

Kazakh-British Technical University

Web Application Development

Midterm

Building a Task Management Application Using Django and Docker

Full Name: Tamiris Abildayeva

ID: 23MD0503

Link to GitHub: <https://github.com/TamirisK/university-web-application-development>

27.10.2024

Table of Contents

Executive Summary	3
Introduction	4
Project Objectives	5
Intro to Containerization: Docker	6
Creating a Dockerfile	7
Using Docker Compose	8
Docker Networking and Volumes	9
Django Application Setup	10
Defining Django Models	11
Conclusion	12
References	13
References Appendices	14

Executive Summary

The task management web application aims to empower users to organize, manage, and monitor their tasks efficiently. Its main **objective** is to provide a user-friendly platform that allows individuals to create, modify, and remove tasks while offering robust features for categorization, searching, and sorting.

Utilizing the Django framework, the application guarantees a strong user authentication and data handling backend. The intuitive user interface is constructed with HTML, CSS, and JavaScript, while a relational database—SQLite—ensures data integrity and accessibility.

Outcomes achieved:

1. To create secure signup and login processes that implement strong password policies to ensure the safety of user information.
2. To develop features that allow users to efficiently manage their tasks, including the ability to add, edit, and delete task details such as titles, descriptions, categories, creation dates, and deadlines.
3. To design an intuitive dashboard that provides users with a comprehensive overview of their tasks, highlighting completed tasks for easy identification.
4. To integrate robust search and sort functionality that enables users to quickly locate and organize tasks based on various criteria, enhancing overall usability.
5. To implement task categorization, allowing users to sort their tasks into specific groups such as “Work,” “Personal,” and “Other” for better organization and management.

Future upgrades will include deadline reminders and a mobile application. There are also plans to create project tasks for multiple users for collaboration.

Introduction

In today's software development world, containerization has become essential for creating applications that run consistently across various environments. By using containers, developers can package an application along with all its dependencies, ensuring it behaves the same whether on a local machine or in the cloud. This approach helps eliminate common issues, such as the infamous "it works on my machine" problem, making collaboration easier and allowing teams to deliver software more reliably.

Docker is a leading containerization platform that simplifies this process, enabling developers to create and manage containers with ease. It streamlines both development and deployment, allowing teams to focus on writing code instead of dealing with infrastructure challenges.

I am motivated to develop a task management application using Django to deepen my understanding of back-end development, as my primary focus has been on front-end development. I want to revisit and strengthen my skills in this area while aiming for a good grade on my midterm exam. With a current GPA of 3.9, I am committed to maintaining strong academic performance, where achieving at least a 90 is essential throughout my studies. This project not only offers a valuable learning opportunity but also helps me meet my academic goals while creating a practical tool for task management.

Project Objectives

1. **Develop “Task management” a web application:** Create a fully operational task management application that allows users to register, log in, and effectively manage their tasks.
2. **Understand Docker's features:** Gain hands-on experience with Docker by containerizing the application to ensure consistent deployment across various environments.
3. **Introduction to containerization Docker:**
 - a. Comprehend the concepts of containerization and its benefits in modern application development.
 - b. Install Docker and set up a simple Docker container to run a basic application.
4. **Create a Dockerfile:** Develop a Dockerfile to define the environment for the Django application, specifying the necessary dependencies, including Python and Django packages.
5. **Utilize Docker Compose:** Use Docker Compose to define and run multi-container applications, creating a `docker-compose.yml` file to manage both the Django application and the database service.
6. **Configure Docker Networking and Volumes:**
 - a. Set up Docker networking to enable communication between containers.
 - b. Use Docker volumes to persist data and manage the state of the database.
7. **Set Up a Django project:** Initialize a Django project and create a basic application structure, configuring the settings to operate seamlessly within the Docker environment.
8. **Define Django Models:** Implement Django models for the task management application, including user and task entities, and execute migrations to create the necessary database tables.
9. **Create user authentication:** Establish secure user authentication processes, including registration and login functionalities that adhere to strong password policies.
10. **Build an intuitive user interface:** Develop a user-friendly front end that enhances the overall user experience, making it easy for users to navigate and manage their tasks.
11. **Incorporate Task Management features:** Implement features that allow users to add, edit, delete, and categorize tasks, as well as mark them as completed.
12. **Integrate search and sort functionality:** Enable users to quickly search for and organize their tasks based on various criteria, improving efficiency in task management.
13. **Test and debug the application** Conduct thorough testing to identify and resolve any bugs or issues, ensuring the application functions smoothly and reliably.

Intro to Containerization: Docker

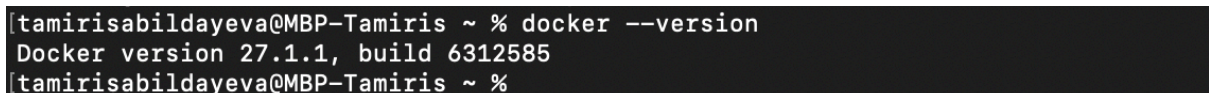
Containerization Overview

Containerization is a technology that allows developers to package an application and its dependencies into a single unit, known as a container. This approach provides numerous benefits, including portability across different environments, isolation of applications for security and stability, and simplified management of dependencies, which reduces the likelihood of conflicts.

