KBTU

KAZAKH-BRITISH
TECHNICAL
UNIVERSITY

Kazakh-British Technical University

Midterm Work

Web Application Development

Building a Task Management Application Using Django and Docker

Done by : Sarsengaliyev Zhaisan

Checked by: Serek Azamat

Almaty, 2024

1. Executive Summary.

My project is about building Task Management Application that allows to user to manage tasks. Features include the create, get, update, delete tasks with task filtering options like completion status and task priority. Also, I added searching the task functionality by the title.

The app was implemented using Django for backend, Bootstrap for frontend styling to make UI the user friendly. For database I used PostgreSQL, and app is containerized in Docker.

Overall, the app successfully covered all critiries providing easy to use task management system.

2. Introduction.

Containerization become a key practice because it helps to make sure that the app can be executed in different environments without any problems, whether it's local computer or development, or production server. For containerization I use Docker.

For this midterm project the goal was to create Task Management Application using Django framework. With this app users can create, edit and organize their tasks effectively. The motivation behind using Django for the app is the simplicity. By using docker I make sure that the app will be consistent in different environments which makes to easier to deploy the app without worrying about differences between systems.

3. Project Objectives.

The main objectives of this midterm project were to achieve the following:

1. Developing functional web application:
   - Building fully working Task Management App where users can create, view, update, delete tasks.
   - Adding features like task priority, due dates, and the ability to search tasks by their title and filtering tasks by their status (completed and not completed).
2. Using Docker:
   - Understanding the concepts of Docker and how the containerization helps in creating consistent different environments.
   - Writing Dockerfile to define the app environment including Python and Django setup.
   - Creating docker-compose.yml to manage the app and database as separate docker containers which can communicate with each other.
3. Implementing Django Models and Views:
   - Creating Django models to represent tasks in the database including fields like title, description, priority and due date.
   - Creating the views for handling task operations like creating, updating, deleting and linking them to frontend templates.
4. Creating user friendly frontend:
   - Using Bootstrap to style frontend of my app to make it clean and professional.
   - Adding features like date picker for choosing due dates and visual indicators (badges) for task priority and completion status.

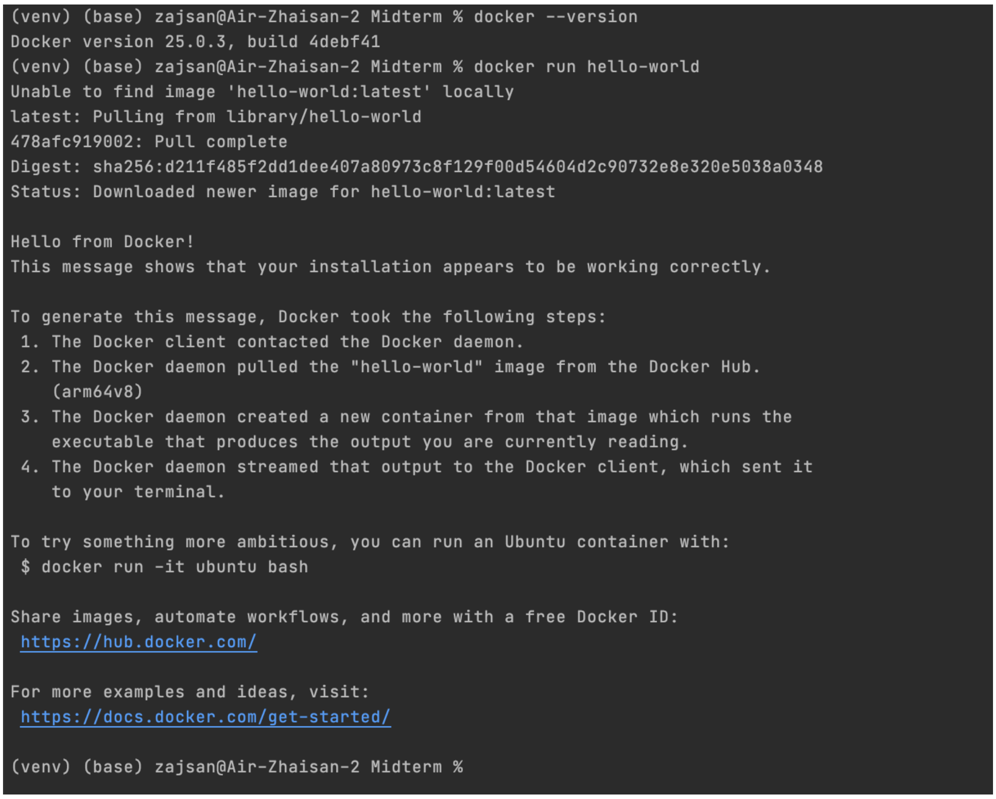
5. Using PostgreSQL for data storing:
    - Using PostgreSQL as database to store data.
    - Making sure that database is properly configured in Docker using docker-compose.

By doing these objectives the app shows the core concepts of web application development, containerization, frontend design for building task management system.

4. Intro to Containerization: Docker

Containerization is a technology which allows to package the application with all its dependencies into container. This provides that application run in the same way in different environments. Key benefits of containerization are:

- Portability: Containers can run everywhere.
- Isolation: Each container runs in own isolated place which means that some problems of container will not affect other containers.
- Efficiency: Containers are lightweight compared to virtual machines.



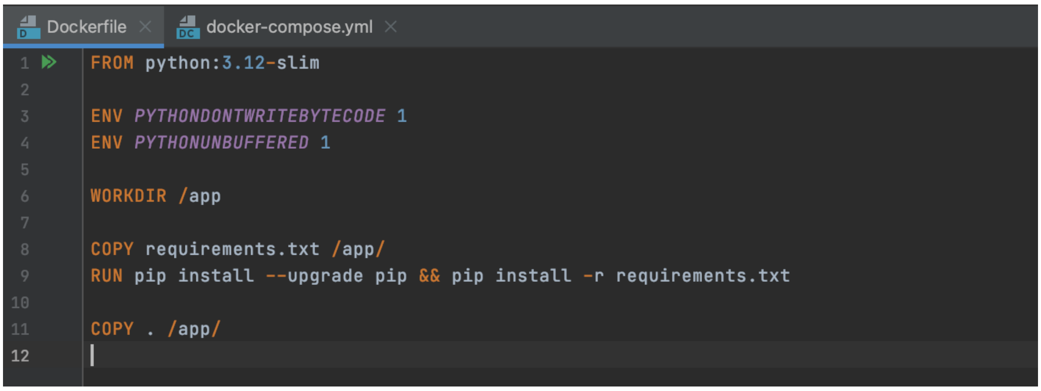
```
(venv) (base) zajsan@Air-Zhaisan-2 Midterm % docker --version
Docker version 25.0.3, build 4debf41
(venv) (base) zajsan@Air-Zhaisan-2 Midterm % docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
478afc919002: Pull complete
Digest: sha256:d211f485f2dd1dee407a80973c8f129f00d54604d2c90732e8e320e5038a0348
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/

(venv) (base) zajsan@Air-Zhaisan-2 Midterm %
```

