KAZAKH KBTU BRITISH TECHNICAL UNIVERSITY

Midterm

Building a task management application using Django and Docker

Prepared by Ibragimov Temirlan

Almaty, 26.10.2024

Table of contents

Executive summary

The goal of this assignment is to use Django with Docker to create a simple project. In this project, I used Python 3, Django, PostgreSQL, and Docker. The project allows users to create, read, update, and delete tasks, fulfilling the requirement of implementing CRUD operations for the assignment. The project covers the fundamentals of containerization, Docker configuration and Django model configuration.

Introduction

Containerization has become a vital part of each new project that's being developed. Docker containerization makes running, building processes much easier and saves developers time. Without docker, developers would have to deal with project building and configuration processes, which potentially could take a lot of time. So, by using Docker, we can ensure that our application has the same system engine and setup, which prevents any potential merge errors in the future.

In this project, I used Docker to containerize my Python application. Specifically, I used Django for CRUD operations and PostgreSQL for data storage. The advantage of Docker is that I'm no longer required to start each component of the project manually, because Docker starts them automatically.

Project objectives

Among the goals of this project, I can highlight the creation of a prototype of the application with the necessary requirements. All this is necessary in order to better understand this subject and master the material.

List of goals:

1. Understand the concepts of containerization and its benefits.
   Install Docker and set up a simple Docker container to run a basic application.
2. Create a Dockerfile to define the environment for the Django application.
   Specify the necessary dependencies, including Python and Django packages.
3. Use Docker Compose to define and run multi-container applications.
   Set up a docker-compose.yml file to manage both the Django app and the database service.
4. Configure Docker networking to allow communication between containers.
   Use Docker volumes to persist data and manage the database state.
5. Set up a Django project and create a basic application structure.
   Configure settings to work with the Docker environment.
6. Define Django models for the task management application.
   Implement migrations to create the necessary database tables.

Intro to containerization: Docker

Containerization is a software deployment process that bundles an application's code with all the files and libraries it needs to run on any infrastructure. Traditionally, to run any application on your computer, you had to install the version that matched your machine's operating system. For example, you needed to install the Windows version of a software package on a Windows machine. However, with containerization, you can create a single software package, or container, that runs on all types of devices and operating systems.

Developers use containerization to build and deploy modern applications because of the following advantages:

1. **Portability** - Software developers use containerization to deploy applications in multiple environments without rewriting the program code. They build an application once and deploy it on multiple operating systems. For example, they run the same containers on Linux and Windows operating systems. Developers also upgrade legacy application code to modern versions using containers for deployment.
2. **Scalability** - Containers are lightweight software components that run efficiently. For example, a virtual machine can launch a containerized application faster because it doesn't need to boot an operating system. Therefore, software developers can easily add multiple containers for different applications on a single machine. The container cluster uses computing resources from the same shared operating system, but one container doesn't interfere with the operation of other containers.
3. **Fault tolerance** - Software development teams use containers to build fault-tolerant applications. They use multiple containers to run microservices on the cloud. Because containerized microservices operate in isolated user spaces, a single faulty container doesn't affect the other containers. This increases the resilience and availability of the application.
4. **Agility** - Containerized applications run in isolated computing environments. Software developers can troubleshoot and change the application code without interfering with the operating system, hardware, or other application services. They can shorten software release cycles and work on updates quickly with the container model.

To install Docker, I used the official documentation. I attached a link to it in the references section. My operating system is Windows, so I downloaded Docker Desktop to handle all the operations with Docker. Docker Desktop is a great application itself, because it has a usable UI, which easies the user experience. I can see all the containers, volumes, images, and also I can run my docker container and see the logs.

Creating a Dockerfile

Dockerfile components:

1. **FROM -** The FROM instruction initiates a new build phase and sets the base image for subsequent instructions. As such, a valid Dockerfile must begin with the FROM instruction.
   FROM <imge_name>:<version>

2. **RUN -** The RUN instruction allows you to install your application and the required packages for it. It executes any command on top of the current image and creates a new layer giving the result.
   RUN <command> (the command is run in a shell)

3. **CMD -** The CMD command defines a default command to run when your container starts.
   CMD <command>

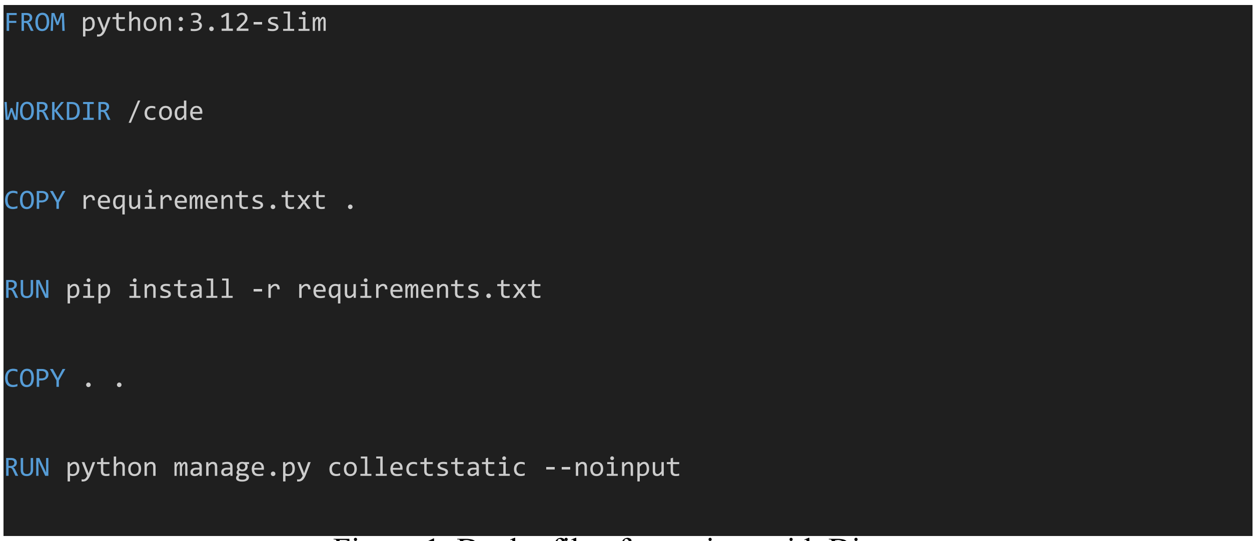
4. **ENTRYPOINT -** An ENTRYPOINT is the same work as CMD. An ENTRYPOINT allows you to the extra argument of the command that will run as an executable.
   ENTRYPOINT <command> <param1> <param2>….

5. **COPY -** The COPY instruction copies files or directories from the Base os <src> and adds them to the filesystem of the container at the path <dest>
   COPY <src> <dest>

6. **WORKDIR -** The WORKDIR instruction sets the working directory for any RUN, CMD, ENTRYPOINT, COPY and ADD instructions that follow it in the

Dockerfile.
WORKDIR </path/to/workdir>

7. And more components to be used.



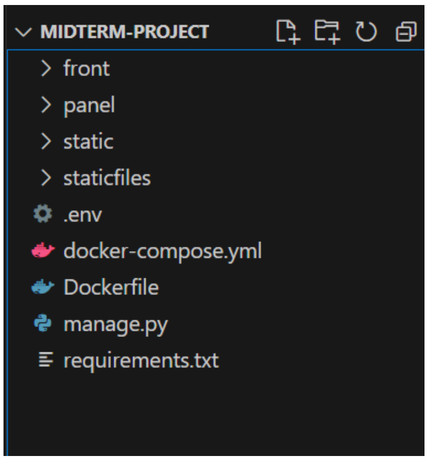
```
FROM python:3.12-slim

WORKDIR /code

COPY requirements.txt .

RUN pip install -r requirements.txt

COPY . .

RUN python manage.py collectstatic --noinput
```

Figure 1. Dockerfile of container with Django

Dockerfile explaination:

