Web development

Midterm Project: Building a Task Management Application Using Django and Docker

Bitanov Assanali (23MD0392)

24.10.2024

Table of Contents

Executive Summary

This project involves developing a task management application using Django and deploying it in a Docker container. The 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.

Introduction

In this midterm project, we are going to practice our development skills in order to build an app with basic task management pages using Django and PostgresSQL inside containers. The topics that are covered within the project Containerization with Docker, Dockerfile, Docker-compose and Django. The pages that we need to build are Task List page, Create Task page, Edit Task page, and Delete task page.

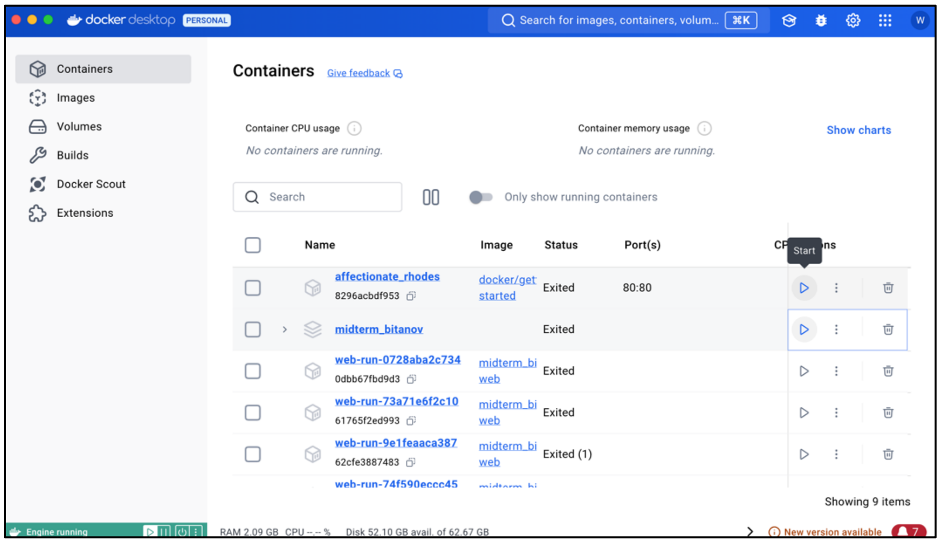
Project Objectives

- Develop a functional task management web application using Django.
- Practice with Docker, including containerization, networking, and volumes.
- Set up and configure a PostgreSQL database inside a Docker container.
- Create and manage Django models to store and handle task data.
- 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.



Container CPU usage
No containers are running.

Container memory usage
No containers are running.

Name	Image	Status	Port(s)
affectionate_rhodes 8296acbdf953	docker/get started	Exited	80:80
midterm_bitanov		Exited	
web-run-0728aba2c734 0dbb67fbd9d3	midterm_bi web	Exited	
web-run-73a71e6f2c10 61765f2ed993	midterm_bi web	Exited	
web-run-9e1feaaca387 62cfe3887483	midterm_bi web	Exited (1)	
web-run-74f590eccc45	midterm_bi		

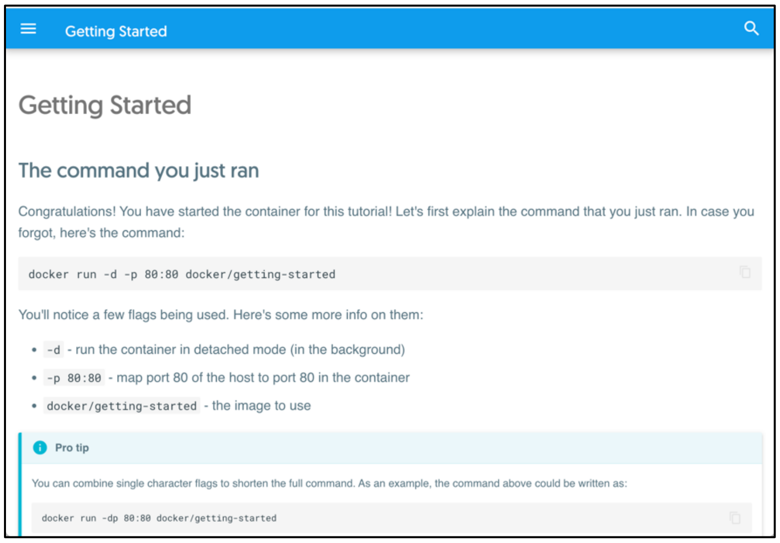
Showing 9 items

RAM 2.09 GB CPU --.-- % Disk 52.10 GB avail. of 62.67 GB

New version available 7

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.



