



**Kazakh-British Technical University**  
**School of Information Technologies and Engineering**

**MidTerm**  
**Web Application Development**  
**Building a Task Management Application Using Django**

Prepared by: Maratuly T.  
Checked by: Serek A.

**Almaty, 2024**

## CONTENT

Introduction .....	3
1. Intro to Containerization: Docker .....	4
2. Creating a Dockerfile .....	5
3. Using Docker Compose .....	6
4. Docker Networking and Volumes .....	7
5. Django Application Setup .....	9
6. Defining Django Models .....	14
7. Back-end views .....	16
7.1 User-related endpoints .....	16
7.2 Task-related endpoints .....	19
Conclusion .....	22

## INTRODUCTION

This project aims to implement a TODO list for handling events such as posting new tasks, fetching own tasks, logging into the system, registering new users, and deleting own tasks. Django was used because of its simplicity, quick realization of the tasks (since it is commonly used for MVP products), and just personal passion to this framework. Containerization via Docker is assumed to quickly develop product transfer into an isolated virtual environment, and easily deploy it on a server or connect unused packages (modules) which are used in the current project with no need to manually install them (here PostgreSQL). It ensures that what works on the developer's machine will also work in production, minimizing conflicts caused by dependencies or system configurations. Docker is used to containerize both the Django task management application and the PostgreSQL database. It combines the following motivation: consistent development environment (both the application and database run identically on local machines and in production), multi-container setup with docker compose (Django web service, here, back-end, and the PostgreSQL database service, named "db", are defined and managed together using Docker Compose. This makes it easy to orchestrate and maintain both services). The usage of volumes for docker is a simple way to store database results, even if we stop and restart the container. Django is chosen for its high-level abstractions (class-based approach), built-in admin panel (to manage models' data), and robust ORM (Object Relational Mapper).

**Executive summary.** The project combines the most popular technologies such as DRF (Django Rest Framework) for back-end development, PostgreSQL as a relation database, JWT (JSON Web Tokens) for user's authentication, Debug Tool Bar for processing database queries, connection for further optimization, Docker to build and launch the project without any forced requirements for packages, technologies or even an operational system. Docker compose is applied to connect both back-end (DRF) and database (PostgreSQL) into one network without manual way of doing it separately for each of them.

**Project goal.** The project aims to create a highly loaded, quickly implemented simple back-end application using REST (Representational State Transfer) architecture style to divide the Web application into separate responsible sides.

**Project outcome.** The project contains several outcomes such as the possibility to create, delete, view and partially update tasks through a simple application (each user has own tasks, so, they separated). Another outcome is that the application is based on the latest technologies such as containerization to quickly build and run a project in several systems (Docker networking ensures that Django can communicate securely with the PostgreSQL database).

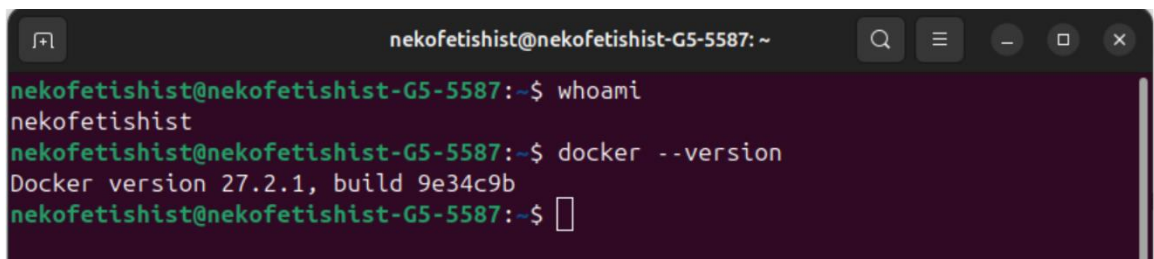
**Project objectives.** The primary objective of the project is to develop a task management Application Programming Interface (API) using Django, Django Rest Framework (DRF) for creating, retrieving, updating, and deleting tasks, integrating PostgreSQL as a database, and deploying it in a Dockerized environment. Moreover, the work involves the usage of JWT (JSON Web Tokens) tokens for secure authentication.

## 1. Introduction to Containerization: Docker

This section is responsible for getting familiar with containerization, docker, its installation and benefits of the usage. Containerization is an approach where you wrap your application, program or any utility into an isolated environment, own operational system, with a separate network. One of the best offers is to use Docker containerization. Docker is an open-source product which is used for developing and wrapping applications into images, then run these images in containers. Docker has several components (Docker engine, Docker CLI, Docker image, Docker container, Docker registry, Docker hub, Docker network, etc.). Docker Engine is a core part that is responsible for running and managing containers. It consists of 2 parts: Docker daemon (docker's server side) and Docker API (interface that communicates between Docker Daemon and other Docker components). Docker CLI or Docker Client or Docker Command-Line-Interface is used to work with Docker, for example, run containers, stop them, etc. Docker Image is a read-only template (e.g., template of ubuntu os, or python interpreter, etc.) with instructions to create a docker container. Dockerfile is your own image. Docker Container is a runnable instance of an image. By default, these containers are isolated from each other. Docker Registry is a place where you host/store different images or group of images. Docker Hub is a registry to store and share images. Docker Network is a way to provide isolation for docker containers. It is possible to set containers to see each other or be in the same network.

Docker installation depends on what operating system (OS) you use (Linux, MacOS or Windows). Here we introduce installation only for Linux (Fedora distributor) since this OS was used. Firstly, install the docker using the command in a terminal “sudo dnf install docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker-compose-plugin”, then you need to start a docker as a process using the command “sudo systemctl start docker”, add your user to a docker group in order to run the commands with no admin privileges. That's it, the installation is over.

As we installed docker via official documentation, we need to make sure that it works. The best way to use ‘docker --version’ which must show the installed version of a docker. The information about docker's version is illustrated in Figure 1.1.

