

**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**

Almaty, 2024

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.

1. **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.

1. **Project Objectives**
2. Develop a web-based task management application using Django.
3. Use Docker to containerize the application, ensuring consistency.
4. Implement database management with PostgreSQL.
5. Explore Docker networking, volumes, and multi-container setups with Docker Compose.
6. **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.

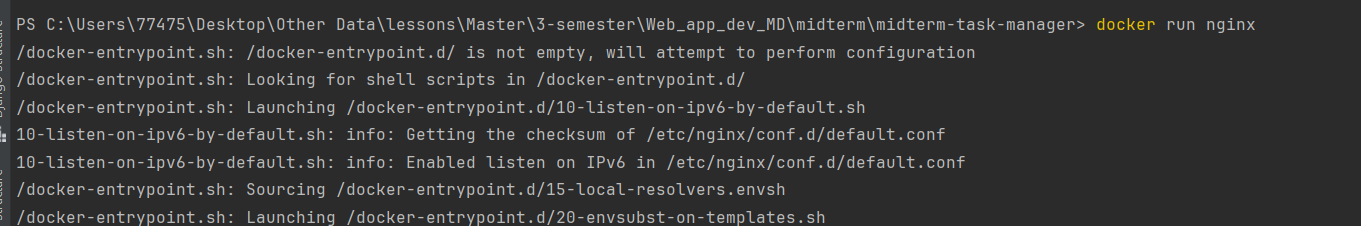


Figure 1. Simple container using nginx

1. **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:

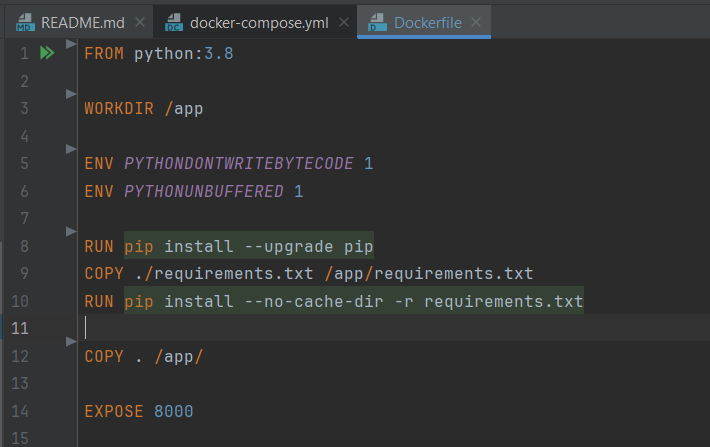


Figure 2. Dockerfile’s structure

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.

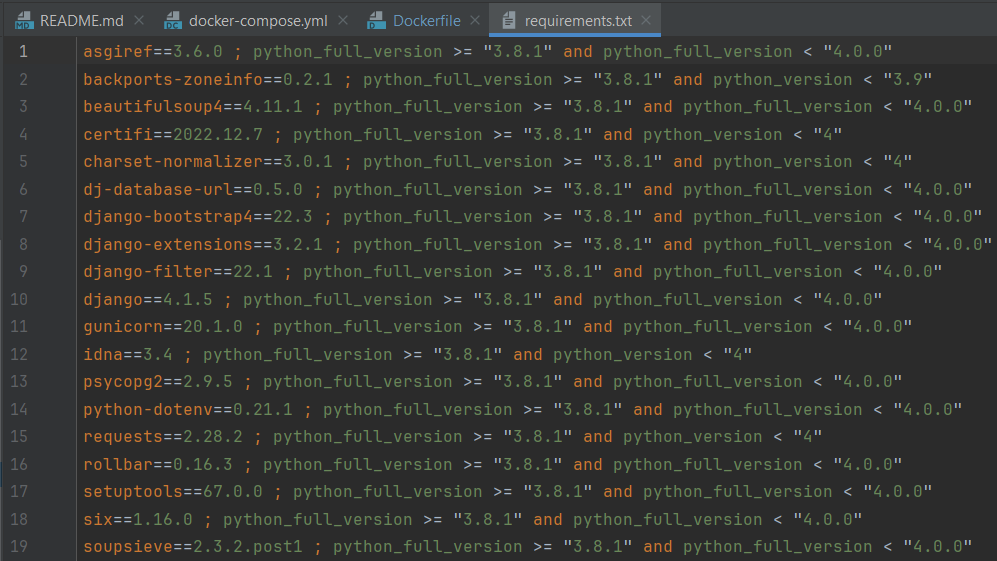


Figure 3. Requirements.txt file

1. **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:

Изображение выглядит как текст, снимок экрана, программное обеспечение, Мультимедийное программное обеспечение

Автоматически созданное описание

Figure 4. Structure of docker-compose file

1. **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.

1. **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.



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.

Изображение выглядит как текст, снимок экрана, программное обеспечение, Мультимедийное программное обеспечение

Автоматически созданное описание

Figure 6. User models

1. **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.

Изображение выглядит как текст, снимок экрана, программное обеспечение, веб-страница

Автоматически созданное описание

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.



Figure 8. Login page

After that users 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 creare task.

Изображение выглядит как снимок экрана, текст, программное обеспечение

Автоматически созданное описание

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.

