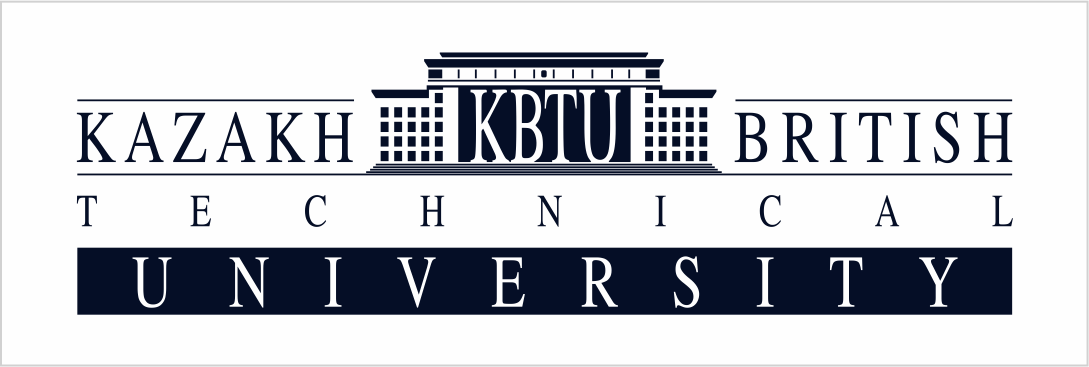
****

**Midterm**

**Building a task management application using Django and Docker**

**Prepared by Ibragimov Temirlan**

**Almaty, 26.10.2024**

**Table of contents**

[**Executive summary 3**](#_heading=h.xz0zs8pje8xc)

[**Introduction 3**](#_heading=h.we7gxlo37vra)

[**Project objectives 3**](#_heading=h.o9ljx6k700xp)

[**Intro to containerization: Docker 3**](#_heading=h.rs4dt3gh6v2n)

[**Creating a Dockerfile 4**](#_heading=h.xrvfh737h1i9)

[**Using Docker Compose 6**](#_heading=h.81o1xr3hq2ib)

[Services 8](#_heading=h.4v80104ddavm)

[Networks 8](#_heading=h.y5qwlx8rmj6e)

[**Docker networking and volumes 8**](#_heading=h.wwzddz7i0ryh)

[**Defining Django models 9**](#_heading=h.bqnn6a9evoo8)

[**Creating views 9**](#_heading=h.vup5mz7fc8mu)

[**Conclusion 15**](#_heading=h.a2alh37606l9)

[**References 16**](#_heading=h.6efb3vdwb30q)

# 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

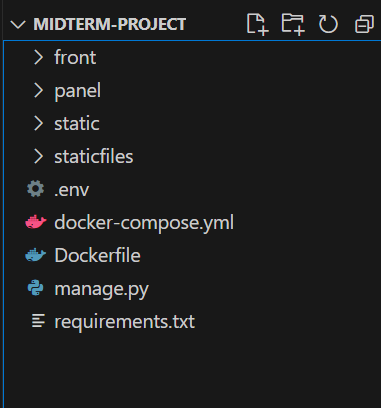


Figure 2. Project structure

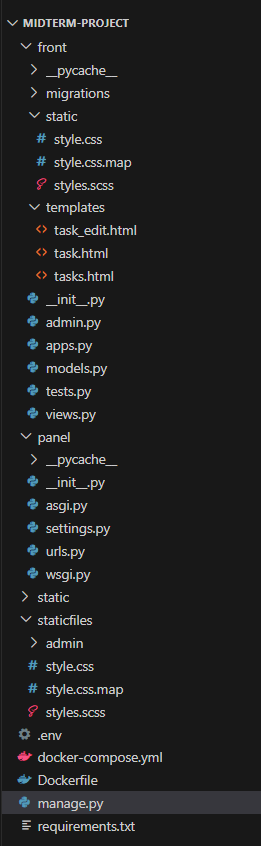


Figure 3. Full project structure

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.

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.
  1. **ports**: Maps port 8000 on the host to port 8000 in the container, making the Django application accessible locally.
  2. **environment**: Sets environment variables for the database connection, which Django uses to connect to PostgreSQL.
  3. **depends\_on**: Ensures the PostgreSQL service starts before Django.
  4. **networks**: Connects the Django app to the django\_network, allowing it to communicate with db.