A terminal window with a dark background and light green text. The window title is 'nekofetishist@nekofetishist-G5-5587: ~'. The terminal shows three commands and their outputs: 'whoami' returns 'nekofetishist', 'docker --version' returns 'Docker version 27.2.1, build 9e34c9b', and the prompt returns to the user. The window has standard Linux window controls (search, menu, zoom, close) in the top right corner.

```
nekofetishist@nekofetishist-G5-5587: ~  
nekofetishist@nekofetishist-G5-5587:~$ whoami  
nekofetishist  
nekofetishist@nekofetishist-G5-5587:~$ docker --version  
Docker version 27.2.1, build 9e34c9b  
nekofetishist@nekofetishist-G5-5587:~$
```

Figure 1.1 - Docker's installed version information

As we installed the docker, we can run ‘hello-world’ container to see that docker works properly with no issues and errors. Figure 1.2 demonstrates the results of this command. Docker will download ‘hello-world’ image if you do not have it.

```
nekofetishist@nekofetishist-G5-5587: ~  
nekofetishist@nekofetishist-G5-5587:~$ docker --version  
Docker version 27.2.1, build 9e34c9b  
nekofetishist@nekofetishist-G5-5587:~$ docker run hello-world  
Unable to find image 'hello-world:latest' locally  
latest: Pulling from library/hello-world  
c1ec31eb5944: 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.  
   (amd64)  
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/
```

Figure 1.2 - Logs after running ‘hello-world’ container

The output of the `docker run hello-world` command was that there was no hello-world image, and it started downloading it. After that it showed the message that the docker is installed correctly and ready to work.

## 2. Creating a Dockerfile

As we know that docker works perfectly, we can create a Dockerfile for our backend application. Let’s now look at “Dockerfile” to understand the project’s building approach. Dockerfile is in root of the project as it is our entry point to build an image from it and run. Figure 2.1 contains information about django’s docker file.

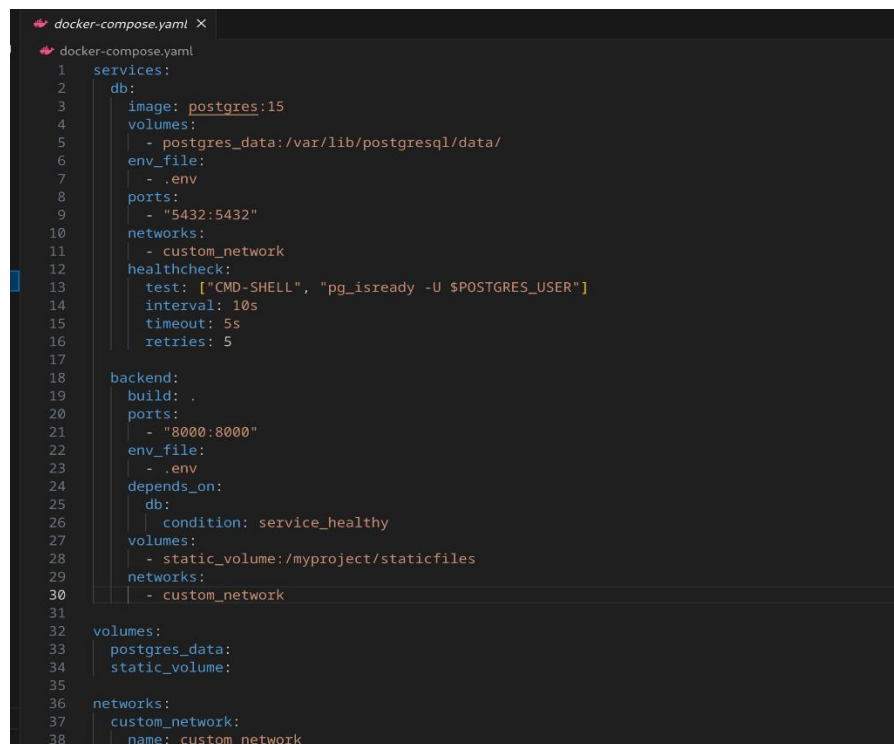
```
Dockerfile X  
Dockerfile > ...  
1 FROM python:3.12-slim  
2  
3 # Set environment variables  
4 ENV PYTHONDONTWRITEBYTECODE 1  
5 ENV PYTHONUNBUFFERED 1  
6  
7 # Set the working directory  
8 WORKDIR /myproject  
9  
10 # Copy Pyproject file  
11 COPY ./pyproject.toml .  
12 COPY ./requirements.txt .  
13 COPY ./env .  
14  
15 # Install dependencies  
16 RUN pip3 install --no-cache-dir -r requirements.txt  
17  
18 # Copy the rest of the code  
19 COPY . .  
20  
21 # Collect static files  
22 RUN python3 manage.py collectstatic --noinput  
23  
24 # Expose the port  
25 EXPOSE 8000  
26  
27 # Run the application  
28 ENTRYPOINT ["sh", "-c", "python3 manage.py migrate && python3 manage.py runserver 0.0.0.0:8000"]
```

Figure 2.1 - Dockerfile’s details for project building

The file contains all the comments about each command but emphasizing some notes it can be added that we use python 3.12 slim version and run the application using guvicorn to bind a specified url. Initially, we copy only “pyproject.toml”, “requirements.txt” and “.env” files because we want to cache these steps as we contain there all necessary dependencies. Moreover, we set environment variables “PYTHONDONTWRITEBYTECODE” to 1 or True as well as “PYTHONUNBUFFERED” to 1 or True in order to not to save the cache into the buffer. Then, we copy all the files from the project root folder and transfer then to a “myproject” directory, collect static files, make migrations to apply the rules for a database and run the application. The description of the “Dockerfile” is now over.

### 3. Using Docker Compose

Now we can start with “docker-compose.yaml” file. It is used locally to manage all the containers which are described there to make the project work correctly using all the services. For our case, we need to include images of “postgresql” (here it was decided to use 15th version) and locally built “Dockerfile” of the project. The information about “docker-compose” file data is shown in Figure 3.1.

The image shows a code editor window titled 'docker-compose.yaml X'. The file content is as follows:

```
1  services:
2    db:
3      image: postgres:15
4      volumes:
5        - postgres_data:/var/lib/postgresql/data/
6      env_file:
7        - .env
8      ports:
9        - "5432:5432"
10     networks:
11       - custom_network
12     healthcheck:
13       test: ["CMD-SHELL", "pg_isready -U $POSTGRES_USER"]
14       interval: 10s
15       timeout: 5s
16       retries: 5
17
18     backend:
19       build: .
20       ports:
21         - "8000:8000"
22       env_file:
23         - .env
24       depends_on:
25         db:
26           condition: service_healthy
27       volumes:
28         - static_volume:/myproject/staticfiles
29       networks:
30         - custom_network
31
32  volumes:
33    postgres_data:
34    static_volume:
35
36  networks:
37    custom_network:
38    name: custom_network
```

