

Assignment 2, Web app dev
Exploring Django with Docker
Amanzhol Temirbolat

10.10.2024

Table of Contents

1. Introduction	1
1.1 Objective of the Assignment	1
1.2 Overview of Docker and Django Integration	1
2. Docker Compose	2
2.1 Docker Compose Configuration	2
2.2 Building and Running Containers	3
3. Docker Networking and Volumes	4
3.1 Custom Network Setup	4
3.2 Implementing Docker Volumes	5
3.3 Findings on Networking and Volumes	6
4. Django Application Setup	7
4.1 Project Structure Overview	7
4.2 Database Configuration	8
4.3 Findings on Application Development	9
5. Conclusion	9
5.1 Key Learnings	9

Introduction

The objective of this assignment is to gain practical experience in deploying a Django application using Docker, focusing on Docker Compose, networking, and volumes. By containerizing the Django application, students will learn how to efficiently manage and orchestrate multiple services, such as the web server and PostgreSQL database, within isolated environments. This assignment also aims to demonstrate the advantages of Docker in simplifying the development workflow and ensuring consistency across different environments.

Using Docker with Django serves several purposes. Docker enables developers to create isolated, reproducible environments, eliminating discrepancies between development, testing, and production setups. With Docker Compose, managing multiple services such as the Django application and its database becomes straightforward. Docker also helps in maintaining version control for the environment, ensuring that all dependencies, configurations, and services are consistent, regardless of the underlying host system. This approach promotes better scalability, portability, and simplified debugging, enhancing overall application development and deployment efficiency.

1. Docker Compose

```
Test.py  docker-compose.yml x
1  >> services:
2  >> db:
3      image: postgres:13
4      container_name: postgres_db
5      environment:
6          POSTGRES_DB: ${DB_NAME}
7          POSTGRES_USER: ${DB_USER}
8          POSTGRES_PASSWORD: ${DB_PASSWORD}
9      volumes:
10         - postgres_data:/var/lib/postgresql/data
11     ports:
12         - "5432:5432"
13     env_file:
14         - .env
15
16 >> web:
17     build: .
18     command: python manage.py runserver 0.0.0.0:8000
19     volumes:
20         - ../code
21     ports:
22         - "8000:8000"
23     depends_on:
24         - db
25     environment:
26         DB_NAME: ${DB_NAME}
27         DB_USER: ${DB_USER}
28         DB_PASSWORD: ${DB_PASSWORD}
29         DB_HOST: db
30         DB_PORT: 5432
31     env_file:
32         - .env
33
34 >> volumes:
35     postgres_data:
36
```

The docker-compose.yml file is a core component of this project, responsible for defining and managing multiple services required for the Django application. In this setup, two main services are defined:

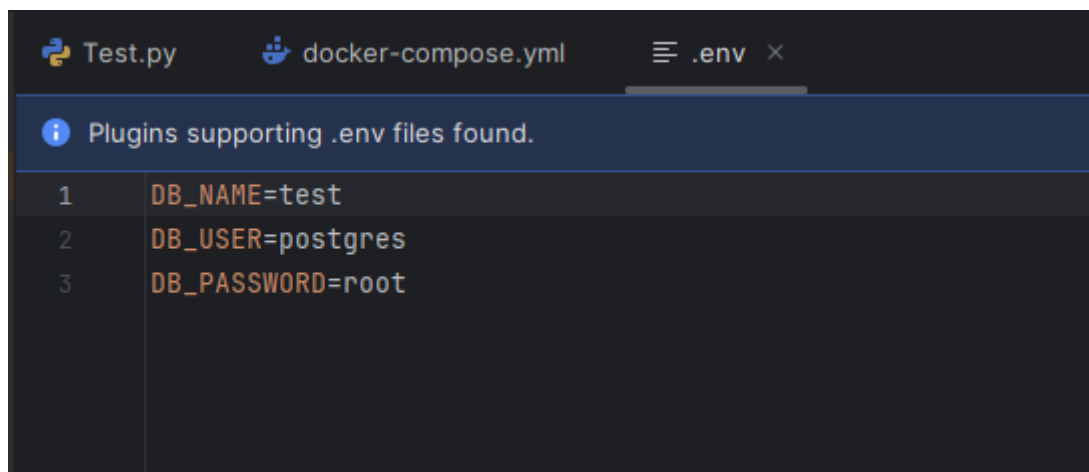
□ **db (PostgreSQL Database):**

- This service pulls the official PostgreSQL image from Docker Hub (postgres:13) to create the database container.
- Environment variables such as POSTGRES_DB, POSTGRES_USER, and POSTGRES_PASSWORD are defined to configure the database. These values are pulled from the .env file to avoid hardcoding sensitive information.
- A volume (postgres_data) is used to persist PostgreSQL data, ensuring that database data is retained even if the container is stopped or removed.
- The PostgreSQL service is also part of a custom Docker network, allowing it to communicate securely with the Django web server.

□ **web (Django Web Server):**

- The web service is responsible for running the Django application.
- It builds the application using the Dockerfile in the project directory, setting up the necessary environment for the web application.
- The service mounts the current working directory into the container, allowing changes made to the code to reflect immediately inside the container without rebuilding.
- Environment variables such as DB_NAME, DB_USER, DB_PASSWORD, DB_HOST, and DB_PORT are used to configure Django to connect to the PostgreSQL database.
- The depends_on directive ensures that the database service (db) starts before the Django service.

The .env file contains the following variables:

A screenshot of a code editor with a dark theme. At the top, there are three tabs: 'Test.py' with a Python icon, 'docker-compose.yml' with a Docker icon, and '.env' with a file icon. The '.env' tab is active. Below the tabs, a blue notification bar says 'Plugins supporting .env files found.' with an information icon. The main area shows the content of the .env file:

```
1 DB_NAME=test
2 DB_USER=postgres
3 DB_PASSWORD=root
```

The command `docker-compose up --build` is used to build and start the containers.

Running the Application:

- After the containers are built, the services are started in the defined order (database first, then the web application).
- The Django application can be accessed via `localhost:8000`, and it communicates with the PostgreSQL database via the custom Docker network.

```
[+] Running 4/2
✓Network hello-docker_default Created 0.0s
✓Volume "hello-docker_postgres_data" Created 0.0s
✓Container postgres_db Created 0.0s
✓Container hello-docker-web-1 Created 0.0s
Attaching to web-1, postgres_db
postgres_db | The files belonging to this database system will be owned by user "postgres".
postgres_db | This user must also own the server process.
postgres_db |
postgres_db | The database cluster will be initialized with locale "en_US.utf8".
postgres_db | The default database encoding has accordingly been set to "UTF8".
postgres_db | The default text search configuration will be set to "english".
postgres_db |
postgres_db | Data page checksums are disabled.
postgres_db |
postgres_db | fixing permissions on existing directory /var/lib/postgresql/data ... ok
postgres_db | creating subdirectories ... ok
postgres_db | selecting dynamic shared memory implementation ... posix
postgres_db | selecting default max_connections ... 100
postgres_db | selecting default shared_buffers ... 128MB
postgres_db | selecting default time zone ... Etc/UTC
postgres_db | creating configuration files ... ok
postgres_db | running bootstrap script ... ok
web-1 | python: can't open file '/code/manage.py': [Errno 2] No such file or directory
postgres_db | performing post-bootstrap initialization ... ok
postgres_db | syncing data to disk ... ok
postgres_db |
postgres_db | Success. You can now start the database server using:
postgres_db |
postgres_db |     pg_ctl -D /var/lib/postgresql/data -l logfile start
postgres_db |
postgres_db | initdb: warning: enabling "trust" authentication for local connections
postgres_db | You can change this by editing pg_hba.conf or using the option -A, or
postgres_db | --auth-local and --auth-host, the next time you run initdb.
postgres_db | waiting for server to start....2024-10-10 06:38:46.111 UTC [48] LOG:  starting PostgreSQL 13.16 (Debian 13.16-1.pgdg120+1) on x86_64-pc-linux-gnu, compiled
it
postgres_db | 2024-10-10 06:38:46.112 UTC [48] LOG:  listening on Unix socket "/var/run/postgresql/.s.PGSQL.5432"
postgres_db | 2024-10-10 06:38:46.159 UTC [49] LOG:  database system was shut down at 2024-10-10 06:38:45 UTC
postgres_db | 2024-10-10 06:38:46.162 UTC [48] LOG:  database system is ready to accept connections
postgres_db | done
postgres_db | server started
web-1 exited with code 2
postgres_db | CREATE DATABASE
postgres_db |
postgres_db | /usr/local/bin/docker-entrypoint.sh: ignoring /docker-entrypoint-initdb.d/*
postgres_db | waiting for server to shut down...2024-10-10 06:38:46.291 UTC [48] LOG:  received fast shutdown request
```

