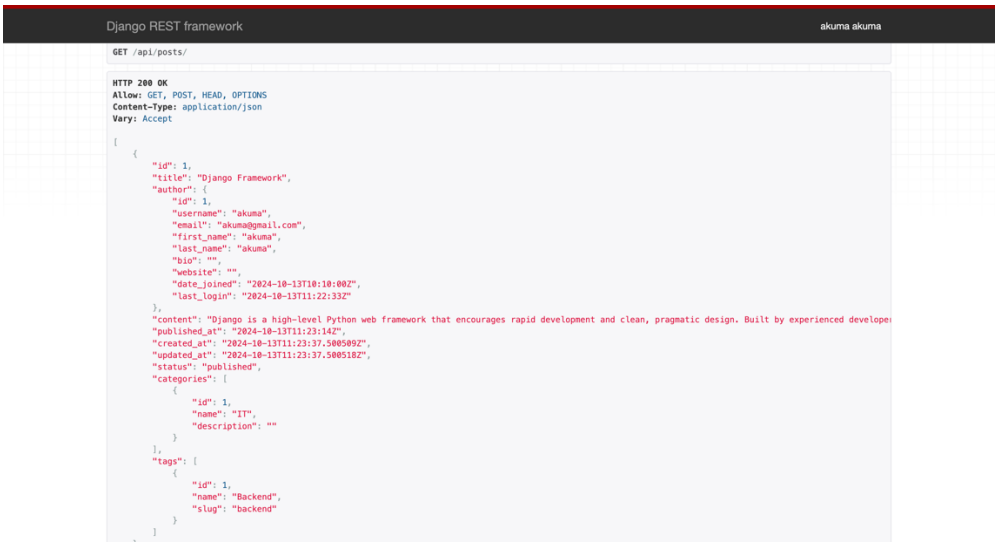
API View: The `PostViewSet` handles the API logic for retrieving, creating, updating, and deleting blog posts. It uses `IsAuthenticatedOrReadOnly` permissions, meaning only authenticated users can modify data, while others can read it.

```
class PostViewSet(viewsets.ModelViewSet):
    queryset = Post.objects.all()
    serializer_class = PostSerializer
    permission_classes = [permissions.IsAuthenticatedOrReadOnly]
```

Django Admin Interface: The admin panel shows the posts with relevant metadata, allowing management of `Posts`, `Authors`, `Categories`, and `Tags` directly from the UI.



Django REST Framework API Output: The API endpoint `/api/posts/` successfully returns data in JSON format for a Post, showing fields such as `title`, `content`, `author`, `categories`, and `tags`, confirming the correct functioning of the serializer and view set.



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

  The Django project runs smoothly within Docker, and its services (Django and PostgreSQL) work in harmony because of the proper configurations in the `docker-compose.yml` file. Docker allows for isolated development environments, which streamlines development and testing.

Findings

The development of the Django application inside Docker created a seamless environment for development and deployment. Docker allowed containerization that isolated the application's dependencies, and the application behaved identically across systems. I could configure service running Django and PostgreSQL with Docker Compose in a way that they could talk to each other without conflicts. The ability to spot networks and volumes gave persistent storage data and locked inter-service communication that would otherwise turn out to be more cumbersome to manage manually. Overall, Docker made the setup more maintainable, portable, and easier to scale.

Conclusion

This lab assignment has provided great insights into deploying a web application using Docker. I learned how to create and configure the docker-compose.yml file in order to orchestrate both Django and PostgreSQL services with environment variables. The volumes with a custom network setup gave a formidable insight into the capabilities of Docker to manage data persistence and service interactions. Docker makes the whole process reproducible and easier, while Docker Compose allows orchestrating multi-service.

References

- Django Documentation: https://docs.djangoproject.com/en/stable/
- Docker Documentation: https://docs.docker.com/
- Docker Compose Reference: https://docs.docker.com/compose/compose-file/
- Django REST Framework Documentation: https://www.django-rest-framework.org/
- PostgreSQL Docker Image Documentation: https://hub.docker.com/_/postgres