Web development
Midterm Project: Building a Task Management Application Using Django and Docker
Bitanov Assanali (23MD0392)
27.10.2024

**Table of Contents**

Executive Summary

This project involves developing a task management application using Django and deploying it in a Docker container. The project will cover the fundamentals of containerization, Docker configuration, and Django model creation. The app will let users create, view, update, and delete tasks. Docker is used to make sure everything works the same in both development and production. In this project, we will learn how to use Docker to create containers, set up Docker configuration files, and connect Django with a database. The task management app will have several pages, like a page for listing tasks, creating new tasks, editing tasks, and deleting tasks. We will also explore how to set up Django models, write migrations, and connect the app with the database inside the container. By the end of the project, we will have a working web app that can manage tasks, with both Django and Docker working together to create a solid environment for development. The project also focuses on understanding containerization and how Docker makes the app more reliable across different systems. We will practice skills like writing Python code, setting up Docker services, and managing database migrations. This project is a great way to learn the basics of Django, Docker, and how to build a web app that can manage data and tasks efficiently.

## Introduction

In this midterm project, we will practice our development skills by building a simple task management app. This app will have basic pages like Task List, Create Task, Edit Task, and Delete Task. We are going to use Django as the framework to build the app, and PostgreSQL as the database. Both Django and PostgreSQL will run inside Docker containers to make sure everything works the same, no matter where we run it.

The project will teach us important topics like how to use Docker, how to write a Dockerfile, and how to use Docker Compose. Docker allows us to package everything the app needs so it can run on any computer. We will also focus on how to create and manage tasks in the app using Django models. Each task will have fields like title, description, and date, and we will use forms to add or edit these tasks.

We will learn how to set up a working environment where the app and the database talk to each other inside containers. This means we can develop the app in a way that is clean, consistent, and easy to manage. The app's pages will allow users to see a list of tasks, create new ones, edit existing tasks, and delete tasks that are no longer needed.

By the end of the project, we will have a working web app with a solid foundation in containerization and web development. We will understand how to create models in Django and how to make sure the database stores all the tasks correctly. Plus, we will see how Docker helps developers work together more easily by creating the same environment for everyone.

Learning to use Docker and Django together will help us build more reliable applications in the future. This project will show us how these tools can make development smoother, and help us become better at building apps that can work in different environments without any problems.

## Project Objectives

1. Develop a functional task management web application using Django.
2. Practice with Docker, including containerization, networking, and volumes.
3. Set up and configure a PostgreSQL database inside a Docker container.
4. Create and manage Django models to store and handle task data.
5. Ensure the application is portable and works consistently across different environments using Docker Compose.

## Intro to Containerization: Docker

Containerization is a method of virtualization that allows you to isolate and package an application and its dependencies into a standardized unit called a container. Docker provides a platform for building, shipping, and running containers, enabling developers to create lightweight, portable, and self-sufficient environments [1].

To install Docker we download the Docker Desktop app from the official page [2]. After the installation, we open the app. The Docker Desktop is shown in Figure 1.
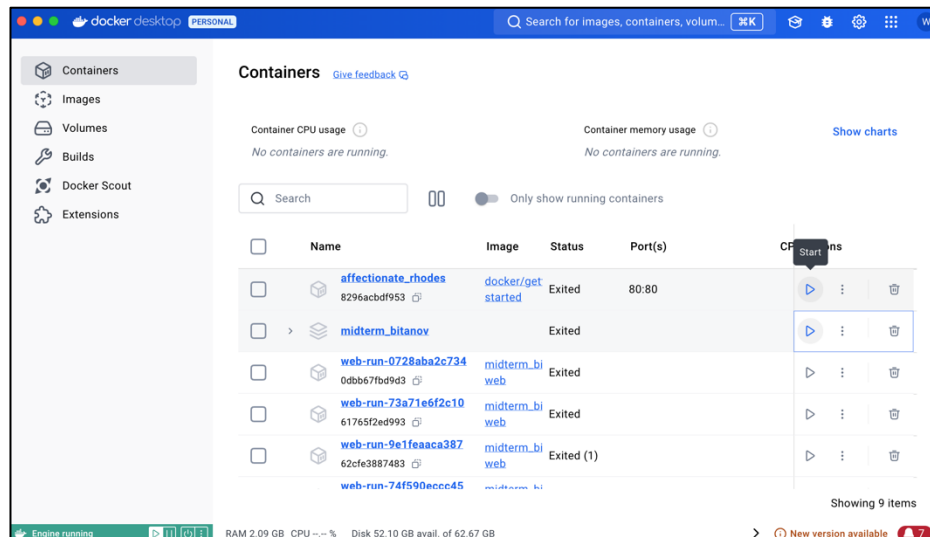


