



Web application development
Building a Task Management Application Using Django and Docker
Student: Almabekuly Aibek
Date of submission: 27.10.2024
Git: https://github.com/Aibek2201/Web_app_dev_MD.git

1. Executive Summary

This project focuses on developing a task management application using Django, deployed with Docker. The goal is to create a CRUD-based task manager that ensures consistent development and production environments using Docker. Key technologies include Django, Docker, Docker Compose, and PostgreSQL. This project helps understand containerization's role in modern development.

2. Introduction

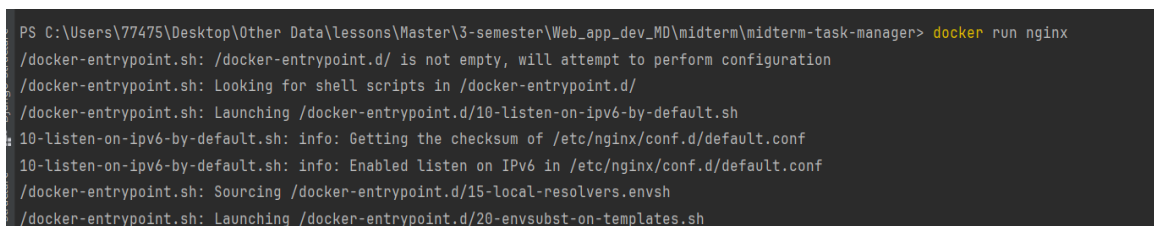
Containerization allows applications to run consistently across different environments. Docker, a popular containerization platform, provides isolation and portability, making deployment straightforward. This project explores these concepts by building a task management application with Django, leveraging Docker for an efficient deployment process.

3. Project Objectives

- 1 Develop a web-based task management application using Django.
- 2 Use Docker to containerize the application, ensuring consistency.
- 3 Implement database management with PostgreSQL.
- 4 Explore Docker networking, volumes, and multi-container setups with Docker Compose.

4. Intro to Containerization: Docker

Containers isolate applications, ensuring they run the same regardless of the environment. This prevents "it works on my machine" issues. Docker Installation: Docker was installed using official documentation. A simple NGINX container was run to verify the setup.



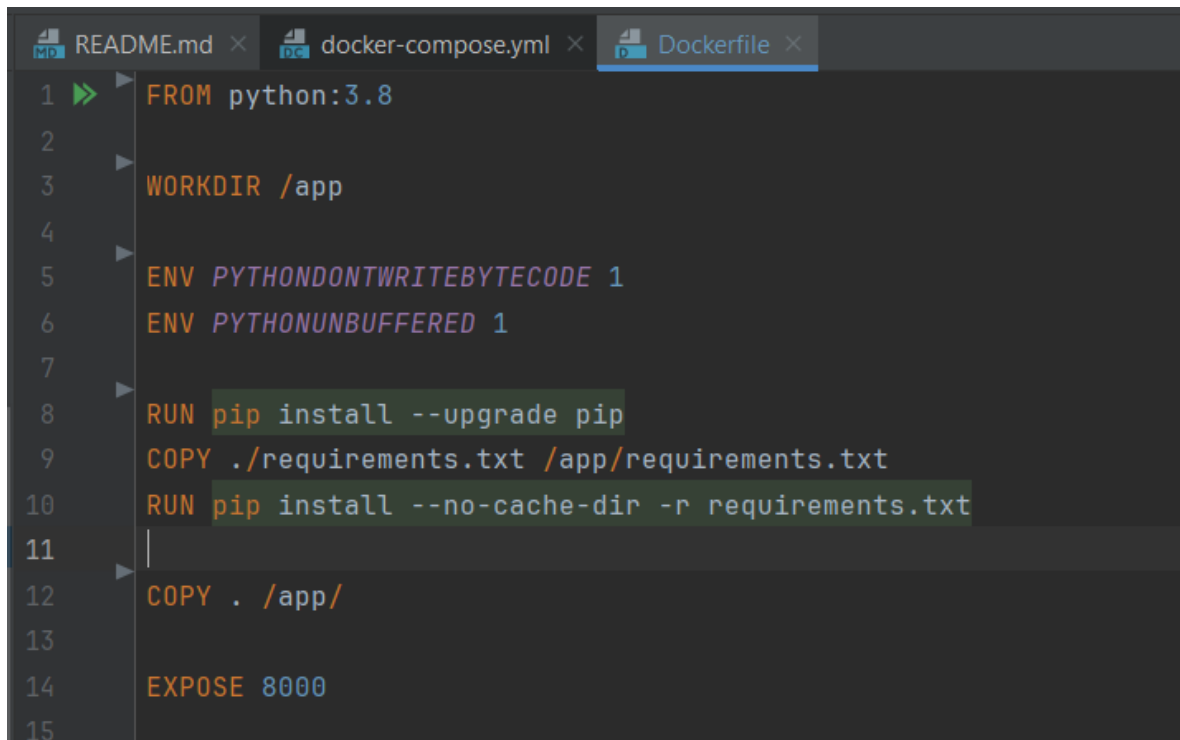
```
PS C:\Users\77475\Desktop\Other Data\lessons\Master\3-semester\Web_app_dev_MD\midterm\midterm-task-manager> docker run nginx
/docker-entrypoint.sh: /docker-entrypoint.d/ is not empty, will attempt to perform configuration
/docker-entrypoint.sh: Looking for shell scripts in /docker-entrypoint.d/
/docker-entrypoint.sh: Launching /docker-entrypoint.d/10-listen-on-ipv6-by-default.sh
10-listen-on-ipv6-by-default.sh: info: Getting the checksum of /etc/nginx/conf.d/default.conf
10-listen-on-ipv6-by-default.sh: info: Enabled listen on IPv6 in /etc/nginx/conf.d/default.conf
/docker-entrypoint.sh: Sourcing /docker-entrypoint.d/15-local-resolvers.envsh
/docker-entrypoint.sh: Launching /docker-entrypoint.d/20-envsubst-on-templates.sh
```