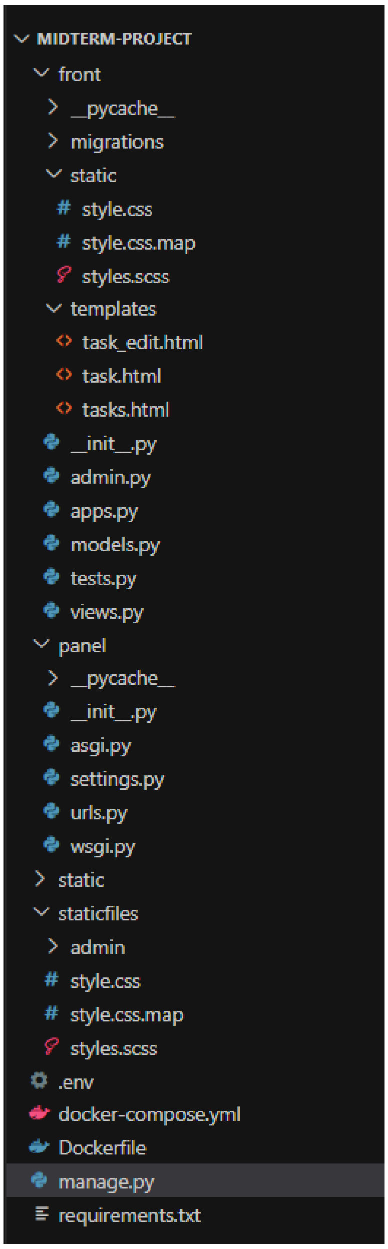
1. **FROM python:3.12-slim** - This line imports the Python 3.12-slim image from the Docker registry.
2. **WORKDIR /code** - This sets the working directory inside the container to /code.
3. **COPY requirements.txt ./** - This copies the requirements.txt file from the local machine to the current directory in the container.
4. **RUN pip install -r requirements.txt** - This installs the Python packages listed in requirements.txt.
5. **COPY . .** - This copies all files from the local directory to the current directory in the container.

Using Docker Compose



```
MIDTERM-PROJECT
  > front
  > panel
  > static
  > staticfiles
  .env
  docker-compose.yml
  Dockerfile
  manage.py
  requirements.txt
```

Figure 2. Project structure



```
MIDTERM-PROJECT
  front
    > __pycache__
    > migrations
    static
      # style.css
      # style.css.map
      styles.scss
    templates
      task_edit.html
      task.html
      tasks.html
    __init__.py
    admin.py
    apps.py
    models.py
    tests.py
    views.py
  panel
    > __pycache__
    __init__.py
    asgi.py
    settings.py
    urls.py
    wsgi.py
  > static
  staticfiles
    > admin
    # style.css
    # style.css.map
    styles.scss
  .env
  docker-compose.yml
  Dockerfile
  manage.py
  requirements.txt
```

Figure 3. Full project structure

This docker-compose.yml file defines a multi-container Docker application with two services: django and postgres. It also sets up a custom network named net.

```yaml
version: '3.9'

services:
  db:
    image: postgres
    environment:
      POSTGRES_DB: ${DB_NAME}
      POSTGRES_USER: ${DB_USER}
      POSTGRES_PASSWORD: ${DB_PASSWORD}
    volumes:
      - postgres_data:/var/lib/postgresql/data
    networks:
      - django_network

  web:
    build: .
    command: python manage.py runserver 0.0.0.0:8000
    volumes:
      - .:/code
      - static_volume:/code/static
    ports:
      - "8000:8000"
    environment:
      DB_NAME: ${DB_NAME}
      DB_USER: ${DB_USER}
      DB_PASSWORD: ${DB_PASSWORD}
      DB_HOST: db
      DB_PORT: 5432
    depends_on:
      - db
    networks:
      - django_network

volumes:
  postgres_data:
  static_volume:

networks:
  django_network:
```

Figure 4. docker-compose.yml

Services

1. **django**
    1. **build: .**: Builds the Docker image from the current directory, using the Dockerfile present.
    2. **command**: Runs the Django development server on port 8000.
    3. **volumes**:
        - Mounts the project directory (.) to /code in the container, allowing code changes without rebuilding the container.
        - Mounts static_volume to /code/static for persisting static files.
    4. **ports**: Maps port 8000 on the host to port 8000 in the container, making the Django application accessible locally.
    5. **environment**: Sets environment variables for the database connection, which Django uses to connect to PostgreSQL.
    6. **depends_on**: Ensures the PostgreSQL service starts before Django.
    7. **networks**: Connects the Django app to the django_network, allowing it to communicate with db.

2. **postgres**
    1. **image**: postgres: Uses the official PostgreSQL Docker image.
    2. **environment**: Sets up environment variables for PostgreSQL by getting values from a .env file:
    3. **POSTGRES_DB**: Sets the database name.
    4. **POSTGRES_USER**: Sets the database user.
    5. **POSTGRES_PASSWORD**: Sets the password for the database user.
    6. **volumes**: Mounts a named volume postgres_data to /var/lib/postgresql/data in the container, persisting data across container restarts.
    7. **networks**: Connects this service to django_network for communication with other services.

Networks

- **django_network**: A custom bridge network used to facilitate communication between the django and postgres services.

Volumes - defines named volumes for my project's data:

- **postgres_data**: Persists PostgreSQL data.
- **static_volume**: Persists static files generated by Django.

This setup ensures that the Django application can communicate with the PostgreSQL database, and the health of the PostgreSQL service is monitored before the Django service starts.
The entire database state is preserved by mounting volumes in the Postgres service. Similarly, mounting helps dynamically apply and reflect changes in the Django project code.

Docker networking and volumes

The docker-compose.yml file defines a custom network named net using the bridge driver. This network sets up communication between the django and postgres services. Let me explain the benefits of this setup:

1. **Isolation**: The custom network isolates the services from other containers running on the host, enhancing security.

2. **Service discovery**: Docker automatically assigns each service a hostname based on the service name, making it easy for services to discover and communicate with each other.
3. **Simplified configuration**: By using a custom network, you avoid the need to manually configure IP addresses and ports for inter-service communication.

Volumes - defines named volumes for my project's data:

- **postgres_data**: Persists PostgreSQL data.
- **static_volume**: Persists static files generated by Django.

Defining Django models

To create models in Django, I created a model in the models.py file. This model is a python class with fields representing columns in the table. In this case, there are 4 columns: title, description, created_at, completed. CharField, TextField, DateTimeField and BooleanField are the data types of the corresponding columns.

To create migrations, follow these steps:

1. **Generate migrations** - run the command *python manage.py makemigrations*. This command inspects the models and creates migration files that describe the changes to be made to the database schema.
2. **Apply migrations** - use the command *python manage.py migrate*. This command applies all pending migrations to the database, ensuring the schema is up-to-date with the current state of the models.
3. **Check migration status** - the command *python manage.py showmigrations* lists all migrations and their applied status.
4. **Rollback migrations** - If needed, migrations can be rolled back using *python manage.py migrate <app_name> <migration_name>* to revert to a specific migration.

Note: If you are trying to migrate the models after docker build, you have to put "**docker exec**" before any command to execute it in a container.

Creating views

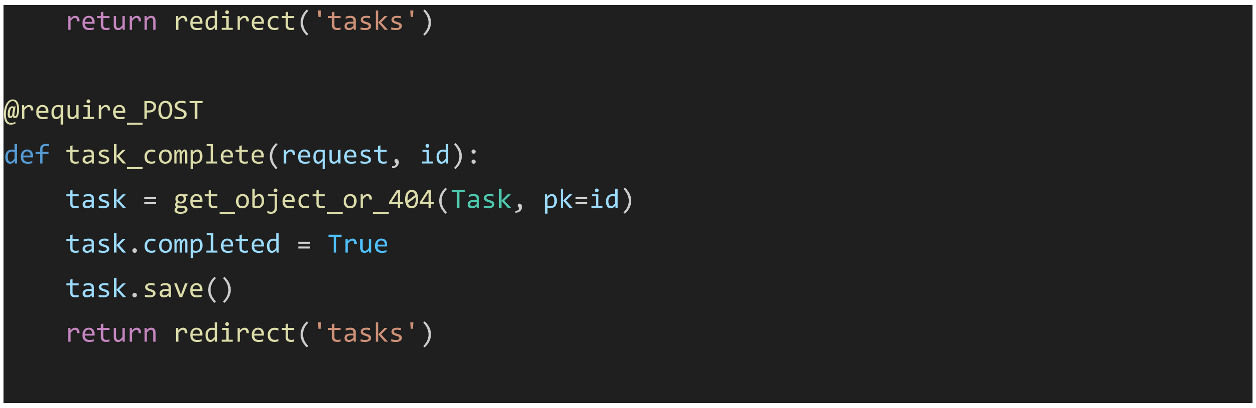
To create a view we should make corresponding changes in views.py file.