Figure 1. The Docker Desktop app

There we can see a bunch of containers. They are stored in the local machine. To launch any of it we press the Start button. For example, we launch a container with the docker/getting-started image. The page is shown in Figure 2.
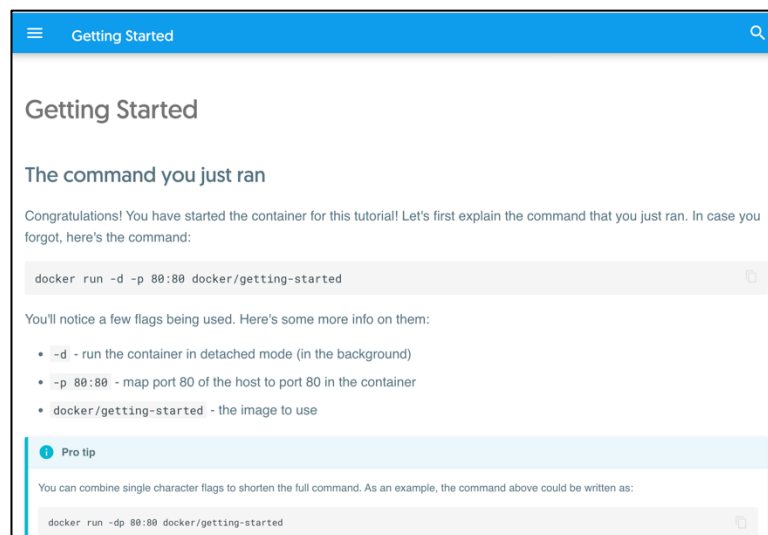


Figure 2. The Docker Getting Started page running at http://localhost/tutorial/

Creating a Dockerfile

Dockerfiles is a text file that list instructions for the Docker daemon to follow when building a container image [3]. In the file we define the version of python. In our case it is python 3.9. The Dockerfile is shown in Figure 2.

```
Dockerfile > ...
         Асанали Битанов, 32 minutes ago | 1 author (Асанали Битанов)
1    FROM python:3.9
2    WORKDIR /app
3    COPY . /app
4    RUN pip install --no-cache-dir -r requirements.txt
5    EXPOSE 8000
6    ENV PYTHONDONTWRITEBYTECODE 1
7    ENV PYTHONUNBUFFERED 1
8    CMD ["python", "manage.py", "runserver", "0.0.0.0:8000"]
```

Figure 3. The Dockerfile

Next, we define the app as workdir. This sets /app as the working directory inside the container. All of the future commands will be executed inside this directory. Then, we add all the files from the project into the /app directory by using ADD . /app command. Next, we write the command that will be executed during the image build process. It runs a command inside the container during the image creation and commits the results to the new image layer. The command will install all the necessary dependencies for the project to work. Then we define EXPOSE 8000, which declares the port that the app will run on. Lastly, we define the command that will be executed when the container will run. The command runs the server on the localhost address.

