KBTU KAZAKH-BRITISH TECHNICAL UNIVERSITY

Prepared by: Serzhan Yerasil

Midterm

Web Development

Building a Task Management Application Using Django and Docker

26.10.2024

Table of Contents

Executive summary

The first goal of the project is to build a task management service that listens to external calls and works with the database using CRUD operations. The second goal is to describe a dockerfile for the project, use it for the service, and raise postgres as a second container, and link them together in docker-compose. Technologies used: Django, PostgreSql, Docker. Finally, we managed to make a task management application

Introduction

Containerization in modern development is one of the most important tools. Its importance lies in isolation, a container that runs locally - will run on another machine with a very high probability, the second big plus is its light weight, unlike a virtual machine. This project is primarily aimed at studying such a well-known and popular framework as Django and connecting postgresql to it, as well as wrapping them in docker, django provides an application skeleton, which is its main feature, you will not need to suffer much with designing the project structure. Postgres is one of the most popular databases in the world, besides it is free, open source. In this project, we interact with it using the operation Create, Read, Update, Delete through ORM

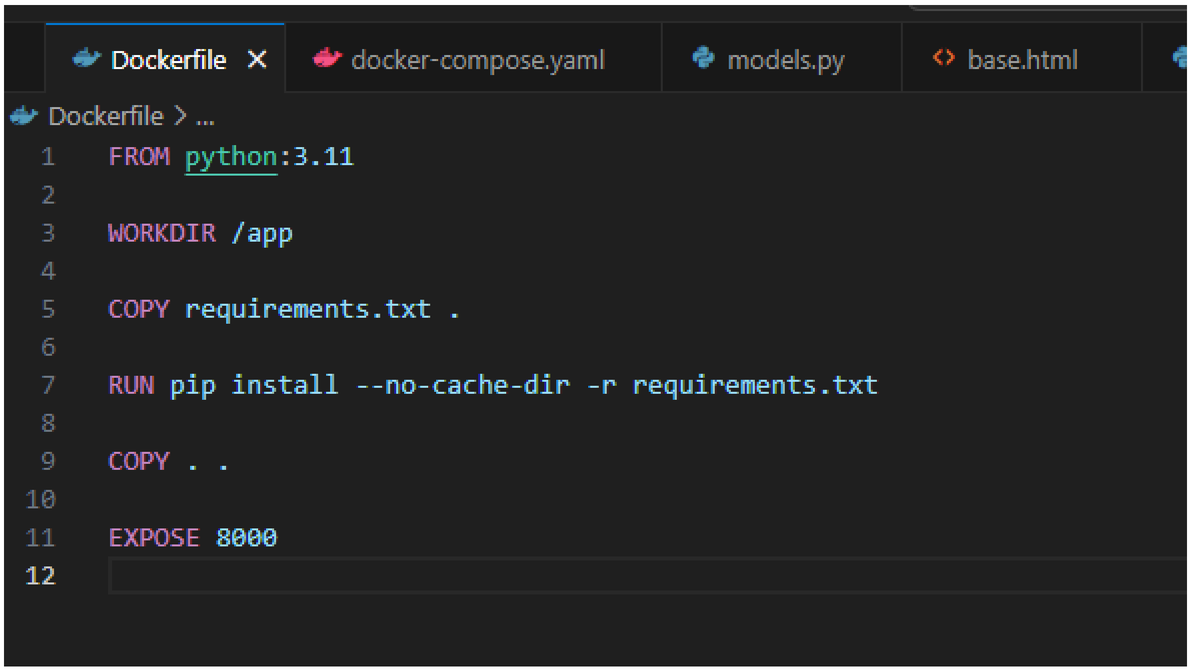
Project Objectives

First objective is to build an application that allows users to create, view, update, and delete tasks, the second objective is to interact with the database using the built-in django orm, for this we define django models, for this in the django class itself we describe the structure of our table in the database, their name and the connections between them, and with the help of migration we perform a rollout. The third objective is to write a simple frontend for interaction with the backend, the fourth objective is to wrap all the services in docker and configure their interaction

Intro to Containerization: Docker

Containerization is the packaging of an application together with its dependencies, libraries and configuration files as a single entity into a single object, its main advantages are lightness and portability - that is, if it runs locally, then with a very high probability it will run on another machine. In order to install Docker, I downloaded Docker Desktop and enabled virtualization on the motherboard. In order to run the container, we use the command docker run image name or path to the Dockerfile, this command will check if such an image exists locally, if not, it will pull it from the repository and run it as a container.