Изображение выглядит как текст, снимок экрана, программное обеспечение, число

Автоматически созданное описание

Figure 10. List of statuses

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

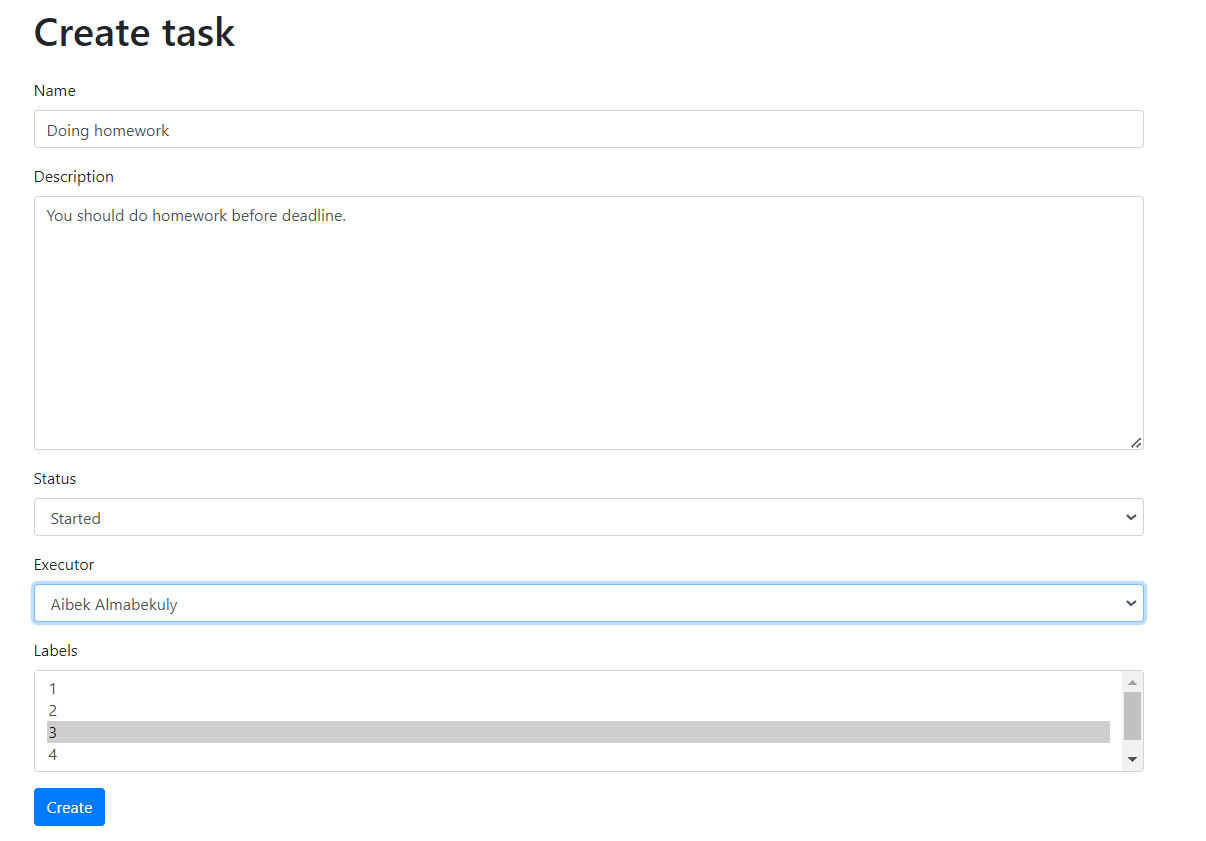


Figure 11. Page for creating tasks

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

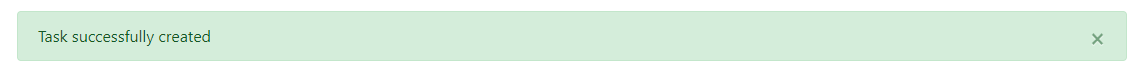


Figure 12. Message about creating task.

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.

Изображение выглядит как текст, снимок экрана, программное обеспечение, число

Автоматически созданное описание

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.

Изображение выглядит как текст, снимок экрана, программное обеспечение, веб-страница

Автоматически созданное описание

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.

Изображение выглядит как текст, снимок экрана, линия, число

Автоматически созданное описание

Figure 15. Changing page

Architecture of our project looks like this:

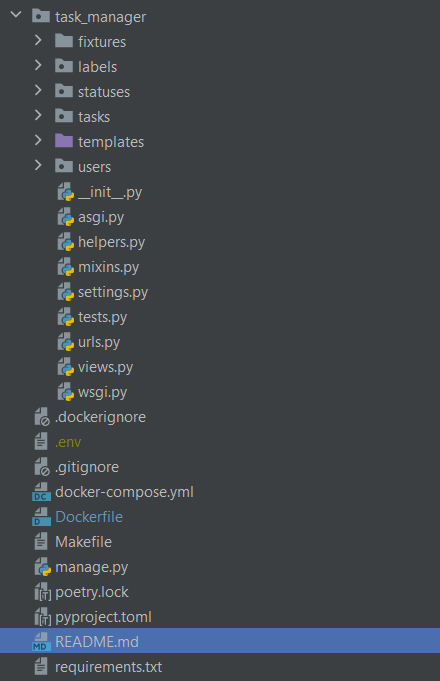


Figure 16. Architecture of our code

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

Изображение выглядит как текст, снимок экрана, программное обеспечение, Мультимедийное программное обеспечение

Автоматически созданное описание

Figure 17. URL addresses of our project

1. **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.

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