There are different views. Let me briefly describe them:

1. **tasks** - retrieves a list of all tasks, ordered by completed status and id in descending order. Uses tasks.html file as a template
2. **create_task** - works for **POST** requests and works only if the **HTTP** request is sent with **POST** method. It gets the title, description, saves them and redirects the user to the main page.
3. **task** - returns the task.html template with complete information about a task. In this function I used ***get_object_or_404*** function to get a task with a specific **ID**.
4. **task_edit** - a task is returned by **ID**, and if the request method is **POST**, the task will be updated with new values from the form data, and after this app will save the changes, and redirect the user to the tasks view. If the request method is not **POST**, it renders the task_edit.html template, allowing the user to edit the task.
5. **task_delete** - gets the task by ID, then proceeds to delete it from the database.

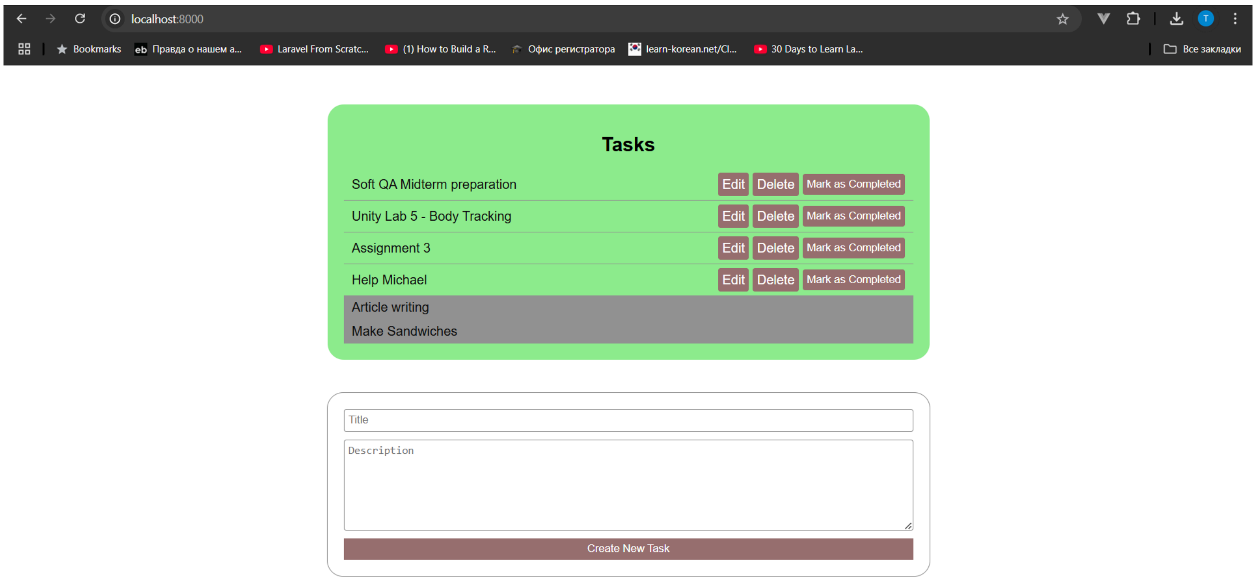
6. task_complete - simple HTTP POST request, which gets the ID of a task and marks them completed in the database.

```python
from django.shortcuts import render, get_object_or_404, redirect
from django.http import HttpResponse
from django.template import loader
from .models import Task
from .models import Task
from django.views.decorators.http import require_POST

# Create your views here.
def tasks(request):
    tasks_list = Task.objects.order_by('completed', '-id').values()
    template = loader.get_template('tasks.html')
    context = {
        'list': tasks_list,
    }
    return HttpResponse(template.render(context, request))

@require_POST
def create_task(request):
    new_task = Task(
        title=request.POST['title'],
        description=request.POST['description']
    )
    new_task.save()
    return redirect('tasks')

def task(request, id):
    task = get_object_or_404(Task, pk=id)
    return render(request, 'task.html', {'task': task})

def task_edit(request, id):
    task = get_object_or_404(Task, pk=id)
    if request.method == 'POST':
        task.title = request.POST['title']
        task.description = request.POST['description']
        task.save()
        return redirect('tasks')
    return render(request, 'task_edit.html', {'task': task})

def task_delete(request, id):
    task = get_object_or_404(Task, pk=id)
    task.delete()
```



```
    return redirect('tasks')

@require_POST
def task_complete(request, id):
    task = get_object_or_404(Task, pk=id)
    task.completed = True
    task.save()
    return redirect('tasks')
```

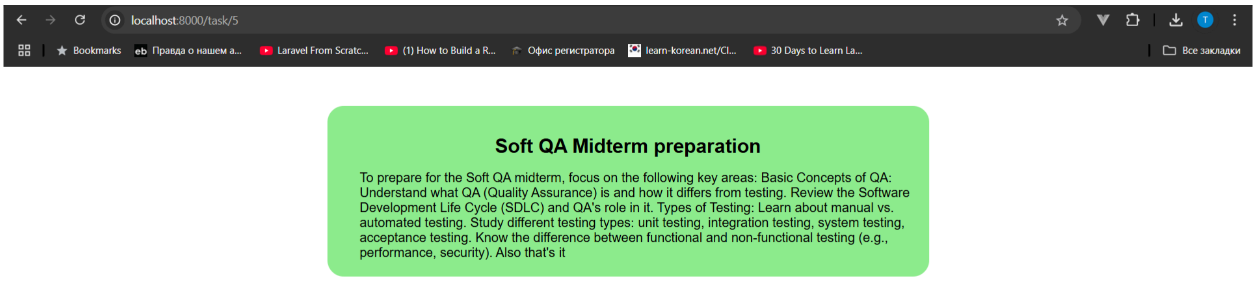
Figure 5. front/views.py



Tasks

Soft QA Midterm preparation
Unity Lab 5 - Body Tracking
Assignment 3
Help Michael
Article writing
Make Sandwiches

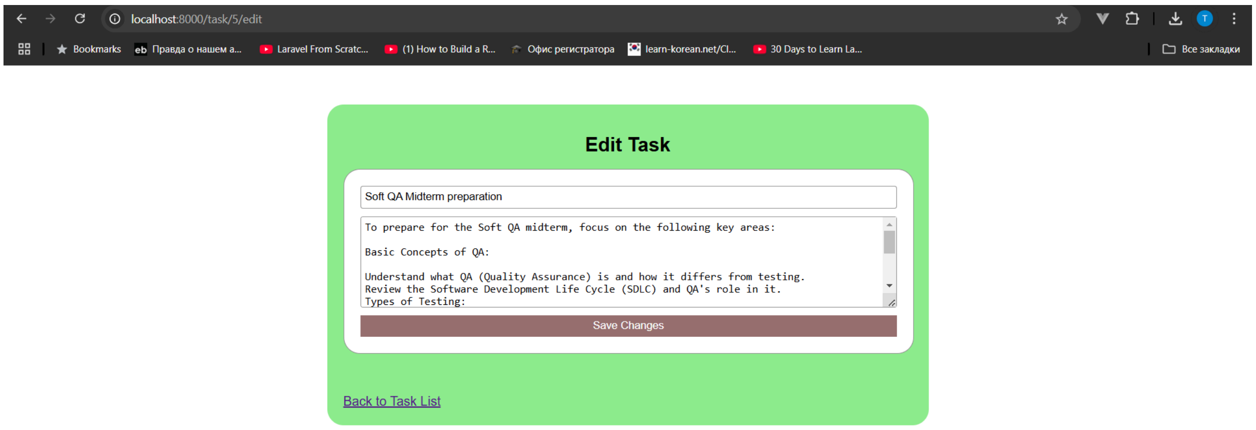
Figure 6. Main page



Soft QA Midterm preparation

To prepare for the Soft QA midterm, focus on the following key areas: Basic Concepts of QA: Understand what QA (Quality Assurance) is and how it differs from testing. Review the Software Development Life Cycle (SDLC) and QA's role in it. Types of Testing: Learn about manual vs. automated testing. Study different testing types: unit testing, integration testing, system testing, acceptance testing. Know the difference between functional and non-functional testing (e.g., performance, security). Also that's it