Creating a Dockerfile


Dockerfile
docker-compose.yaml
models.py
base.html
Dockerfile > ...

```dockerfile
FROM python:3.11

WORKDIR /app

COPY requirements.txt .

RUN pip install --no-cache-dir -r requirements.txt

COPY . .

EXPOSE 8000

```

The Dockerfile I used for this application.

FROM python:3.11 - this command will start a container with a Linux operating system and a Python interpreter installed in it

WORKDIR /app set the working directory inside the container

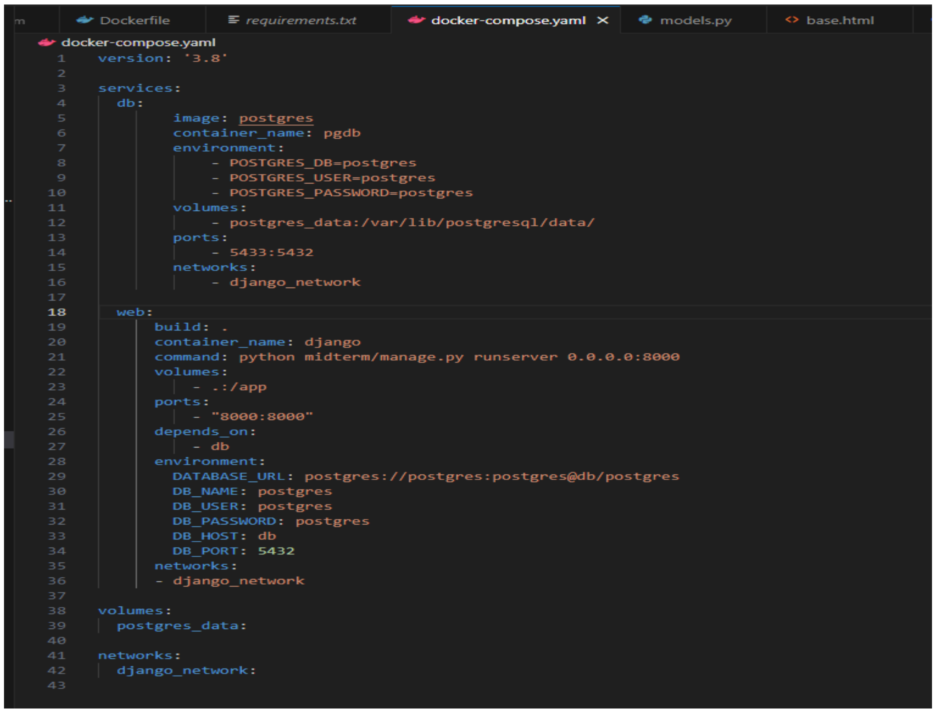
COPY requirements.txt . copy the requirements with a separate command to the directory so that each time the code changes, you don't have to copy and install it again

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

COPY . . copy the remaining files to the working directory

EXPOSE 8000 specify which port the container will use

Using Docker Compose



```yaml
version: '3.8'

services:
  db:
      image: postgres
      container_name: pgdb
      environment:
          - POSTGRES_DB=postgres
          - POSTGRES_USER=postgres
          - POSTGRES_PASSWORD=postgres
      volumes:
          - postgres_data:/var/lib/postgresql/data/
      ports:
          - 5433:5432
      networks:
          - django_network

  web:
    build: .
    container_name: django
    command: python midterm/manage.py runserver 0.0.0.0:8000
    volumes:
      - .:/app
    ports:
      - "8000:8000"
    depends_on:
      - db
    environment:
      DATABASE_URL: postgres://postgres:postgres@db/postgres
      DB_NAME: postgres
      DB_USER: postgres
      DB_PASSWORD: postgres
      DB_HOST: db
      DB_PORT: 5432
    networks:
    - django_network

volumes:
  postgres_data:

networks:
  django_network:
```

here we have two services, db and django app. In the db service we use image postgres and set credentials for it, and set volumes for saving data to local machine for further new containers, because after removing container all data will disappear. Also we map ports from docker container to local machine and specify network where db will defined. In web service we use our Dockerfile as our image, also define volumes and ports and specify that it depends from db. Setting up env variables for connecting to db and define his network. And command that will start our app. I wrote a Dockerfile for Django, wrapped two services in docker compose, configured them to be in the same network, so that they were in the same domain, configured environment variables for the Django application so that it could pull secret data from environment variables inside the container. I configured docker volumes for the Django database and application so that the data in the container and on the local host were mapped to each other, this is done so that if the container is deleted, then when creating a new container, it will pull data from the local host files