Figure 4.1: Docker version and running simple container

Firstly I verified that docker is installed as you see in the figure 1.1 by running the command '*docker --version*'. It showed that Docker version is 25.0.3. Next I tested Docker by running the *hello-world* container. Actually the *hello-world* image wasn't on my machine, but Docker automatically pulled the latest version from Docker hub. Then docker created new container from image and ran it which showing '*Hello from Docker!*'. This confirms that Docker is installed correctly.

5. Creating a Dockerfile

Dockerfile is file which has a list of instructions to build Docker image.



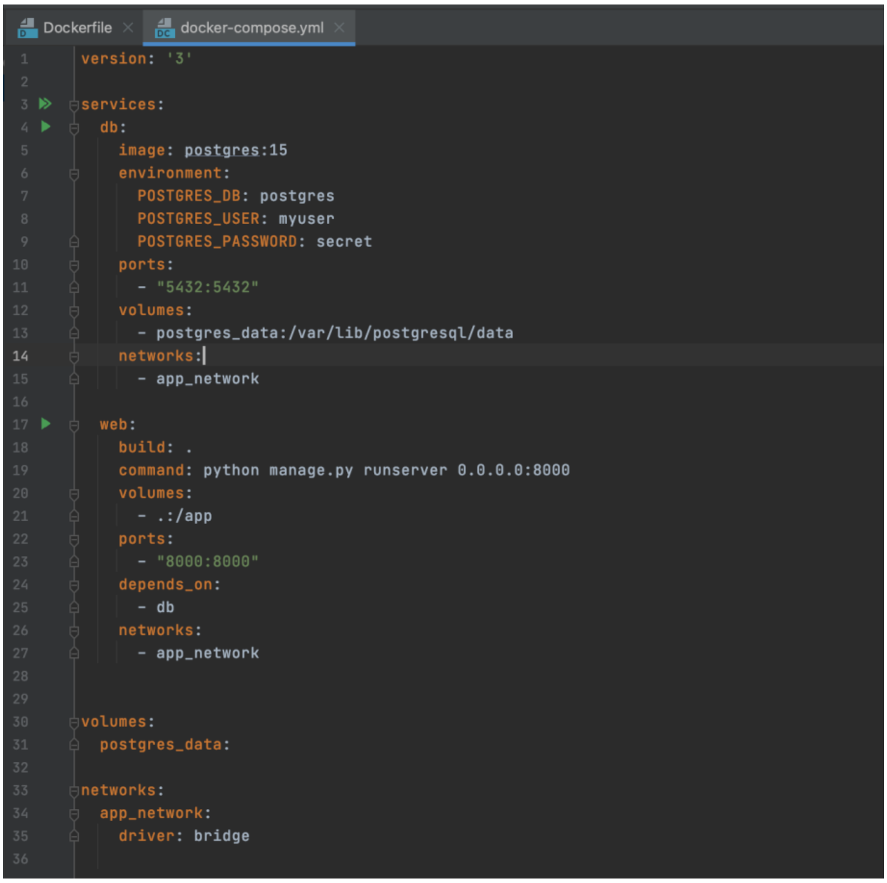
```
FROM python:3.12-slim

ENV PYTHONDONTWRITEBYTECODE 1
ENV PYTHONUNBUFFERED 1

WORKDIR /app

COPY requirements.txt /app/
RUN pip install --upgrade pip && pip install -r requirements.txt

COPY . /app/

```

Figure 5.1: Dockerfile

So, my Dockerfile starts from '*FROM python:3.12-slim*'. This means that Docker uses official Python 3.12-slim image as the base of my app. Next is '*ENV PYTHONDONTWRITEBYTECODE 1*'. It prevents python not to download unnecessary .pyc files inside container. According to '*ENV PYTHONUNBUFFERED 1'* command this makes sure that the ouput is sent directly to the terminal without biffering. It provides easy debugging and logging. Next is '*WORKDIR /app'* instruction sets the working directory to '/app', which means that all proccesses will be run in this folder. Next command is '*COPY requirements.txt /app, RUN pip install --upgrade pip && pip install -r requirements.txt'* copies the requirements.txt into container's /app/ folder. The 'RUN' command installs last version of pip and installs all dependencies lised in requirements.txt file.

6. Using Docker Compose

My compose file has two services:

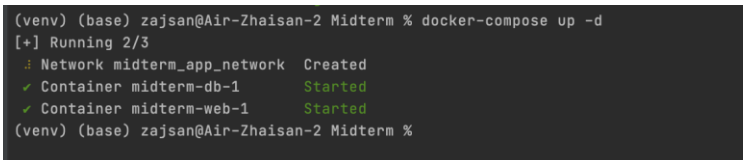


```yaml
version: '3'

services:
  db:
    image: postgres:15
    environment:
      POSTGRES_DB: postgres
      POSTGRES_USER: myuser
      POSTGRES_PASSWORD: secret
    ports:
      - "5432:5432"
    volumes:
      - postgres_data:/var/lib/postgresql/data
    networks:
      - app_network

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


volumes:
  postgres_data:

networks:
  app_network:
    driver: bridge

```

Figure 6.1: Docker-compose.yml

First service is web (Django application). This service is used to run the application. The ‘*command*’ tells the container to run the app using ‘*python manage.py runserver 0.0.0.0:8000*’ which makes the app accessible on port 8000. The volume ‘*.:/app*’ maps the project folder to /app folder inside the container. The ‘*ports*’ section exposes the Django app’s default port to the host machine. The ‘*depends_on:db*’ section says that db service starts before the web service. The ‘*networks: app_network*’ means that service connected to this network.