Docker Installation

To install Docker, I followed these detailed steps:

1. **Download Docker Desktop:** I visited the official Docker website and downloaded the Docker Desktop installer appropriate for my macOS operating system.
2. **Run the installer:** After downloading, I double-clicked the installer to begin the installation process. I followed the on-screen prompts to complete the installation, making sure to grant any required permissions.
3. **Start Docker:** Once the installation was complete, I launched Docker Desktop. It took a moment to initialize, and I waited until the Docker icon in the system tray indicated that Docker was running.
4. **Verify installation:** To confirm that Docker was installed correctly, I opened a terminal and ran the following command:



```
[tamirisabildayeva@MBP-Tamiris ~ % docker --version  
Docker version 27.1.1, build 6312585  
[tamirisabildayeva@MBP-Tamiris ~ %
```

Fig. 1: Screenshot of the terminal displaying the installed Docker version.

This command displayed the installed version of Docker, confirming that the installation was successful.

5. **Run a Test Container:** Next, I executed a simple test by running the following command:

```
tamirisabildayeva@MBP-Tamiris ~ % docker run hello-world
[Unable to find image 'hello-world:latest' locally]
latest: Pulling from library/hello-world
478afc919002: Pull complete
Digest: sha256:91fb4b041da273d5a3273b6d587d62d518300a6ad268b28628f74997b93171b2
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
1. The Docker client contacted the Docker daemon.
2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
   (arm64v8)
3. The Docker daemon created a new container from that image which runs the
   executable that produces the output you are currently reading.
4. The Docker daemon streamed that output to the Docker client, which sent it
   to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
$ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
https://hub.docker.com/

For more examples and ideas, visit:
https://docs.docker.com/get-started/

tamirisabildayeva@MBP-Tamiris ~ %
```

Fig. 2: Screenshot of the terminal displaying the output of the docker run hello-world command.

This command downloaded the "hello-world" image and ran it in a new container. Upon successful execution, I received a message indicating that Docker was functioning correctly, verifying the installation process.

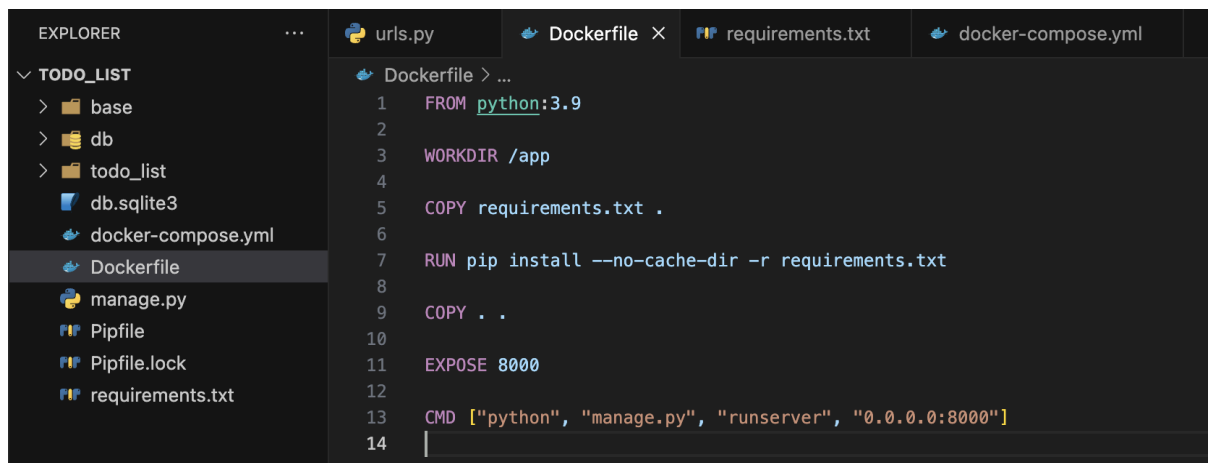
These steps ensured that Docker was properly set up and ready to use in my development environment.

Creating a Dockerfile

Dockerfile Structure

The Dockerfile for a Django application defines the environment in which it runs and includes the following components:

1. **Base Image:** Specifies the official Python image using the `FROM` instruction.
2. **Working directory:** Sets the working directory inside the container to `/app` with `WORKDIR`.
3. **Install Dependencies:** Copies `requirements.txt` into the container and installs dependencies with `RUN pip install --no-cache-dir -r requirements.txt`.
4. **Copy Project Files:** Transfers the entire project directory using the second `COPY` instruction.
5. **Expose Port:** Uses `EXPOSE` to signal that the application listens on port 8000.
6. **Run Command:** The `CMD` instruction runs the Django development server on container start.



```
1 FROM python:3.9
2
3 WORKDIR /app
4
5 COPY requirements.txt .
6
7 RUN pip install --no-cache-dir -r requirements.txt
8
9 COPY . .
10
11 EXPOSE 8000
12
13 CMD ["python", "manage.py", "runserver", "0.0.0.0:8000"]
14 |
```