Getting Started

The command you just ran

Congratulations! You have started the container for this tutorial! Let's first explain the command that you just ran. In case you forgot, here's the command:

```
docker run -d -p 80:80 docker/getting-started
```

You'll notice a few flags being used. Here's some more info on them:

- `-d` - run the container in detached mode (in the background)
- `-p 80:80` - map port 80 of the host to port 80 in the container
- `docker/getting-started` - the image to use

Pro tip

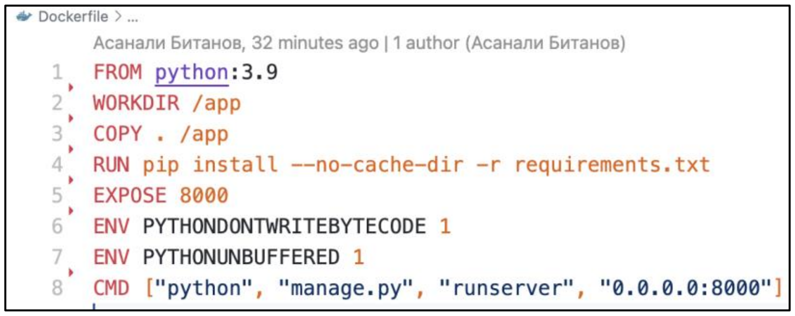
You can combine single character flags to shorten the full command. As an example, the command above could be written as:

```
docker run -dp 80:80 docker/getting-started
```

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)

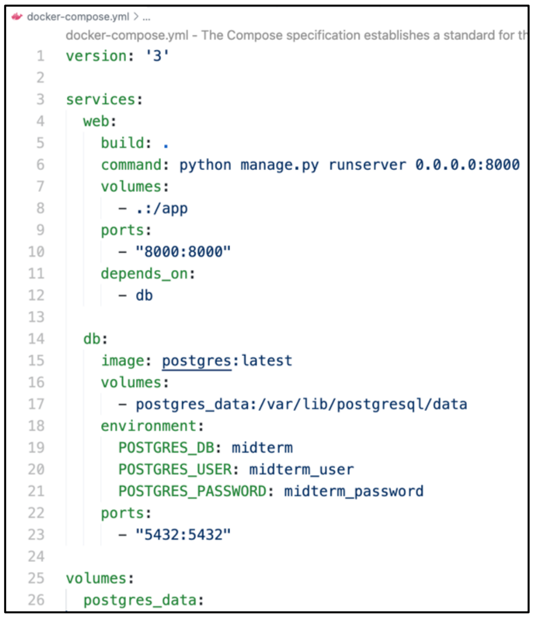
```
FROM python:3.9
WORKDIR /app
COPY . /app
RUN pip install --no-cache-dir -r requirements.txt
EXPOSE 8000
ENV PYTHONDONTWRITEBYTECODE 1
ENV PYTHONUNBUFFERED 1
CMD ["python", "manage.py", "runserver", "0.0.0.0:8000"]
```

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

```yaml
version: '3'

services:
  web:
    build: .
    command: python manage.py runserver 0.0.0.0:8000
    volumes:
      - .:/app
    ports:
      - "8000:8000"
    depends_on:
      - db

  db:
    image: postgres:latest
    volumes:
      - postgres_data:/var/lib/postgresql/data
    environment:
      POSTGRES_DB: midterm
      POSTGRES_USER: midterm_user
      POSTGRES_PASSWORD: midterm_password
    ports:
      - "5432:5432"

volumes:
  postgres_data:
```