Figure 1. Simple container using nginx

5. Creating a Dockerfile

The Dockerfile defines the environment for the Django application, specifying the Python version, dependencies, and the commands required to run the app. It ensures that the application is isolated and runs consistently regardless of the underlying system. The requirements.txt file lists necessary packages like Django and database libraries. These are installed during the Docker image build process to ensure all required modules are available in the container environment.

Now We can see our Dockerfile here:



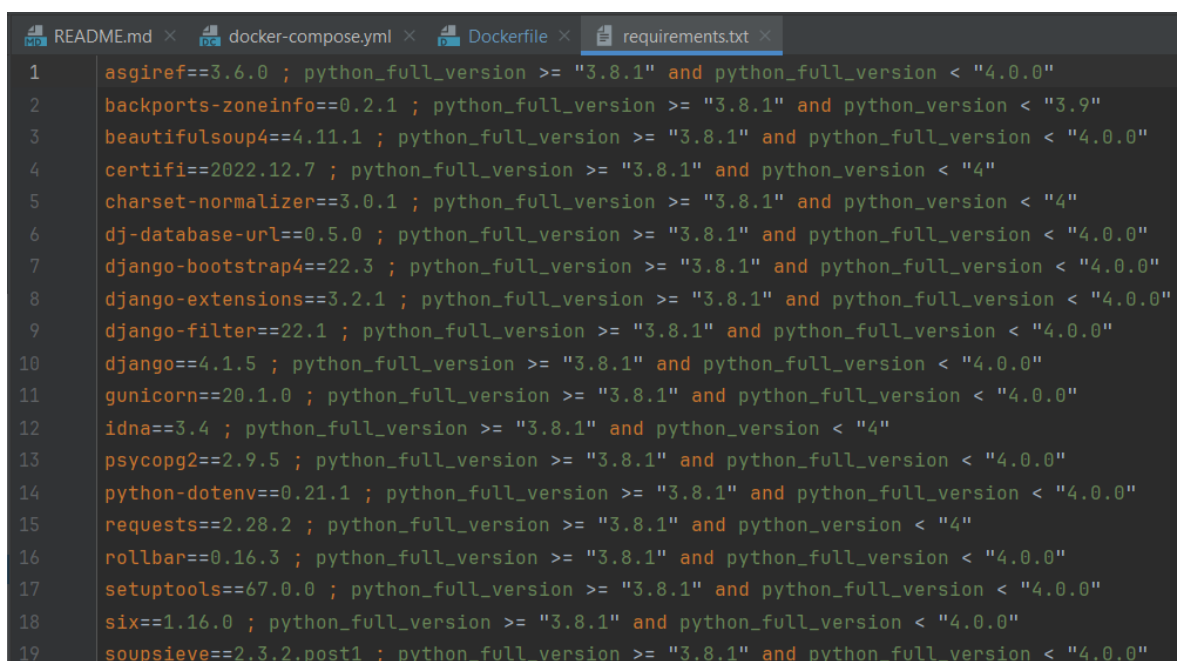
```

1 FROM python:3.8
2
3 WORKDIR /app
4
5 ENV PYTHONDONTWRITEBYTECODE 1
6 ENV PYTHONUNBUFFERED 1
7
8 RUN pip install --upgrade pip
9 COPY ./requirements.txt /app/requirements.txt
10 RUN pip install --no-cache-dir -r requirements.txt
11
12 COPY . /app/
13
14 EXPOSE 8000
15