Using Docker Compose

Docker Compose allows us to create a multi-layer container. We can manage services like Django web application and the database. In our case it is Postgresql. The code of this file is shown in Figure 4.

```
docker-compose.yml > ...
    docker-compose.yml - The Compose specification establishes a standard for th
1   version: '3'
2
3   services:
4     web:
5       build: .
6       command: python manage.py runserver 0.0.0.0:8000
7       volumes:
8         - .:/app
9       ports:
10        - "8000:8000"
11      depends_on:
12        - db
13
14    db:
15      image: postgres:latest
16      volumes:
17        - postgres_data:/var/lib/postgresql/data
18      environment:
19        POSTGRES_DB: midterm
20        POSTGRES_USER: midterm_user
21        POSTGRES_PASSWORD: midterm_password
22      ports:
23        - "5432:5432"
24
25  volumes:
26    postgres_data:
```

Figure 4. The docker-compose

Here we define the version of the Docker compose file format. In our case it is version 3. Then we define web service, build . tells that the file has to use Dockerfile to build the image. The volumes section mounts the current directory on the container's /app directory. It allows the container to access and use the project files from the host machine in real time. This enables live editing and debugging of the application while it's running in the container. The next two lines of code maps port 8000 of the host machine to port 8000 inside the container, allowing access to the Django development server via http://localhost:8000 on the host.

Next the db section. There we define the environment variables, such as the name of the database, username and password. The named volume postgres_data is listed at the bottom of the file. It saves PostgreSQL data outside of the container, so the data stays safe even if the container is restarted or rebuilt.

Docker Networking and Volumes

Docker networking lets containers talk to each other using an internal network. In this project, Docker Compose automatically creates a default network so the web and database containers can connect. The depends_on key makes sure the database container starts before the web container. The web container can find the database by using the service name db as the hostname. The running container is shown in Figure 5.
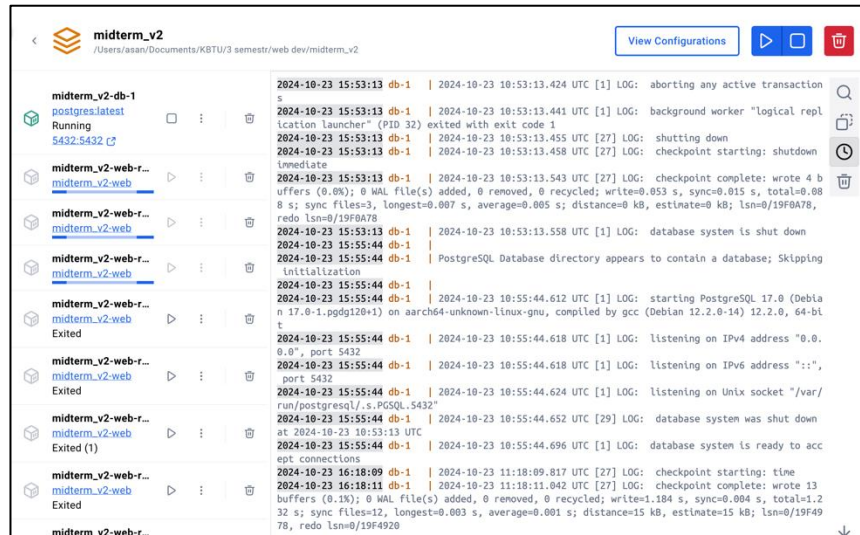


Figure 5. The Docker Container

Volumes are used to save data outside the container so it doesn't get lost when containers restart. In this project, the named volume postgres_data stores PostgreSQL data. This means that even if the database container stops or is recreated, the data stays safe and can still be used.