1. **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.

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

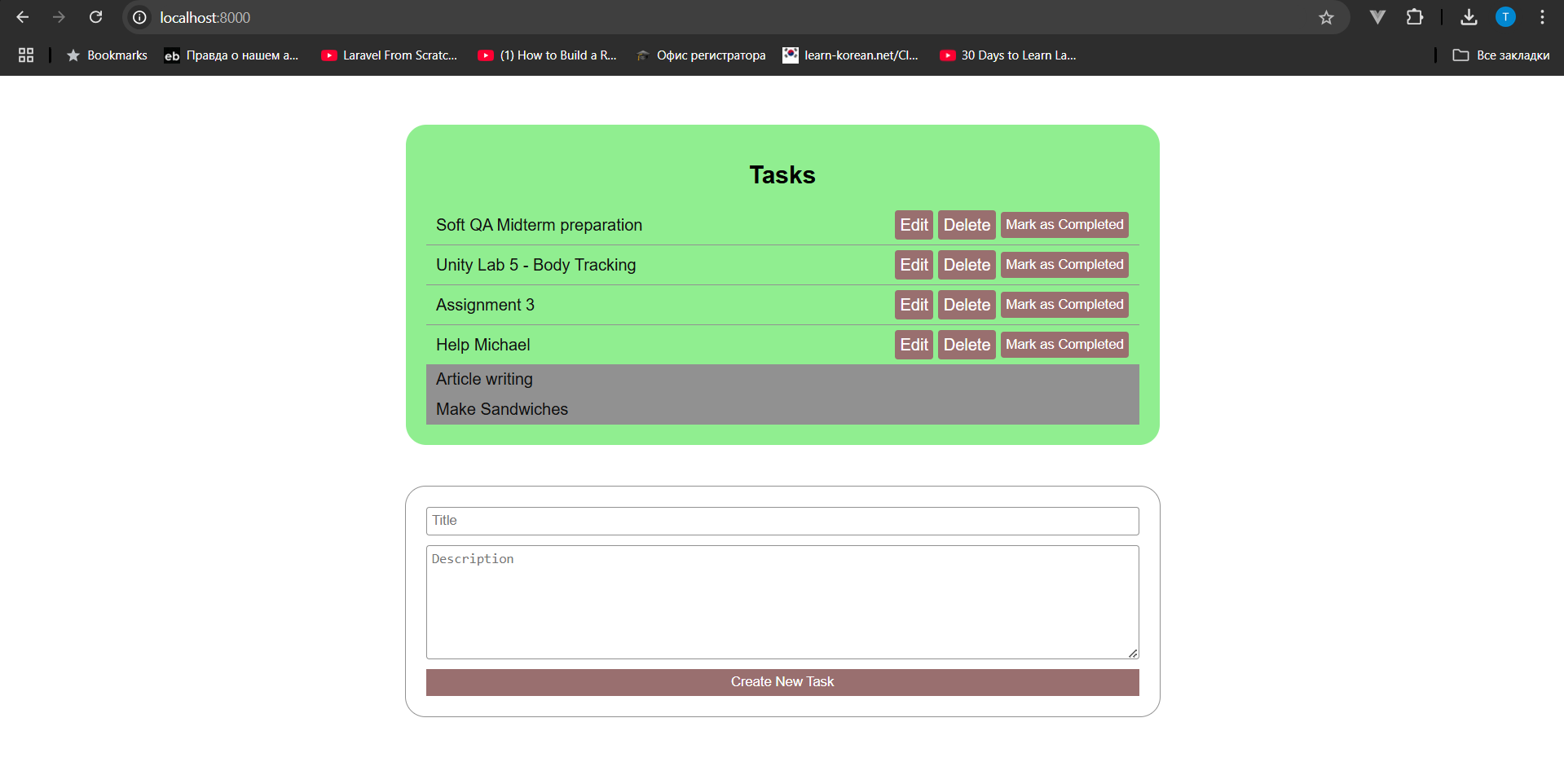