```

Figure 2. Dockerfile's structure

As we can see here we did some commands inside of dockerfile like `pip install -r requirements.txt`. Using this command we can download require packages inside our container. In the next figure we see require packages and modules for our application.



```

1 asgiref==3.6.0 ; python_full_version >= "3.8.1" and python_full_version < "4.0.0"
2 backports-zoneinfo==0.2.1 ; python_full_version >= "3.8.1" and python_version < "3.9"
3 beautifulsoup4==4.11.1 ; python_full_version >= "3.8.1" and python_full_version < "4.0.0"
4 certifi==2022.12.7 ; python_full_version >= "3.8.1" and python_version < "4"
5 charset-normalizer==3.0.1 ; python_full_version >= "3.8.1" and python_version < "4"
6 dj-database-url==0.5.0 ; python_full_version >= "3.8.1" and python_full_version < "4.0.0"
7 django-bootstrap4==22.3 ; python_full_version >= "3.8.1" and python_full_version < "4.0.0"
8 django-extensions==3.2.1 ; python_full_version >= "3.8.1" and python_full_version < "4.0.0"
9 django-filter==22.1 ; python_full_version >= "3.8.1" and python_full_version < "4.0.0"
10 django==4.1.5 ; python_full_version >= "3.8.1" and python_full_version < "4.0.0"
11 gunicorn==20.1.0 ; python_full_version >= "3.8.1" and python_full_version < "4.0.0"
12 idna==3.4 ; python_full_version >= "3.8.1" and python_version < "4"
13 psycopg2==2.9.5 ; python_full_version >= "3.8.1" and python_full_version < "4.0.0"
14 python-dotenv==0.21.1 ; python_full_version >= "3.8.1" and python_full_version < "4.0.0"
15 requests==2.28.2 ; python_full_version >= "3.8.1" and python_version < "4"
16 rollbar==0.16.3 ; python_full_version >= "3.8.1" and python_full_version < "4.0.0"
17 setuptools==67.0.0 ; python_full_version >= "3.8.1" and python_full_version < "4.0.0"
18 six==1.16.0 ; python_full_version >= "3.8.1" and python_full_version < "4.0.0"
19 soupsieve==2.3.2.post1 ; python_full_version >= "3.8.1" and python_full_version < "4.0.0"