Django Application Setup

To set up the Django project, we started by creating a new Django project using the command django-admin startproject midterm. Then, we created a new app within the project called tasks using python manage.py startapp tasks. The app is where the core functionality of the task management system is built, including the models, views, and templates. The structure of the project is shown in Figure 6.
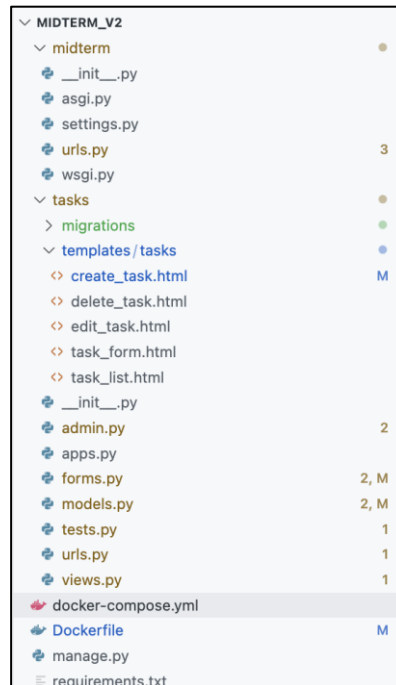
Figure 6. The structure of the Midterm_v2 project in VSCode

To connect Django to the PostgreSQL database defined in Docker Compose, we updated the settings.py file. In the DATABASES section, we specified the database engine as PostgreSQL, and added the name of the database (midterm), the user (midterm_user), and the password (midterm_password) from the Docker Compose file. The settings.py is shown in Figure 7.

```
77  DATABASES = {
78      'default': {
79          'ENGINE': 'django.db.backends.postgresql',
80          'NAME': 'midterm',
81          'USER': 'midterm_user',
82          'PASSWORD': 'midterm_password',
83          'HOST': 'db',
84          'PORT': '5432',
85      }
86  }
```

Figure 6. The settings.py file

The host is set to db, which is the service name for the database container in Docker Compose, and the port is 5432 as defined in the Compose file. This allows Django to interact with the PostgreSQL database running in the container.

Defining Django Models

We created a Task model in the models.py file under the tasks app. This model defines the structure of a task, with fields like title (for the task name), description (for details about the task), and due_date (to specify when the task is due). The Task class is shown in Figure 7.

```python
from django.db import models

You, 59 minutes ago | 2 authors (Асанали Битанов and others)
class Task(models.Model):
    title = models.CharField(max_length=255)
    description = models.TextField()
    completed = models.BooleanField(default=False)
    created_at = models.DateTimeField(auto_now_add=True)
    due_date = models.DateTimeField(null=True, blank=True)

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

Figure 7. The Task class in the models.py file

Each field represents a column in the database, and Django automatically handles the conversion of these fields into database columns. This model serves as the foundation for managing tasks within the application. After defining the Task model, we created migrations using the command python manage.py makemigrations. This generates migration files that tell Django how to modify the database to match the new model structure. The output of the command is shown in Figure 8.



**midterm_v2-web-run-ec97e504d8b3**

‹  midterm_v2-web

aa875a9dc80d

| Logs | Inspect | Bind mounts | Exec | Files | Stats |