Figure 6. Main page

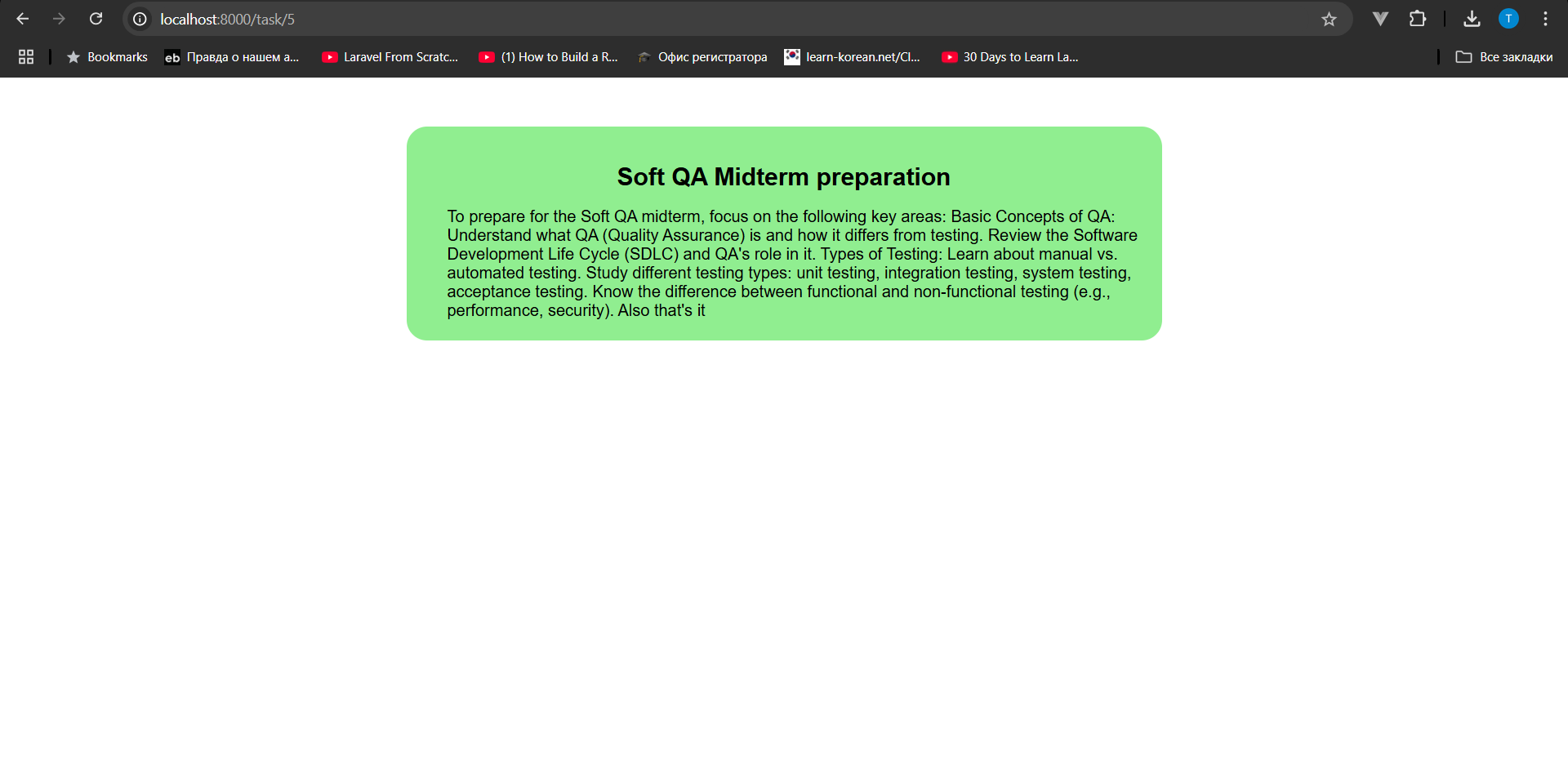


Figure 7. Task page

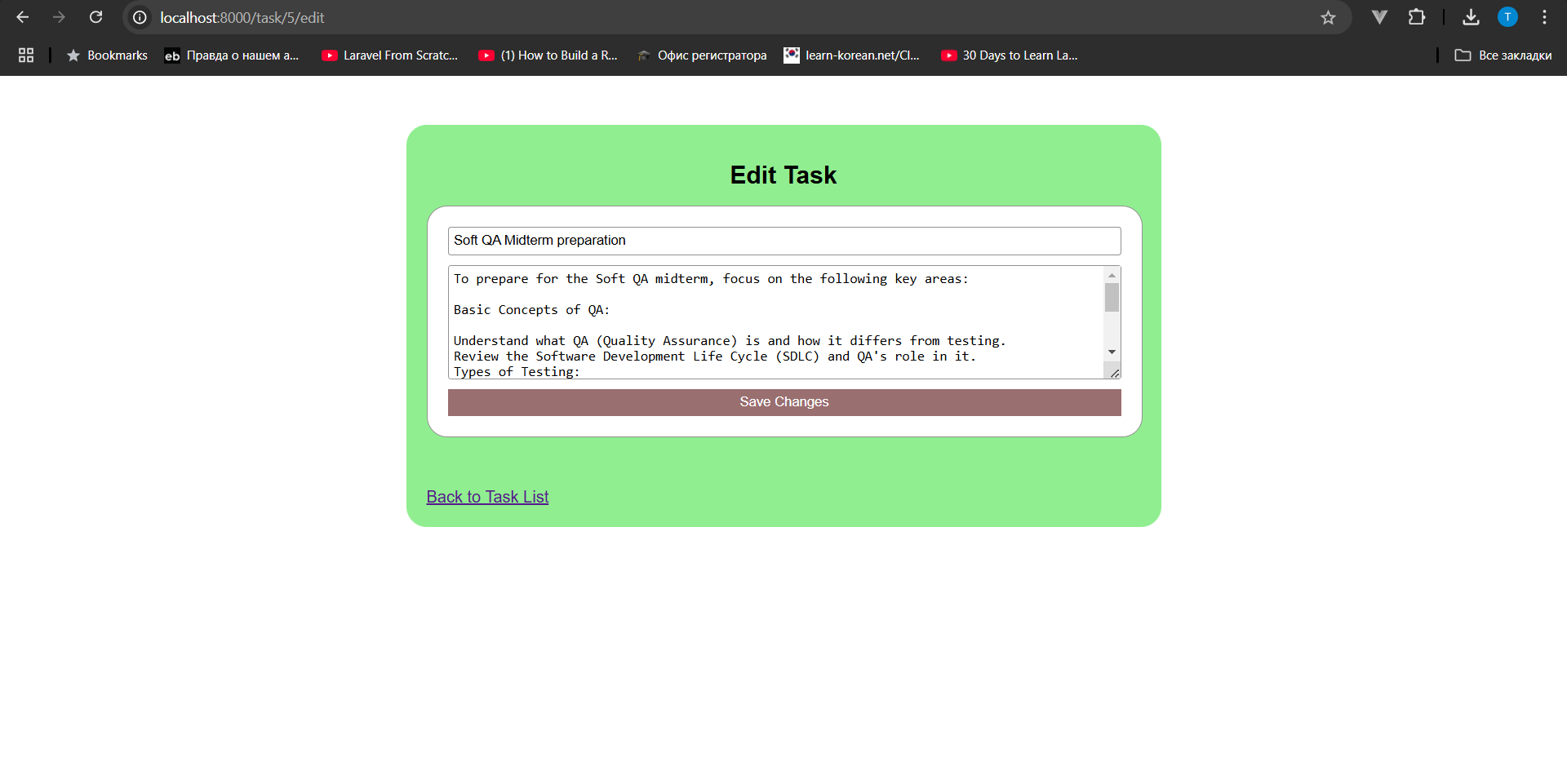