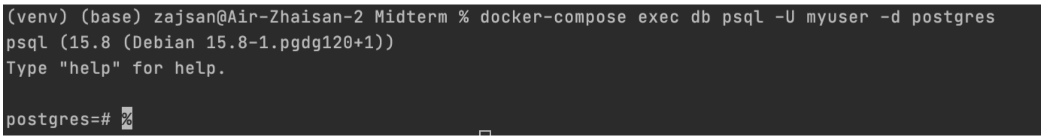
Second service is db service. It uses official PostgreSQL image. The ‘*environment*’ section sets up environment variables of database like database name, the database user, the password. The ‘*ports*’ maps database’s default port (5432) of container to the host machine. The volume ‘*postgres_data*’ makes sure that the database is stored on the host machine even if the container is stopped or removed. The ‘*networks*’ allows to communicate with other services.



```
(venv) (base) zajsan@Air-Zhaisan-2 Midterm % docker-compose up -d
[+] Running 2/3
 ⠿ Network midterm_app_network  Created
 ✔ Container midterm-db-1       Started
 ✔ Container midterm-web-1      Started
(venv) (base) zajsan@Air-Zhaisan-2 Midterm %
```

Figure 6.2: Command to run compose file

As you see in Figure 6.2, I run the services in detached mode starting both the db and web containers.



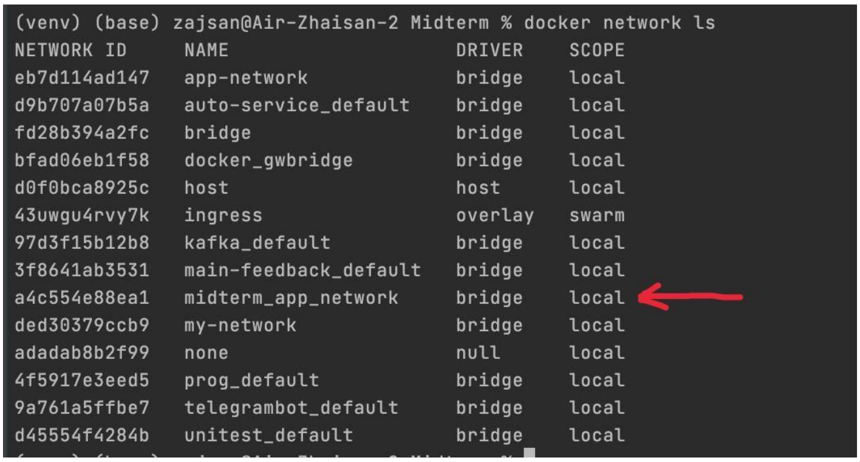
```
(venv) (base) zajsan@Air-Zhaisan-2 Midterm % docker-compose exec db psql -U myuser -d postgres
psql (15.8 (Debian 15.8-1.pgdg120+1))
Type "help" for help.

postgres=# %
```

Figure 6.3: Connecting to database

In figure 6.3, I interact with PostgreSQL to ensure that it works fine.

7. Docker Networking and Volumes

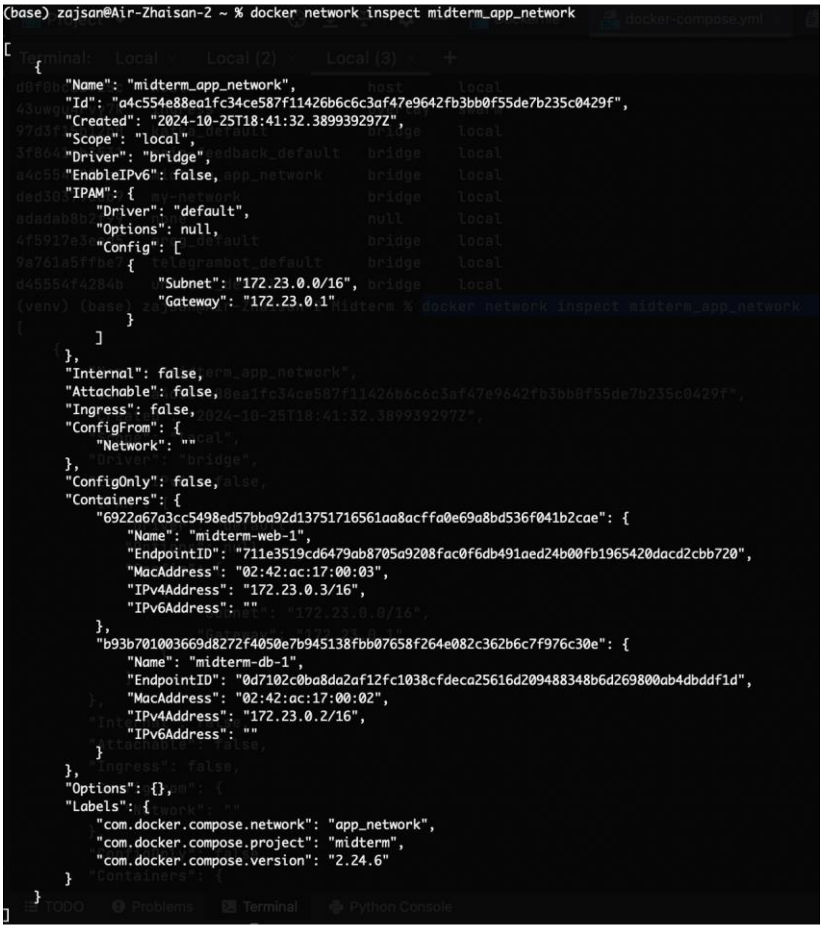
Networking allows different containets to communicate with each other. I created network called app_network using Docker's bridge driver. This driver used for creating isolated networks that allows containers to communicate in the same project.



```
(venv) (base) zajsan@Air-Zhaisan-2 Midterm % docker network ls
NETWORK ID     NAME                    DRIVER    SCOPE
eb7d114ad147   app-network             bridge    local
d9b707a07b5a   auto-service_default    bridge    local
fd28b394a2fc   bridge                  bridge    local
bfad06eb1f58   docker_gwbridge         bridge    local
d0f0bca8925c   host                    host      local
43uwgu4rvy7k   ingress                 overlay   swarm
97d3f15b12b8   kafka_default           bridge    local
3f8641ab3531   main-feedback_default   bridge    local
a4c554e88ea1   midterm_app_network     bridge    local
ded30379ccb9   my-network              bridge    local
adadab8b2f99   none                    null      local
4f5917e3eed5   prog_default            bridge    local
9a761a5ffbe7   telegrambot_default     bridge    local
d45554f4284b   unitest_default         bridge    local
```