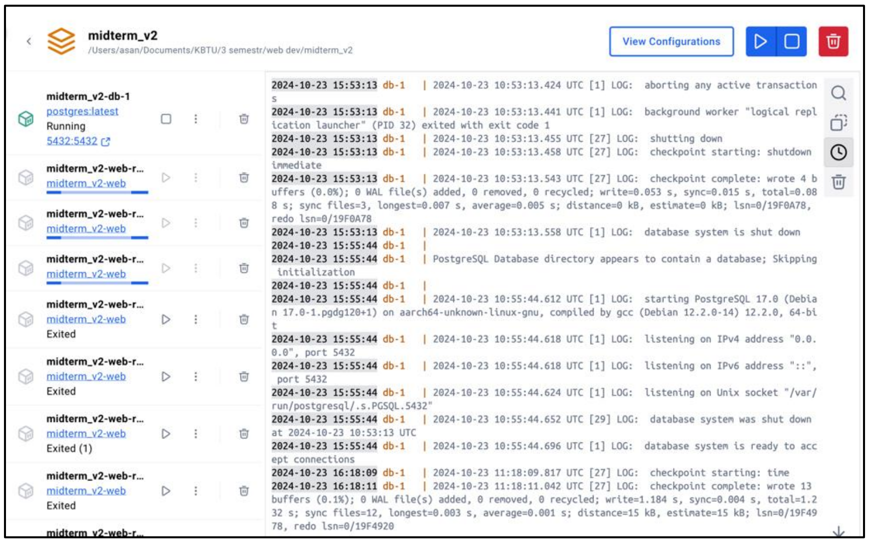
Figure 4. The docker-compose

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.



```
2024-10-23 15:53:13 db-1   | 2024-10-23 10:53:13.424 UTC [1] LOG:  aborting any active transaction
s
2024-10-23 15:53:13 db-1   | 2024-10-23 10:53:13.441 UTC [1] LOG:  background worker "logical repl
ication launcher" (PID 32) exited with exit code 1
2024-10-23 15:53:13 db-1   | 2024-10-23 10:53:13.455 UTC [27] LOG:  shutting down
2024-10-23 15:53:13 db-1   | 2024-10-23 10:53:13.458 UTC [27] LOG:  checkpoint starting: shutdown
immediate
2024-10-23 15:53:13 db-1   | 2024-10-23 10:53:13.543 UTC [27] LOG:  checkpoint complete: wrote 4 b
uffers (0.0%); 0 WAL file(s) added, 0 removed, 0 recycled; write=0.053 s, sync=0.015 s, total=0.08
8 s; sync files=3, longest=0.007 s, average=0.005 s; distance=0 kB, estimate=0 kB; lsn=0/19F0A78,
redo lsn=0/19F0A78
2024-10-23 15:53:13 db-1   | 2024-10-23 10:53:13.558 UTC [1] LOG:  database system is shut down
2024-10-23 15:55:44 db-1   |
2024-10-23 15:55:44 db-1   | PostgreSQL Database directory appears to contain a database; Skipping
 initialization
2024-10-23 15:55:44 db-1   |
2024-10-23 15:55:44 db-1   | 2024-10-23 10:55:44.612 UTC [1] LOG:  starting PostgreSQL 17.0 (Debia
n 17.0-1.pgdg120+1) on aarch64-unknown-linux-gnu, compiled by gcc (Debian 12.2.0-14) 12.2.0, 64-bi
t
2024-10-23 15:55:44 db-1   | 2024-10-23 10:55:44.618 UTC [1] LOG:  listening on IPv4 address "0.0.
0.0", port 5432
2024-10-23 15:55:44 db-1   | 2024-10-23 10:55:44.618 UTC [1] LOG:  listening on IPv6 address "::",
 port 5432
2024-10-23 15:55:44 db-1   | 2024-10-23 10:55:44.624 UTC [1] LOG:  listening on Unix socket "/var/
run/postgresql/.s.PGSQL.5432"
2024-10-23 15:55:44 db-1   | 2024-10-23 10:55:44.652 UTC [29] LOG:  database system was shut down
at 2024-10-23 10:53:13 UTC
2024-10-23 15:55:44 db-1   | 2024-10-23 10:55:44.696 UTC [1] LOG:  database system is ready to acc
ept connections
2024-10-23 16:18:09 db-1   | 2024-10-23 11:18:09.817 UTC [27] LOG:  checkpoint starting: time
2024-10-23 16:18:11 db-1   | 2024-10-23 11:18:11.042 UTC [27] LOG:  checkpoint complete: wrote 13
buffers (0.1%); 0 WAL file(s) added, 0 removed, 0 recycled; write=1.184 s, sync=0.004 s, total=1.2
32 s; sync files=12, longest=0.003 s, average=0.001 s; distance=15 kB, estimate=15 kB; lsn=0/19F49
78, redo lsn=0/19F4920
```

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.

