



School of Information Technology and Engineering

**Midterm Project: Building a Task Management Application Using Django
and Docker**

Discipline: Web Application Development
Student: Makhatbek Ilyas
ID: 23MD0502

Table of Contents

1. Executive Summary	3
2. Introduction	3
3. Project Objectives	3
4. Intro to Containerization: Docker	3
4.1 Containerization Overview	4
4.2 Docker Installation	4
5. Creating a Dockerfile	5
5.1 Dockerfile Structure	5
5.2 Dependencies	5
6. Using Docker Compose	6
6.1 Compose File Structure	6
6.2 Service Definitions	6
7. Docker Networking and Volumes	7
7.1 Networking	7
7.2 Volumes	7
8. Django Application Setup	8
8.1 Project Initialization	8
8.2 Configuration	8
9. Defining Django Models	9
9.1 Model Creation	9
9.2 Migrations	9
10. Conclusion	10
11. References	11

1. Executive Summary

The goal of this project was to build a task management web application using Django and to use Docker to make sure that the environment for development and production was consistent. The app allows users to create, view, update, and delete tasks. Docker was used to package the application into a container, which made it easy to run on different systems without compatibility problems. We also used Docker Compose to manage both the application and the database (PostgreSQL) together. This report details the steps taken to complete the project, the difficulties encountered, and the final outcome, with screenshots to provide a better understanding. The process was both educational and challenging, involving a lot of trial and error, which ultimately led to a deeper understanding of containerization, web development, and multi-container applications.

2. Introduction

In the world of software development today, containerization has become very important. It helps make sure that applications can run the same way, no matter where they are deployed. Docker is one of the tools that make containerization possible. The idea behind this project was to learn about Docker and containerization by building a task management application using Django, a popular web framework for Python, and then deploying it in a Docker container. This makes it easy to run the same application on different computers, without running into issues like missing dependencies or different operating systems.

The task management app allows users to add tasks, view a list of tasks, edit details, and delete them once they are done. This report explains the development process, using Docker to containerize the app, and connecting it with a PostgreSQL database. By using Docker, the whole project could be moved from one environment to another without having to make any changes to the setup or code. This helps developers avoid the typical issues of "it works on my machine" because the container includes everything needed for the application to run.

This project gave me an opportunity to understand the workflow of deploying a Django application in a containerized environment, providing a practical understanding of how professional development teams use these technologies to ensure consistency across different stages of development, testing, and production.

3. Project Objectives

1. Build a simple but functional task management web application using Django.
2. Understand and use containerization with Docker.
3. Set up a PostgreSQL database and connect it to the Django app.
4. Use Docker Compose to handle both the Django app and the database together in a multi-container setup.
5. Deploy the application in a containerized environment.
6. Learn how to use Docker networking and volumes to manage data and make sure the different containers can communicate with each other.
7. Develop a workflow that other developers could easily follow and replicate using Docker.

4. Intro to Containerization: Docker

4.1 Containerization Overview

Containerization is a technology that allows an application and all its dependencies to be packaged into a small, portable unit called a container. This packaging ensures that the application will work exactly the same, no matter where it is deployed, because everything it needs to run is included in the container itself. Containers are also more lightweight than traditional virtual machines since they share the operating system kernel, which makes them much more efficient and faster to start up.

One of the biggest advantages of containerization is the portability it provides. This means that developers can build an application, put it into a container, and then share that container with others, who can run it without any extra configuration or dependency issues. This greatly simplifies the development and deployment process, especially in teams. Docker is the most popular tool for managing these containers and is widely used across the industry. Containers also help maintain isolation between different parts of an application, which means that any issues in one container will not affect others, leading to more stable and secure systems overall.