Figure 7.1: Docker networks

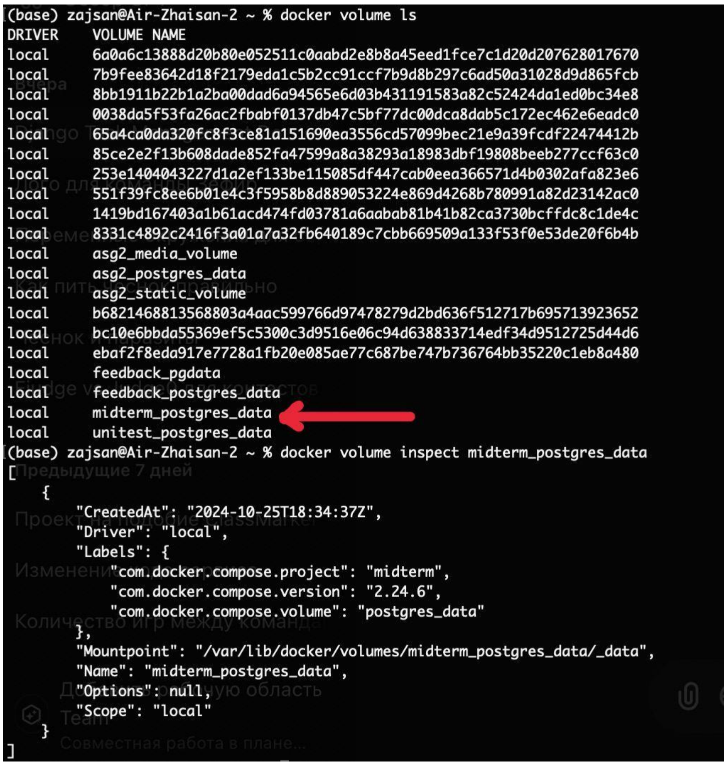
I have many networks, but for my this app my network is '*midterm_app_network'*.



```
(base) zajsan@Air-Zhaisan-2 ~ % docker network inspect midterm_app_network
[
    {
        "Name": "midterm_app_network",
        "Id": "a4c554e88ea1fc34ce587f11426b6c6c3af47e9642fb3bb0f55de7b235c0429f",
        "Created": "2024-10-25T18:41:32.389939297Z",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": null,
            "Config": [
                {
                    "Subnet": "172.23.0.0/16",
                    "Gateway": "172.23.0.1"
                }
            ]
        },
        "Internal": false,
        "Attachable": false,
        "Ingress": false,
        "ConfigFrom": {
            "Network": ""
        },
        "ConfigOnly": false,
        "Containers": {
            "6922a67a3cc5498ed57bba92d13751716561aa8acffa0e69a8bd536f041b2cae": {
                "Name": "midterm-web-1",
                "EndpointID": "711e3519cd6479ab8705a9208fac0f6db491aed24b00fb1965420dacd2cbb720",
                "MacAddress": "02:42:ac:17:00:03",
                "IPv4Address": "172.23.0.3/16",
                "IPv6Address": ""
            },
            "b93b701003669d8272f4050e7b945138fbb07658f264e082c362b6c7f976c30e": {
                "Name": "midterm-db-1",
                "EndpointID": "0d7102c0ba8da2af12fc1038cfdeca25616d209488348b6d269800ab4dbddf1d",
                "MacAddress": "02:42:ac:17:00:02",
                "IPv4Address": "172.23.0.2/16",
                "IPv6Address": ""
            }
        },
        "Options": {},
        "Labels": {
            "com.docker.compose.network": "app_network",
            "com.docker.compose.project": "midterm",
            "com.docker.compose.version": "2.24.6"
        }
    }
]
```

Figure 7.2: Inspecting our network

This command shows details like IP addresses of containers and how the connected to network.



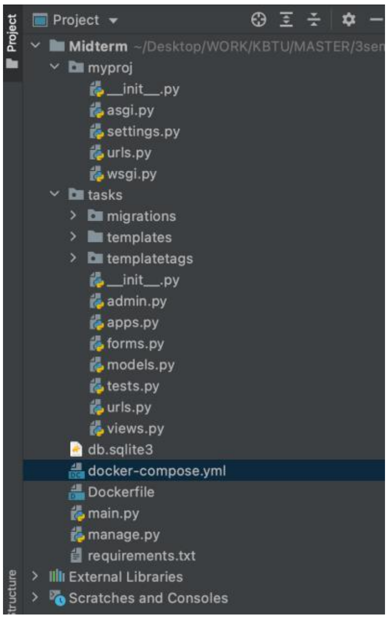
```
(base) zajsan@Air-Zhaisan-2 ~ % docker volume ls
DRIVER    VOLUME NAME
local     6a0a6c13888d20b80e052511c0aabd2e8b8a45eed1fce7c1d20d207628017670
local     7b9fee83642d18f2179eda1c5b2cc91ccf7b9d8b297c6ad50a31028d9d865fcb
local     8bb1911b22b1a2ba00dad6a94565e6d03b431191583a82c52424da1ed0bc34e8
local     0038da5f53fa26ac2fbabf0137db47c5bf77dc00dca8dab5c172ec462e6eadc0
local     65a4ca0da320fc8f3ce81a151690ea3556cd57099bec21e9a39fcdf22474412b
local     85ce2e2f13b608dade852fa47599a8a38293a18983dbf19808beeb277ccf63c0
local     253e1404043227d1a2ef133be115085df447cab0eea366571d4b0302afa823e6
local     551f39fc8ee6b01e4c3f5958b8d889053224e869d4268b780991a82d23142ac0
local     1419bd167403a1b61acd474fd03781a6aabab81b41b82ca3730bcffdc8c1de4c
local     8331c4892c2416f3a01a7a32fb640189c7cbb669509a133f53f0e53de20f6b4b
local     asg2_media_volume
local     asg2_postgres_data
local     asg2_static_volume
local     b6821468813568803a4aac599766d97478279d2bd636f512717b695713923652
local     bc10e6bbda55369ef5c5300c3d9516e06c94d638833714edf34d9512725d44d6
local     ebaf2f8eda917e7728a1fb20e085ae77c687be747b736764bb35220c1eb8a480
local     feedback_pgdata
local     feedback_postgres_data
local     midterm_postgres_data
local     unitest_postgres_data
(base) zajsan@Air-Zhaisan-2 ~ % docker volume inspect midterm_postgres_data
[
    {
        "CreatedAt": "2024-10-25T18:34:37Z",
        "Driver": "local",
        "Labels": {
            "com.docker.compose.project": "midterm",
            "com.docker.compose.version": "2.24.6",
            "com.docker.compose.volume": "postgres_data"
        },
        "Mountpoint": "/var/lib/docker/volumes/midterm_postgres_data/_data",
        "Name": "midterm_postgres_data",
        "Options": null,
        "Scope": "local"
    }
]
```

Figure 7.3: Docker volume and inspecting it

Volumes in Docker are used to persist data which means that data will not be lost even when the container stopped or removed. In 7.3 Figure we see my volume name ‘*midterm_postgres_data’*. Also, we can inspect it to see where the data is stored on my local machine.