Figure 7. Task page

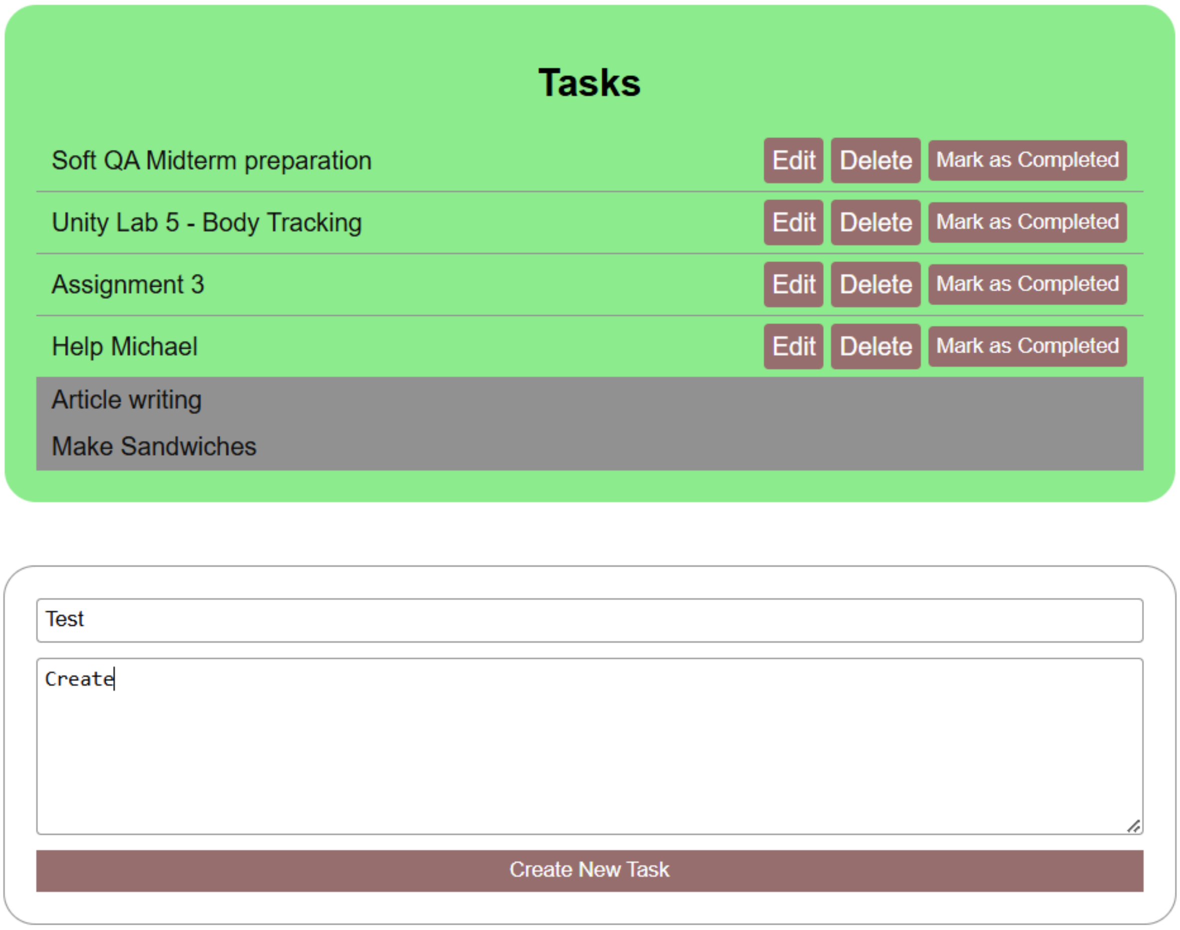


Edit Task

Soft QA Midterm preparation

```
To prepare for the Soft QA midterm, focus on the following key areas:

Basic Concepts of QA:

Understand what QA (Quality Assurance) is and how it differs from testing.
Review the Software Development Life Cycle (SDLC) and QA's role in it.
Types of Testing:
```

Figure 8. Task edit page



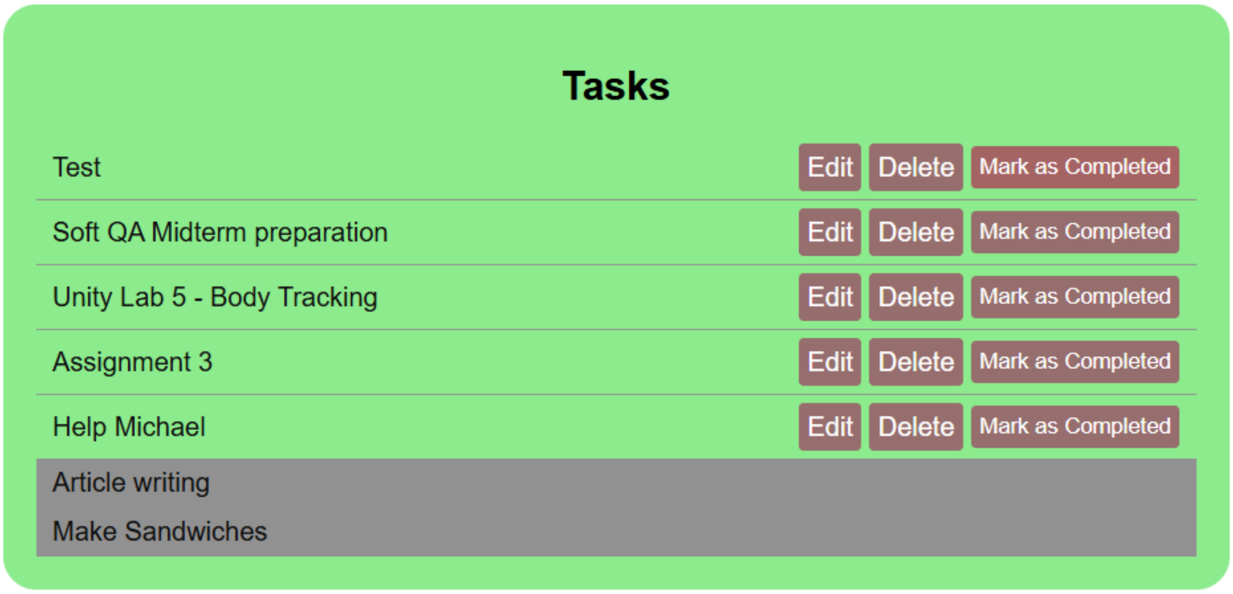
Tasks

Soft QA Midterm preparation	Edit	Delete	Mark as Completed
Unity Lab 5 - Body Tracking	Edit	Delete	Mark as Completed
Assignment 3	Edit	Delete	Mark as Completed
Help Michael	Edit	Delete	Mark as Completed
Article writing			
Make Sandwiches			