Figure 8. Task edit page

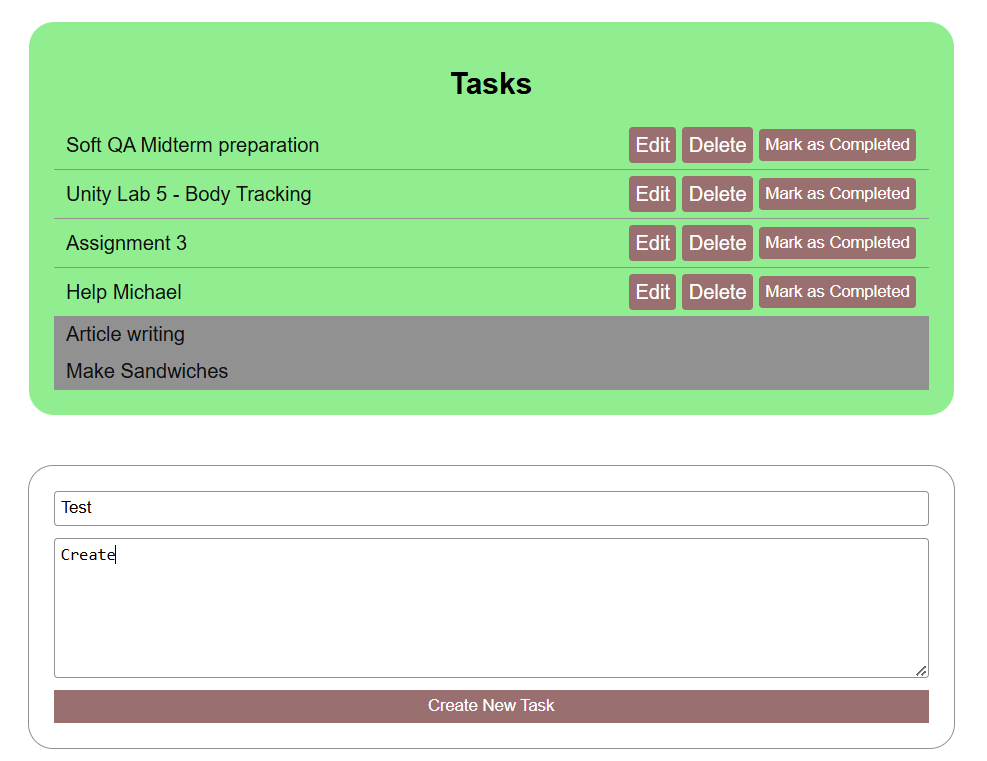


Figure 9. Creating a new task

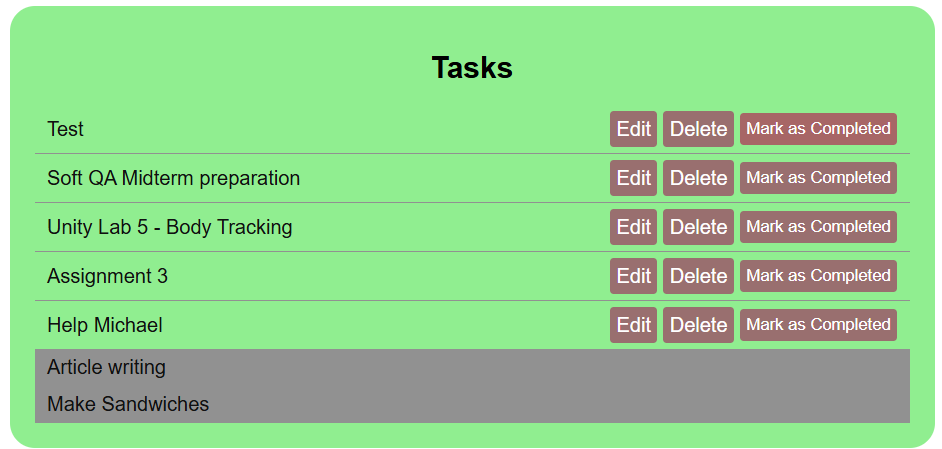