8. Django Application Setup

This section is about initializing my project. The project was created by the ‘*docker-compose run web django-admin startproject myproj’* command. After creating project, I created Django app ‘*tasks’* using command ‘*docker-compose run web python manage.py startapp tasks’*.

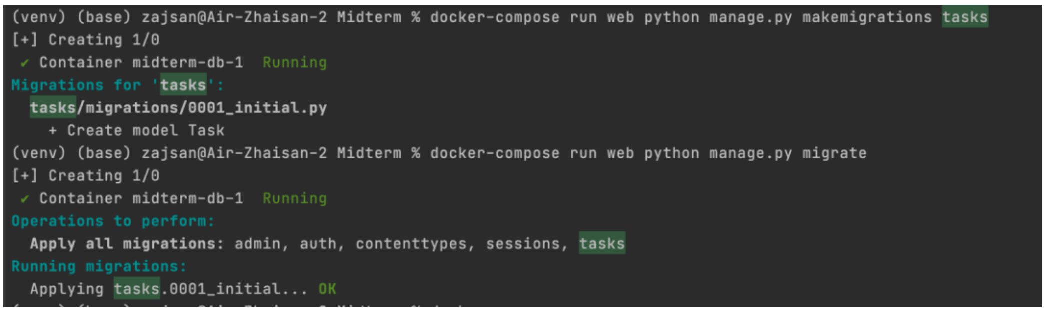


```
Project ▾
v Midterm ~/Desktop/WORK/KBTU/MASTER/3ser
  v myproj
      __init__.py
      asgi.py
      settings.py
      urls.py
      wsgi.py
  v tasks
    > migrations
    > templates
    > templatetags
      __init__.py
      admin.py
      apps.py
      forms.py
      models.py
      tests.py
      urls.py
      views.py
    db.sqlite3
    docker-compose.yml
    Dockerfile
    main.py
    manage.py
    requirements.txt
> External Libraries
> Scratches and Consoles
```

Figure 8.1: Project folder hierarchy

This app is where main functionality of the task management system was created including models, views, templates.

After this I run migration commands to create the necessary database tables.

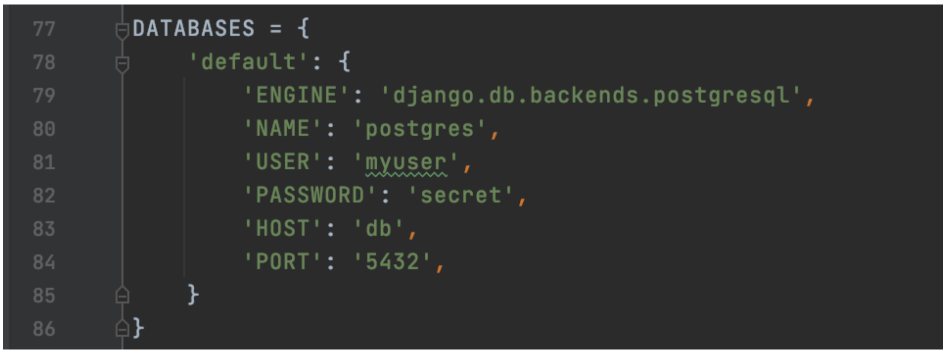


```
(venv) (base) zajsan@Air-Zhaisan-2 Midterm % docker-compose run web python manage.py makemigrations tasks
[+] Creating 1/0
 ✓ Container midterm-db-1  Running
Migrations for 'tasks':
  tasks/migrations/0001_initial.py
    + Create model Task
(venv) (base) zajsan@Air-Zhaisan-2 Midterm % docker-compose run web python manage.py migrate
[+] Creating 1/0
 ✓ Container midterm-db-1  Running
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, sessions, tasks
Running migrations:
  Applying tasks.0001_initial... OK
```

Figure 8.2: Migration commands

These commands generated the migrations for database schema and applied them to set up tables in databse.

Configuration. To connect the django app to PostgreSQL database I added some settings to *settings.py* file



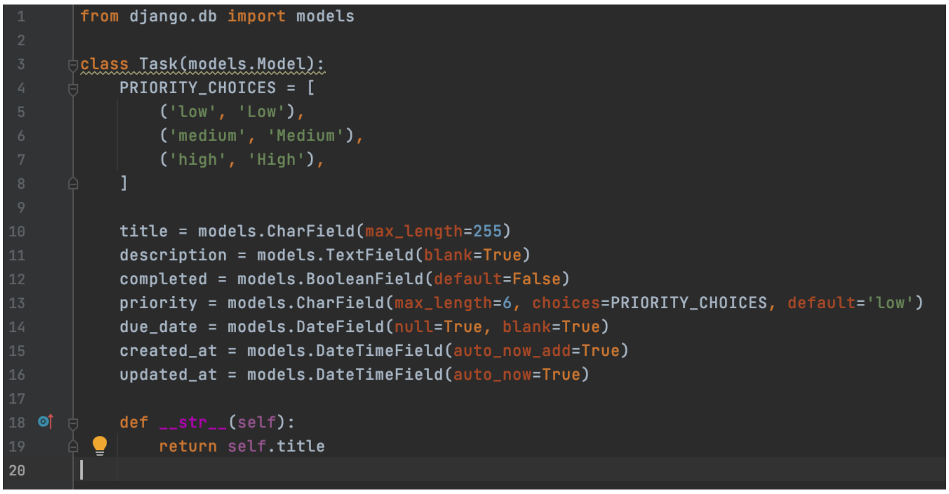
```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'postgres',
        'USER': 'myuser',
        'PASSWORD': 'secret',
        'HOST': 'db',
        'PORT': '5432',
    }
}
```

Figure 8.3: Database settings

Here the '*ENGINE'* means that Django should use PostgreSQL to connect to database. '*NAME'* is database name, '*USER* and *PASSWORD'* are db credentials, '*HOST'* tells Django where to find database service. '*PORT'* is default port for PostgreSQL.