```

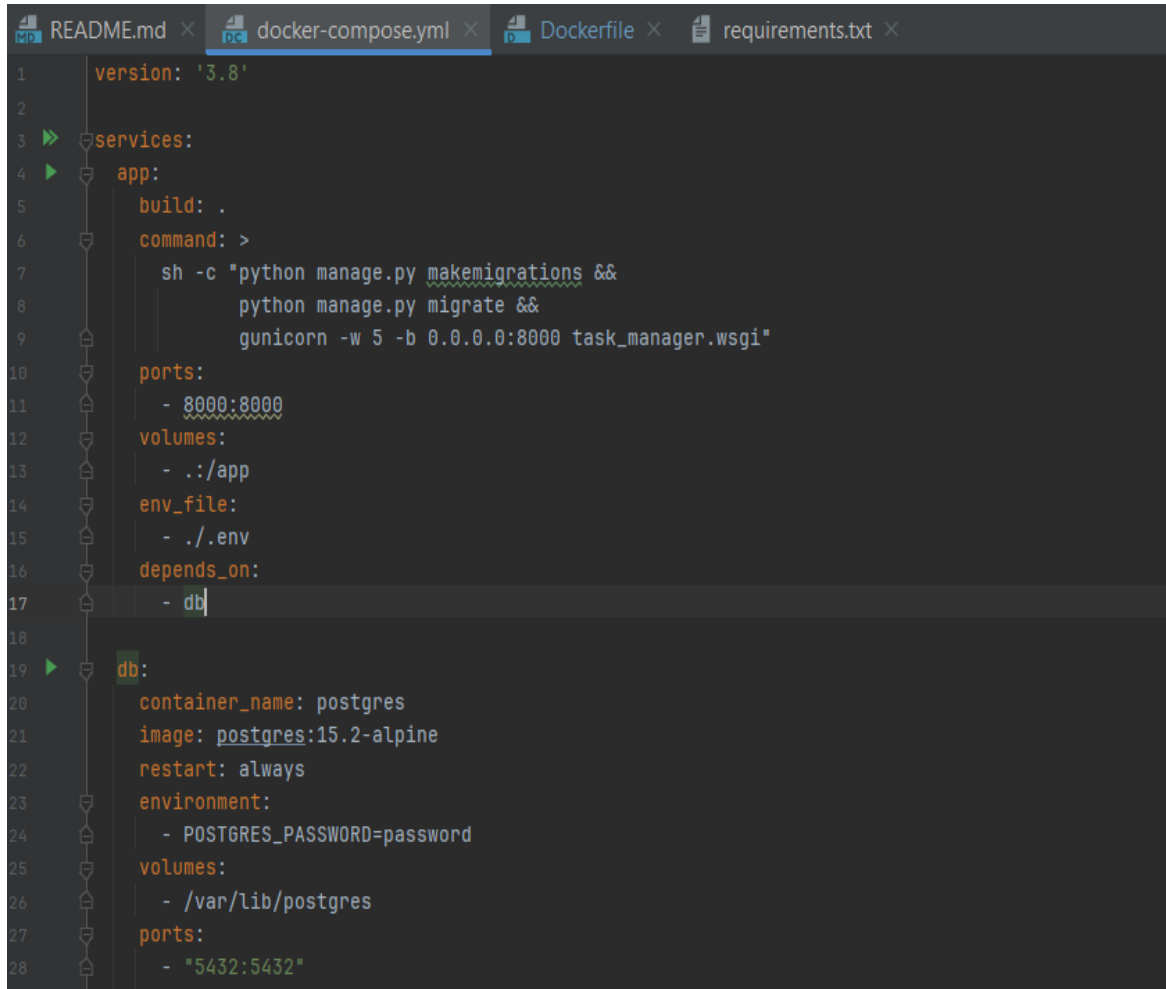
Figure 3. Requirements.txt file

6. Using Docker Compose

Docker Compose is used to manage multi-container setups, simplifying the process of running the Django application alongside a PostgreSQL database. By defining both services in a single `docker-compose.yml` file, it becomes easier to start, stop, and manage the entire environment with just a few commands.

The docker-compose.yml file defines two primary services—web for the Django application and db for the PostgreSQL database. The web service builds the Django app from the Dockerfile and exposes port 8000, allowing access to the app. The db service uses the official PostgreSQL image and sets up the database with specified environment variables like database name, user, and password.

Our docker-compose file looks like this:



```
1  version: '3.8'
2
3  services:
4    app:
5      build: .
6      command: >
7        sh -c "python manage.py makemigrations &&
8          python manage.py migrate &&
9          gunicorn -w 5 -b 0.0.0.0:8000 task_manager.wsgi"
10     ports:
11       - 8000:8000
12     volumes:
13       - ./app
14     env_file:
15       - ./.env
16     depends_on:
17       - db
18
19   db:
20     container_name: postgres
21     image: postgres:15.2-alpine
22     restart: always
23     environment:
24       - POSTGRES_PASSWORD=password
25     volumes:
26       - /var/lib/postgres
27     ports:
28       - "5432:5432"
```

Figure 4. Structure of docker-compose file

7. Docker Networking and Volumes

Docker networking allows the web and db services to communicate seamlessly within the same network. In the docker-compose.yml, the two services are automatically placed on a bridge network, allowing Django to connect to PostgreSQL using the service name db as the host. This simplifies database connection settings, as no external configuration is needed for the containers to find each other.

Docker volumes are used to persist data from the PostgreSQL database. By using the postgres_data volume, database files are stored outside of the container. This means that even if the database container is stopped, the data remains intact and can be reused when the container is restarted. Volumes are crucial for preventing data loss and ensuring data integrity during development and deployment. In this project I used /var/lib/midterm directory inside of docker container.

8. Django Application Setup

The Django project was initialized using `django-admin startproject`, creating a structured folder containing core files for the application. A new Django app named `tasks` was created to handle task management features, such as creating, viewing, updating, and deleting tasks.

The Django settings were adjusted to integrate with the PostgreSQL database defined in the `docker-compose.yml`. This includes specifying database credentials in `settings.py`, setting up `DATABASES` to connect to the db service, and ensuring that `ALLOWED_HOSTS` includes the Docker network settings. Additionally, the static and media files were configured to work seamlessly with Docker volumes for easier management during development.