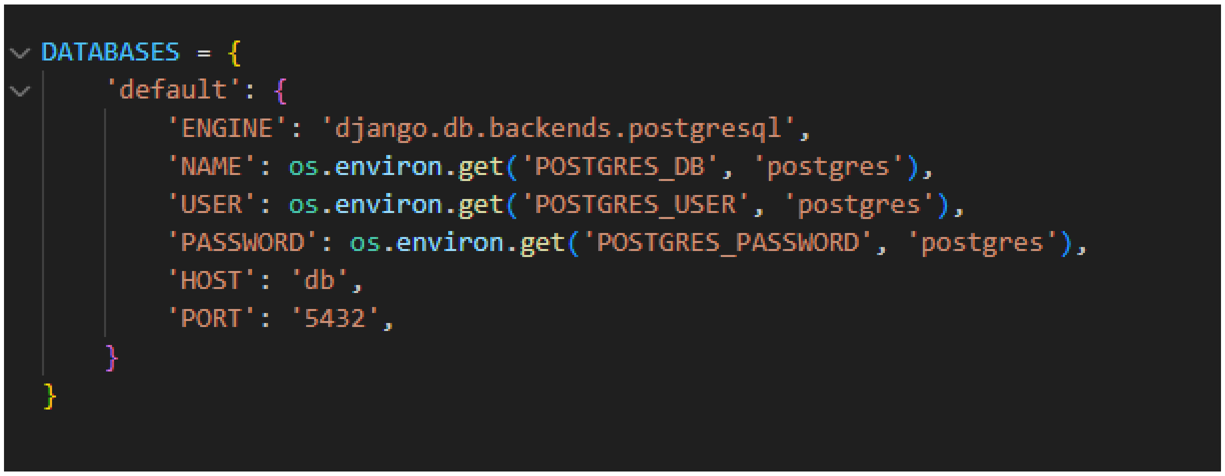
Docker Networking and Volumes

Docker networks allow containers to interact if they are on the same network, this is like a definition area for them. Within the same network, they access each other by the service name described in the docker compose file, and not via an IP address

Docker volumes are needed so that the data inside the container is not lost after they are completed or deleted. To do this, a folder is created on the local machine that is associated with the container folder, after deleting the container, all data inside the container is lost, but they are on the local machine, as soon as a new container is launched, the data from the local directory will be displayed in the container directory.

