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: Turganbek Y.
Checked by: Serek A.

Almaty, 2024

CONTENT

Introduction 3
1. Docker Compose 4
2. Docker Networking and Volumes..... 6
3. Django Application Setup..... 7
Conclusion 9

INTRODUCTION

This assignment focuses on learning how to set up a Django web application using Docker. The goal is to help us understand how Docker works with web applications by using Docker Compose, networking, and volumes. By the end, we will have a working Django app inside Docker containers. This will help us get practical experience with Docker and make managing web applications easier.

1. Docker Compose

- **Creating docker-compose file with Django And Postgres:**

docker-compose.yml

```yaml
version: '3.8'

services:
  web:
    build: .
    volumes:
      - .:/app
    expose:
      - 8088
    environment:
      - DEBUG=1
      - DJANGO_DB_HOST=db
      - DJANGO_DB_PORT=5432
      - DJANGO_DB_NAME=demo
      - DJANGO_DB_USER=demo
      - DJANGO_DB_PASSWORD=demo
    depends_on:
      - db
      - redis

  db:
    image: postgres:13
    volumes:
      - postgres_data:/var/lib/postgresql/data/
    environment:
      - POSTGRES_DB=demo
      - POSTGRES_USER=demo
      - POSTGRES_PASSWORD=demo
volumes:
  postgres_data:
```

```
turgan6ek@Yerulans-MacBook-Pro demo % docker-compose up
[+] Building 0.0s (0/0)                                                                                                 docker:desktop-linux
[+] Running 2/2
 ✔ Container demo-db-1   Created                                                                                                    0.0s
 ✔ Container demo-web-1  Created                                                                                                    0.0s
Attaching to demo-db-1, demo-web-1
demo-db-1   |
demo-db-1   | PostgreSQL Database directory appears to contain a database; Skipping initialization
demo-db-1   |
demo-db-1   | 2024-10-13 11:17:55.102 UTC [1] LOG:  starting PostgreSQL 13.16 (Debian 13.16-1.pgdg120+1) on aarch64-unknown-linux-gnu, compiled by gcc (Debian 12.2.0-14) 12.2.0
, 64-bit
demo-db-1   | 2024-10-13 11:17:55.102 UTC [1] LOG:  listening on IPv4 address "0.0.0.0", port 5432
demo-db-1   | 2024-10-13 11:17:55.102 UTC [1] LOG:  listening on IPv6 address "::", port 5432
demo-db-1   | 2024-10-13 11:17:55.104 UTC [1] LOG:  listening on Unix socket "/var/run/postgresql/.s.PGSQL.5432"
demo-db-1   | 2024-10-13 11:17:55.106 UTC [27] LOG:  database system was shut down at 2024-10-13 11:16:47 UTC
demo-db-1   | 2024-10-13 11:17:55.110 UTC [1] LOG:  database system is ready to accept connections
demo-web-1  | [2024-10-13 11:17:55 +0000] [1] [INFO] Starting gunicorn 23.0.0
demo-web-1  | [2024-10-13 11:17:55 +0000] [1] [INFO] Listening at: http://0.0.0.0:8000 (1)
demo-web-1  | [2024-10-13 11:17:55 +0000] [1] [INFO] Using worker: sync
demo-web-1  | [2024-10-13 11:17:55 +0000] [7] [INFO] Booting worker with pid: 7
```

Short explanation of the configuration:
version: ‘3.8’: Specifies the version of Docker Compose being used.
services:
- **web**:
- **build:** . : Builds the Docker image from the current directory (‘.’) where the ‘Dockerfile’ is located.
- **volumes**: Maps the local directory (‘.’) to ‘/app’ inside the container, allowing code changes to reflect in real time.
- **expose**: Exposes port 8088 for internal communication within Docker containers.
- **environment**: Sets environment variables for Django, including enabling debug mode (‘DEBUG=1’) and database connection settings (‘DJANGO_DB_HOST’, ‘DJANGO_DB_PORT’, etc.).
- **depends_on**: Ensures that the ‘db’ service is started before the ‘web’ service.
- **db**:
- **image**: Uses the PostgreSQL 13 image.
- **volumes**: Persists database data by mounting ‘postgres_data’ to ‘/var/lib/postgresql/data/’.
- **environment**: Sets up the PostgreSQL database with the name ‘demo’ and user credentials.

volumes:
- **postgres_data**: A volume for persistent storage of PostgreSQL data.