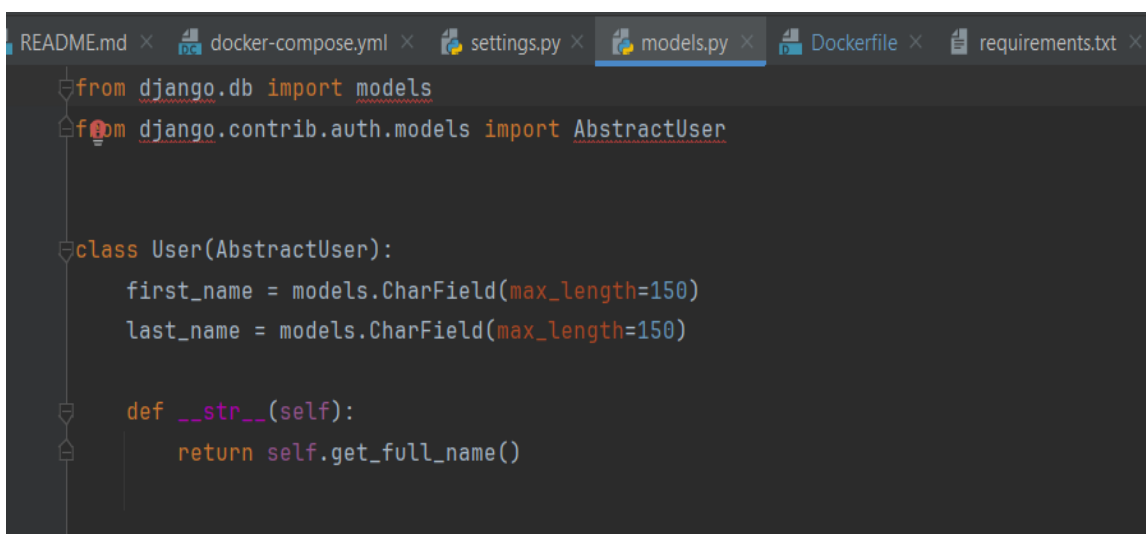
After creating Django project we should add database for working with data. In this project I used default SQLite3 database and settings we can see here.



```
DATABASES = {  
    'default': dj_database_url.config(  
        default='sqlite:///db.sqlite3',  
        conn_max_age=600,  
    )  
}
```

Figure 5. Database settings

So after initializing database we must create some models using Django models for work with data. As we know Django can send database models using custom ORM model. The main purpose of Object Relational Mapping is to convert python code to SQL. So using python code we can structure SQL database.



```
from django.db import models  
from django.contrib.auth.models import AbstractUser  
  
class User(AbstractUser):  
    first_name = models.CharField(max_length=150)  
    last_name = models.CharField(max_length=150)  
  
    def __str__(self):  
        return self.get_full_name()
```

Figure 6. User models

9. Defining Django Models

The core of the task management application revolves around defining models that represent database tables. The Task model includes fields like title, description, created_at, and completed, which map to columns in the database. Each field is given a specific data type, such as CharField for titles and BooleanField for the completion status.