Figure 10. Marking the new task as completed

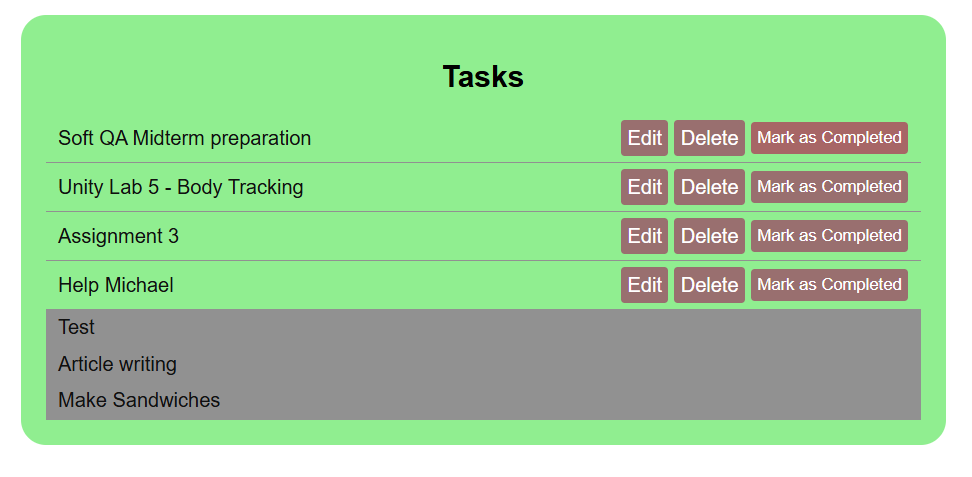


Figure 11. Final result