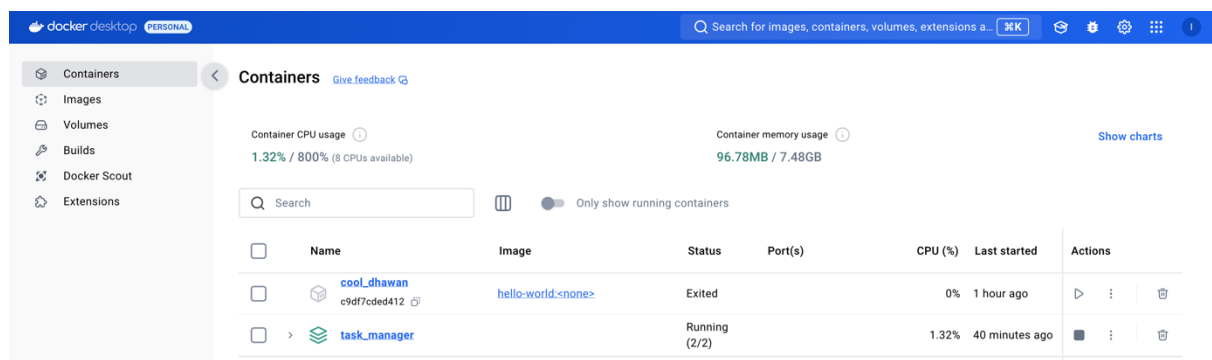
Another advantage of containerization is the ease of scalability. Since containers are lightweight, multiple containers can run on a single host machine, allowing applications to scale easily. If more computing power is needed, developers can simply add more containers. This flexibility is one of the reasons containerization has become so popular in recent years, especially for large applications that need to handle many users.

4.2 Docker Installation

To begin the project, I needed to install Docker. Here are the steps I followed:

I went to the official Docker website and downloaded Docker Desktop for macOS.

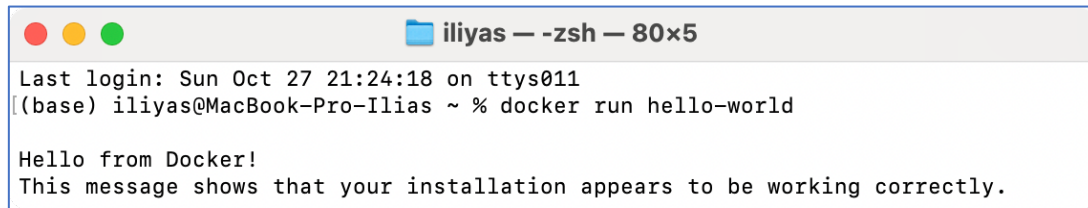
After installing Docker Desktop, I checked that it was installed properly by running `docker --version` in the terminal, which showed the current version of Docker.



I then ran a simple Docker container to verify the installation:

```
docker run hello-world
```

This command pulled a Docker image called "hello-world" and ran it, which printed a message in the terminal to show that Docker was working as expected.

A terminal window titled 'iliyas — -zsh — 80x5'. The output shows the last login time and the command 'docker run hello-world' being executed. The response is 'Hello from Docker!' followed by a confirmation message: 'This message shows that your installation appears to be working correctly.'

```
iliyas — -zsh — 80x5
Last login: Sun Oct 27 21:24:18 on ttys011
[(base) iliyas@MacBook-Pro-Ilias ~ % docker run hello-world

Hello from Docker!
This message shows that your installation appears to be working correctly.
```

This initial step was important because it confirmed that Docker was correctly set up and ready to use for the project. Ensuring that Docker was properly installed helped avoid issues later on when creating and running containers for the application.

5. Creating a Dockerfile

5.1 Dockerfile Structure

A Dockerfile is a script containing instructions for building a Docker image. In this project, I created a Dockerfile to set up the environment for the Django application. Below is the Dockerfile used for our task management application:

```
FROM python:3.9

# Set the working directory
WORKDIR /app

# Install dependencies
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# Copy the project files
COPY . .

# Expose the port
EXPOSE 8000

# Command to run the application
CMD ["python", "manage.py", "runserver", "0.0.0.0:8000"]
```

5.2 Dependencies

The Dockerfile starts by using an official Python image (FROM python:3.9). Then it sets the working directory to /app, which means all the commands will run in that directory. The next step is to copy the requirements.txt file into the container and install the dependencies listed in it, like Django. After that, all the project files are copied into the container, and finally, the command to start the server is added.