9. Defining Django Models

Models are for defining the structure of data stored in database. For my project I created Task model to manage tasks in the task management app. Each task has some several fields.

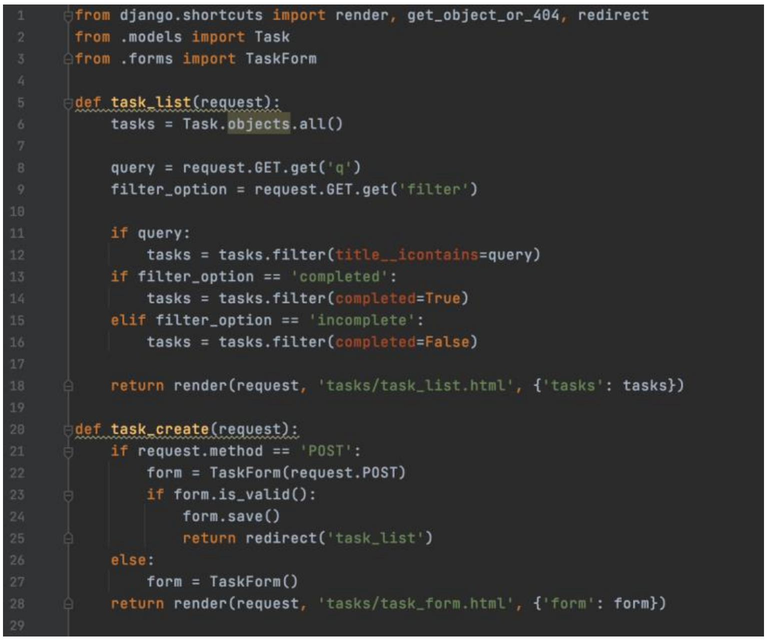


```
from django.db import models

class Task(models.Model):
    PRIORITY_CHOICES = [
        ('low', 'Low'),
        ('medium', 'Medium'),
        ('high', 'High'),
    ]

    title = models.CharField(max_length=255)
    description = models.TextField(blank=True)
    completed = models.BooleanField(default=False)
    priority = models.CharField(max_length=6, choices=PRIORITY_CHOICES, default='low')
    due_date = models.DateField(null=True, blank=True)
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)

    def __str__(self):
        return self.title

```

Figure 9.1: Task model

Task has fields like title, description, completed, priority, due_date, created_at, updated_at. After defining the model I can generate migration files with the command that I executed earlier.

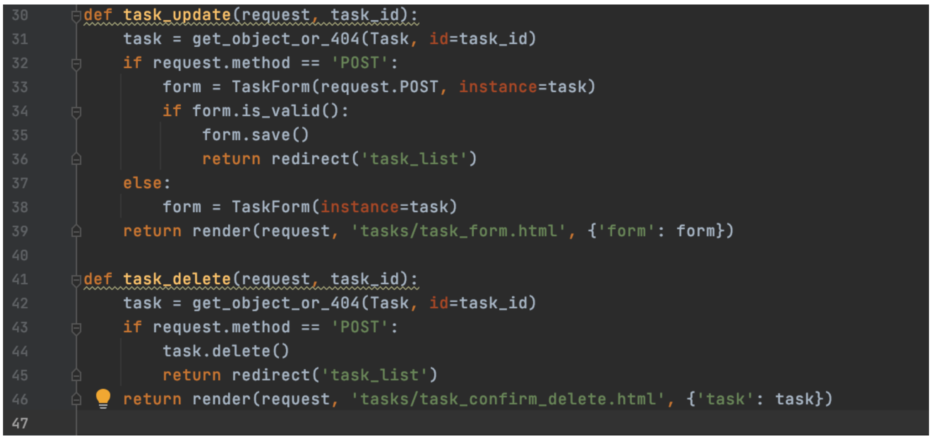
Views. Views are for handling requests and returning responses. In my app I created several views to manage tasks, to create, view, update, delete tasks.



```python
from django.shortcuts import render, get_object_or_404, redirect
from .models import Task
from .forms import TaskForm

def task_list(request):
    tasks = Task.objects.all()

    query = request.GET.get('q')
    filter_option = request.GET.get('filter')

    if query:
        tasks = tasks.filter(title__icontains=query)
    if filter_option == 'completed':
        tasks = tasks.filter(completed=True)
    elif filter_option == 'incomplete':
        tasks = tasks.filter(completed=False)

    return render(request, 'tasks/task_list.html', {'tasks': tasks})

def task_create(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/task_form.html', {'form': form})
```

Figure 9.2: List, Create views

As you see, in the list I added filtering by '*completed'* field and searching by '*title'* field.

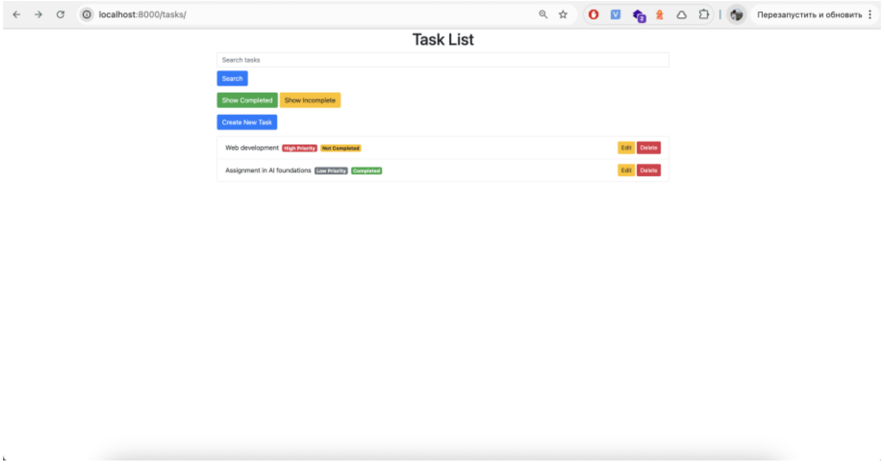


```python
def task_update(request, task_id):
    task = get_object_or_404(Task, id=task_id)
    if request.method == 'POST':
        form = TaskForm(request.POST, instance=task)
        if form.is_valid():
            form.save()
            return redirect('task_list')
    else:
        form = TaskForm(instance=task)
    return render(request, 'tasks/task_form.html', {'form': form})

def task_delete(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/task_confirm_delete.html', {'task': task})
```

Figure 9.3: Update, Delete views

In this figure the update, delete requests are implemented. By the way, all this views are connected to htmls, I will describe them in Appendices section.