Figure 12. Deleting a task

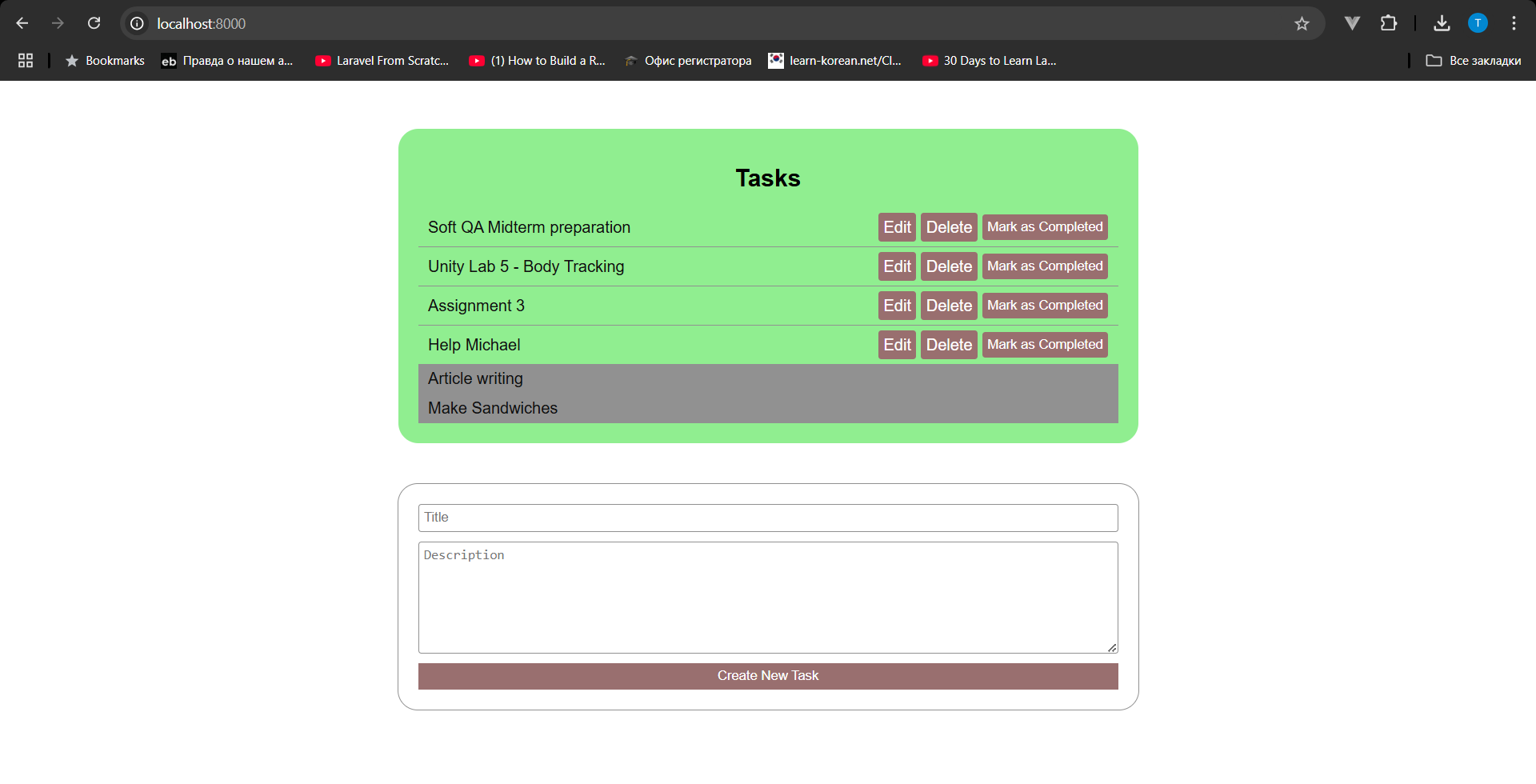


Figure 13. Result

# 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**