```
2024-10-23 15:21:54  Migrations for 'tasks':
2024-10-23 15:21:54    tasks/migrations/0001_initial.py
2024-10-23 15:21:54      - Create model Task
```
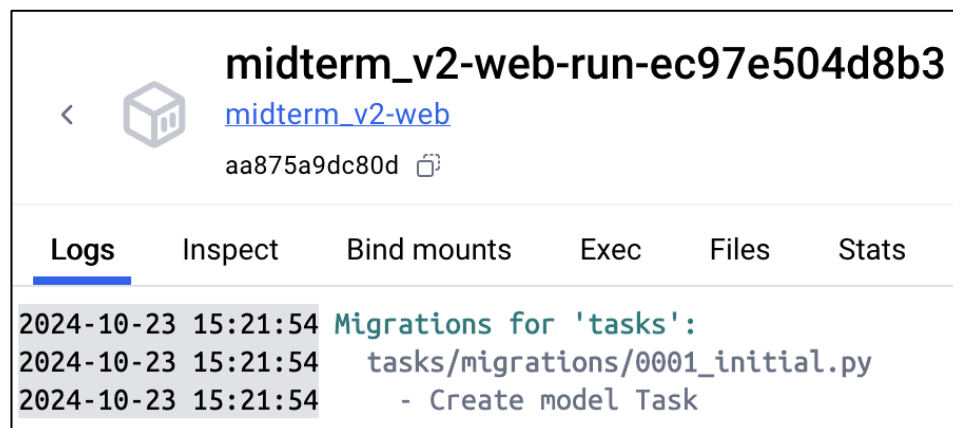
Figure 8. The output of the python manage.py makemigrations command

To apply these migrations and set up the database schema, we ran python manage.py migrate. This command applies the changes defined in the migration files to the PostgreSQL

database, creating the necessary tables and columns to store task data. The output of the command is shown in Figure 9.
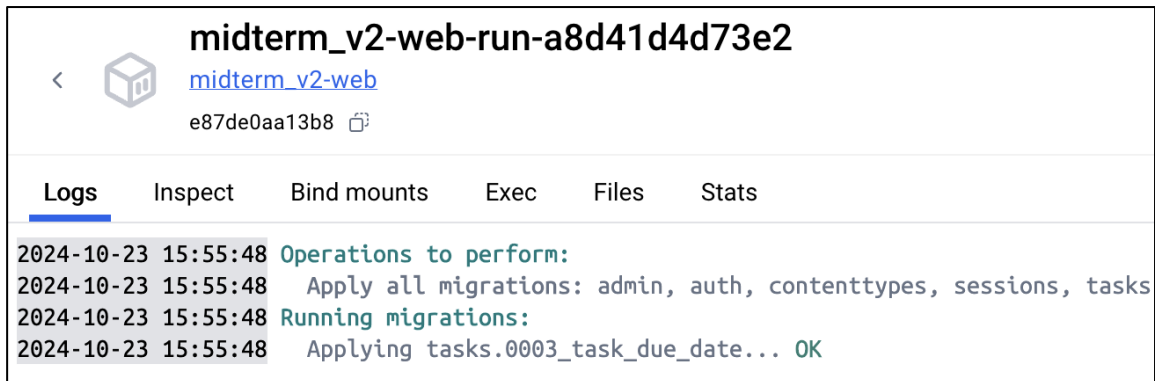


Figure 9. The output of the python manage.py migrate command

Next, we set up the views.py file. In Django, views are the part of the app that handles user requests and sends back responses. The views.py file is shown in Figure 10.



Figure 10. The output of the python manage.py makemigrations command

Here we can see task_list, create_task, edit_task and delete_task. Each of them stands for their purpose respectively. Task_list shows us the list of the created tasks. From here, we can edit them or delete them. The page of tasks list is shown in Figure 11.
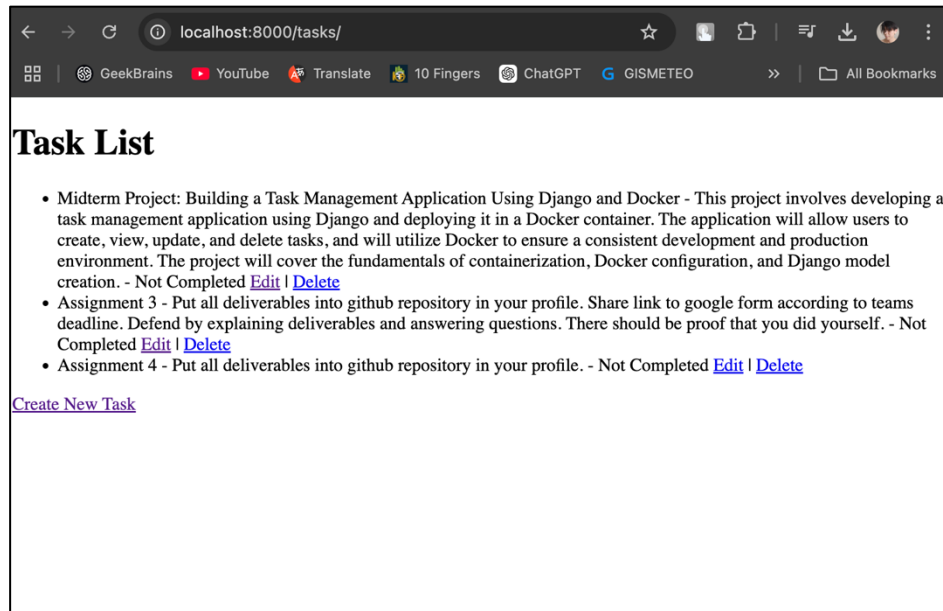
Figure 11. The page of tasks list

The page is located at http://localhost:8000/tasks/. Here we see three tasks: Midterm Project, Assignment 3 and Assignment 4. Each of them has a description and state of completion. In this page we can create a new task, edit or delete the existing ones. Create new task page is shown in Figure 12.
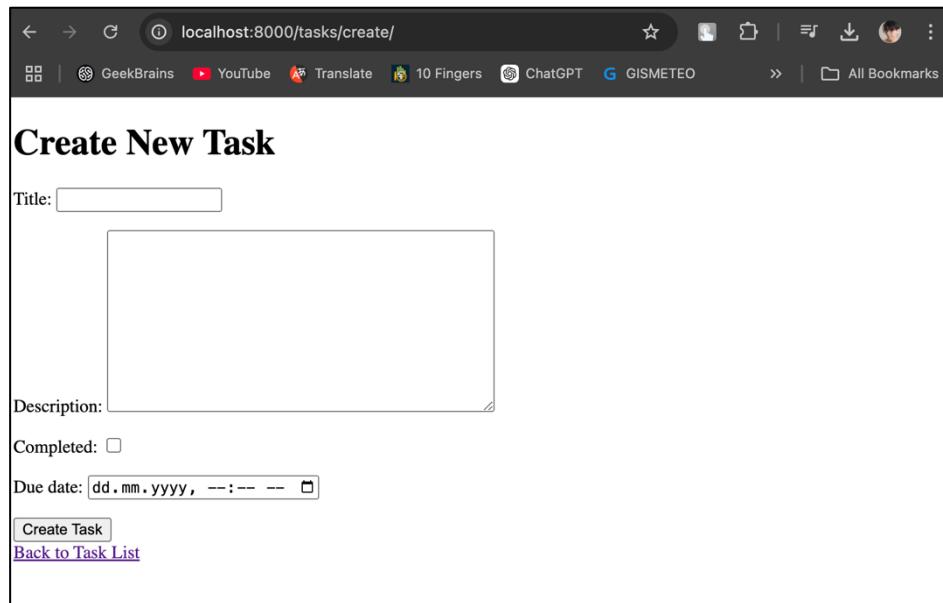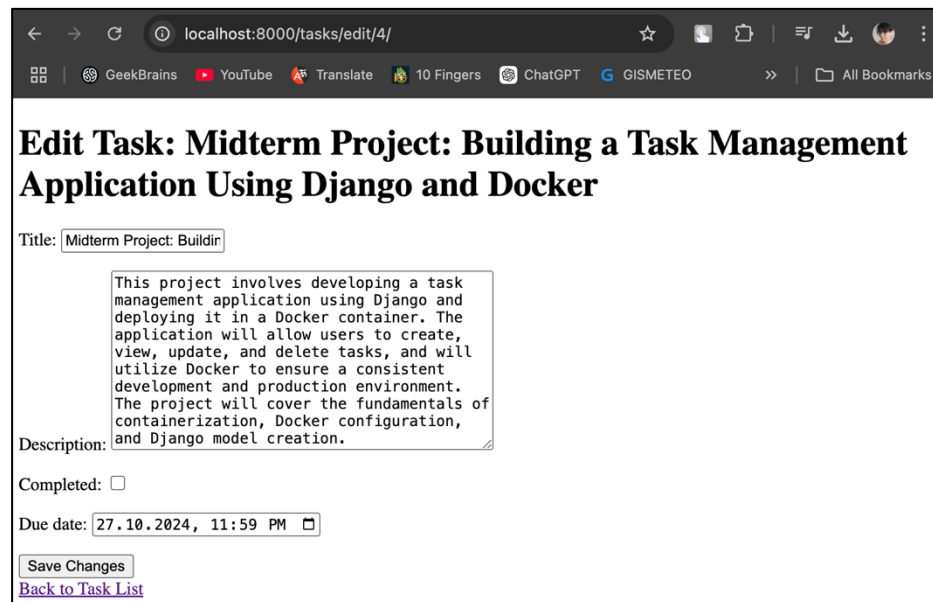


Figure 12. The Create New Task page