Figure 3.1 - Docker compose file’s configuration

This file defines a multi-service setup with Docker for a project using the latest version (not specified at the top). It contains two services: "db" and "backend". The "db" service uses the “postgres:15 image”, its data stored in a volume called "postgres\_data". It also uses an environment file named ".env" for configuration and exposes port "5432" to communicate with other services or the host. Additionally,

before the database service is launched it passes the health check to be sure that database works with not problems. We can run the docker compose file using the command “docker compose up -d” which means to run a docker compose file in detach mode. Figure 3.2 illustrates the results as the container was run.

```
nekofetishist@fedora:~/Study/Master/Web-Application-Development/MidTerm$ docker compose up -d
[+] Running 2/2
 ✓ Container midterm-db-1      Healthy
 ✓ Container midterm-backend-1 Started
```

Figure 3.2 – Statuses of the database and back-end containers

The "backend" service is set to build from the current directory where the “docker-compose” is located, and it also references the ".env" file for its configuration. It depends on the "db" service, ensuring that the database is ready before the backend starts. The "backend" service is exposed on port "8000" and uses a volume named "static\_volume" to store static files at the specified path in the container.

#### 4. Docker Networking and Volumes

Docker networking and volumes are key elements of a Docker infrastructure. Docker networking allows a developer to create their own isolated networks and let the services see and communicate with each other. Figure 4.1 shows the list of the current existing networks on a local machine.

```
nekofetishist@fedora:~/Study/Master/Web-Application-Development/MidTerm$ docker network ls
```

NETWORK ID	NAME	DRIVER	SCOPE
2df2c34dc887	bridge	bridge	local
a5537c542c58	host	host	local
cd8a6ae4cbd9	none	null	local

Figure 4.1 – List of the current existing networks

After running the docker compose file we get a new isolated network which combines our both services “db” and “backend” in bridge mode. The updated list of the networks is shown in Figure 4.2.

```
nekofetishist@fedora:~/Study/Master/Web-Application-Development/MidTerm$ docker network ls
```

NETWORK ID	NAME	DRIVER	SCOPE
2df2c34dc887	bridge	bridge	local
c04fb7fe5779	custom_network	bridge	local
a5537c542c58	host	host	local
cd8a6ae4cbd9	none	null	local

Figure 4.2 – List of the updated networks



So, our new network “custom\_network” has been successfully added. Additionally, we can refer from one service to another by just using the name of the service there since they are in the same network (database connection is proof). Moving to a concept of volumes we can understand as a storage where we can save data, but it is necessary to refer to it where we use it. The project has custom networks which are called “postgres\_data” and “static\_volume” namely. So, volumes are the preferred mechanism for persisting data generated by and used by Docker containers. Figure 4.3 demonstrates the list of the volumes before docker compose is run.

```
nekofetishist@fedora:~/Study/Master/Web-Application-Development/MidTerm$ docker volume ls
DRIVER      VOLUME NAME
local       7cd464521aeee05bd1eb4b4cb161976c9f59b46cfdb7fb7c239fd40e5d5f94a6
local       7fc699f84682e2f21fe238b34dc210ef2a2422d5e3f055f373490062d6181841
local       8a92a00cb1b8f6332c359f6a24bebd9be40cdd829322a64bbb4c4ebb6856705c
local       432c76967c234b8e4fb2be70b7eec76ef45752305a61b90a8897551ade556422
local       542d6aa2b58b0025297a88a900fb12acb6afb4a59372922175184911a92f91f7
local       e4547da59ed051b595ae462858d50a48d3a84d276fc6ad018846f03cd9586dbd
local       e36493d67bd5492ae417041851259b8d80411e3b3efd5a23f4f1fa0ed197c39c
local       e441452e07ad9d7a998ea6916362353f63c123e92ff77ec77c809eede94e93b1
```

Figure 4.3 - List of the volumes before docker compose is run

After we run the docker compose file we can see that new volume has been added to our list of volumes. In addition, volumes are often a better choice than persisting data in a container's writable layer, because a volume doesn't increase the size of the containers using it, and the volume's contents exist outside the lifecycle of a given container. Figure 4.4 shows the list of the volumes after docker compose is run.

```
nekofetishist@fedora:~/Study/Master/Web-Application-Development/MidTerm$ docker volume ls
DRIVER      VOLUME NAME
local       7cd464521aeee05bd1eb4b4cb161976c9f59b46cfdb7fb7c239fd40e5d5f94a6
local       7fc699f84682e2f21fe238b34dc210ef2a2422d5e3f055f373490062d6181841
local       8a92a00cb1b8f6332c359f6a24bebd9be40cdd829322a64bbb4c4ebb6856705c
local       432c76967c234b8e4fb2be70b7eec76ef45752305a61b90a8897551ade556422
local       542d6aa2b58b0025297a88a900fb12acb6afb4a59372922175184911a92f91f7
local       e4547da59ed051b595ae462858d50a48d3a84d276fc6ad018846f03cd9586dbd
local       e36493d67bd5492ae417041851259b8d80411e3b3efd5a23f4f1fa0ed197c39c
local       e441452e07ad9d7a998ea6916362353f63c123e92ff77ec77c809eede94e93b1
local       midterm_postgres_data
local       midterm_static_volume
```

Figure 4.4 - List of the volumes after docker compose is run

Additionally, we can inspect the volumes using the command “docker volume inspect <volume-name>”. The example of an inspection for the “postgres\_data” is represented in Figure 4.5. It gives us the information about when the volume was created, its drive (here “local”), path at this point it is located, scope of the volume and so on.



```

nekofetishist@fedora: ~/Study/Master/Web-Application-Development/MidTerm$ docker volume inspect midterm_postgres_data
[
  {
    "CreatedAt": "2024-10-25T15:38:03+05:00",
    "Driver": "local",
    "Labels": {
      "com.docker.compose.project": "midterm",
      "com.docker.compose.version": "2.29.7",
      "com.docker.compose.volume": "postgres_data"
    },
    "Mountpoint": "/var/lib/docker/volumes/midterm_postgres_data/_data",
    "Name": "midterm_postgres_data",
    "Options": null,
    "Scope": "local"
  }
]

```

Figure 4.5 - Information of the volume

So, we can gather information about drivers, labels, names, and available options of a selected network.