Testing endpoints. CRUD operations are attached below.



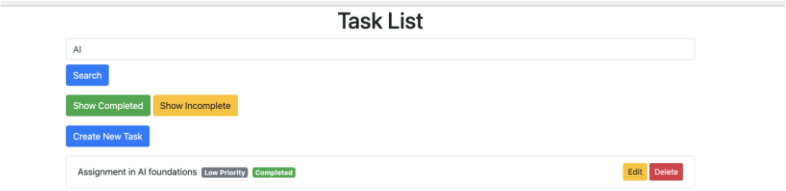
Task List

Search tasks

Web development High Priority Not Completed

Assignment in AI foundations Low Priority Completed

Figure 9.4: Listing tasks

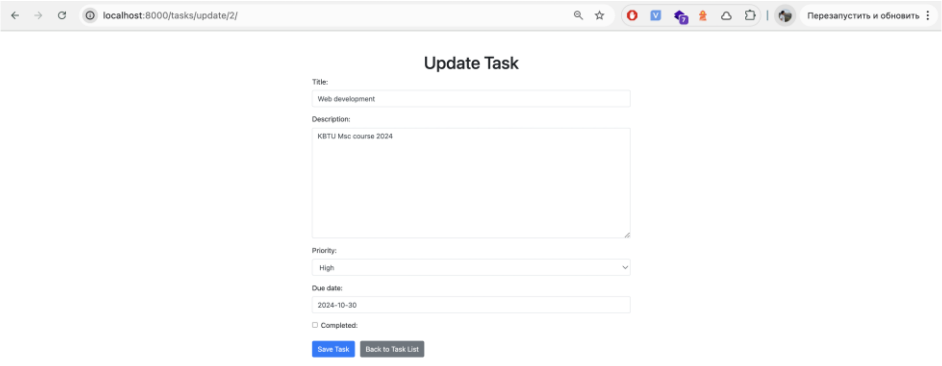


Task List

AI

Assignment in AI foundations Low Priority Completed

Figure 9.5: Search by title



Update Task

Title: Web development

Description: KBTU Msc course 2024

Priority: High

Due date: 2024-10-30

□ Completed:

Figure 9.6: Update task

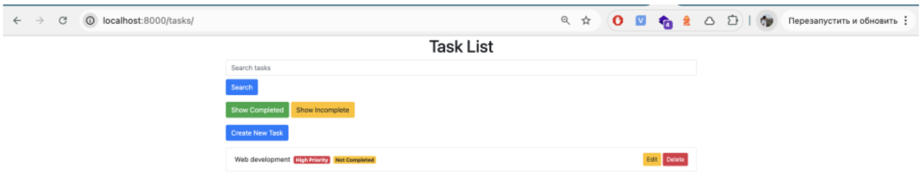


Figure 9.7: Deleting task

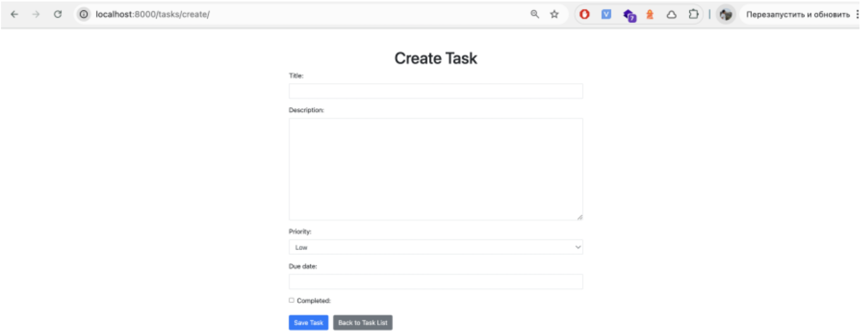
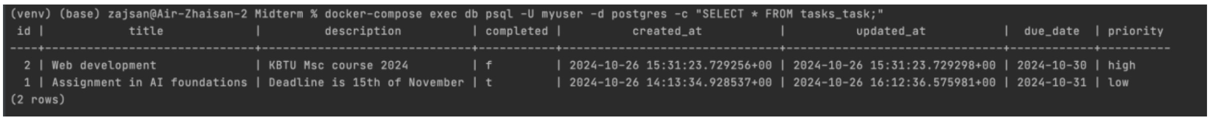


Figure 9.8: Creating task

So, in the figures above we can see the UI of task management system. Everything works correctly and fine.

We can see the results in database:



```
(venv) (base) zajsan@Air-Zhaisan-2 Midterm % docker-compose exec db psql -U myuser -d postgres -c "SELECT * FROM tasks_task;"
 id |             title             |         description          | completed |          created_at           |          updated_at           |  due_date  | priority
----+-------------------------------+------------------------------+-----------+-------------------------------+-------------------------------+------------+----------
  2 | Web development               | KBTU Msc course 2024         | f         | 2024-10-26 15:31:23.729256+00 | 2024-10-26 15:31:23.729298+00 | 2024-10-30 | high
  1 | Assignment in AI foundations  | Deadline is 15th of November | t         | 2024-10-26 14:13:34.928537+00 | 2024-10-26 16:12:36.575981+00 | 2024-10-31 | low
(2 rows)
```

Figure 9.9: My tasks in database

10. Conclusion

In my project I successfully developed Django application (with Frontend) using Docker. By containerizing application I ensured that the running the app was easier. It allowed me to run Django app and PostgreSQL database in separate isolated containers, also, managing the dependencies was easy through Dockerfile and Docker Compose. Overall, I achieved the objectives by creating task management system with clear functionality while also using development tools like Django framework and Docker to reach efficiency and organization.