Here we can create a new task by entering the forms. We can write down a title, a description, and select a date. Then, if it's done we can edit the task. The edit task page is shown in Figure 13.



Figure 13. The Edit Task page

If we want to delete a task, we can do so by pressing the Delete button on the page of tasks list. When we do so there is a confirmation page. The page is shown in Figure 14.



Figure 14. The delete confirmation page

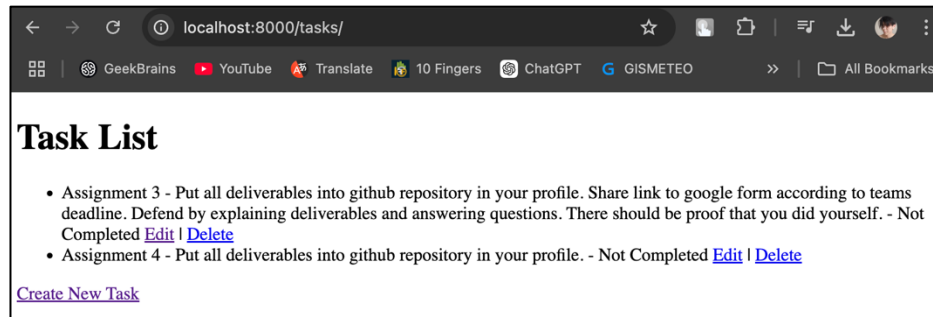If we press Yes, Delete the task will be deleted. We can see that in Figure 15.

Figure 14. The delete confirmation page

The Midterm Project task is no longer on the Task List page.

## Conclusion

In this midterm project, we worked on improving our development skills by building a simple task management app. The app has important pages like Task List, Create Task, Edit Task, and Delete Task. We used Django as the framework for the app and PostgreSQL as the database. Both were set up inside Docker containers, which helped us keep the environment consistent. This project focused on key topics like Containerization with Docker, writing a Dockerfile, and using Docker Compose to link the app and the database.

We learned how to create a multi-layered container where both the Django app and the PostgreSQL database work together smoothly. This is important because using Docker makes sure that the app behaves the same no matter where it's running, whether it's on a personal computer or a production server. Each part of the project was an opportunity to practice skills that are important for real-world development.

The pages we created allow users to view tasks, create new tasks, edit tasks, and delete them. For example, the Task List page shows all the tasks, and the Create Task page allows users to add new tasks by filling out a form. We also learned how to set up forms in Django and how to connect them to the database, making sure the data is saved and updated correctly. The project gave us a good understanding of how to build and manage web applications with modern tools.

This project provided valuable experience with technologies that are widely used in web development. Learning to manage both the app and the database inside containers made the process smoother and more reliable. It taught us how to work with Docker, how to set up a PostgreSQL database, and how to build functional pages in Django. These are skills we can apply in future projects, making our development process more efficient and professional.

By working through the project, we gained a better understanding of how to use these tools together. We can now use Docker to make sure our applications run the same way across different environments, and we can use Django to build web apps that are easy to manage and extend. This experience will help us in future projects, especially when building applications that need both a front-end and a back-end, and when working in teams.

## References

1. What        is        Docker        Containerization?        [PubNub]
   https://www.pubnub.com/guides/containerization/
2. Get Started with DockerComparison [Docker] https://www.docker.com/get-started/
3. What Is a Dockerfile And How To Build It – Best Practices [spacelift]
   https://spacelift.io/blog/dockerfile