## 5. Django application setup

The section is responsible for describing the project's configuration, settings, created applications, approach, used package manager, etc. Before the beginning of the own representation of the work done, we need to set up the initial project using python. Here "python3", especially, "python3.12" version applied for the whole work. We need to create a virtual environment to store all the required packages along with their specific version to be used, especially in the current version. To achieve this role the command "python3 -m venv venv" was executed to create a folder "venv" that contains all the information about interpreter. To run a virtual environment, run the command: "source venv/bin/activate". To install django using the command "pip3 install django" (latest django version) and run the command "django-admin startproject <project-name>". Figure 5.1 illustrates customized django's project configuration.

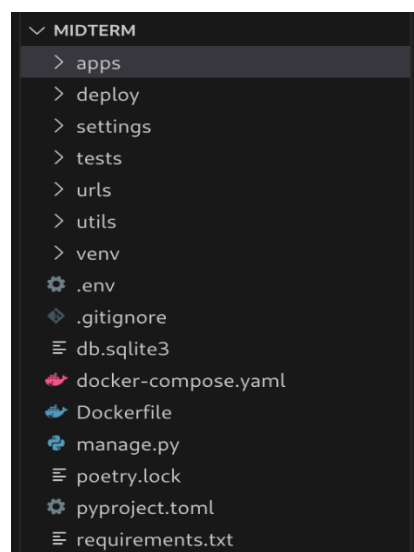


Figure 5.1 – Project's files tree

The project's default configuration of the project has been changed for a custom one. The project's configuration has been changed because of personal experience as well as a native understanding of files locations as the project would get bigger. Inside the "apps/" folder, you'll find all the project's applications separated by its names. The "settings/" folder stores the project's configuration files, such as separate settings for development and production. There's also a "tests/" folder, which contains tests to make sure everything in the project works as expected. The "urls/" folder manages the links in the app and tells the system which page to load for each URL.

The "utils/" folder contains reusable helper functions used across the project. Files like ".env" store private settings (like passwords) safely, and ".gitignore" ensures unnecessary files are not tracked by Git (such as the virtual environment or temporary files). For deployment, there are Docker tools available: "docker-compose.yaml" helps run multiple services (like databases and the Django app) together, and the "Dockerfile" explains how to build the app inside a container. Finally, the "manage.py" file helps you interact with Django from the command line (e.g., running the server or making database changes). "requirements.txt" lists the libraries the app needs, while "pyproject.toml" and "poetry.lock" help manage these dependencies using Poetry, a modern tool for handling project libraries. This structure ensures everything is well-organized, easy to develop, and ready for deployment. The project itself includes DRF (Django Rest Framework), it also uses JWT (JavaScript Web Tokens) for authentication. Let's now examine project's settings ("settings" folder). Figure 5.2 shows the files under the "settings" folder.

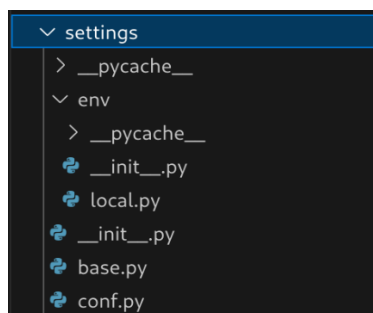
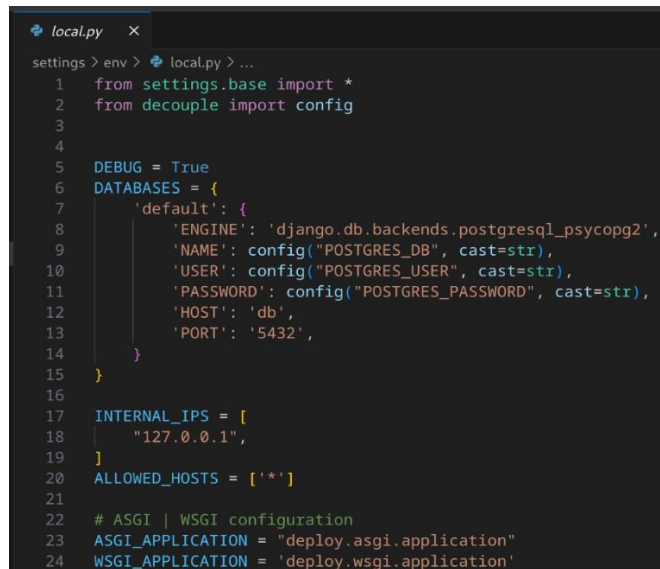


Figure 5.2 – Structure of the project settings

The project's settings are divided into 2 groups: environmental and common ones. Environmental settings are located into "env" folder, here, especially, "local.py" file which contains such parameters like (debug mode, database configuration, wsgi and asgi paths) for the local machine running. Moreover, another group contains the same files configuration for all the projects modes (production or test, does not matter). It contains 2 files: "base.py" and "conf.py". The former has only base configuration for the projects such as "INSTALLED\_APPS", "MIDDLEWARE" and other main variables, while the latter is about third party libraries configuration such as "DJANGO\_DEBUG\_TOOLBAR", JWT configuration, etc. The "base.py" file just exports everything from the "conf.py" file. Let's examine "local.py" file to see more details (Figure 5.3).



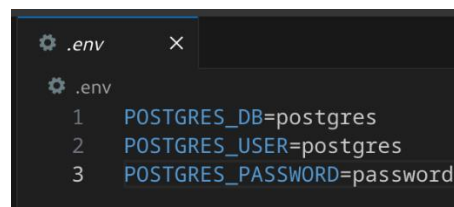
```

local.py
settings > env > local.py > ...
1 from settings.base import *
2 from decouple import config
3
4
5 DEBUG = True
6 DATABASES = {
7     'default': {
8         'ENGINE': 'django.db.backends.postgresql_psycopg2',
9         'NAME': config("POSTGRES_DB", cast=str),
10        'USER': config("POSTGRES_USER", cast=str),
11        'PASSWORD': config("POSTGRES_PASSWORD", cast=str),
12        'HOST': 'db',
13        'PORT': '5432',
14    }
15 }
16
17 INTERNAL_IPS = [
18     '127.0.0.1',
19 ]
20 ALLOWED_HOSTS = ['*']
21
22 # ASGI | WSGI configuration
23 ASGI_APPLICATION = "deploy.asgi.application"
24 WSGI_APPLICATION = 'deploy.wsgi.application'

```

Figure 5.3 – Local file configuration

It can be clearly seen that files contain specific configuration related to the current environment, which is set, e.g. for the local machine we use postgres as a database, debug mode is enabled to see the trace back of the code as an error appears. It could be noticed that we just extend the “base.py” file which also include “conf.py” inside, and the required variables such as database name, user’s name in a database or password are set “.env” (located in the root project folder) file for managing some hidden variables from the developer. Figure 5.4 contains all the environmental variables which are used for both the database and back-end parts.