Challenges Faced

1. Environment Variable Issues:

- During the initial setup, environment variables from the .env file were not loading correctly into the containers. This issue was resolved by adding the env_file directive explicitly to both the db and web services in the docker-compose.yml file to ensure that Docker Compose correctly referenced the environment variables.

2) Docker Networking and Volumes

Set Up Docker Networking

To allow communication between the Django application and the PostgreSQL database, a custom Docker network is defined in the docker-compose.yml file. This ensures that both services can interact with each other securely and efficiently, avoiding external exposure unless explicitly needed.

Network Configuration:

In the docker-compose.yml file, a custom network named mynetwork is defined. Both the db (PostgreSQL) and web (Django) services are added to this network, allowing them to communicate using service names rather than IP addresses. This makes the configuration more flexible and robust.

```

1  networks:
2    mynetwork:
3      driver: bridge
4
5  > services:
6    > db:
7      image: postgres:13
8      container_name: postgres_db
9      networks:
10     - mynetwork
11     environment:
12       POSTGRES_DB: ${DB_NAME}
13       POSTGRES_USER: ${DB_USER}
14       POSTGRES_PASSWORD: ${DB_PASSWORD}
15     volumes:
16       - postgres_data:/var/lib/postgresql/data
17     ports:
18       - "5432:5432"
19     env_file:
20       - .env
21
22  > web:
23     build: .
24     networks:
25       - mynetwork
26     command: python manage.py runserver 0.0.0.0:8000
27     volumes:
28       - ./code
29       - django_static:/code/static
30       - django_media:/code/media
31     ports:
32       - "8000:8000"
33     depends_on:
34       - db
35     environment:
36       DB_NAME: ${DB_NAME}
37       DB_USER: ${DB_USER}
38       DB_PASSWORD: ${DB_PASSWORD}
39       DB_HOST: db
40       DB_PORT: 5432
41     env_file:
42       - .env

```

In this setup:

- The network named mynetwork is created with the default bridge driver.
- Both services (db and web) are attached to the mynetwork, enabling communication between them using service names (db as the host for the PostgreSQL database).

Volumes: Implementation for Data Persistence

Volumes in Docker are crucial for persisting data generated by applications. In this setup, two types of volumes were implemented:

1. PostgreSQL Data Volume:

- A volume named postgres_data was created to persist PostgreSQL data. This volume maps to the PostgreSQL data directory (/var/lib/postgresql/data), ensuring that the database retains its state even when the container is stopped or removed.
- Configuration in the docker-compose.yml file looks like this:

```

volumes:
  postgres_data:

```

2. Django Static and Media Volumes:

- Two additional volumes, `django_static` and `django_media`, were defined to handle the storage of static and media files:

```
volumes:
  django_static:
  django_media:
```

- This ensures that user-uploaded files and static assets are preserved across container restarts, providing a reliable user experience.

By mounting these volumes, the application can maintain a consistent state and user-generated data, which is crucial for applications like Django that often handle dynamic content.

Findings: Advantages of Using Docker Networking and Volumes

1. Data Persistence:

- The use of volumes allowed the PostgreSQL database to retain its data between container lifecycles, protecting against data loss during updates or restarts. This reliability is essential for any application that requires persistent data storage.

2. Enhanced Security and Performance:

- Custom networking ensured that only necessary services were exposed to each other, improving security while optimizing communication performance between services.