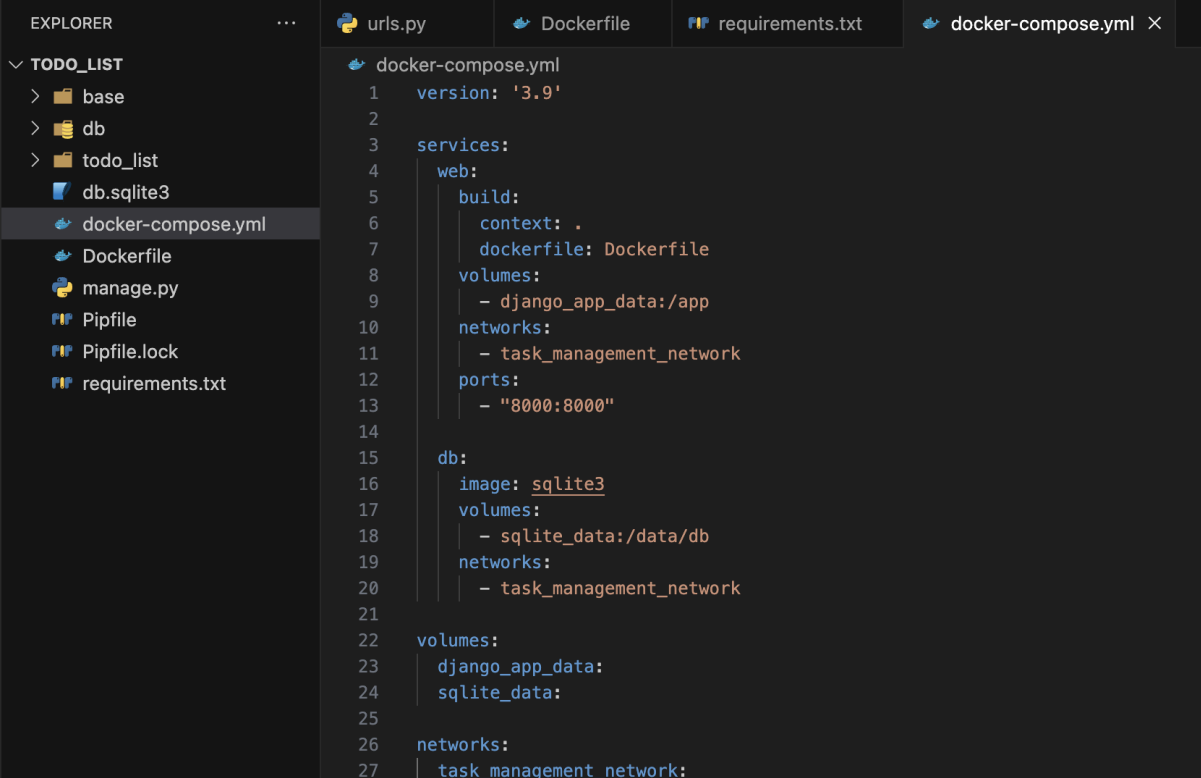
Fig. 3: Screenshot of the Dockerfile in this project.

The Dockerfile for the Django application is carefully designed to create a reliable and efficient environment for running the app. It defines important parts like the base image, working directory, dependencies, and commands to execute. This setup helps ensure smooth deployment and easy access.

Dependencies

The Dockerfile lists the main dependencies for the Django application in requirements.txt. The primary dependency is Django, the framework for the web app. Since it

uses an SQLite3 database, no extra drivers are needed. The command `RUN pip install --no-cache-dir -r requirements.txt` installs everything while avoiding unnecessary cache files.



```
1  version: '3.9'
2
3  services:
4    web:
5      build:
6        context: .
7        dockerfile: Dockerfile
8      volumes:
9        - django_app_data:/app
10     networks:
11       - task_management_network
12     ports:
13       - "8000:8000"
14
15     db:
16       image: sqlite3
17       volumes:
18         - sqlite_data:/data/db
19       networks:
20         - task_management_network
21
22     volumes:
23       django_app_data:
24       sqlite_data:
25
26     networks:
27       task_management_network:
```

Fig. 4: Screenshot of the docker-compose.yml in this project.

This Dockerfile effectively sets up the environment required for the Django application using SQLite3, ensuring that all necessary components are included for successful deployment.

Docker Networking and Volumes

Networking

Docker networking allows containers to communicate effortlessly, enabling those on the same network to connect using their service names as hostnames. This feature enhances collaboration between different services, such as a web server interacting with a database container.

Creating a custom network in a Docker environment greatly improves both functionality and security. By adding a custom network to the ``docker-compose.yml`` file with the bridge driver, services like Django and SQLite can communicate effectively. Here are key benefits of using custom networks:

1. **Isolation:** Custom networks enhance security by ensuring that only authorized services can communicate with each other.
2. **Service Discovery:** Docker's automatic service discovery simplifies communication between services.
3. **Simplified Configuration:** Custom networks streamline inter-service communication, reducing the risk of configuration errors.
4. **Load Balancing:** They support load balancing and help facilitate the scaling of services.
5. **Monitoring:** Custom networks can be integrated with monitoring tools, making it easier to identify and resolve issues.

In summary, custom Docker networks are essential for optimizing service interaction while maintaining a secure and efficient environment.

Volumes

Docker volumes are crucial for maintaining data persistence across container lifecycles, ensuring data integrity and continuity in applications.

By storing data outside the container's filesystem, Docker volumes allow data to persist even when containers are stopped, removed, or updated. This is essential for database-driven applications like SQLite3, where mounting a volume to the database directory protects against data loss. Benefits of Volumes:

1. **Backup and Restore:** Simplifies data backup and restoration, vital for maintaining integrity in databases.
2. **Data Sharing:** Enables multiple containers to share the same data, reducing duplication and enhancing collaboration.
3. **Version Control:** Facilitates tracking changes and reverting to previous data states.
4. **Security:** Improves security by separating sensitive data from transient containers.
5. **Performance Optimization:** Enhances performance by optimizing data access and reducing storage overhead.

In summary, Docker volumes are essential for managing persistent data in containerized environments, offering benefits like backup, data sharing, version control, security, and performance optimization. Understanding their use is crucial for ensuring data integrity and operational efficiency.