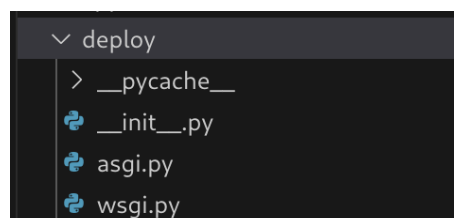
```

.env
.env
1 POSTGRES_DB=postgres
2 POSTGRES_USER=postgres
3 POSTGRES_PASSWORD=password

```

Figure 5.4 – Database and back-end environment variables

The naming for variables would be described in project’s launching section. It is left to explain what “deploy” folder contains. Figure 5.5 demonstrates the explorer of the files of a “deploy” folder.



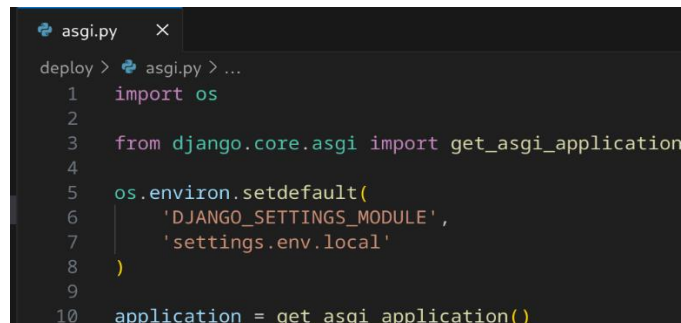
```

deploy
├── __pycache__
├── __init__.py
├── asgi.py
└── wsgi.py

```

Figure 5.5 – Deployment files

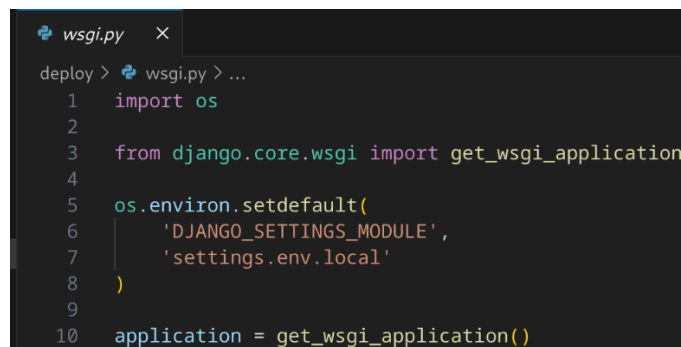
So, there are 2 main files and empty “\_\_init\_\_.py” file for the Django to see the files. There are 2 files “asgi.py” and “wsgi.py” which are named as Asynchronous Server Gateway (ASGI) Interface and Web Server Gateway Interface (WSGI) respectively. Figure 5.6 represents the customization for the ASGI.

A screenshot of a code editor showing the content of the asgi.py file. The file is named 'asgi.py' and is located in a directory named 'deploy'. The code starts with 'import os', followed by 'from django.core.asgi import get\_asgi\_application'. Then, 'os.environ.setdefault()' is used to set 'DJANGO\_SETTINGS\_MODULE' to 'settings.env.local'. Finally, 'application = get\_asgi\_application()' is assigned.

```
asgi.py
deploy > asgi.py > ...
1 import os
2
3 from django.core.asgi import get_asgi_application
4
5 os.environ.setdefault(
6     'DJANGO_SETTINGS_MODULE',
7     'settings.env.local'
8 )
9
10 application = get_asgi_application()
```

Figure 5.6 – ASGI file settings

ASGI files allow Django to communicate with asynchronous web servers, supporting features like WebSockets and background tasks. It's used if you plan to run Django with an asynchronous server like Daphne. Figure 5.7 demonstrates the code of the WSGI file.

A screenshot of a code editor showing the content of the wsgi.py file. The file is named 'wsgi.py' and is located in a directory named 'deploy'. The code starts with 'import os', followed by 'from django.core.wsgi import get\_wsgi\_application'. Then, 'os.environ.setdefault()' is used to set 'DJANGO\_SETTINGS\_MODULE' to 'settings.env.local'. Finally, 'application = get\_wsgi\_application()' is assigned.

```
wsgi.py
deploy > wsgi.py > ...
1 import os
2
3 from django.core.wsgi import get_wsgi_application
4
5 os.environ.setdefault(
6     'DJANGO_SETTINGS_MODULE',
7     'settings.env.local'
8 )
9
10 application = get_wsgi_application()
```

Figure 5.7 – WSGI file configuration

WSGI file is an entry point for synchronous web servers like Gunicorn or WSGI to communicate with your Django app. It's used when you are deploying Django in a production environment. By default, all the synchronous operations of the Django go through the Web Server Gateway Interface.

Let's discuss the URL paths which direct our queries to our logic processing. There is a single file “urls.py” which is used to collect all the paths and redirect them to a set view (here ViewSet). The file “urls.py” is located in the “urls” folder (Figure 5.8).

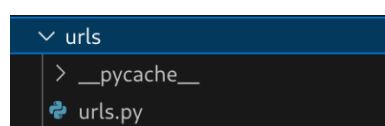


Figure 5.8 – Location of the URL file

Generally, it plays a role in routing web requests to the appropriate views (functions or classes that generate a response). It defines the URL patterns that your application will respond to. Figure 5.9 shows the code of the “urls.py” file.

```
from rest_framework.routers import DefaultRouter

from django.contrib import admin
from django.urls import path, include

from apps.tasks.views import UserViewSet, TaskViewSet

urlpatterns = [
    path('admin/', admin.site.urls),
    path('api-auth/', include('rest_framework.urls')),
    path('api/token/', TokenObtainPairView.as_view(), name='token_obtain_pair'),
    path('api/token/refresh/', TokenRefreshView.as_view(), name='token_refresh'),
    path('api/token/verify/', TokenVerifyView.as_view(), name='token_verify'),
]

# -----
# API Endpoints
#
router: DefaultRouter = DefaultRouter(trailing_slash=False)

router.register(
    prefix="auths/users",
    viewset=UserViewSet,
    basename="user"
)
router.register(
    prefix="tasks/tasks",
    viewset=TaskViewSet,
    basename="task"
)

urlpatterns += [
    path("api/v1/", include(router.urls)),
]
```

Figure 5.9 – Code for the URL routing

Initially, it is necessary to import all the packages, such as, router, path, and include in order to register all the endpoints’ paths. It creates several paths: one for the Django admin interface, another for authentication-related endpoints, and specific JWT token management endpoints. The code then uses a “DefaultRouter” to automatically generate RESTful API routes. It registers two “viewsets”: one for user-related operations under the path “auths/users” and one for task-related operations under “tasks/tasks”. Finally, it includes the router’s URLs under the API version (here “v1”). So, if a user decides to register, or login, he/she will follow “http://host-name/api/v1/auths/users” and then an appropriate name like “login” and “register” as a suffix of a route.