Using a Dockerfile makes sure that everyone working on the project has the same environment, with the same versions of all the software, which prevents errors that might occur because of version mismatches.

Creating a Dockerfile also simplifies the process of sharing the application with others. Once the Dockerfile is written, anyone can build the Docker image from it, which guarantees consistency in the environment. This is especially useful for development teams, as it eliminates the need to manually install dependencies or troubleshoot environment-related issues.

6. Using Docker Compose

6.1 Compose File Structure

Docker Compose is used to define and run multi-container Docker applications. For this project, I created a `docker-compose.yml` file to manage both the Django app and the PostgreSQL database.

Below is the content of the `docker-compose.yml` file:

```
version: '3.8'

services:
  web:
    build: .
    ports:
      - "8000:8000"
    volumes:
      - ./app
    depends_on:
      - db

  db:
    image: postgres
    environment:
      POSTGRES_DB: task_db
      POSTGRES_USER: user
      POSTGRES_PASSWORD: password
    volumes:
      - postgres_data:/var/lib/postgresql/data

volumes:
  postgres_data:
```

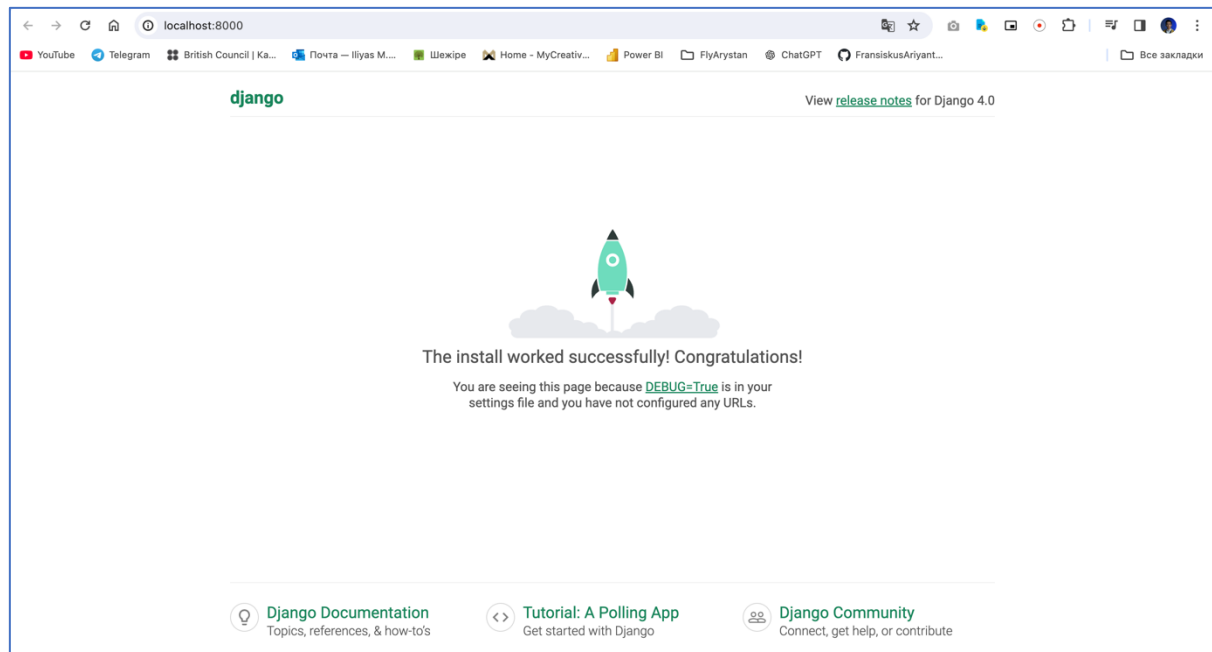
6.2 Service Definitions

The web service uses the Dockerfile to build the Django app and maps port 8000 from the container to port 8000 on my computer so that I can access the app in my browser. It also

uses volumes to make sure any changes made to the code are immediately reflected in the container. The db service uses the official PostgreSQL image to create a database. I added environment variables to set up the database name, user, and password.