Django's migration system automatically generates database schemas based on the models. After defining the Task model, migrations were created using `python manage.py makemigrations` and applied with `python manage.py migrate`. This process ensures that the database structure aligns with the Django models, allowing smooth updates to the schema as the project evolves. In the next code we see example model in Django.

```
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 main page of our project looks like this. Task manager application must be simple for using.

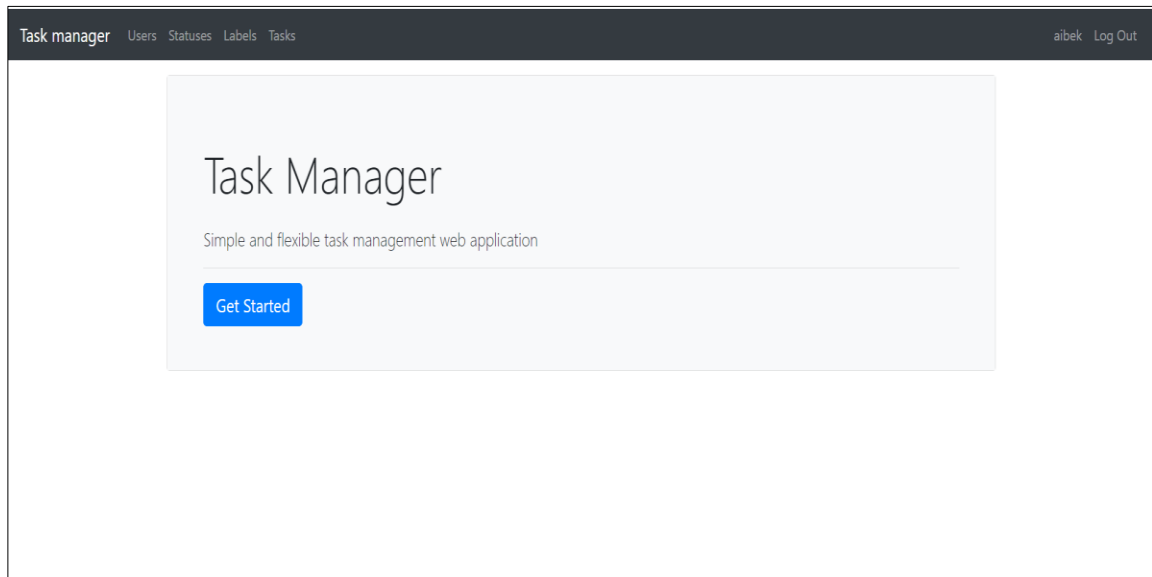


Figure 7. Main page of task manager

To use this project for each user We made login simple login page. To log in to individual page each user should use their login and password, which created in the beginning.

Login

Username

aibek

Password

...

Enter

Figure 8. Login page

After that users log in to task manager page. When we sign in to the account first time it will be empty. So if we need to create some tasks, we can push button create task.

Tasks

Create task

Status Executor Label ☐ Only own tasks

ID	Name	Status	Author	Executor	Creation date
----	------	--------	--------	----------	---------------

Figure 9. Page with tasks

To create tasks firstly we must create statuses to the tasks. Here we can see some created statuses for tasks.

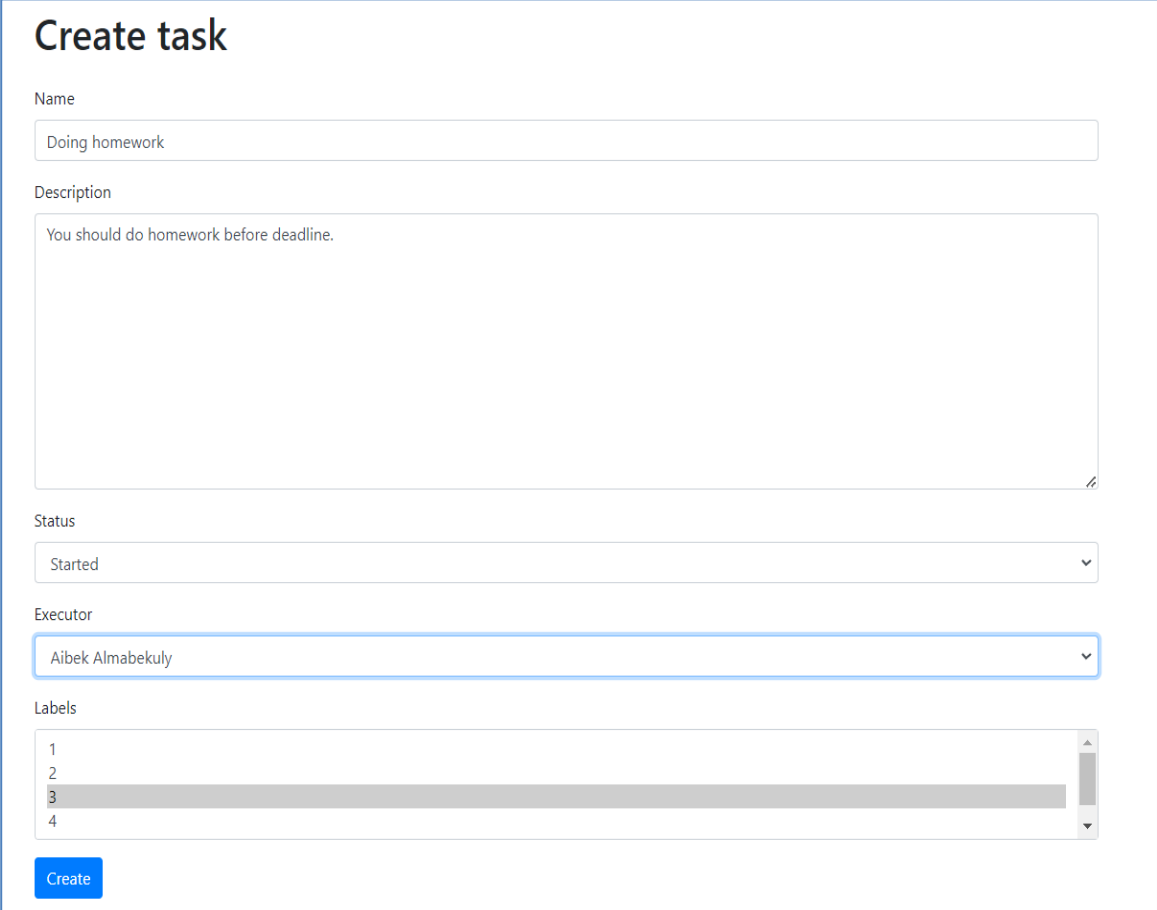
Statuses

Create status

ID	Name	Creation date	
1	In progress	27.10.2024 15:16	Update Delete
2	Done	27.10.2024 15:16	Update Delete
3	Started	27.10.2024 15:16	Update Delete
4	Жасауға ойланып отырмын	27.10.2024 15:17	Update Delete

Figure 10. List of statuses

After creating dependencies we can create tasks for users. In the creating-page we see some cells.

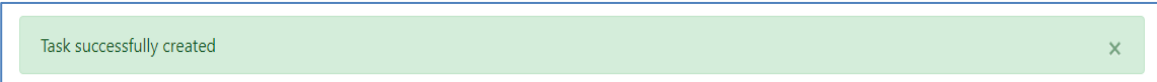