Test

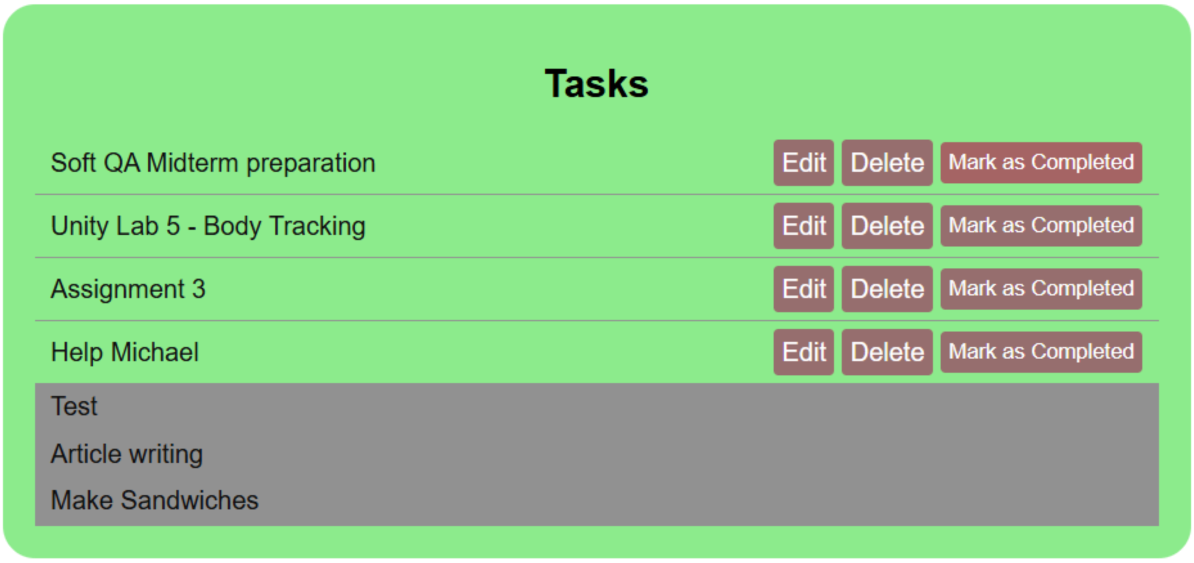
Create

Figure 9. Creating a new task


Tasks

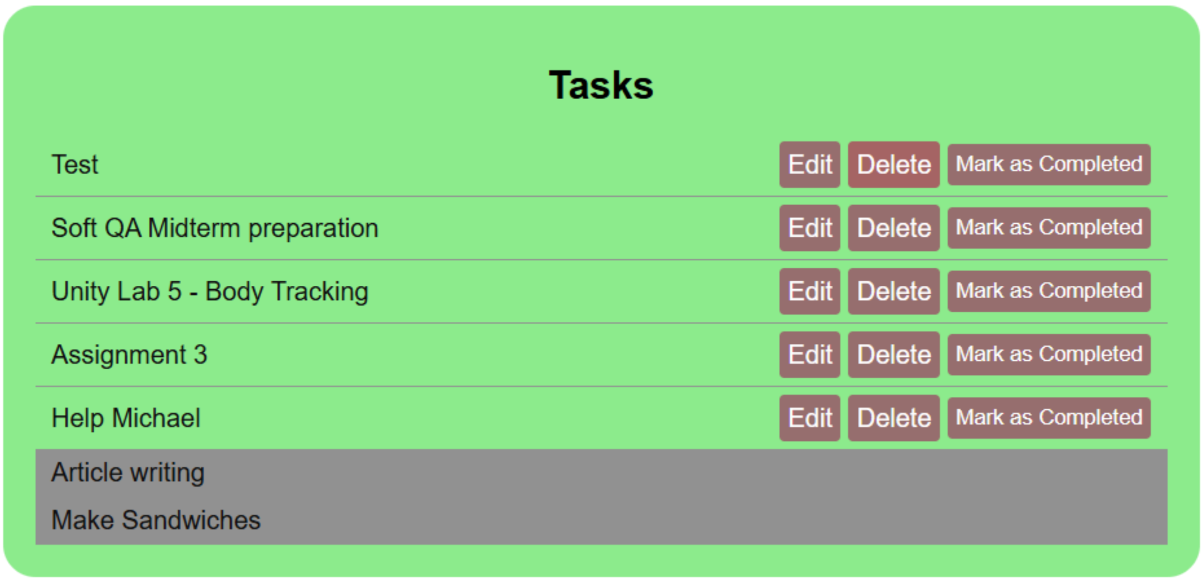
Test
Soft QA Midterm preparation
Unity Lab 5 - Body Tracking
Assignment 3
Help Michael
Article writing
Make Sandwiches

Figure 10. Marking the new task as completed


Tasks

Soft QA Midterm preparation
Unity Lab 5 - Body Tracking
Assignment 3
Help Michael
Test
Article writing
Make Sandwiches

Figure 11. Final result


Tasks

Test

Soft QA Midterm preparation

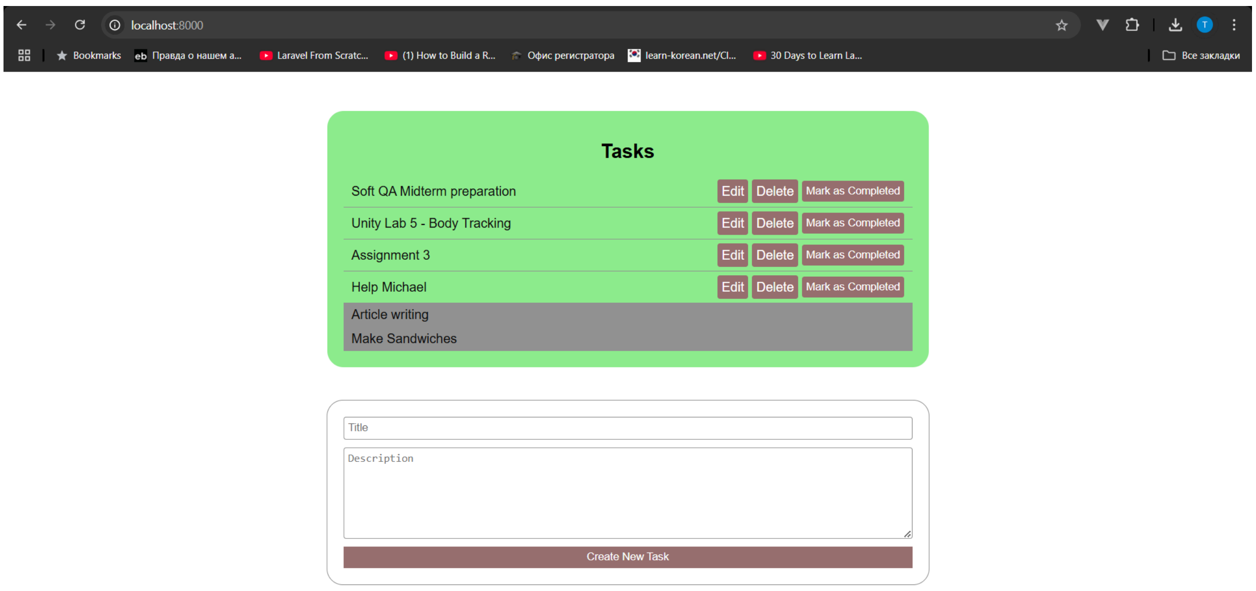
Unity Lab 5 - Body Tracking

Assignment 3

Help Michael

Article writing
Make Sandwiches

Figure 12. Deleting a task



Tasks

Soft QA Midterm preparation

Unity Lab 5 - Body Tracking

Assignment 3

Help Michael

Article writing
Make Sandwiches

Figure 13. Result

Conclusion

As a result, we received an application that fully complies with the requirements of this work, namely "ask management application using Django and deploying it in a Docker container. The application will allow users to create, view, update, and delete tasks, and will utilize Docker to ensure a consistent development and production environment. The project will cover the fundamentals of containerization, Docker configuration, and Django model creation."

By using Docker as a containerization tool, this project has gained all the benefits of applications using this approach:

1. Portability
2. Efficiency
3. Agility
4. Faster delivery
5. Improved security
6. Faster app startup
7. Easier management
8. Flexibility

References

Links to Django tutorials from official documentation:

1. https://docs.djangoproject.com/en/5.1/intro/tutorial01/
2. https://docs.djangoproject.com/en/5.1/intro/tutorial02/

Link to official Docker documentation:

- https://docs.docker.com/

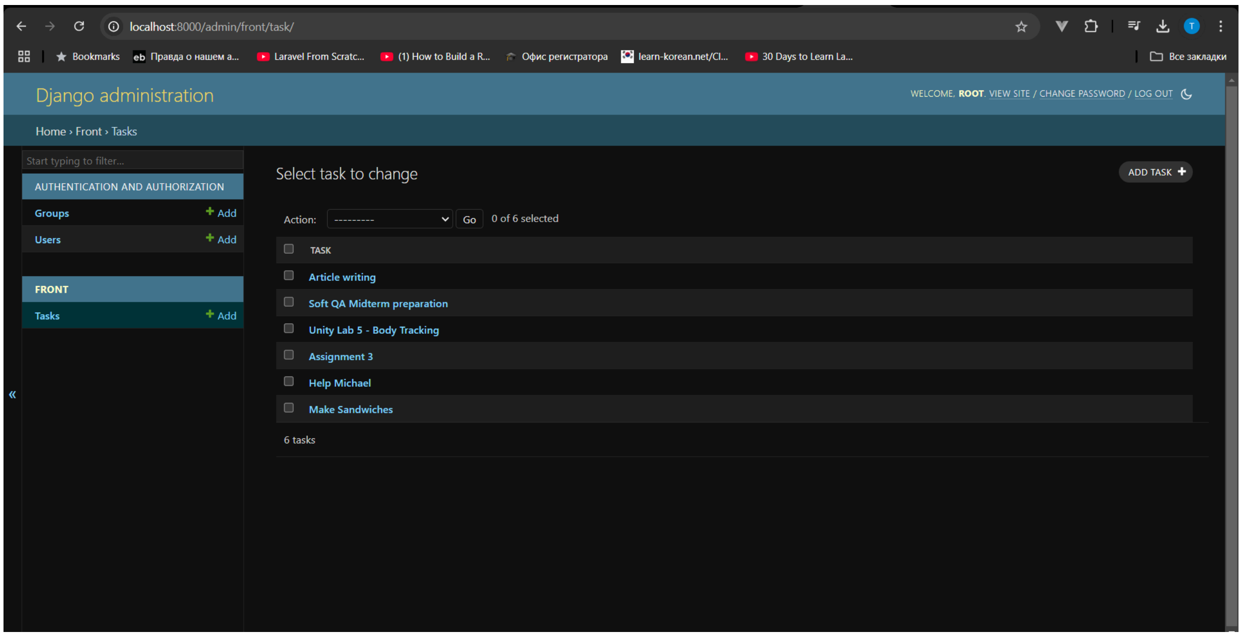
Links to some topics in StackOverflow:

1. https://stackoverflow.com/questions/44487590/django-shell-mode-in-docker
2. https://stackoverflow.com/questions/9181047/django-static-files-development

Link to my GitHub repository:

- https://github.com/bozzbala/webapps-midterm-project

Screenshots


localhost:8000/admin/front/task/

Bookmarks · Правда о нашем а... · Laravel From Scratc... · (1) How to Build a R... · Офис регистратора · learn-korean.net/Cl... · 30 Days to Learn La... · Все закладки

Django administration

WELCOME, ROOT. VIEW SITE / CHANGE PASSWORD / LOG OUT

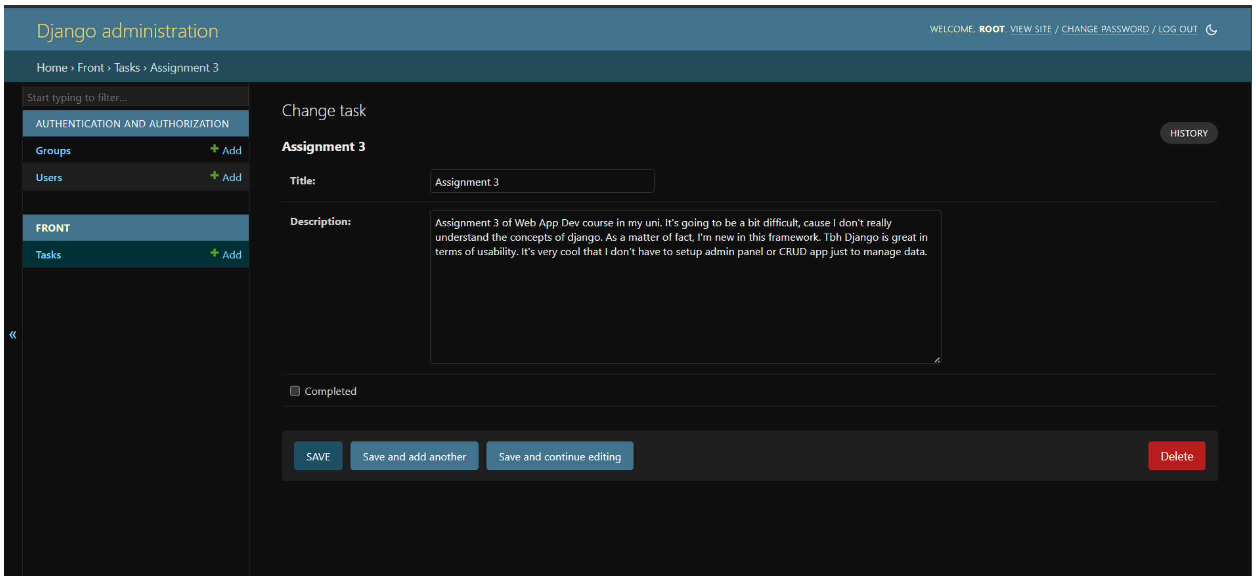
Home › Front › Tasks

Select task to change

Action: --------- Go 0 of 6 selected

TASK
Article writing
Soft QA Midterm preparation
Unity Lab 5 - Body Tracking
Assignment 3
Help Michael
Make Sandwiches

6 tasks


Django administration

WELCOME, ROOT. VIEW SITE / CHANGE PASSWORD / LOG OUT

Home › Front › Tasks › Assignment 3

Change task

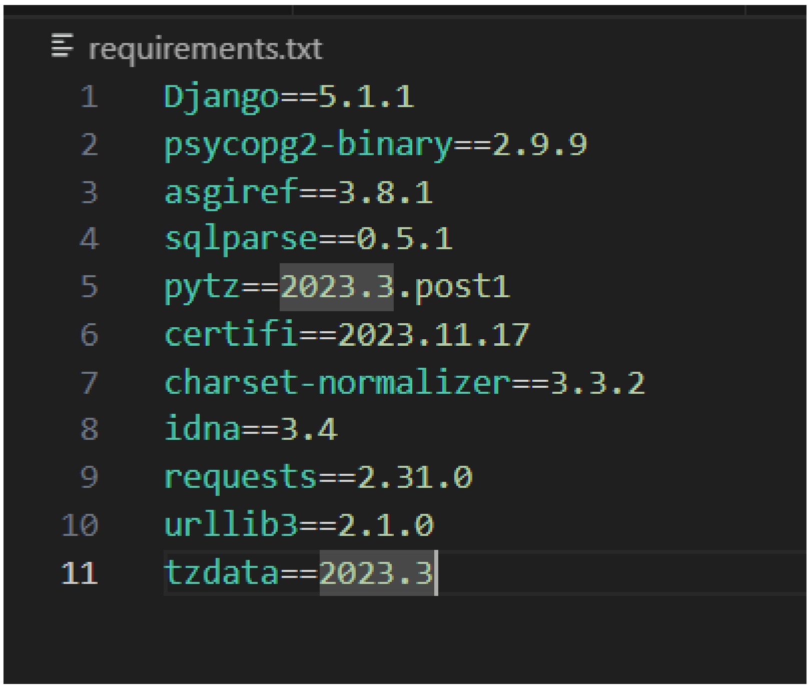
Assignment 3

Title: Assignment 3

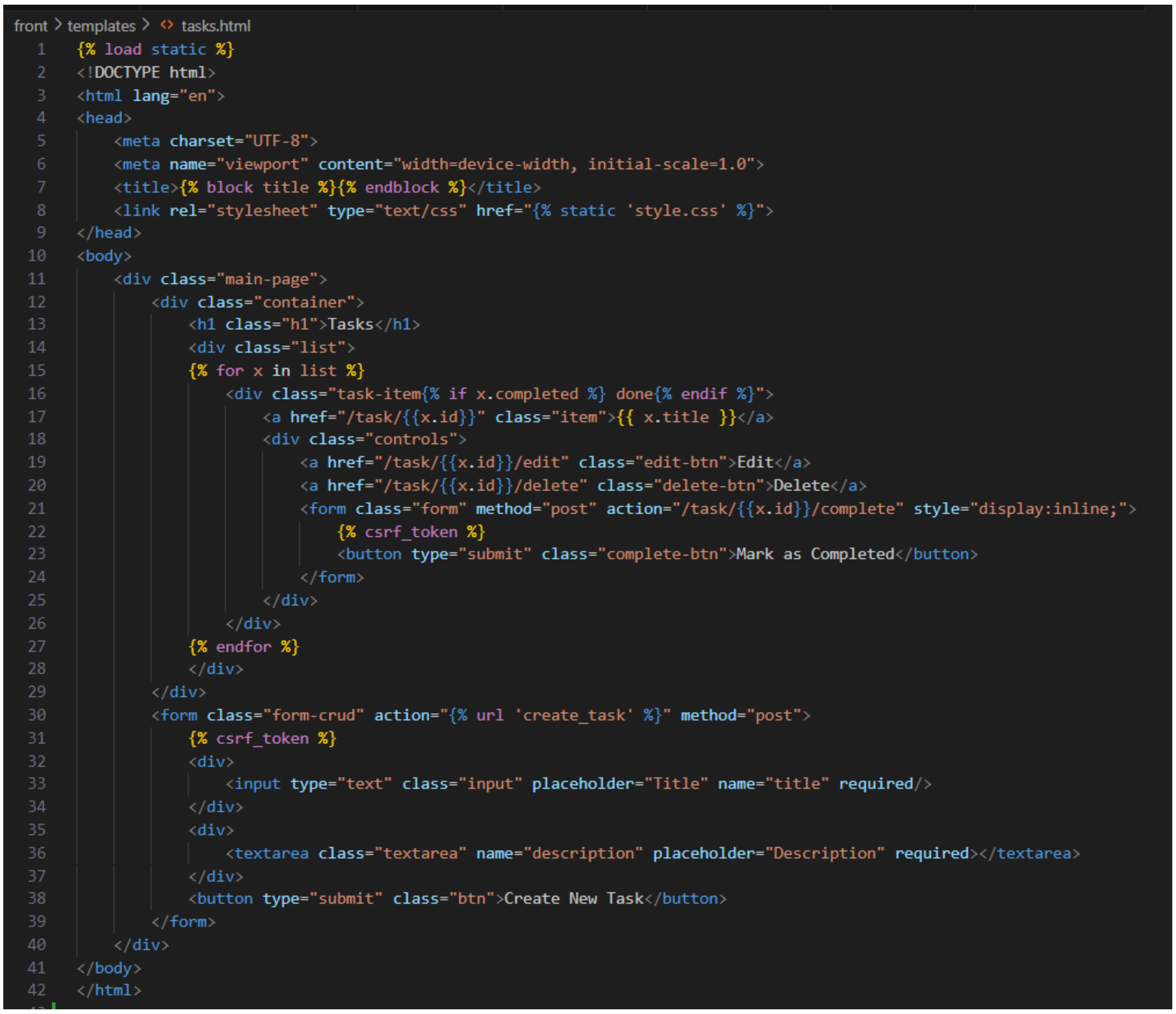
Description: Assignment 3 of Web App Dev course in my uni. It's going to be a bit difficult, cause I don't really understand the concepts of django. As a matter of fact, I'm new in this framework. Tbh Django is great in terms of usability. It's very cool that I don't have to setup admin panel or CRUD app just to manage data.