## 6. Defining Django Models

The current section contains information about Django’s models, their application, and describes the models which were used here. When we talk about models in Django, we mean the tables which we want to create in a database. Django’s approach for creating tables in a database is to use classes where single class may represent only one model. Figure 6.1 shows “AbstractBaseModel” in our application.

```
1 from django.db.models import (
2     DateTimeField,
3     Model,
4 )
5
6
7 class AbstractBaseModel(Model):
8     """
9     AbstractBaseModel database model, holding data we need for every model
10    in the different apps.
11    """
12
13    created_at: DateTimeField = DateTimeField(
14        auto_now_add=True, verbose_name="Date and time of creation"
15    )
16    updated_at: DateTimeField = DateTimeField(
17        auto_now=True,
18        verbose_name="Date and time of last update",
19    )
20
21    class Meta:
22        """Customization of the table view."""
23
24        abstract: bool = True
25
```

Figure 6.1 – Abstract base model

The model class above is meant to be an abstract base model that other models can inherit. It provides common fields that can be reused across multiple models. The field “created\_at” stores the date and time when a record is first created, while “updated\_at” store the date and time when the record was last updated.

All the model classes have a subclass “Meta” to emphasize the meta data of the model like naming in admin panel, etc. Here we set the parameter “abstract” to be on which means that this class will not create a database table. Instead, it will serve as a base for other models to inherit these common fields. Meta class, basically, is applied to add the configuration related to the whole table as one entity. For example, in meta class we can determine a verbose name of the class, unique columns, default ordering of the columns, constraints related to a model when we add a new row into the table. Let’s move to the last class model which is shown in Figure 6.2.



```

class Task(AbstractBaseModel):
    """class to represent a task database model."""

    TITLE_MAX_LENGTH: int = 150
    STATUS_DONE = 1
    STATUS_DONE_TEXT = 'Done'
    STATUS_IN_PROGRESS = 2
    STATUS_IN_PROGRESS_TEXT = 'In progress'
    STATUS_PLANNED = 3
    STATUS_PLANNED_TEXT = 'Planned'
    STATUS_CANCELLED = 4
    STATUS_CANCELLED_TEXT = 'Cancelled'
    STATUS_CHOICES = (
        (STATUS_DONE, STATUS_DONE_TEXT),
        (STATUS_IN_PROGRESS, STATUS_IN_PROGRESS_TEXT),
        (STATUS_PLANNED, STATUS_PLANNED_TEXT),
        (STATUS_CANCELLED, STATUS_CANCELLED_TEXT),
    )

    # Title of the task.
    title: str = CharField(
        max_length=TITLE_MAX_LENGTH,
        db_index=True,
        verbose_name='Title',
        help_text="Title of the task."
    )

    # Description of the task.
    description: str = TextField(
        blank=True,
        null=True,
        verbose_name='Description',
        help_text="Description of the task (detailed explanation)."
    )

    # Owner of the task.
    owner: User = ForeignKey(
        to=User,
        on_delete=CASCADE,

```

Figure 6.2 – Task database class model

The “Task” model inherits from “AbstractBaseModel”. This means it automatically gets the “created\_at” and “updated\_at” fields from the base class, which track when a task was created and last updated. The constants for the whole class are stored on the top of a model. The model “Task” by itself involves the “tasks” database table where each field instance by itself describes the columns of this table. Let’s look at the fields. The field “title” stores the task title (a short text) with a maximum length of 150 characters, “description” is an optional text field for a detailed explanation of the task, “owner” is a foreign key linking the task to a user (built in model class in Django), “status” stores the task's status as an integer field with predefined choices.

As you implement the models based on which you want to create your database tables, you need to create migration files and then migrate them. Creating migration files in Django involves generating files that track changes to your database schema, such as adding, modifying, or removing models, fields, or constraints. To create migration files, you need to execute the following command in the terminal “python3 manage.py makemigrations”. After that there are several (or one) new generated files in the “migrations” folder. Figure 6.3 demonstrates the project’s initial migration file.

```

0001_initial.py
apps > tasks > migrations > 0001_initial.py
1 # Generated by Django 5.1.1 on 2024-09-24 06:34
2
3 import django.db.models.deletion
4 from django.conf import settings
5 from django.db import migrations, models
6
7
8 class Migration(migrations.Migration):
9
10     initial = True
11
12     dependencies = [
13         migrations.swappable_dependency(settings.AUTH_USER_MODEL),
14     ]
15
16     operations = [
17         migrations.CreateModel(
18             name='Task',
19             fields=[
20                 ('id', models.BigAutoField(auto_created=True, primary_key=True, serialize=False, verbose_name='ID')),
21                 ('created_at', models.DateTimeField(auto_now_add=True, verbose_name='Date and time of creation')),
22                 ('updated_at', models.DateTimeField(auto_now=True, verbose_name='Date and time of last update')),
23                 ('title', models.CharField(db_index=True, help_text='Title of the task.', max_length=150, verbose_name='Title')),
24                 ('description', models.TextField(blank=True, help_text='Description of the task (detailed explanation).', null=True, verbose_name='Description')),
25                 ('status', models.IntegerField(choices=[(1, 'Done'), (2, 'In progress'), (3, 'Planned'), (4, 'Cancelled')], default=3, verbose_name='Status')),
26                 ('owner', models.ForeignKey(help_text='Owner of the task.', on_delete=django.db.models.deletion.CASCADE, related_name='tasks', verbose_name='Owner')),
27             ],
28             options={
29                 'verbose_name': 'Task',
30                 'verbose_name_plural': 'Tasks',
31             },
32         ),
33     ]

```

Figure 6.3 – Initial migration file for the “Task” model

No need to understand what happens in these files (but here we just create new “tasks” table) while nothing is crashed there. Migration files are special pythonic files which do DDL actions related to a database. It is just a bridge between a class model and a real database table. Then as new migrations are created you can apply the command “python3 manage.py migrate”. This command is used to apply the migration changes to your database and updates the actual database schema according to the migration file.