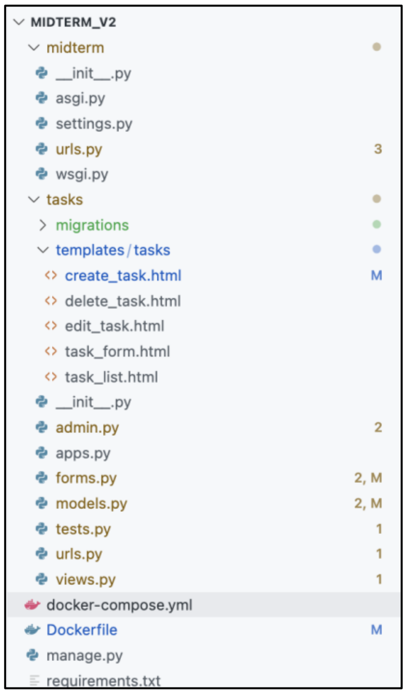
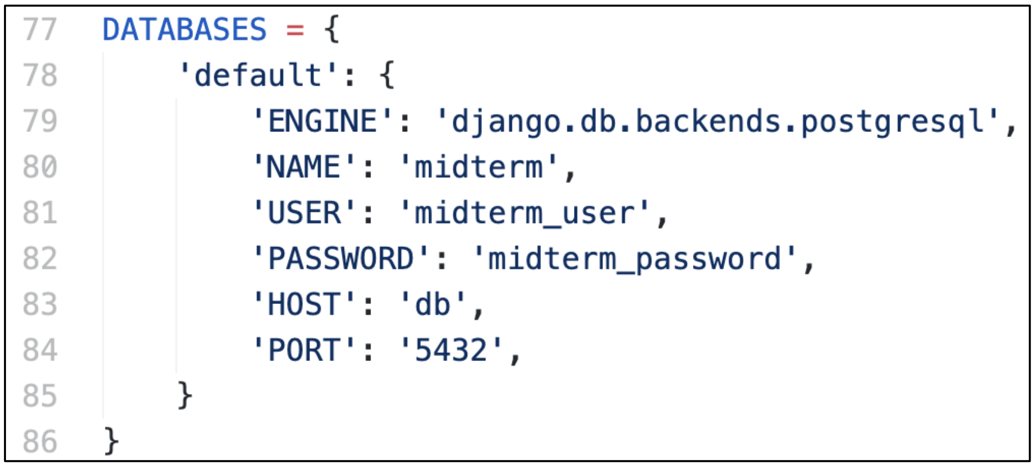


Figure 6. The structure of the Midterm_v2 project in VSCode

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.



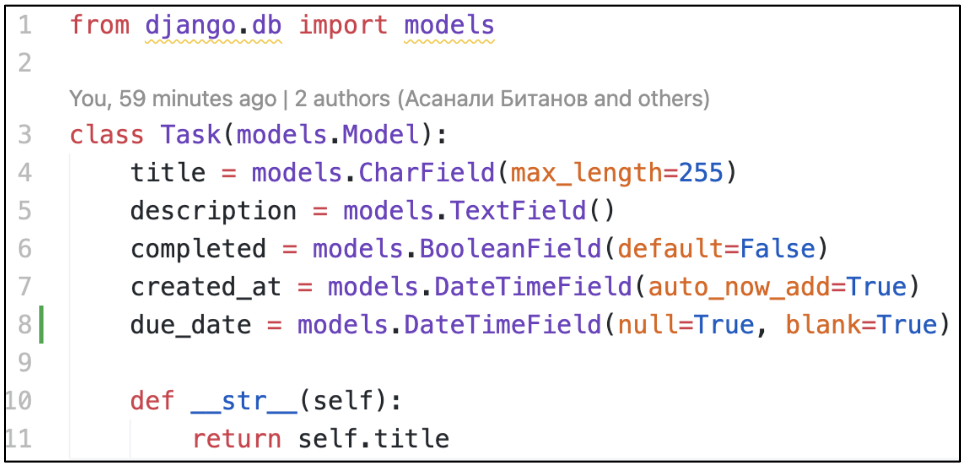
```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'midterm',
        'USER': 'midterm_user',
        'PASSWORD': 'midterm_password',
        'HOST': 'db',
        'PORT': '5432',
    }
}
```

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.