Django Application Setup

Project Initialization

The steps to create the Django project and app involved the following:

1. **Create a Django Project:** Using the command line, I initiated a new Django project named *todo_list* by running:

```
tamirisabildayeva@T-MacBook-Pro DockerStart % django-admin startproject todo_list
tamirisabildayeva@T-MacBook-Pro DockerStart % cd todo_list
```

Fig. 5: Screenshot of the terminal displaying the output *startproject*.

This command created a new directory containing the essential files and structure needed for a Django application.

2. **Navigate to Project Directory:** I changed the directory to the newly created project folder:

```
tamirisabildayeva@T-MacBook-Pro DockerStart % django-admin startproject todo_list
tamirisabildayeva@T-MacBook-Pro DockerStart % cd todo_list
```

Fig. 6: Screenshot of the terminal displaying the output *cd* into the project.

This step allowed me to work within the context of the project directory.

3. **Create a Django App:** Next, I created a new app within the project called *todo_list* to handle task management functionalities:

```
tamirisabildayeva@T-MacBook-Pro todo_list % python3 manage.py runserver
Watching for file changes with StatReloader
Performing system checks...
[
System check identified no issues (0 silenced).

You have 18 unapplied migration(s). Your project may not work properly until you apply the migrations for app(s): admin
Run 'python manage.py migrate' to apply them.
October 27, 2024 - 14:20:01
Django version 5.1.2, using settings 'todo_list.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.

[27/Oct/2024 14:20:08] "GET / HTTP/1.1" 200 12068
```

Fig. 7: Screenshot of the terminal displaying the output *runserver*.



The install worked successfully! Congratulations!

View [release notes](#) for Django 5.1

You are seeing this page because `DEBUG=True` is in your settings file and you have not configured any URLs.

django

Fig. 8: Screenshot of the browser when Django project starts.

startapp command generated a new folder with files that will help manage task-related models, views, and templates.

4. **Set Up the SQLite Database:** Although SQLite is the default database for Django, I ensured that the settings were aligned with the Docker Compose configuration. The SQLite database is lightweight and perfect for development, making it easy to get started without additional setup.

Configuration

Configuring the Django settings to connect to the database defined in Docker Compose involved the following steps:

1. **Update Database Settings:** In the *settings.py* file, I modified the 'DATABASES' setting to configure the SQLite database. Since the SQLite database file is stored within the volume mounted at */data/db*, the configuration appears as follows:

```
74 # Database
75 # https://docs.djangoproject.com/en/3.0/ref/settings/#databases
76
77 DATABASES = {
78     'default': {
79         'ENGINE': 'django.db.backends.sqlite3',
80         'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
81     }
82 }
83
```

Fig. 9: Screenshot of the *setting.py* file.

This setting ensures that Django looks for the database file in the correct location, allowing data to persist across container restarts.

2. **Migrate the Database:** After setting up the database configuration, I executed the following command to apply the necessary migrations and create the initial database structure:

```
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, sessions
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... OK
  Applying admin.0003_logentry_add_action_flag_choices... OK
  Applying contenttypes.0002_remove_content_type_name... OK
  Applying auth.0002_alter_permission_name_max_length... OK
  Applying auth.0003_alter_user_email_max_length... OK
  Applying auth.0004_alter_user_username_opts... OK
  Applying auth.0005_alter_user_last_login_null... OK
  Applying auth.0006_require_contenttypes_0002... OK
  Applying auth.0007_alter_validators_add_error_messages... OK
  Applying auth.0008_alter_user_username_max_length... OK
  Applying auth.0009_alter_user_last_name_max_length... OK
  Applying auth.0010_alter_group_name_max_length... OK
  Applying auth.0011_update_proxy_permissions... OK
  Applying auth.0012_alter_user_first_name_max_length... OK
  Applying sessions.0001_initial... OK
tamirisabildayeva@T-MacBook-Pro todo_list %
```

Fig. 10: Screenshot of the terminal displaying the output *python manage.py migrate*.

This command applies all pending migrations, setting up the default tables required for Django's functionality.

3. **Create a Superuser:** To manage the application through the Django admin interface, I created a superuser account by running:

```
tamirisabildayeva@T-MacBook-Pro todo_list % python manage.py createsuperuser
Username (leave blank to use 'tamirisabildayeva'):
Email address:
Password:
Password (again):
Superuser created successfully.
tamirisabildayeva@T-MacBook-Pro todo_list %
```

Fig. 11: Screenshot of the terminal displaying the output *python manage.py createsuperuser*.

This step allows access to the admin panel for creating and managing tasks, users, and other application features.

With these steps completed, the Django application was successfully initialized and configured to connect to the SQLite database defined in the Docker Compose setup. This integration ensures a seamless development experience within the containerized environment, enabling efficient task management and user interaction. The foundational setup paves the way for implementing the core features of the application, such as task creation, editing, and categorization.

Defining Django Models

Project Initialization

In this section, define a *Task* model for a Django application designed to facilitate task management. The model leverages Django's ORM to establish a structured database schema.

```
base > models.py > ...
1  from django.db import models
2  from django.contrib.auth.models import User
3
4  class Task(models.Model):
5      CATEGORY_CHOICES = [
6          ('Work', 'Work'),
7          ('Personal', 'Personal'),
8          ('Other', 'Other')
9      ]
10
11     user = models.ForeignKey(User, on_delete=models.CASCADE, null=True, blank=True)
12     title = models.CharField(max_length=200)
13     description = models.TextField(null=True, blank=True)
14     complete = models.BooleanField(default=False)
15     created = models.DateTimeField(auto_now_add=True)
16     category = models.CharField(max_length=50, choices=CATEGORY_CHOICES, default='Other')
17     deadline = models.DateTimeField(null=True, blank=True)
18
19     def __str__(self):
20         return self.title
21
22     class Meta:
23         order_with_respect_to = 'user'
24
25     # tamirisabildayeva
```

Fig. 12: Screenshot of the *models.py* file.