2. Docker Networking and Volumes

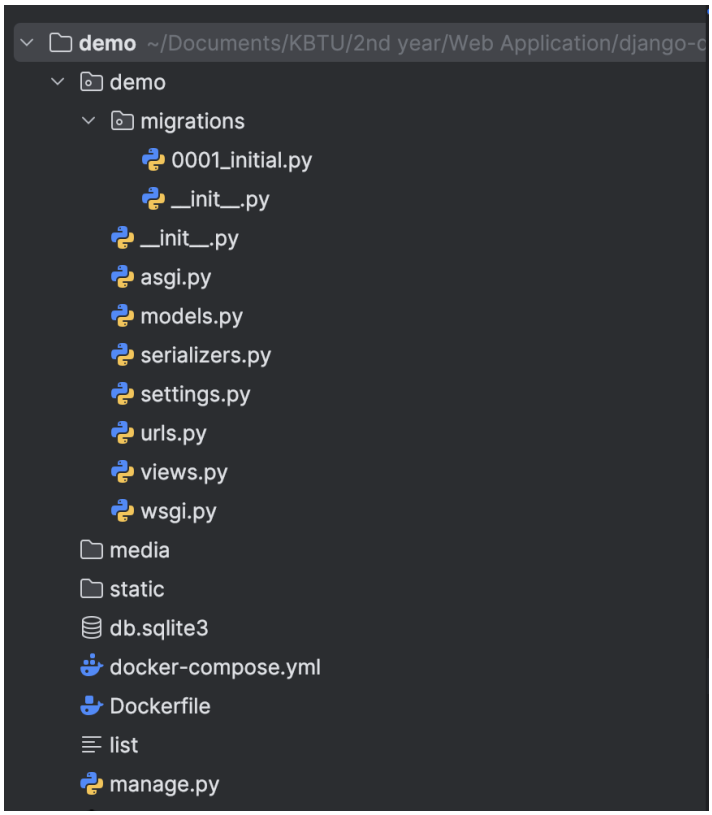
```yaml
networks:
  app_network:
    driver: bridge

services:
  web:
    build: .
    volumes:
      - .:/app
      - static_volume:/app/static
      - media_volume:/app/media
    expose:
      - 8088
    environment: <6 items>
    depends_on: <1 item>
    networks:
      - app_network

  db:
    image: postgres:13
    volumes:
      - postgres_data:/var/lib/postgresql/data/
    environment: <3 items>
    networks:
      - app_network

volumes:
  postgres_data:
  static_volume:
  media_volume:
```

```
turgan6ek@Yerulans-MacBook-Pro demo % docker-compose up
[+] Building 0.0s (0/0)                                                                                    docker:desktop-linux
[+] Running 4/4
[+] Running 5/5_app_network     Created                                                                                    0.0s
 ✔ Network demo_app_network     Created                                                                                    0.0s
 ✔ Volume "demo_media_volume"   Created                                                                                    0.0s
 ✔ Volume "demo_static_volume"  Created                                                                                    0.0s
 ✔ Container demo-db-1          Recreated                                                                                  0.0s
 ✔ Container demo-web-1         Recreated                                                                                  0.1s
Attaching to demo-db-1, demo-web-1
demo-db-1   |
demo-db-1   | PostgreSQL Database directory appears to contain a database; Skipping initialization
demo-db-1   |
demo-db-1   | 2024-10-13 11:31:55.681 UTC [1] LOG:  starting PostgreSQL 13.16 (Debian 13.16-1.pgdg120+1) on aarch64-unknown-linux-gnu, compiled by gcc (Debian 12.2.0-14) 12.2.0
, 64-bit
demo-db-1   | 2024-10-13 11:31:55.681 UTC [1] LOG:  listening on IPv4 address "0.0.0.0", port 5432
demo-db-1   | 2024-10-13 11:31:55.681 UTC [1] LOG:  listening on IPv6 address "::", port 5432
demo-db-1   | 2024-10-13 11:31:55.682 UTC [1] LOG:  listening on Unix socket "/var/run/postgresql/.s.PGSQL.5432"
demo-db-1   | 2024-10-13 11:31:55.684 UTC [26] LOG:  database system was shut down at 2024-10-13 11:23:09 UTC
demo-db-1   | 2024-10-13 11:31:55.688 UTC [1] LOG:  database system is ready to accept connections
demo-web-1  | [2024-10-13 11:31:55 +0000] [1] [INFO] Starting gunicorn 23.0.0
demo-web-1  | [2024-10-13 11:31:55 +0000] [1] [INFO] Listening at: http://0.0.0.0:8000 (1)
demo-web-1  | [2024-10-13 11:31:55 +0000] [1] [INFO] Using worker: sync
demo-web-1  | [2024-10-13 11:31:55 +0000] [7] [INFO] Booting worker with pid: 7
```