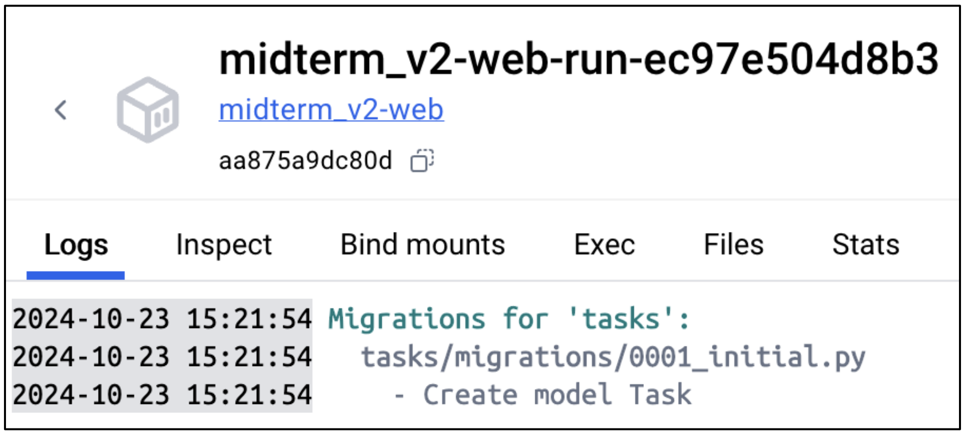
```python
from django.db import models

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

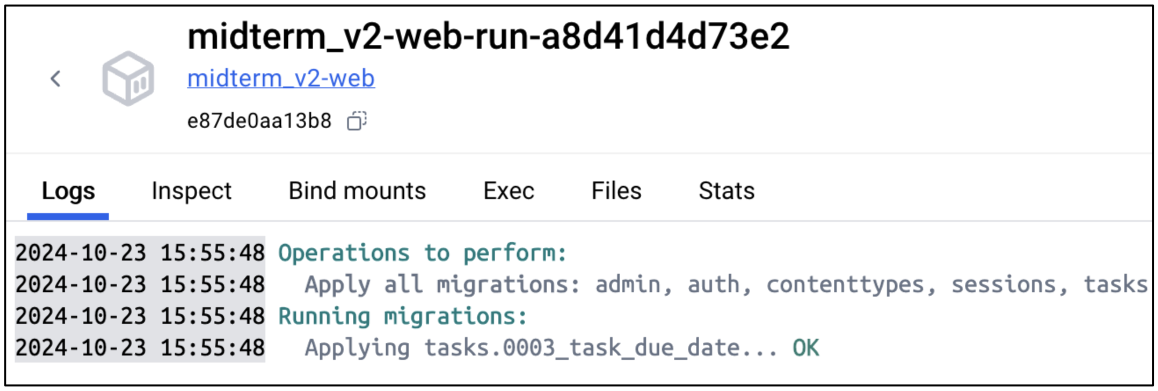
Each field represents a column in the database, and Django automatically handles the conversion of these fields into database columns. This model serves as the foundation for managing tasks within the application. After defining the Task model, we created migrations using the command python manage.py 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.