The screenshot shows a web form titled "Create task". It contains several input fields and a button:

- Name:** A text input field containing "Doing homework".
- Description:** A larger text area containing "You should do homework before deadline." with a small icon in the bottom right corner.
- Status:** A dropdown menu with "Started" selected.
- Executor:** A dropdown menu with "Aibek Almabekuly" selected.
- Labels:** A list of four labels (1, 2, 3, 4) with a scrollbar. Label 3 is highlighted.
- Create:** A blue button at the bottom left.

Figure 11. Page for creating tasks

After creating task we see message about our task created successfully or something went wrong.



The screenshot shows a green notification bar with the text "Task successfully created" and a close button (X) on the right.

Figure 12. Message about creating task.

So We have successfully created several tasks for today, each with its own status, author, and assigned executor. In this page, users can view the complete list of tasks, allowing for easy tracking and management of each item. The page features a filter system, enabling users to sort tasks by their status, executor, or specific labels, and even limit the view to only their own tasks for a more personalized experience. Additionally, while creating a new task, users have the option to assign specific executors, ensuring that responsibilities are clearly defined. Each task is displayed with its ID, name, current status, author, executor, and creation date, making the management process more efficient and organized. Moreover, the options to update or delete tasks directly from this interface add flexibility and control for the user.

Tasks

Create task

Status Executor Label ☐ Only own tasks

ID	Name	Status	Author	Executor	Creation date	
1	Doing homework	Started	Aibek Almabekuly	Aibek Almabekuly	27.10.2024 15:25	Update Delete