Key Components of the Task Model:

Fields:

1. **user:** A foreign key that links each task to a specific *User* instance. This relationship allows each task to be associated with a user. The *on_delete=models.CASCADE* option ensures that if a user is deleted, all of their associated tasks will also be removed. The *null=True* and *blank=True* options permit the creation of tasks that are not yet assigned to a user.
2. **title:** A character field with a maximum length of 200 characters, representing the title of the task.
3. **description:** A text field that allows for an extended description of the task and can be left blank.
4. **complete:** A boolean field that indicates whether the task is complete, with a default value of *False*.
5. **created:** A datetime field that records when the task was created, automatically set to the current date and time upon task creation.

6. **category**: A character field for categorizing the task, offering predefined choices (Work, Personal, Other) with a default value of 'Other'.
7. **deadline**: A datetime field used to set a deadline for task completion, which can also be left blank.

String Representation:

The `__str__` method returns the title of the task, providing a human-readable representation of the object.

Meta Options:


The *Meta* class includes options that influence the model's behavior. Here, `order_with_respect_to = 'user'` ensures that tasks are ordered according to their associated user.

Migrations

Migrations are an essential aspect of Django's schema management, allowing developers to apply changes to the database schema in a controlled manner.

1. Creating Migrations:

After defining the `Task` model, I can create migrations by running the following command in the terminal:



```
tamirisabildayeva@T-MacBook-Pro todo_list % python3 manage.py makemigrations
```

Fig. 13: Screenshot of the terminal displaying output `python3 manage.py makemigrations`.

This command generates migration files based on the changes detected in the models. For example, a new migration file will be created to establish the *Task* model and its fields in the database.

2. Applying Migrations:

To apply the created migrations and update the database schema, use the following command:



```
tamirisabildayeva@T-MacBook-Pro todo_list % python3 manage.py migrate
```

Fig. 14: Screenshot of the terminal displaying output `python3 manage.py migrate`.

This command executes the SQL commands necessary to create the `tasks_task` table in the database, complete with the defined fields and constraints.

The defined *Task* model and the processes of creating and applying migrations illustrate how Django simplifies database schema management through its ORM. This approach enables developers to focus more on application logic rather than manual database manipulation, facilitating the efficient development of task management applications.

Conclusion

The development of the task management web application has yielded impressive results, clearly demonstrating the advantages of using Docker for deployment and the effectiveness of building a Django application in a containerized environment. Throughout this project, the benefits of containerization have been unmistakable, particularly in the seamless management of dependencies and the eradication of compatibility issues that often plague traditional deployment methods.

By leveraging Docker, we not only streamlined the development process but also significantly enhanced the application's portability. The creation of a Dockerfile and the implementation of Docker Compose made environment configuration straightforward, ensuring consistent performance across various systems. This approach simplified the integration of services, such as the Django application and the SQLite database, facilitating smooth communication and efficient data management.

The containerized environment accelerated the development of the Django application, allowing for rapid iteration and testing. This structured setup reinforced best practices, including user authentication and task management features, culminating in a robust and user-friendly platform. The application successfully meets its objectives, empowering users to efficiently organize and track their tasks.

In summary, this project has not only deepened my expertise in back-end development but also showcased the transformative potential of containerization in modern software development. Future upgrades, such as deadline reminders and collaborative project tasks, are set to elevate the application's functionality even further. The insights we have gained here will serve as a strong foundation for continued exploration in both Django and Docker, paving the way for the development of even more sophisticated applications in the future.

Reference

- <https://docs.docker.com/get-started/docker-overview/>
- <https://www.docker.com/products/docker-desktop/>
- <https://docs.docker.com/>
- <https://www.docker.com/blog/docker-documentation-ai-powered-assistant/>
- <https://www.w3schools.com/django/index.php>
- <https://rai-shahnawaz.medium.com/creating-a-django-project-the-right-way-14d230358d72>

Appendices

```
tamirisabildayeva@T-MacBook-Pro todo_list % docker-compose build
WARN[0000] /Users/tamirisabildayeva/Projects/WebProjects/todo_list/docker-compose.yml: the attribute `version` is obso
[+] Building 3.3s (12/12) FINISHED                                docker:desktop-linux
=> [web internal] load build definition from Dockerfile          0.0s
=> => transferring dockerfile: 263B                               0.0s
=> [web internal] load metadata for docker.io/library/python:3.9 1.8s
=> [web auth] library/python:pull token for registry-1.docker.io 0.0s
=> [web internal] load .dockerignore                             0.0s
=> => transferring context: 2B                                     0.0s
=> [web 1/5] FROM docker.io/library/python:3.9@sha256:ed8b9dd4e9f89c111f 0.0s
=> [web internal] load build context                             0.0s
=> => transferring context: 6.11kB                                0.0s
=> CACHED [web 2/5] WORKDIR /app                                0.0s
=> [web 3/5] COPY requirements.txt .                              0.0s
=> [web 4/5] RUN pip install --no-cache-dir -r requirements.txt 1.3s
=> [web 5/5] COPY . .                                           0.0s
=> [web] exporting to image                                     0.0s
=> => exporting layers                                           0.0s
=> => writing image sha256:749361d68201568fe02476a1cb61cd5086e07eec7f7d8 0.0s
=> => naming to docker.io/library/todo_list-web                 0.0s
=> [web] resolving provenance for metadata file                 0.0s
tamirisabildayeva@T-MacBook-Pro todo_list %
```