## 7. Back-end views

The chapter describes the views (endpoints) that the project has and demonstrates their successful work. The “views.py” file contains class-based view sets that are used to process specific endpoints to decide: retrieve data, create new data, change or delete them. We used Django Rest Framework (DRF) to create RESful API. All the crucial endpoints will be described here. To install Django Rest Framework in your environment, use the following command: “pip3 install djangorestframework”. Additionally, you need to add “rest\_framework” attribute into the installed app of your application.

### 7.1 User-related endpoints

As we have a user, we need to create endpoints to register a new user and login him/her. Figure 7.1 shows the code for the endpoint to register a new user.

```

@action(
    methods=["POST"],
    detail=False,
    url_path="register",
    url_name="register",
    permission_classes=(AllowAny,),
    authentication_classes=[],
)
@validate_serializer_data(
    serializer_class=CreateUserModelSerializer
)
def register_user(
    self,
    request: DRFRequest,
    *args: tuple[Any, ...],
    **kwargs: dict[Any, Any]
) -> DRFResponse:
    """Method to handle register user request."""
    serializer: CreateUserModelSerializer = kwargs.get("serializer")
    new_user: User = serializer.save()
    new_user.password = make_password(password=request.data["password"])
    new_user.save(update_fields=["password"])
    return DRFResponse(
        data=DetailUserModelSerializer(
            instance=new_user,
            many=False
        ).data,
        status=HTTP_200_OK
    )

```

Figure 7.1 – Registration endpoint for a new user

According to a Figure, the method (since it is a function of a class) is wrapped by the decorator “action” which determines that we process only “POST” type requests with the path which suffix is “register” and allow any user to be registered. Moreover, it is wrapped by one more serializer which validates the input data for POST requests. “CreateUserModelSerializer” is used to validate data. Serializers are used to validate JSON typed data to understand whether they are valid or not. Figure 7.25 illustrates the implementation of “validate\_serializer\_data” decorator.

```

@wraps(func)
def wrapper(
    request: DRFRequest | Any,
    *args: tuple[Any, ...],
    **kwargs: dict[Any, Any],
):
    # Get the serializer's validated data and check
    req: Any = request

    if not isinstance(request, DRFRequest):
        for arg in args:
            if isinstance(arg, DRFRequest):
                req = arg
                break

    if not "request" in context:
        context["request"] = req

    serializer: Serializer = serializer_class(
        data=req.query_params
        if check_query_params
        else req.data,
        context=context,
    )
    if serializer.is_valid():
        kwargs["validated_data"] = serializer.validated_data.copy()
        kwargs["serializer"] = serializer
        return func(req, *args, **kwargs)
    else:
        return DRFResponse(
            data=serializer.errors,
            status=HTTP_400_BAD_REQUEST,
        )

```

Figure 7.2 - Implementation of “validate\_serializer\_data” decorator

These decorators check and validate the request data using a DRF serializer before running the main function. If the data is valid, it passes the validated data to the function; if not, it returns an error with HTTP “400” (BAD REQUEST). This ensures the function only gets correct data to work with. After we get valid data, we create a new user, set his/her password to be hashed and return this data to a client back. The simulation of the registration process is shown in Figure 7.3.

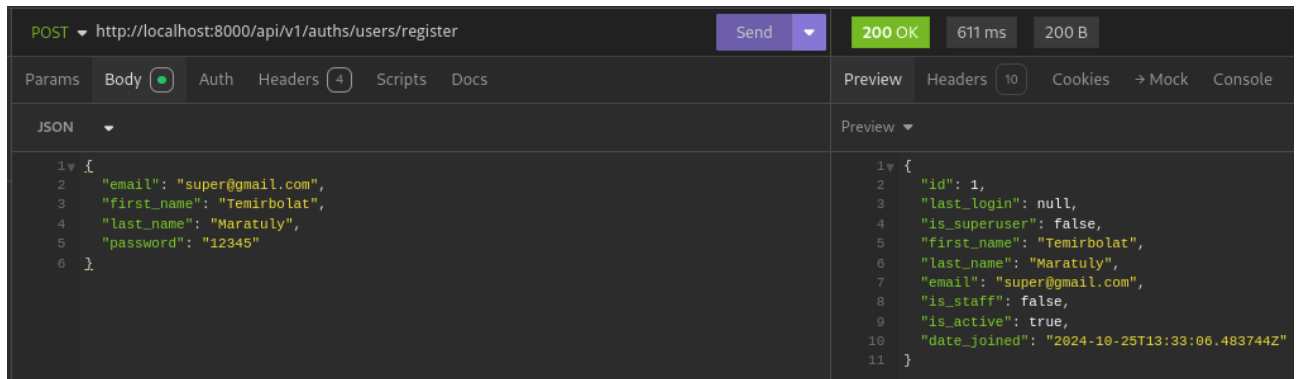


Figure 7.3 – Registration process successful simulation

As we registered a new user, we need to login now. Figure 7.4 shows the code implementation for the login endpoint.

```
@action(
    methods=["POST"],
    detail=False,
    url_path="login",
    url_name="login",
    permission_classes=(AllowAny,),
    authentication_classes=[],
)
@validate_serializer(data[
    serializer_class=LoginUserSerializer
])
def login(
    self,
    request: DRFRequest,
    *args: tuple[Any, ...],
    **kwargs: dict[Any, Any]
) -> DRFResponse:
    """Method to handle login request."""
    user: User = kwargs.get("validated_data").get("user")
    # login(request=request, user=user)

    # Generate refresh and access tokens for the user and set into response data
    refresh_token: RefreshToken = RefreshToken.for_user(user=user)
    response: DRFResponse = DRFResponse(
        data=DetailUserModelSerializer(
            instance=user,
            many=False
        ).data,
        status=HTTP_200_OK
    )
    response.set_cookie(key='refresh_token', value=str(refresh_token), httponly=True, secure=True, samesite='Lax')
    response.set_cookie(key='access_token', value=f"JWT {str(refresh_token.access_token)}", httponly=True, secure=True, samesite='Lax')
    response.set_cookie(key='csrftoken', value=generate_secure_random_string(length=64), httponly=False, secure=True, samesite='Lax')
    return response
```

Figure 7.4 – Login endpoint implementation

Here we validate the incoming POST data of the URL with the suffix “login”. As the data is valid (email and password) we generate access and refresh tokens since we use JWT as an authentication. We set both JWT tokens and CSRF into response’s

cookie to make sure they are secret. The simulation of the login process is shown in Figure 7.5.