Completed

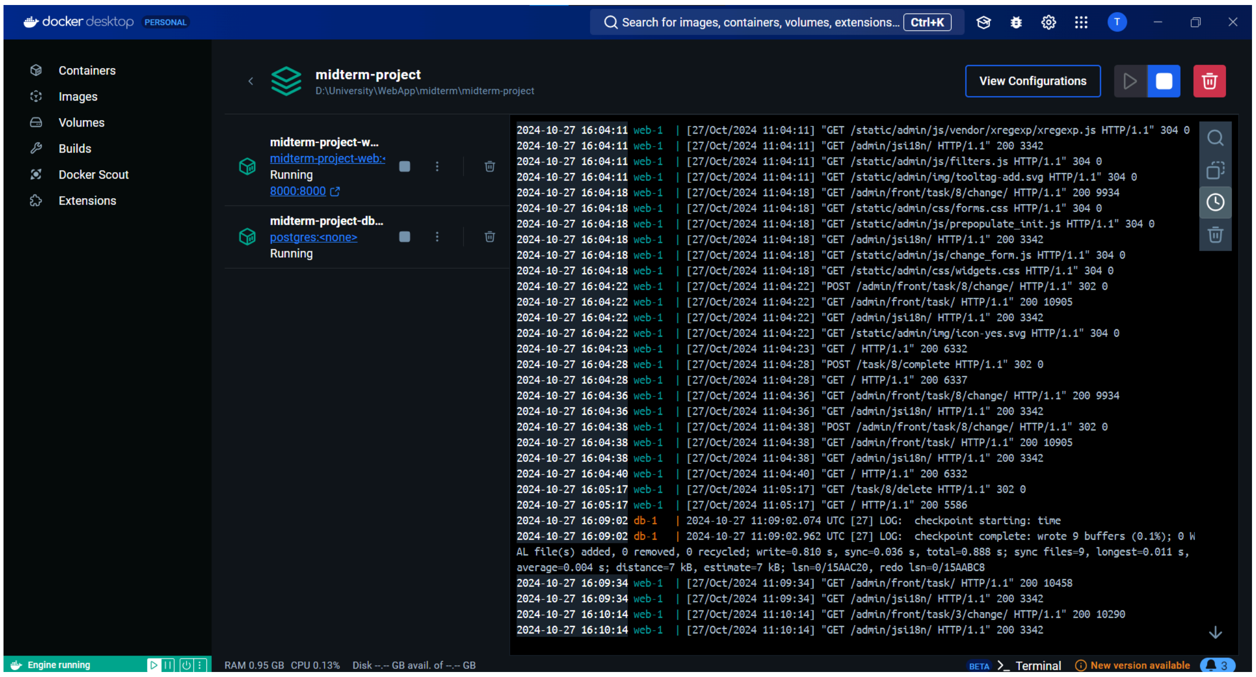
Start typing to filter...
AUTHENTICATION AND AUTHORIZATION
Groups + Add
Users + Add
FRONT
Tasks + Add
HISTORY
SAVE
Save and add another
Save and continue editing
Delete


requirements.txt

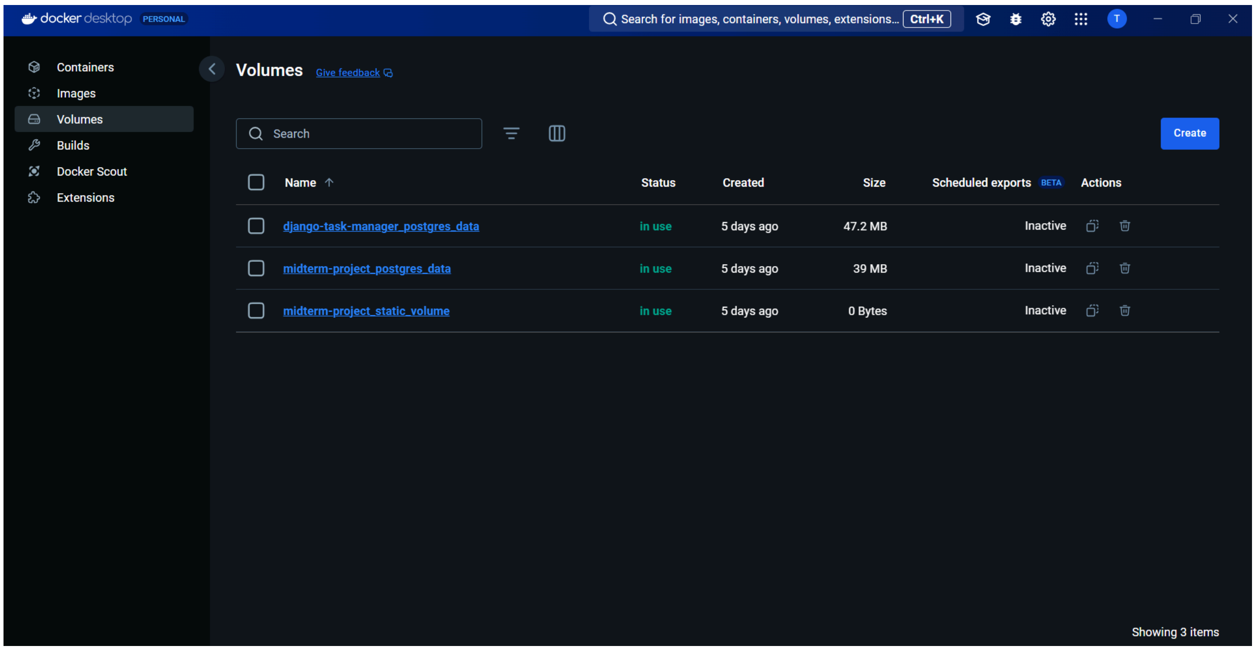
```
Django==5.1.1
psycopg2-binary==2.9.9
asgiref==3.8.1
sqlparse==0.5.1
pytz==2023.3.post1
certifi==2023.11.17
charset-normalizer==3.3.2
idna==3.4
requests==2.31.0
urllib3==2.1.0
tzdata==2023.3
```


front > templates > tasks.html

```
{% load static %}
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>{% block title %}{% endblock %}</title>
    <link rel="stylesheet" type="text/css" href="{% static 'style.css' %}">
</head>
<body>
    <div class="main-page">
        <div class="container">
            <h1 class="h1">Tasks</h1>
            <div class="list">
            {% for x in list %}
                <div class="task-item{% if x.completed %} done{% endif %}">
                    <a href="/task/{{x.id}}" class="item">{{ x.title }}</a>
                    <div class="controls">
                        <a href="/task/{{x.id}}/edit" class="edit-btn">Edit</a>
                        <a href="/task/{{x.id}}/delete" class="delete-btn">Delete</a>
                        <form class="form" method="post" action="/task/{{x.id}}/complete" style="display:inline;">
                            {% csrf_token %}
                            <button type="submit" class="complete-btn">Mark as Completed</button>
                        </form>
                    </div>
                </div>
            {% endfor %}
            </div>
        </div>
        <form class="form-crud" action="{% url 'create_task' %}" method="post">
            {% csrf_token %}
            <div>
                <input type="text" class="input" placeholder="Title" name="title" required/>
            </div>
            <div>
                <textarea class="textarea" name="description" placeholder="Description" required></textarea>
            </div>
            <button type="submit" class="btn">Create New Task</button>
        </form>
    </div>
</body>
</html>
```