```
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

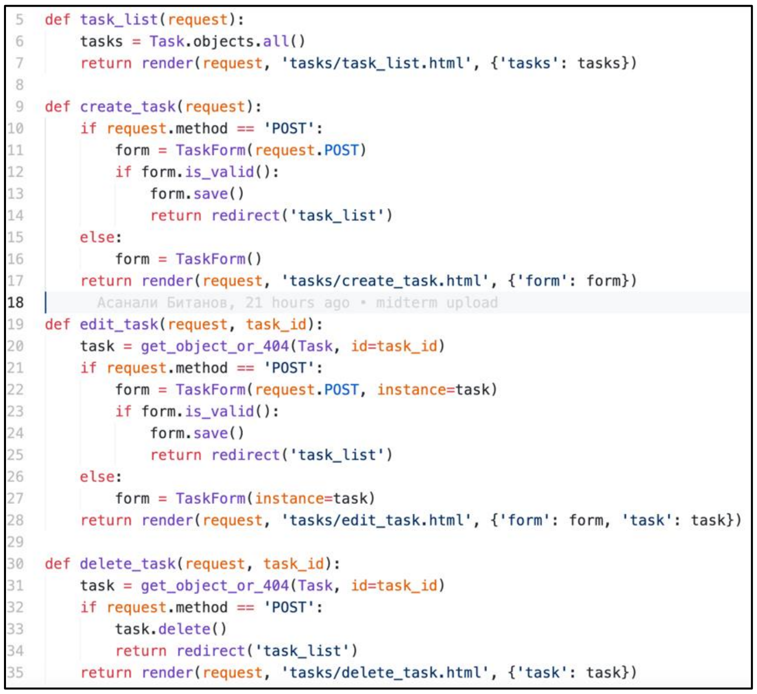
To apply these migrations and set up the database schema, we ran python manage.py migrate. This command applies the changes defined in the migration files to the PostgreSQL database, creating the necessary tables and columns to store task data. The output of the command is shown in Figure 9.



```
2024-10-23 15:55:48 Operations to perform:
2024-10-23 15:55:48   Apply all migrations: admin, auth, contenttypes, sessions, tasks
2024-10-23 15:55:48 Running migrations:
2024-10-23 15:55:48   Applying tasks.0003_task_due_date... OK
```

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.



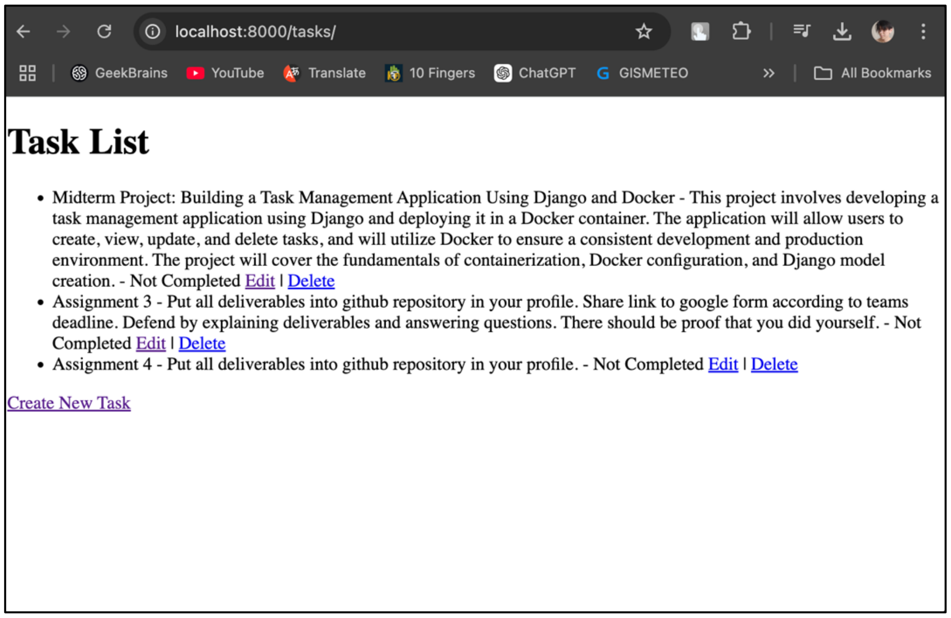
```python
def task_list(request):
    tasks = Task.objects.all()
    return render(request, 'tasks/task_list.html', {'tasks': tasks})

def create_task(request):
    if request.method == 'POST':
        form = TaskForm(request.POST)
        if form.is_valid():
            form.save()
            return redirect('task_list')
    else:
        form = TaskForm()
    return render(request, 'tasks/create_task.html', {'form': form})
      Асанали Битанов, 21 hours ago • midterm upload
def edit_task(request, task_id):
    task = get_object_or_404(Task, id=task_id)
    if request.method == 'POST':
        form = TaskForm(request.POST, instance=task)
        if form.is_valid():
            form.save()
            return redirect('task_list')
    else:
        form = TaskForm(instance=task)
    return render(request, 'tasks/edit_task.html', {'form': form, 'task': task})

def delete_task(request, task_id):
    task = get_object_or_404(Task, id=task_id)
    if request.method == 'POST':
        task.delete()
        return redirect('task_list')
    return render(request, 'tasks/delete_task.html', {'task': task})
```