11. References [optional]

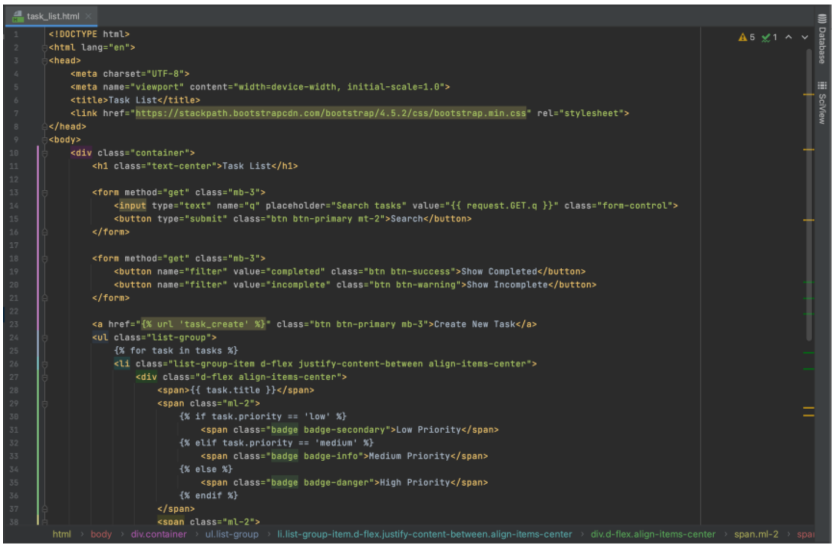
- https://hub.docker.com/

- https://www.w3schools.com/django/
- canva.com (used to create the high-level architecture diagram of the application)

12. Appendices [optional]

In addition to the backend functionality, I implemented frontend part of the project. The features are

- UI using bootstrap

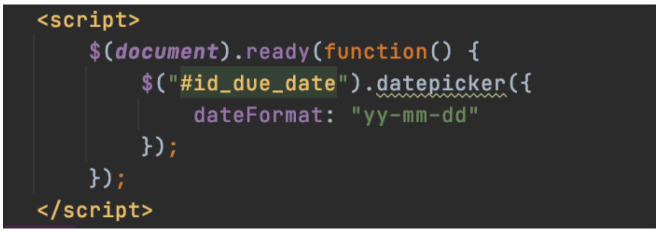


task_list.html
Database
SciView
html › body › div.container › ul.list-group › li.list-group-item.d-flex.justify-content-between.align-items-center › div.d-flex.align-items-center › span.ml-2 › spa

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Task List</title>
    <link href="https://stackpath.bootstrapcdn.com/bootstrap/4.5.2/css/bootstrap.min.css" rel="stylesheet">
</head>
<body>
    <div class="container">
        <h1 class="text-center">Task List</h1>

        <form method="get" class="mb-3">
            <input type="text" name="q" placeholder="Search tasks" value="{{ request.GET.q }}" class="form-control">
            <button type="submit" class="btn btn-primary mt-2">Search</button>
        </form>

        <form method="get" class="mb-3">
            <button name="filter" value="completed" class="btn btn-success">Show Completed</button>
            <button name="filter" value="incomplete" class="btn btn-warning">Show Incomplete</button>
        </form>

        <a href="{% url 'task_create' %}" class="btn btn-primary mb-3">Create New Task</a>
        <ul class="list-group">
            {% for task in tasks %}
            <li class="list-group-item d-flex justify-content-between align-items-center">
                <div class="d-flex align-items-center">
                    <span>{{ task.title }}</span>
                    <span class="ml-2">
                        {% if task.priority == 'low' %}
                            <span class="badge badge-secondary">Low Priority</span>
                        {% elif task.priority == 'medium' %}
                            <span class="badge badge-info">Medium Priority</span>
                        {% else %}
                            <span class="badge badge-danger">High Priority</span>
                        {% endif %}
                    </span>
                    <span class="ml-2">
```

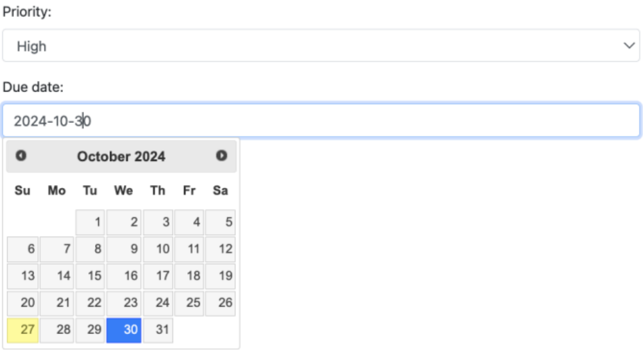
Figure 12.1: List html

- Date picker to choose due date using jQuery UI



```html
<script>
    $(document).ready(function() {
        $("#id_due_date").datepicker({
            dateFormat: "yy-mm-dd"
        });
    });
</script>
```

Figure 12.2: Date picker code



Priority:

High

Due date:

2024-10-30

October 2024

Su	Mo	Tu	We	Th	Fr	Sa
		1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30	31		

Figure 12.3: Data picker to choose date

Also, this is simple high-level architecture diagram of our app

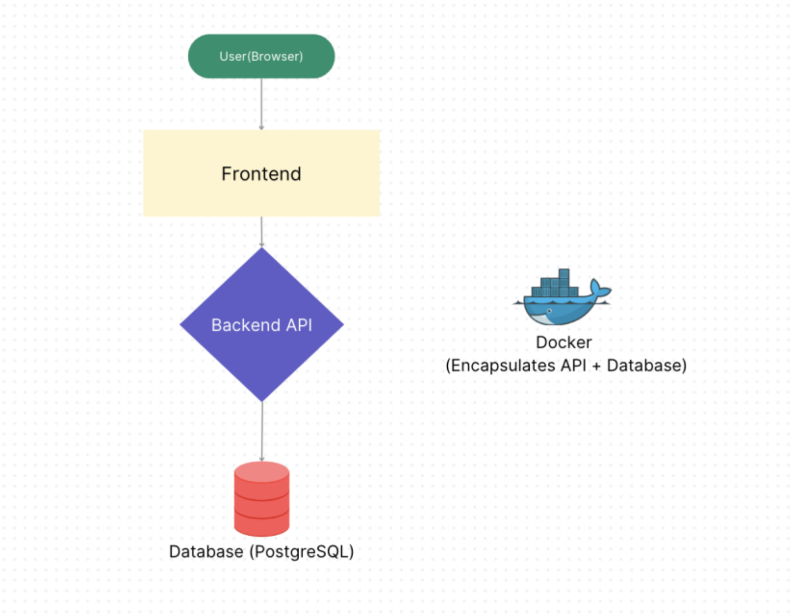


Figure 12.4: Architecture diagram