Django Application Setup

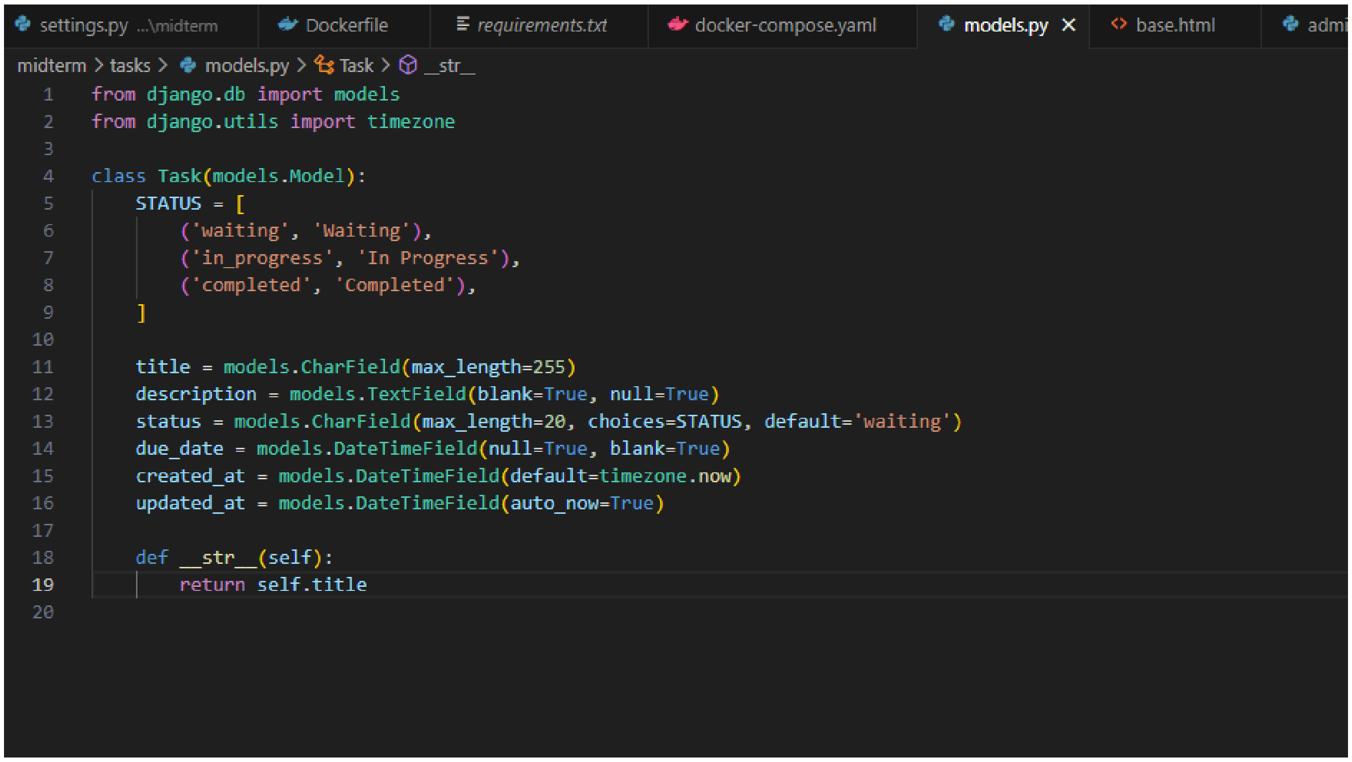
To create a django project I entered django-admin startproject myproject the skeleton of the project is created. Then python manage.py startapp tasks to create an application inside the project, then this application was added to INSTALLED_APPS so that the project recognizes this application. In order to connect to the database that we raised via docker, you need to change the credentials in the variable database



```python
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': os.environ.get('POSTGRES_DB', 'postgres'),
        'USER': os.environ.get('POSTGRES_USER', 'postgres'),
        'PASSWORD': os.environ.get('POSTGRES_PASSWORD', 'postgres'),
        'HOST': 'db',
        'PORT': '5432',
    }
}
```