Docker Compose made it easy to manage both the Django application and the database. Using `depends_on`, I made sure that the database starts before the web application to avoid any issues.

Another important feature of Docker Compose is its ability to create isolated networks for containers, ensuring that different services can communicate securely. This is particularly helpful when working on larger projects with multiple components.



7. Docker Networking and Volumes

7.1 Networking

Docker Compose automatically creates a network so that the services can communicate with each other. In this case, the web service (Django) needed to communicate with the db service (PostgreSQL database). The network created by Docker Compose makes it easy because each service can refer to the other by its name (db in this case), and Docker takes care of the networking behind the scenes.

The default network allows containers to communicate without needing extra configuration, which makes the setup process a lot easier, especially for beginners.

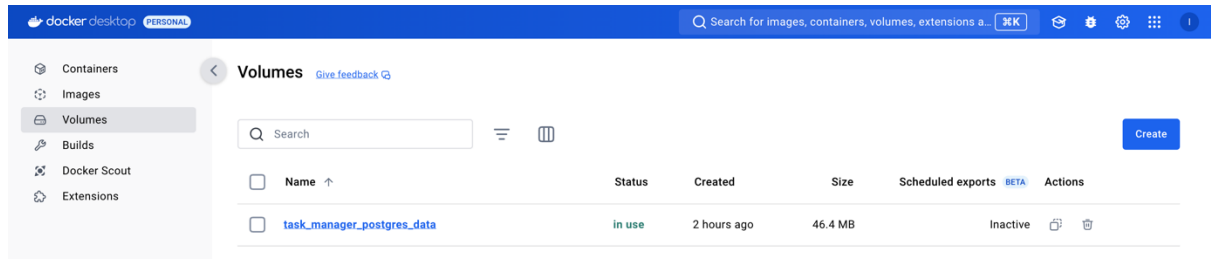
Another benefit of Docker networking is security. Each service is isolated, and only services that need to communicate with each other are allowed to do so. This helps in maintaining a secure environment, particularly in production setups where limiting access is crucial.

7.2 Volumes

In Docker Compose, I used a volume called `postgres_data` to store the data from the PostgreSQL database. The reason for using a volume is to make sure that the data is not lost

if the container is stopped or deleted. This was important because without a volume, I would lose all the tasks I added every time the container restarted.

By using volumes, I could keep the data persistent, which made it easier to work on the project without having to re-enter the data every time. Volumes also allow the database data to be easily backed up or moved, providing flexibility in case I need to migrate to a new system.



8. Django Application Setup

8.1 Project Initialization

To start the Django project, I used the following command:

```
django-admin startproject task_manager
```

This created the basic structure of the Django project. I then created a new app called 'task_manager_app' to handle all the task-related functionalities:

```
docker-compose exec web python manage.py startapp task_manager_app
```