3. Simplified Development Workflow:

- The combination of networking and volumes simplified the development and deployment processes. Developers can focus on building features without worrying about environment discrepancies, as everything runs in a consistent Docker environment.

4. Scalability and Flexibility:

- Docker networking and volumes provide a scalable architecture that can easily accommodate additional services or changes to existing services, making it ideal for future development.

3. Django Application Setup

In `blog/views.py`, create a simple view to list all posts

```
from django.shortcuts import render
from .models import Post

</>
def post_list(request): 1 usage
    posts = Post.objects.all()
    return render(request, 'blog/post_list.html', context={'posts': posts})
```

In `blog/models.py`, define a simple model called `Post` with fields for `title` and `content`:

```
from django.db import models

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
```

In `myproject/urls.py`, include the blog app URLs

```
import ...
from blogs import views

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', views.post_list, name='post_list'),
]
```

The `settings.py` file in the `myproject` directory is configured to connect to the PostgreSQL database using environment variables. This approach allows for flexible configurations based on different environments (development, testing, production).

```
import os

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': os.environ.get('DB_NAME'),
        'USER': os.environ.get('DB_USER'),
        'PASSWORD': os.environ.get('DB_PASSWORD'),
        'HOST': os.environ.get('DB_HOST'),
        'PORT': os.environ.get('DB_PORT'),
    }
}
```

Explanation of Django Application Structure and Interaction with Docker

1. Project Structure:

- The Django project (`myproject`) contains essential files, including `settings`, `URLs`, and `WSGI` configurations, which define the core structure and behavior of the application.
- The `blog` app contains the `models`, `views`, and `templates` specific to the blogging feature of the application.

2. Database Interaction:

- The application uses environment variables defined in the .env file to configure the PostgreSQL database connection. This integration allows for seamless communication between the Django application and the database, utilizing the custom Docker network for internal connectivity.
3. **Container Interaction:**
- The Django application runs inside a Docker container, isolated from the host system. Using Docker Compose, the application can be built, run, and managed easily alongside its database service.
 - By utilizing Docker volumes, any user-uploaded files or static assets can persist even when containers are restarted, enhancing the overall reliability of the application.

Conclusion

This assignment has provided an in-depth exploration of developing a Django application within a Docker environment, highlighting the advantages and functionalities offered by these technologies. The following key learnings emerged from this experience:

1. **Understanding Docker and Docker Compose:**
 - The assignment emphasized the importance of Docker as a containerization platform, allowing applications to run consistently across different environments. Learning to use Docker Compose for orchestrating multi-container applications simplified the management of interconnected services, such as a Django web application and a PostgreSQL database.
2. **Environment Isolation:**
 - By using Docker, the application benefits from an isolated environment that avoids dependency conflicts and simplifies the setup process. This isolation is crucial for maintaining a clean development environment and ensures that applications behave consistently, regardless of where they are deployed.
3. **Data Persistence with Volumes:**
 - The implementation of Docker volumes for data persistence reinforced the importance of maintaining database state across container restarts. This feature is vital for applications that handle user-generated content or require long-term data storage, ensuring that important information is retained even when the application is redeployed or updated.
4. **Seamless Integration of Services:**
 - The project demonstrated how Docker networking facilitates seamless communication between services. By defining a custom network, the Django application was able to interact efficiently with the PostgreSQL database, enhancing both security and performance.
5. **Rapid Development and Testing:**
 - The use of Docker enabled rapid iteration and testing cycles. Developers can make changes to the codebase, rebuild the containers, and test the application in a consistent environment with minimal setup time, fostering an efficient development workflow.
6. **Scalability and Flexibility:**
 - Docker's architecture allows for easy scalability, enabling developers to add new services and functionalities without significant reconfiguration. This flexibility is

particularly beneficial in modern web development, where applications often need to adapt to changing requirements and increased user demand.

In conclusion, this assignment has not only deepened my understanding of Django and Docker but also highlighted the transformative potential of containerization in modern application development. The combination of these technologies empowers developers to create robust, scalable, and maintainable applications that meet the demands of today's dynamic web environment.