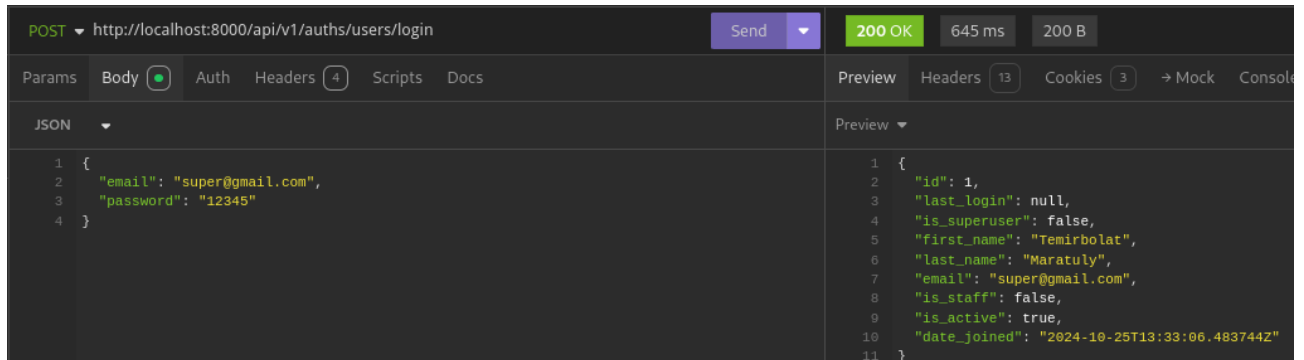


Figure 7.5 – Login process successful simulation

So, all the endpoints work properly.

## 10.2 Task related endpoints

We covered all the user’s related endpoints (login and registration) and now we can move to task’s related endpoints. Generally, any user wants to view his/her own created list of tasks, add a new task, delete their own task or change its status. All these endpoints are created. Figure 7.6 shows the code about viewing user’s own tasks.

```
class TaskViewSet(DRFResponseHandler, ViewSet):
    """View set to handler Task related requests."""

    queryset = Task.objects
    permission_classes = tuple(
        Type[BasePermission],
        Type[BasePermission]
    ) = (
        IsAuthenticated,
        IsTaskOwner,
    )

    serializer_class = Type[TaskBaseModelSerializer] = TaskBaseModelSerializer
    authentication_classes = list[Any] = []

    def list(
        self,
        request: DRFRequest,
        *args: tuple[Any, ...],
        **kwargs: dict[Any, Any]
    ) -> DRFResponse:
        """Handle GET-request to get all the tasks."""

        return self.get_drf_response(
            request=request,
            data=request.user.tasks.all(),
            serializer_class=TaskListModelSerializer,
            many=True
        )
```

Figure 7.6 – Listing endpoint to view own tasks

According to the Figure, we use a built-in method from “ViewSet” class but override it. To reach this endpoint we must be authenticated and be the owner of this

task. As a response we will return a list of all users' own tasks. Next, discuss the implementation of "IsTaskOwner" permission because it is a custom permission. Figure 7.7 shows the code of a custom permission.

```
class IsTaskOwner(BasePermission):
    message: str = "You are not the owner of the task."

    def has_object_permission(
        self, request: DRFRequest, view: Any, obj: Task
    ) -> bool:
        return bool(
            request.user and
            request.user.is_authenticated and
            request.user.id == obj.owner_id
        )
```

Figure 7.7 – Implementation of a custom permission

So, to create a custom permission we need to create a class and extend "BasePermission" class from DRF. Moreover, there are 2 methods ("has\_object\_permission" and "has\_permission") where you need to override them based on the situation. if the current user is allowed to access a specific object. In this case, it returns True if the user is authenticated and their ID matches the owner ID of the task; otherwise, it returns False. If the permission check fails, the user will see the message: "You are not the owner of the task." This ensures that only the task owner can access or modify their task.

As we understood how the permissions work, let's return to the tasks' endpoints. The next step is to create a new task. An endpoint for creating a new task is shown in Figure 7.8.

```
@validate_serializer_data(
    serializer_class=TaskCreateModelSerializer
)
def create(
    self,
    request: DRFRequest,
    *args: tuple[Any, ...],
    **kwargs: dict[Any, Any]
) -> DRFResponse:
    """Handler POST-request to create a new task instance."""

    serializer: TaskCreateModelSerializer = kwargs.get("serializer")
    new_task: Task = serializer.save()
    return self.get_drf_response(
        request=request,
        data=new_task,
        serializer_class=TaskDetailModelSerializer
    )
```

Figure 7.8 - Endpoint to create a new task



Firstly, the code validates the request data using the “TaskCreateModelSerializer”. If the data is valid, it saves the new task to the database. Then, it returns a response with the task’s details, formatted by the “TaskDetailModelSerializer”. The decorator ensures only valid data is processed, making the creation process smooth and error-free. As a user decides to delete a task, he/she can direct the request to an endpoint “delete” to get rid of this endpoint. The realization of deleting a task is shown in Figure #.

```
@find_queryset_object_by_query_pk(
    queryset=Task.objects,
    class_name=Task,
    entity_name="Task"
)
def delete(
    self,
    request: DRFRequest,
    *args: tuple[Any, ...],
    **kwargs: dict[Any, Any]
) -> DRFResponse:
    """Handle DELETE-request to drop your owned task."""
    task: Task = kwargs.get("object")
    self.check_object_permissions(request=request, obj=task)
    task.delete()
    return DRFResponse(
        data="Your task has been succesfully deleted",
        status=HTTP_204_NO_CONTENT
    )
```

Figure 7.9 - Realization of a delete method

After all the above realizations of the endpoints we can understand that it is much easier and more convenient to implement general decorators, handlers or mixins to reuse them, so we do not create a huge code and use deduplication.

## CONCLUSION

To sum up, we managed to create a task management application successfully using Django along with the Django Rest Framework (DRF), integrated it with PostgreSQL as a powerful database, and deployed it with Docker. The project demonstrated the practical usage, benefits of applying containerization with Docker to have an isolated, with minimal amount of necessary packets environment, ensuring that the application behaves the same regardless of the host system.

Moreover, JWT (JSON Web Tokens) were involved into the project for authentication aims, Docker compose simplified the management of multiple containers which created custom volumes to store static files, database data and networks to organize the looped communication between the backend (Django) and the PostgreSQL database.