8.2 Configuration

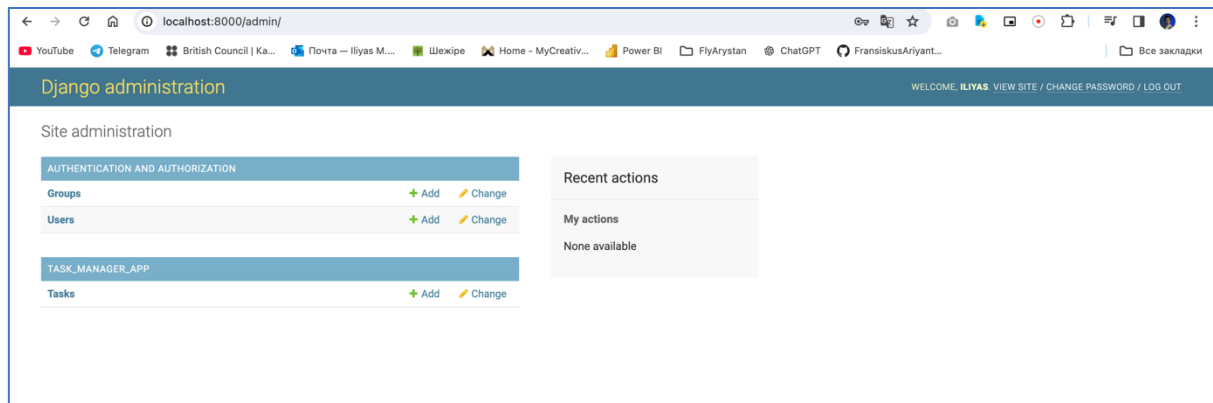
The next step was to configure the database settings in the settings.py file. Instead of using the default SQLite database, I configured it to use PostgreSQL, which was running in a Docker container.

Here is the database configuration I used:

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

The HOST is set to db, which is the name of the service in Docker Compose. This setup allows Django to connect to the PostgreSQL database container.

This configuration was crucial for ensuring that the Django application could successfully connect to the PostgreSQL database. Any issues in this part would prevent the application from storing or retrieving tasks, which is the core functionality of the app.



9. Defining Django Models

9.1 Model Creation

To represent tasks in the application, I created a model in the models.py file of task_manager_app:

```
from django.db import models

class Task(models.Model):
    title = models.CharField(max_length=200)
    description = models.TextField()
    created_at = models.DateTimeField(auto_now_add=True)
    completed = models.BooleanField(default=False)

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

The Task model has fields for the title, description, created date, and whether the task is completed. This is a simple model, but it covers all the basic functionalities needed for a task management application.

Adding a `__str__` method to the model made it easier to work with tasks in the Django admin interface, as it allowed each task to be represented by its title.

9.2 Migrations

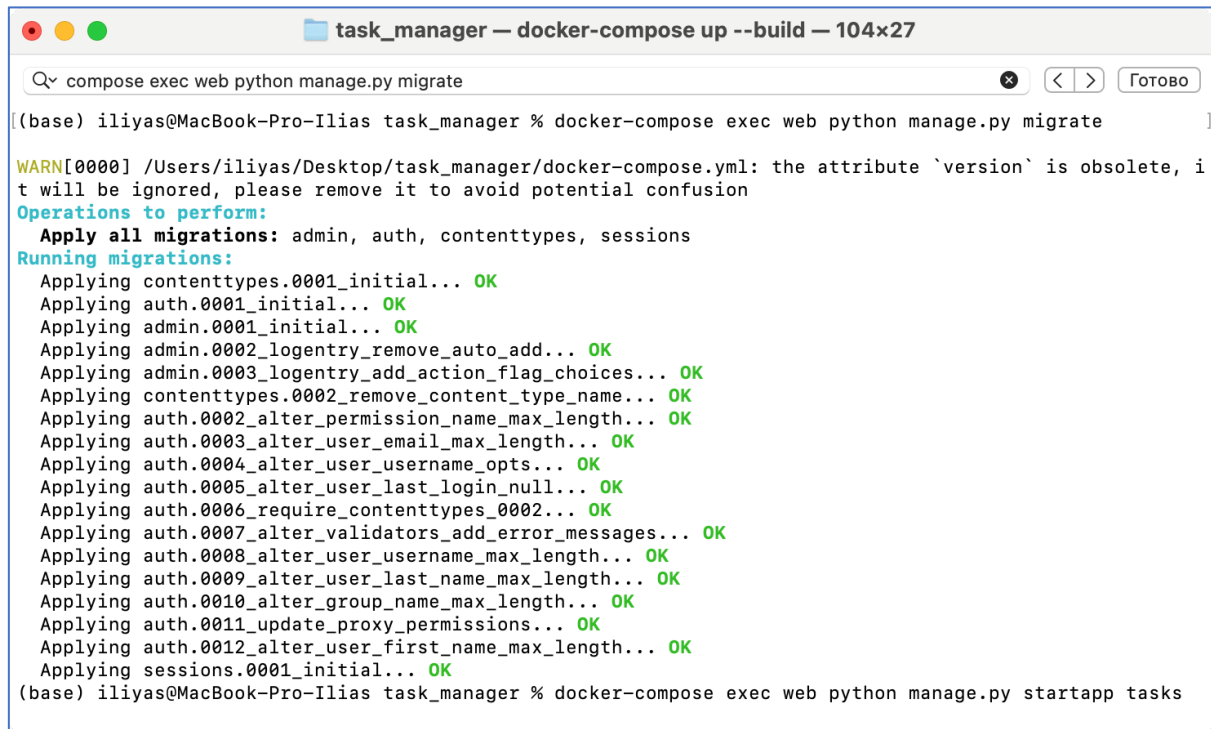
After defining the model, I used the following commands to create and apply the database migrations:

```
docker-compose exec web python manage.py makemigrations
```

```
docker-compose exec web python manage.py migrate
```

Migrations are used to create the database schema that matches the models in the code. This ensures that the database has the correct structure to store the data.

Running these commands generated the necessary migration files and applied them, creating the tables in the PostgreSQL database where the tasks would be stored.



```
task_manager — docker-compose up --build — 104x27
compose exec web python manage.py migrate

(base) iliyas@MacBook-Pro-Ilias task_manager % docker-compose exec web python manage.py migrate

WARN[0000] /Users/iliyas/Desktop/task_manager/docker-compose.yml: the attribute `version` is obsolete, it will be ignored, please remove it to avoid potential confusion
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
(base) iliyas@MacBook-Pro-Ilias task_manager % docker-compose exec web python manage.py startapp tasks
```

10. Conclusion

This project was a great learning experience in both web development and using Docker for containerization. Building a task management application helped me understand how to work with Django, including creating models and configuring a database. Docker made it easy to set up the environment, ensuring that the application could be run on different computers without needing to change the setup.

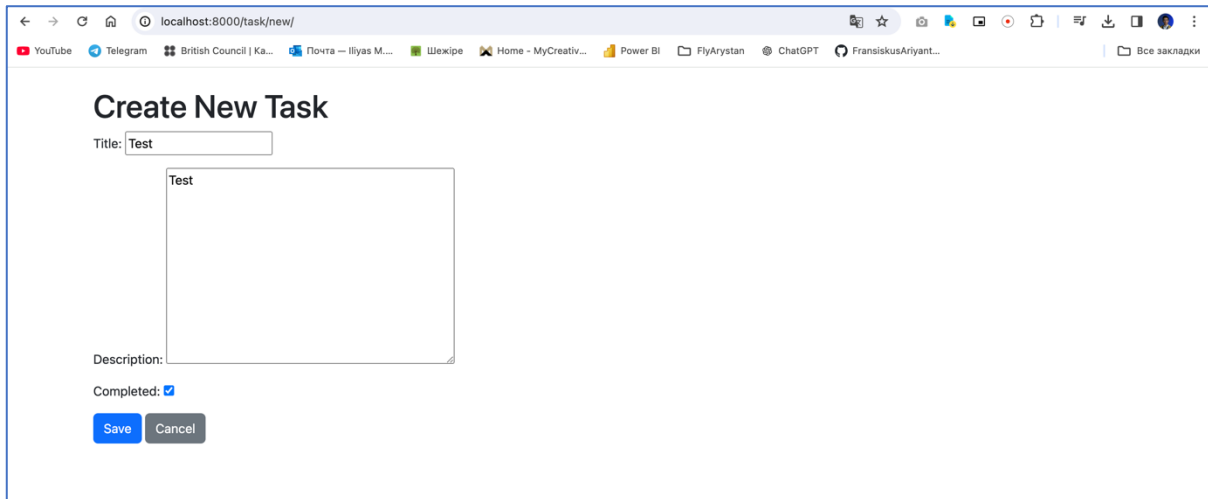
One of the biggest challenges I faced was understanding how Docker Compose handles multiple containers and getting the database connection to work correctly. There were a few times when the application couldn't connect to the database because it wasn't started yet. By using `depends_on` in the Docker Compose file, I was able to solve this problem.