Fig. 15: Screenshot of the terminal displaying output *docker-compose build*.

```
tamirisabildayeva@T-MacBook-Pro todo_list % docker-compose up
WARN[0000] /Users/tamirisabildayeva/Projects/WebProjects/todo_list/docker-compose.yml: the attribute `version` is obso
[+] Running 3/3
✔ Network todo_list_default          Created          0.1s
✔ Volume "todo_list_sqlite_data"     Cre...          0.0s
✔ Container todo_list-web-1          Created          0.0s
Attaching to web-1
web-1 | Traceback (most recent call last):
web-1 |   File "/app/manage.py", line 10, in main
web-1 |     from django.core.management import execute_from_command_line
web-1 | ModuleNotFoundError: No module named 'django'
web-1 |
web-1 | The above exception was the direct cause of the following exception:
web-1 |
web-1 | Traceback (most recent call last):
web-1 |   File "/app/manage.py", line 21, in <module>
web-1 |     main()
web-1 |   File "/app/manage.py", line 12, in main
web-1 |     raise ImportError(
web-1 | ImportError: Couldn't import Django. Are you sure it's installed and available on your PYTHONPATH environment
web-1 exited with code 1
tamirisabildayeva@T-MacBook-Pro todo_list %
```

Fig. 16: Screenshot of the terminal displaying output *docker-compose up*.

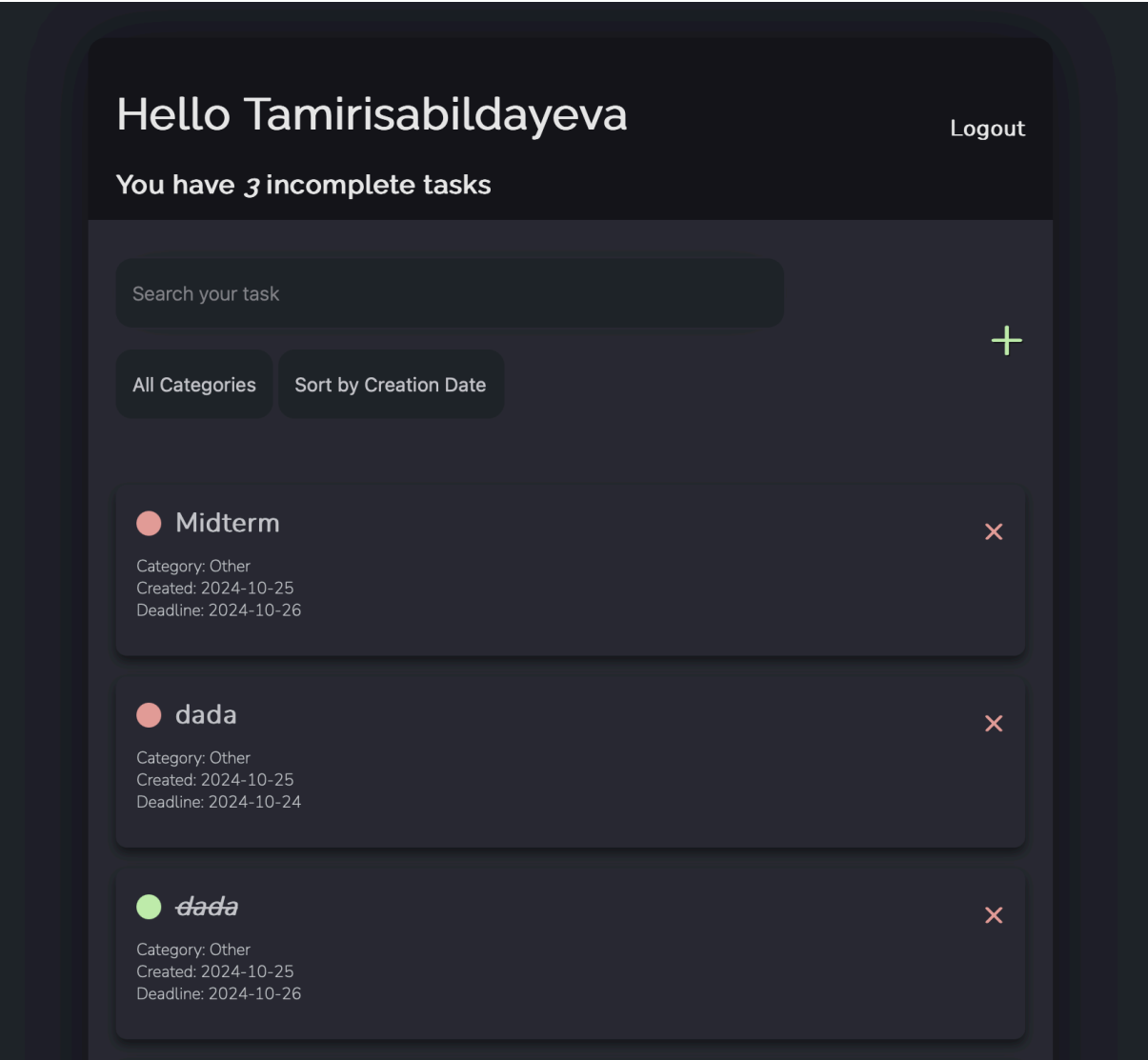


Fig. 17: Screenshot of the Web application.