The custom network (app_network) allows the Django app (web) and the PostgreSQL database (db) to communicate securely and easily using service names. This ensures the services are isolated from external networks. The volumes (postgres_data, static_volume, media_volume) persist data for the database, static files, and media files, so no data is lost when the containers are stopped or restarted, ensuring a stable application environment.

3. Django Application Setup

Directory of my Django Project (Task from classwork)




```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'demo',
        'USER': 'demo',
        'PASSWORD': 'demo',
        'HOST': 'db',
        'PORT': '5432',
    }
}
```

After configuring Django Db settings, we run migrations using docker-compose.

```
turganбek@Yerulans-MacBook-Pro demo % docker-compose run web python manage.py migrate

[+] Building 0.0s (0/0)                                                                                    docker:desktop-linux
[+] Creating 1/1
 ✔ Container demo-db-1  Running                                                                                          0.0s
[+] Building 0.0s (0/0)                                                                                    docker:desktop-linux
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, demo, 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 demo.0001_initial... OK
  Applying sessions.0001_initial... OK
turganбek@Yerulans-MacBook-Pro demo %
```

Models are:

```python
class Migration(migrations.Migration):

    initial = True

    dependencies = [
    ]

    operations = [
        migrations.CreateModel(
            name='Category',
            fields=[
                ('id', models.BigAutoField(auto_created=True, primary_key=True, serialize=False, verbose_name='ID')),
                ('name', models.CharField(max_length=100)),
            ],
        ),
        migrations.CreateModel(
            name='Customer',
            fields=[
                ('id', models.BigAutoField(auto_created=True, primary_key=True, serialize=False, verbose_name='ID')),
                ('name', models.CharField(max_length=100)),
                ('email', models.EmailField(max_length=254, unique=True)),
            ],
        ),
        migrations.CreateModel(
            name='Product',
            fields=[
                ('id', models.BigAutoField(auto_created=True, primary_key=True, serialize=False, verbose_name='ID')),
                ('name', models.CharField(max_length=100)),
                ('description', models.TextField()),
                ('price', models.DecimalField(decimal_places=2, max_digits=10)),
                ('category', models.ForeignKey(on_delete=django.db.models.deletion.CASCADE, related_name='products', to='demo.category')),
            ],
        ),
        migrations.CreateModel(
            name='Order',
            fields=[
```

The Django application is structured with a project directory containing the main configuration files and apps like Ecommerce. The project directory includes settings (`settings.py`), URL routing (`urls.py`), and WSGI configuration (`wsgi.py`) and etc.

The application interacts with Docker through a `docker-compose.yml` file, which defines services (like the Django web server and PostgreSQL database) and their configurations. Docker manages the containers, allowing the Django app to run in an isolated environment while connecting to the database using the specified service names. This setup ensures easy deployment, scalability, and consistent environments across development and production.

8

CONCLUSION

In this assignment, we gained valuable insights into integrating Django with Docker. One of the key learnings was setting up a Django project within a Docker container, which highlighted the benefits of containerization, such as ease of deployment and consistent development environments. We learned how to use Docker Compose to manage multiple services, like the Django web application and PostgreSQL database, enabling efficient communication between them.

Reflecting on the significance of using Docker with Django for application development, we realized how it streamlines the development process. Docker allows us to package applications with all their dependencies, ensuring they run the same way in different environments. This eliminates the frustrating "it works on my machine" problem, enhances collaboration among team members, and simplifies the deployment process to production. Overall, integrating Docker into our Django projects not only improves efficiency but also contributes to building more robust and scalable applications.