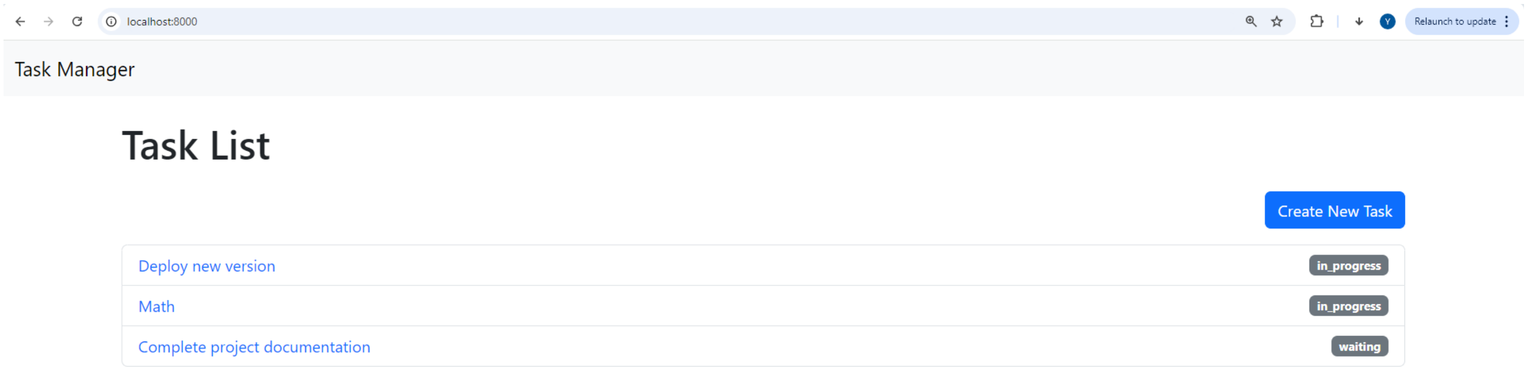
Defining Django Models

The structure and functionality of the database in our Django application is defined in models. Django models allow us to represent the structure of our data, easily manage it via an ORM, and provide the flexibility to make changes as needed. Migrations in Django allow us to apply changes to the database schema based on our models, keeping the database structure in sync with the code. Once we have described the structure in the models, we write python manage.py makemigrations to have Django generate migration files and apply them to the database with python manage.py migrate



```python
from django.db import models
from django.utils import timezone

class Task(models.Model):
    STATUS = [
        ('waiting', 'Waiting'),
        ('in_progress', 'In Progress'),
        ('completed', 'Completed'),
    ]

    title = models.CharField(max_length=255)
    description = models.TextField(blank=True, null=True)
    status = models.CharField(max_length=20, choices=STATUS, default='waiting')
    due_date = models.DateTimeField(null=True, blank=True)
    created_at = models.DateTimeField(default=timezone.now)
    updated_at = models.DateTimeField(auto_now=True)

    def __str__(self):
        return self.title

```

here are described the fields that will be in the database, here is also set a limitation on the status field

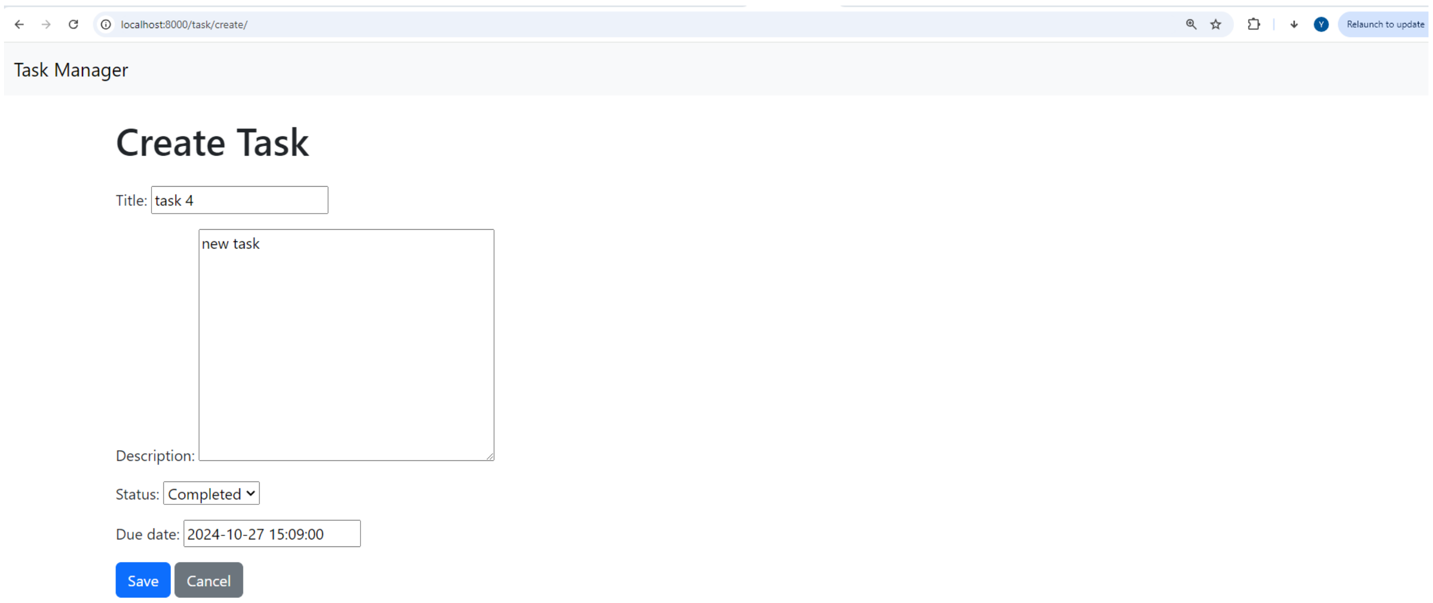


Task Manager

Task List

Deploy new version	in_progress
Math	in_progress
Complete project documentation	waiting