Another thing I learned was the importance of volumes in Docker. By using a volume for the database, I was able to keep the data even if the container stopped. This made the development process much smoother, as I didn't have to re-enter the data each time.

Going forward, there are a few things that could be improved in the project. For example, adding user authentication so that users can only see their own tasks would be a good next step. Another improvement could be deploying the application to a cloud platform like AWS, which would make it accessible from anywhere. This would also involve setting up a

production-ready environment, including a web server like Gunicorn and using HTTPS for security.

Overall, this project gave me a good understanding of how to build a web application and how to use Docker to make the development process easier and more reliable.



localhost:8000/task/new/

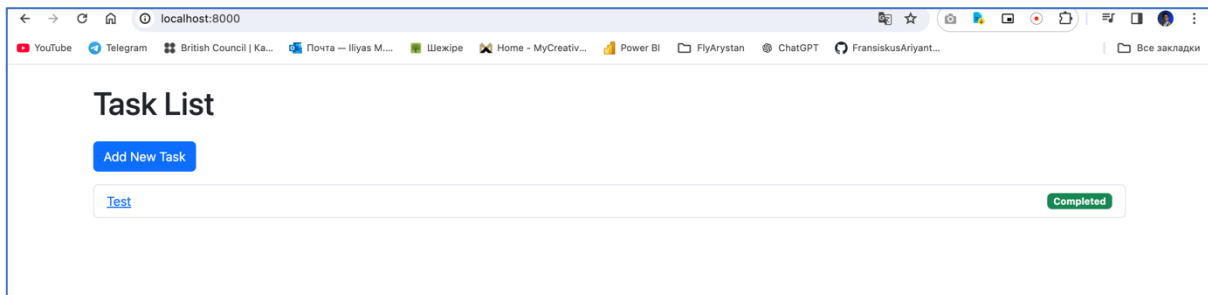
Create New Task

Title:

Description:

Test

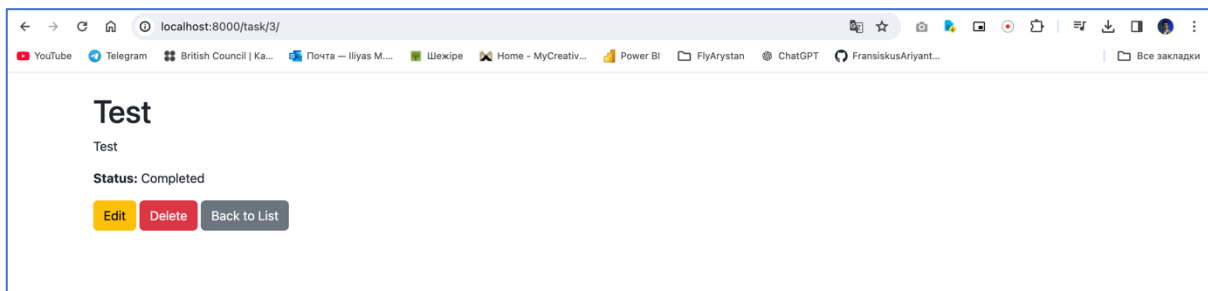
Completed: ☒



localhost:8000

Task List

Test	Completed
------	-----------



localhost:8000/task/3/

Test

Test

Status: Completed

11. References

1. Django Documentation - <https://docs.djangoproject.com/>
2. Docker Documentation - <https://docs.docker.com/>
3. PostgreSQL Documentation - <https://www.postgresql.org/docs/>
4. Docker Compose Documentation - <https://docs.docker.com/compose/>
5. Official Python Docker Image - https://hub.docker.com/_/python
6. Bootstrap Documentation - <https://getbootstrap.com/docs/>