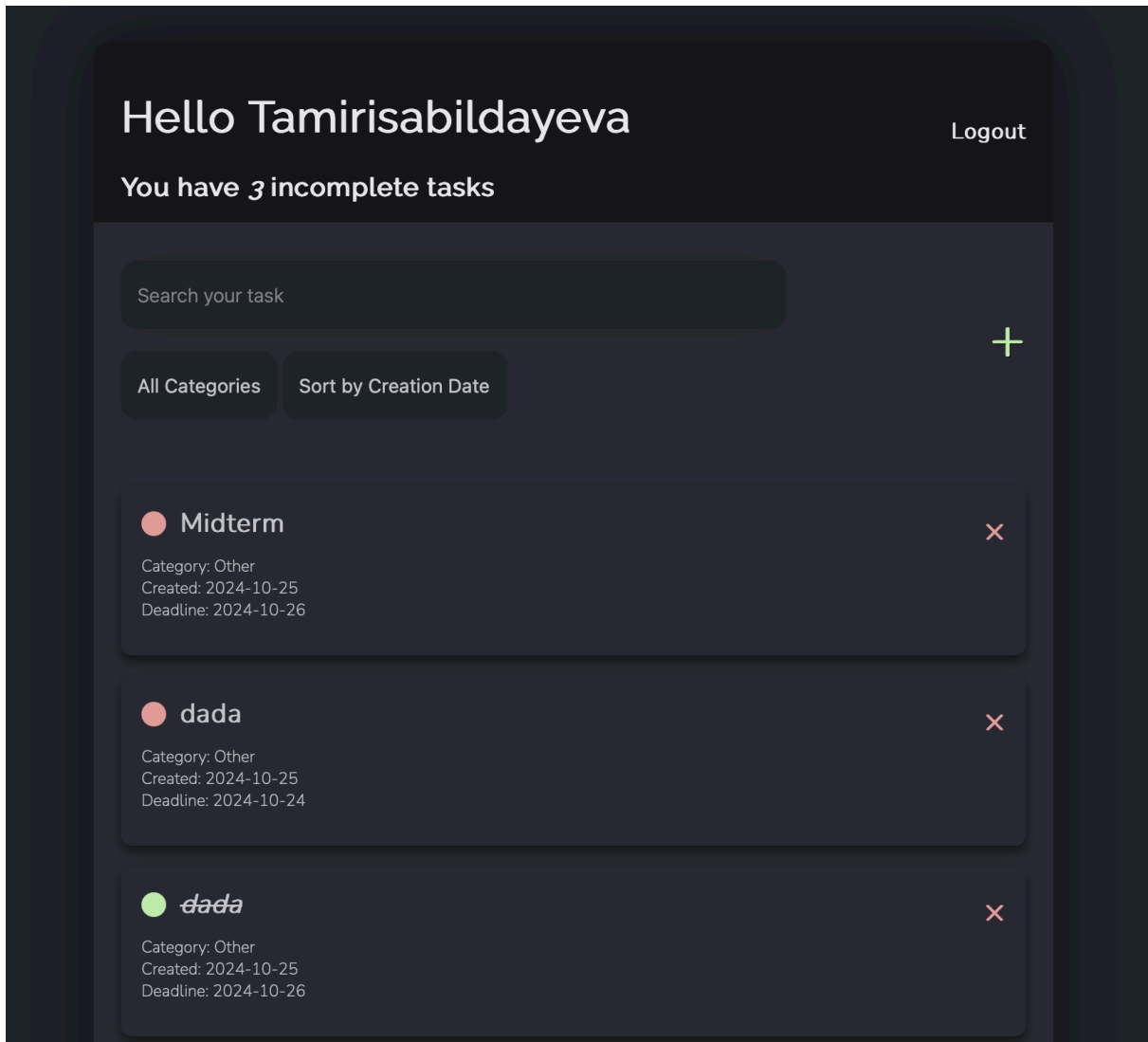


Fig. 18: Screenshot of the Web application while sorting by Creation Date.