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.


localhost:8000/tasks/

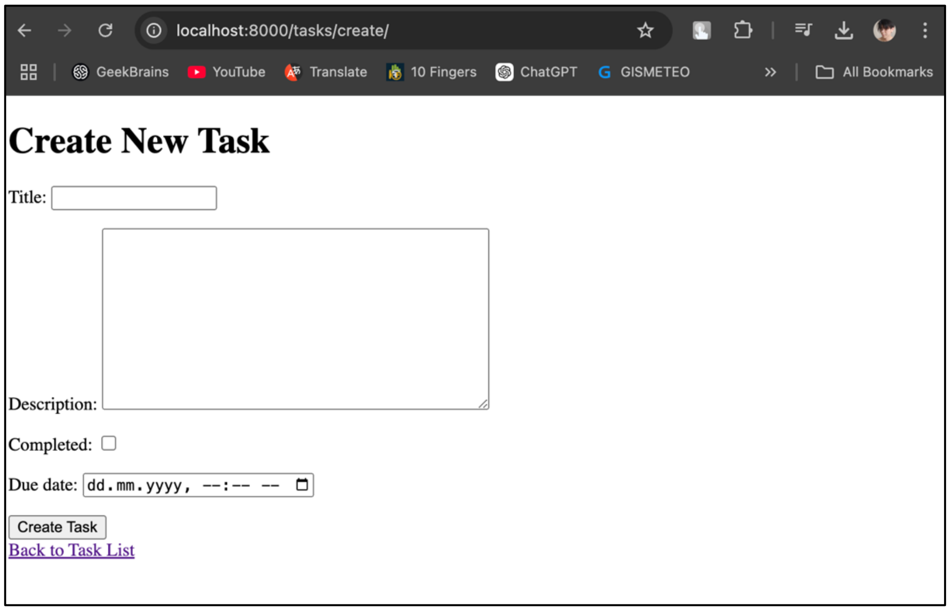
Task List

- Midterm Project: Building a Task Management Application Using Django and Docker - This project involves developing a task 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. - Not Completed Edit | Delete
- Assignment 3 - Put all deliverables into github repository in your profile. Share link to google form according to teams deadline. Defend by explaining deliverables and answering questions. There should be proof that you did yourself. - Not Completed Edit | Delete
- Assignment 4 - Put all deliverables into github repository in your profile. - Not Completed Edit | Delete

Create New Task

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.


localhost:8000/tasks/create/

Create New Task

Title:

Description:

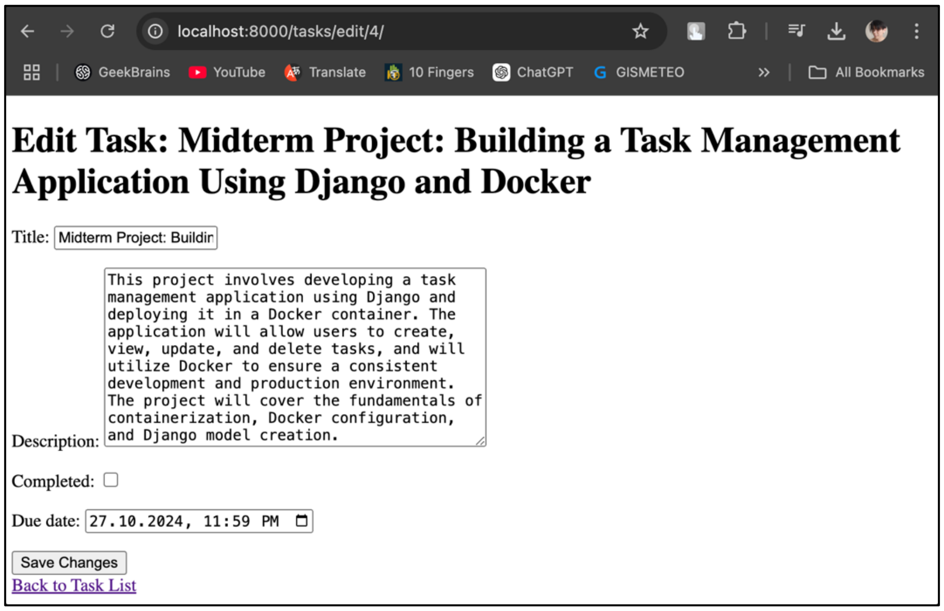
Completed: ☐

Due date: dd.mm.yyyy, --:-- --

Back to Task List

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.



Edit Task: Midterm Project: Building a Task Management Application Using Django and Docker

Title: Midterm Project: Buildir

Description:
```
This project involves developing a task
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.
```

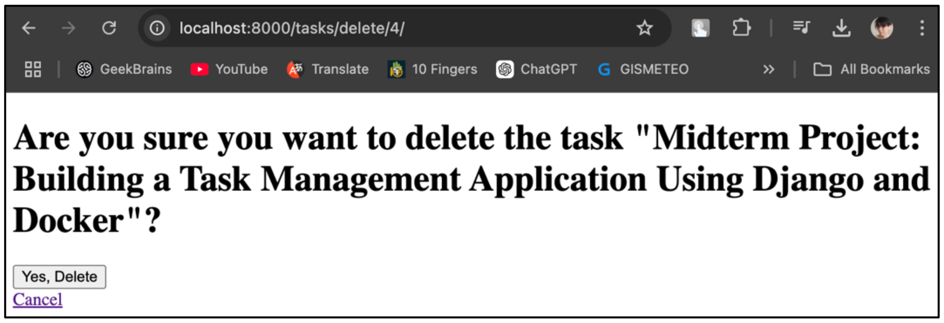
Completed: ☐

Due date: 27.10.2024, 11:59 PM

Save Changes
Back to Task List

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.

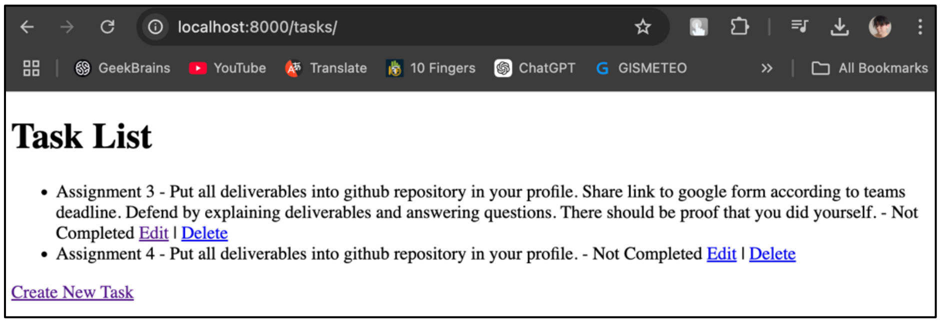


Are you sure you want to delete the task "Midterm Project: Building a Task Management Application Using Django and Docker"?

Yes, Delete
Cancel

Figure 14. The delete confirmation page

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



Task List

- Assignment 3 - Put all deliverables into github repository in your profile. Share link to google form according to teams deadline. Defend by explaining deliverables and answering questions. There should be proof that you did yourself. - Not Completed Edit | Delete
- Assignment 4 - Put all deliverables into github repository in your profile. - Not Completed Edit | Delete

Create New Task

Figure 14. The delete confirmation page

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

Conclusion

In this midterm project, we practiced our development skills in order to build an app with basic task management pages using Django and PostgresSQL inside containers. The topics that are covered within the project are Containerization with Docker, Dockerfile, Docker-compose and Django. We created these pages: Task List page, Create Task page, Edit Task page, and Delete task page. Also, we successfully created a multi-layered container with Django and PostgreSQL images. This experience gave us a valuable skills that we can use in our future projects.

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