2	Work	Done	Aibek Almabekuly		27.10.2024 15:26	Update Delete
3	Doing something	Done	Aibek Almabekuly	Aibek Almabekuly	27.10.2024 15:27	Update Delete

Figure 13. List of task

This list of all tasks. For sort tasks which executor is me, we can use sort options in top of page. I put executor Aibek, and result we see here.

Tasks

Create task

Status ✓ Executor ✓ Label ✓ ☐ Only own tasks

ID	Name	Status	Author	Executor	Creation date	
1	Doing homework	Started	Aibek Almabekuly	Aibek Almabekuly	27.10.2024 15:25	Update Delete
3	Doing something	Done	Aibek Almabekuly	Aibek Almabekuly	27.10.2024 15:27	Update Delete

Figure 14. My tasks

If we need to update some data in tasks, we can push update button in the right side of tasks. And we will see next page again.

Task change

Name

Description

Status

Executor

Labels

Figure 15. Changing page

Architecture of our project looks like this:

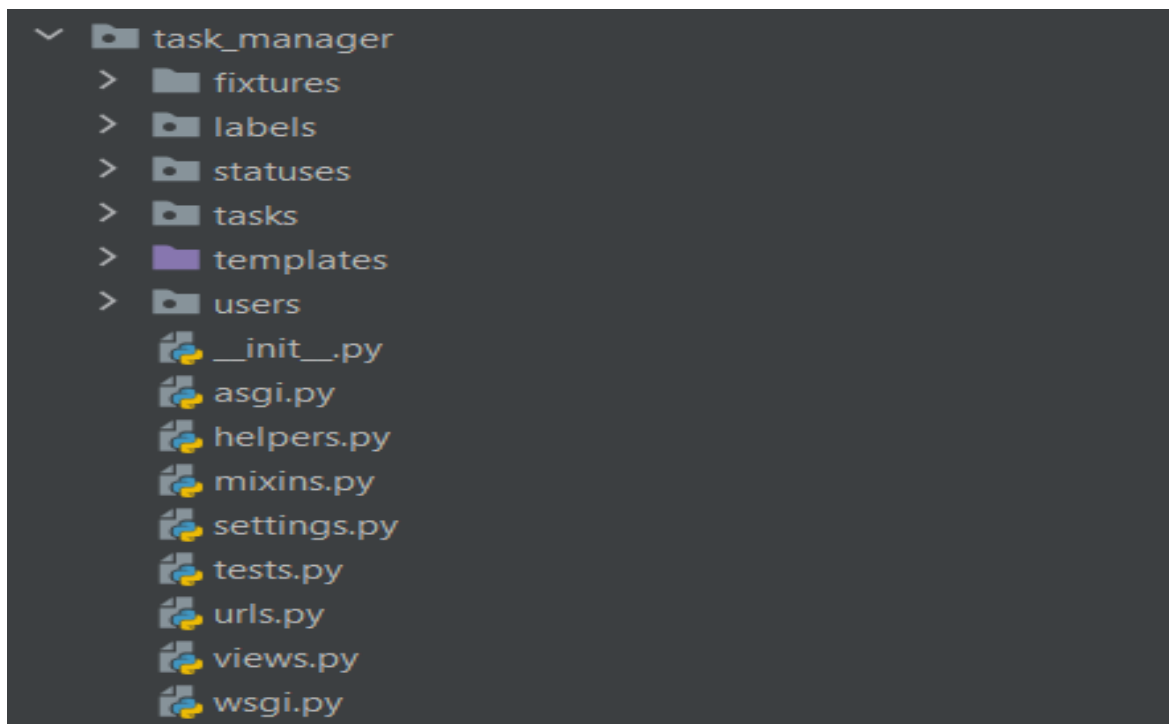


Figure 16. Architecture of our code

To organize urls for this project we created urls.py file and put all url addresses here. In the next figure see it.

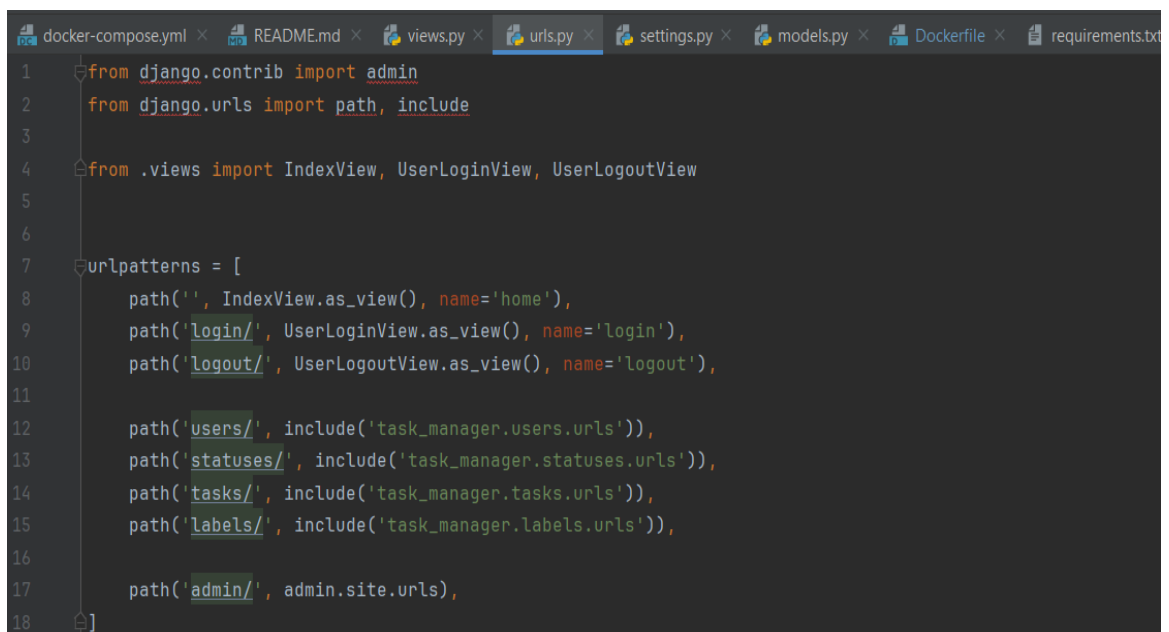


Figure 17. URL addresses of our project

10. Conclusion

The task management application project demonstrated the power of using Docker for containerization in web development. By combining Django's robust framework with Docker's containerization capabilities, the project achieved a streamlined and consistent deployment process. Docker Compose simplified managing multiple services, while Docker volumes ensured that database data was preserved across sessions. This approach significantly enhanced the development workflow, making the application more portable and resilient across different environments.

11. References

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