Fig 1. Main page, list all tasks



localhost:8000/task/create/

Relaunch to update

Task Manager

Create Task

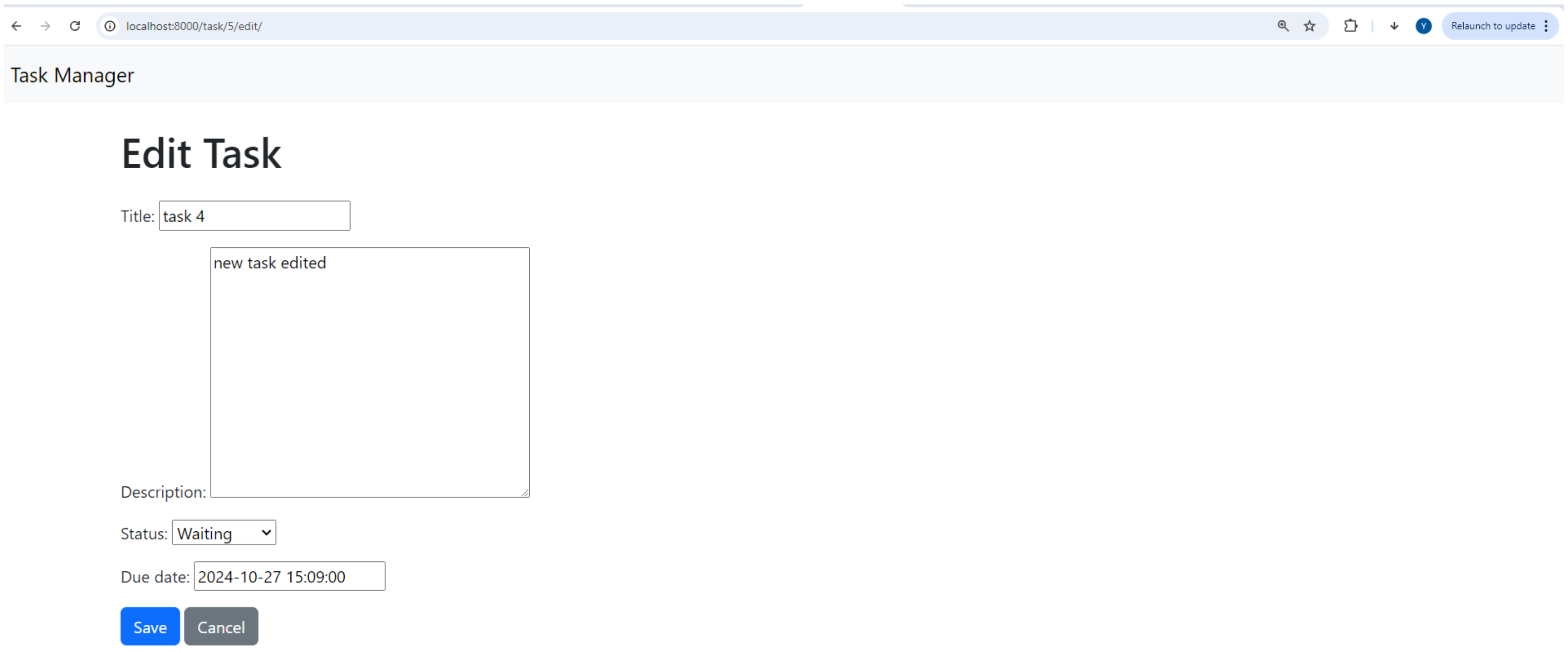
Title: task 4

Description: new task

Status: Completed

Due date: 2024-10-27 15:09:00

Fig 2.   Create tak


localhost:8000/task/5/edit/

Task Manager

Edit Task

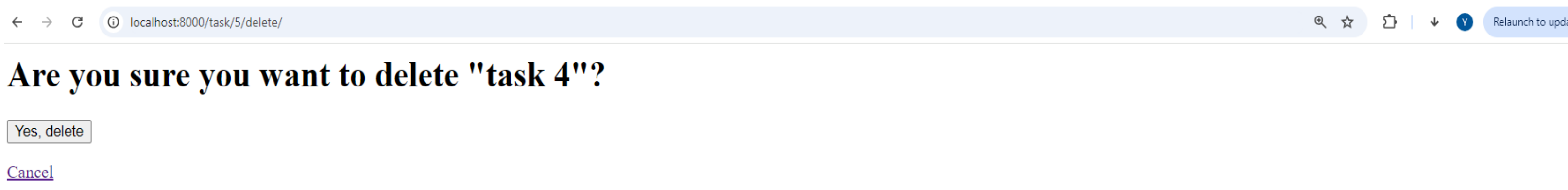
Title: task 4

Description: new task edited

Status: Waiting

Due date: 2024-10-27 15:09:00

Fig 3. Edit task



Are you sure you want to delete "task 4"?

Fig 5. Delete task

Conclusion

This project was a great learning experience in building task management app using Django and Docker. Working with Django made it easy to write the backend, especially with features like models, migrations, and the ORM