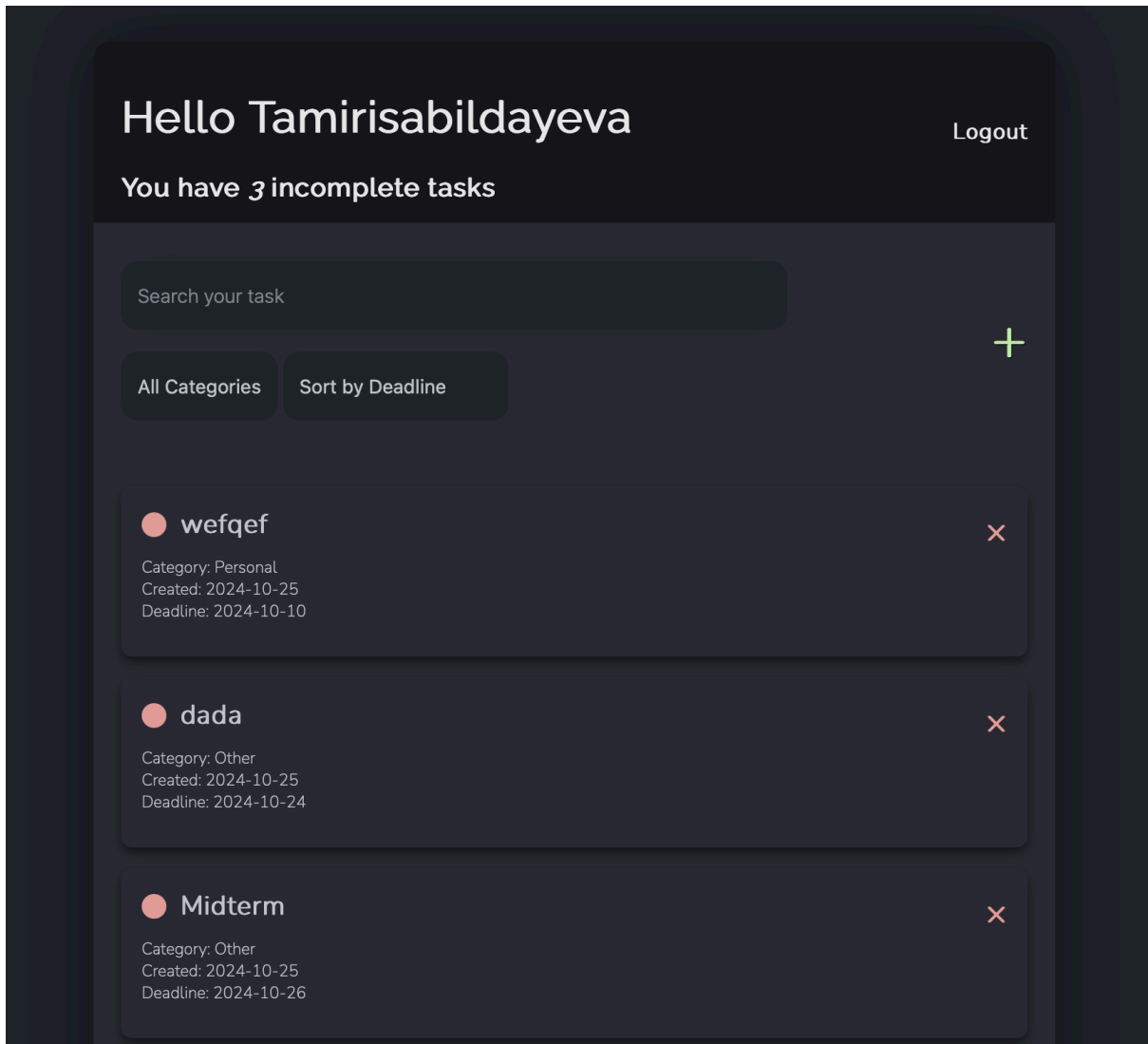


Fig. 19: Screenshot of the Web application while sorting by Deadline.

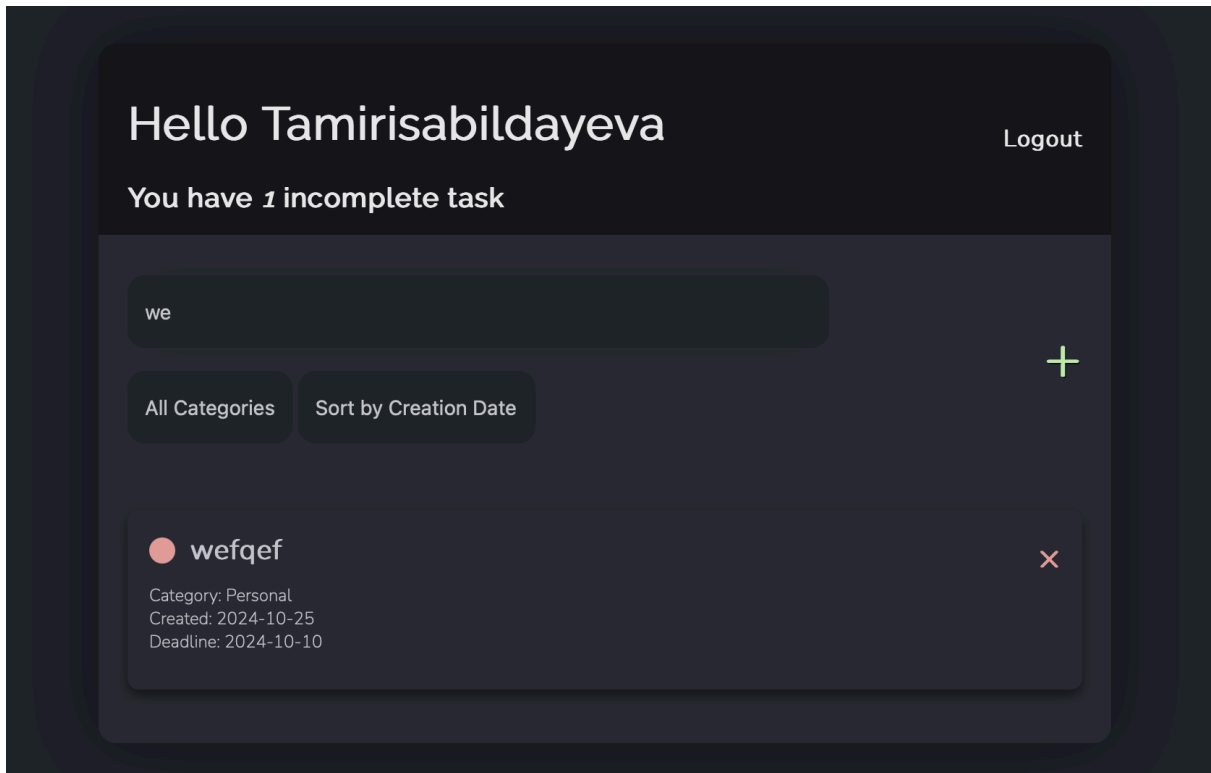


Fig. 20: Screenshot of the Web application while searching *we*.