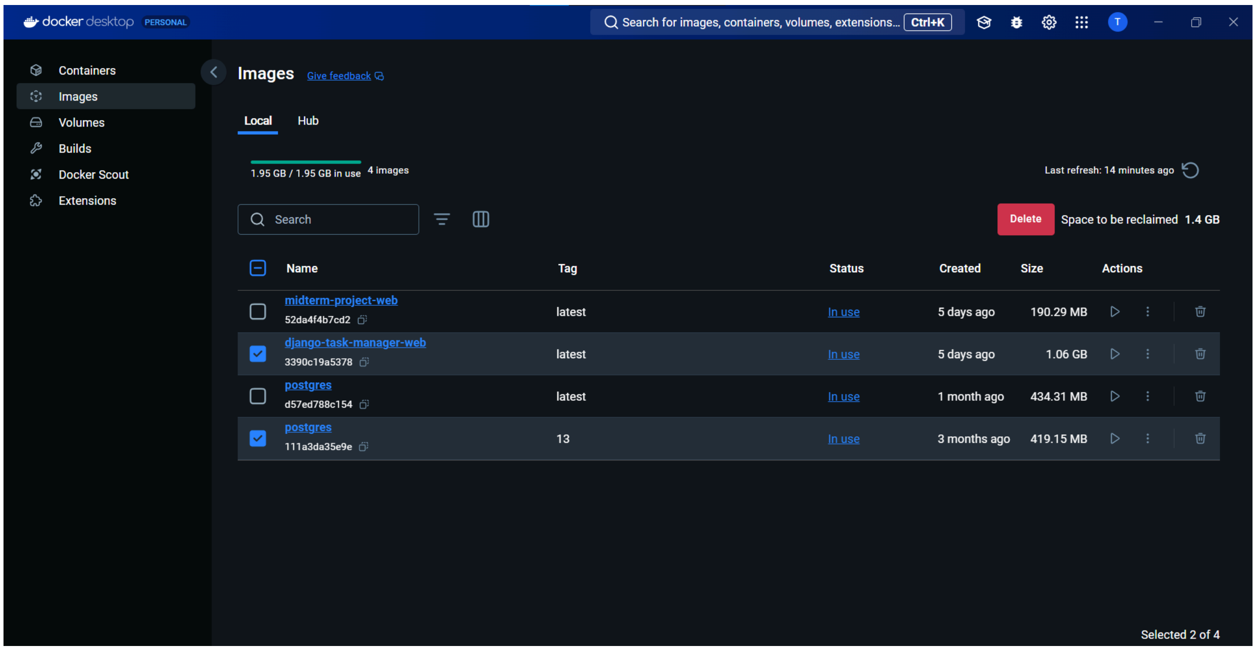


```
2024-10-27 16:04:11 web-1  | [27/Oct/2024 11:04:11] "GET /static/admin/js/vendor/xregexp/xregexp.js HTTP/1.1" 304 0
2024-10-27 16:04:11 web-1  | [27/Oct/2024 11:04:11] "GET /admin/jsi18n/ HTTP/1.1" 200 3342
2024-10-27 16:04:11 web-1  | [27/Oct/2024 11:04:11] "GET /static/admin/js/filters.js HTTP/1.1" 304 0
2024-10-27 16:04:11 web-1  | [27/Oct/2024 11:04:11] "GET /static/admin/img/tooltag-add.svg HTTP/1.1" 304 0
2024-10-27 16:04:18 web-1  | [27/Oct/2024 11:04:18] "GET /admin/front/task/8/change/ HTTP/1.1" 200 9934
2024-10-27 16:04:18 web-1  | [27/Oct/2024 11:04:18] "GET /static/admin/css/forms.css HTTP/1.1" 304 0
2024-10-27 16:04:18 web-1  | [27/Oct/2024 11:04:18] "GET /static/admin/js/prepopulate_init.js HTTP/1.1" 304 0
2024-10-27 16:04:18 web-1  | [27/Oct/2024 11:04:18] "GET /admin/jsi18n/ HTTP/1.1" 200 3342
2024-10-27 16:04:18 web-1  | [27/Oct/2024 11:04:18] "GET /static/admin/js/change_form.js HTTP/1.1" 304 0
2024-10-27 16:04:18 web-1  | [27/Oct/2024 11:04:18] "GET /static/admin/css/widgets.css HTTP/1.1" 304 0
2024-10-27 16:04:22 web-1  | [27/Oct/2024 11:04:22] "POST /admin/front/task/8/change/ HTTP/1.1" 302 0
2024-10-27 16:04:22 web-1  | [27/Oct/2024 11:04:22] "GET /admin/front/task/ HTTP/1.1" 200 10905
2024-10-27 16:04:22 web-1  | [27/Oct/2024 11:04:22] "GET /admin/jsi18n/ HTTP/1.1" 200 3342
2024-10-27 16:04:22 web-1  | [27/Oct/2024 11:04:22] "GET /static/admin/img/icon-yes.svg HTTP/1.1" 304 0
2024-10-27 16:04:23 web-1  | [27/Oct/2024 11:04:23] "GET / HTTP/1.1" 200 6332
2024-10-27 16:04:28 web-1  | [27/Oct/2024 11:04:28] "POST /task/8/complete HTTP/1.1" 302 0
2024-10-27 16:04:28 web-1  | [27/Oct/2024 11:04:28] "GET / HTTP/1.1" 200 6337
2024-10-27 16:04:36 web-1  | [27/Oct/2024 11:04:36] "GET /admin/front/task/8/change/ HTTP/1.1" 200 9934
2024-10-27 16:04:36 web-1  | [27/Oct/2024 11:04:36] "GET /admin/jsi18n/ HTTP/1.1" 200 3342
2024-10-27 16:04:38 web-1  | [27/Oct/2024 11:04:38] "POST /admin/front/task/8/change/ HTTP/1.1" 302 0
2024-10-27 16:04:38 web-1  | [27/Oct/2024 11:04:38] "GET /admin/front/task/ HTTP/1.1" 200 10905
2024-10-27 16:04:38 web-1  | [27/Oct/2024 11:04:38] "GET /admin/jsi18n/ HTTP/1.1" 200 3342
2024-10-27 16:04:40 web-1  | [27/Oct/2024 11:04:40] "GET / HTTP/1.1" 200 6332
2024-10-27 16:05:17 web-1  | [27/Oct/2024 11:05:17] "GET /task/8/delete HTTP/1.1" 302 0
2024-10-27 16:05:17 web-1  | [27/Oct/2024 11:05:17] "GET / HTTP/1.1" 200 5586
2024-10-27 16:09:02 db-1   | 2024-10-27 11:09:02.074 UTC [27] LOG:  checkpoint starting: time
2024-10-27 16:09:02 db-1   | 2024-10-27 11:09:02.962 UTC [27] LOG:  checkpoint complete: wrote 9 buffers (0.1%); 0 W
AL file(s) added, 0 removed, 0 recycled; write=0.810 s, sync=0.036 s, total=0.888 s; sync files=9, longest=0.011 s,
average=0.004 s; distance=7 kB, estimate=7 kB; lsn=0/15AAC20, redo lsn=0/15AABC8
2024-10-27 16:09:34 web-1  | [27/Oct/2024 11:09:34] "GET /admin/front/task/ HTTP/1.1" 200 10458
2024-10-27 16:09:34 web-1  | [27/Oct/2024 11:09:34] "GET /admin/jsi18n/ HTTP/1.1" 200 3342
2024-10-27 16:10:14 web-1  | [27/Oct/2024 11:10:14] "GET /admin/front/task/3/change/ HTTP/1.1" 200 10290
2024-10-27 16:10:14 web-1  | [27/Oct/2024 11:10:14] "GET /admin/jsi18n/ HTTP/1.1" 200 3342
```



Name	Status	Created	Size	Scheduled exports BETA	Actions
django-task-manager_postgres_data	in use	5 days ago	47.2 MB	Inactive	
midterm-project_postgres_data	in use	5 days ago	39 MB	Inactive	
midterm-project_static_volume	in use	5 days ago	0 Bytes	Inactive	

Showing 3 items



Images

1.95 GB / 1.95 GB in use 4 images

Last refresh: 14 minutes ago

Space to be reclaimed 1.4 GB

	Name	Tag	Status	Created	Size	Actions
☐	midterm-project-web 52da4f4b7cd2	latest	In use	5 days ago	190.29 MB	
☑	django-task-manager-web 3390c19a5378	latest	In use	5 days ago	1.06 GB	
☐	postgres d57ed788c154	latest	In use	1 month ago	434.31 MB	
☑	postgres 111a3da35e9e	13	In use	3 months ago	419.15 MB	

Selected 2 of 4