KBTU KAZAKH-BRITISH TECHNICAL UNIVERSITY

Midterm

Web Application Development

Building a Task Management Application

Using Django

Prepared by:
Seitbekov S.
Checked by:
Serek A.

Almaty, 26.10.2024

Table of Contents

Executive Summary

In this rapid pace cycle of software development, the consistency of the development, test, and production environments is highly demanded. This project is powered by containerization, showing how one can build a task management application in Django and then deploy it into Docker containers. The developed Task Management Application will provide the user with clear-cut ways to perform all CRUD operations: Create, Read, Update, Delete. Besides Django, other technologies such as Redis for caching, Swagger for API documentation, Grafana for monitoring, and GitHub Actions for automating CI/CD make the project more full-bodied.

Through Docker Compose, one orchestrates a multi-container setup whereby each of these services run in an isolated container. The integration of PostgreSQL shall be implemented to have persistent data storage using Docker Volumes for persistence. GitHub Actions automate the deployment pipeline, pushing the application into the production server. The production server will be installed with SSL certificates to ensure safety while communicating with the client. This project will deploy a web application that is scalable, secure, ready for production, and uses tools like Redis for caching, with monitoring through Grafana combined with Docker.

Introduction

The ever-growing importance of the role it plays in today's software industry has made containerization indispensable in modern-day web development. Packing an application and its dependencies into isolated, portable containers solves typical problems of environment inconsistencies. The application would thus behave in the same way across a local development, staging, and production environment. Where Docker plays a huge role in that matter, Docker Compose overtakes the orchestration of several services like a Web Application and Database.

The Django-based Task Management application remains the centerpiece of this project, where a user gets an opportunity to manage tasks by performing common CRUD operations. For this exercise, Django was chosen because all its features are of high order: comfortable ORM for manipulations with the database in the easiest way, and API development within this framework. Redis is used as the cache engine-in other words, to increase application speed by unloading it from the most frequent queries. It decreases response time. The Swagger format applied to API documentation makes working with endpoints of this application easier for a developer.

GitHub Actions were used for the application deployment; thus, automated building and testing are performed, and new code is pushed to the server. That continuous integration/continuous deployment pipeline means every change in the code will go through proper testing and deployment without requiring any manual intervention. Hence, it increases the productivity of the whole team. The server also has SSL certificates, meaning communication between the clients and the application is encrypted. On top of this, application performance is monitored in real time using Grafana; in other words, a provider of insight into system metrics to keep the service slick.

The combination of Django, Redis, Swagger, Docker, and Grafana, having automated deployments via GitHub Actions, tells volumes about modern DevOps practices: how web applications should be built, how they could be deployed, and how they can be monitored. The task management application serves not only learning purposes but testifies as to how one is able to produce a production-ready, secure, and scalable application.

Project Objectives

The objective of the project is not to just build a task management application but develop a scalable, production-ready solution with proper infrastructure and automation. Key goals listed here:

1. Build Functional Web Application: Create a task management application using Django, which shall be capable of enabling the users to add, edit, delete, and view tasks through an intuitive interface.
2. Redis Caching: Redis caching to frequently accessed data to improve performance and reduce response time.
3. GitHub Actions for Automation of CI/CD: The building, and application deployment should be automated using GitHub Actions for enabling continuous delivery to the production server, pushing to Docker Hub and sending notification to Telegram.
4. API Swagger Documentation: Document APIs using Swagger; thus, API endpoints will be understandable and easy to handle.
5. Grafana - Setup Monitoring: Container system metrics could be tracked by including Grafana for maintaining an application stable under various loads and scenarios.
6. Dockerize the Application: Create a Dockerfile that will package the Django application and all its dependencies into a Docker image so that it works consistently across different environments.
7. Multi-Container Setup: The docker-compose.yaml should be instrumented to run multiple services, such as a Django web application, PostgreSQL database, and Redis, by defining how they will talk to each other.
8. Setup SSL Certificates for Securing Application: The app is to be deployed with valid SSL certificates to ensure that a user and the server can communicate securely.
9. Persistent Data with Docker Volumes: To persist the data created in PostgreSQL, Docker Volumes will be used.
10. Server Deployment: The containerized application should be ready to deploy on any server or cloud platforms, such as DigitalOcean or Kubernetes, and easily scalable to meet future needs.

The project is also meant to expose developers to the process and experience of web development, containerization, caching, automation, monitoring, and deployment. It provides hands-on experience in how modern technologies work together to construct and maintain safe, reliable, high-performance web applications.

Intro to Containerization: Docker

Containerization is one of the recent strategies in deploying software, whereby an application and its related dependencies get packaged in one standardized, light-weight unit called a container. Assuredly, the containers ensure that software behavior remains consistent across the board, from when a developer operates it on his or her local machine to higher environments, such as test and production environments.

Unlike Virtual Machines, containers share the kernel of the host operating system and, therefore, are much more resource-efficient and faster to deploy. The key benefits of containerization include:

1. Portability: Containers behave the same on any system that has Docker installed, hence

solving the "it works on my machine" problem.

2. Isolation: Each container runs its processes in isolation, and all dependencies of a container are installed in that one single container, hence preventing conflicts.
3. Scalability: More containers can be run side by side with an individual container to distribute workload; thus, horizontal scaling is easily achieved.
4. Uniformity: Containerization keeps the same setting across development, testing, and production. This minimizes unexpected bugs.
5. Faster Deployment: Containers start up much faster, since all they package is the essentials, leveraging the host OS for everything else. Improvement in Collaboration: Developers will also package their code into containers and share them with teams, which is a nice way of working because everyone is in the same environment.

In this project, Docker is used for containerization to ensure that the Django-based task management application works seamlessly everywhere. Docker keeps the app, and its dependencies insulated from each other so that modification or updating those does not interfere with other components or services.

Installation of Docker

At the very beginning of the containerization process, Docker was installed on the development machine.

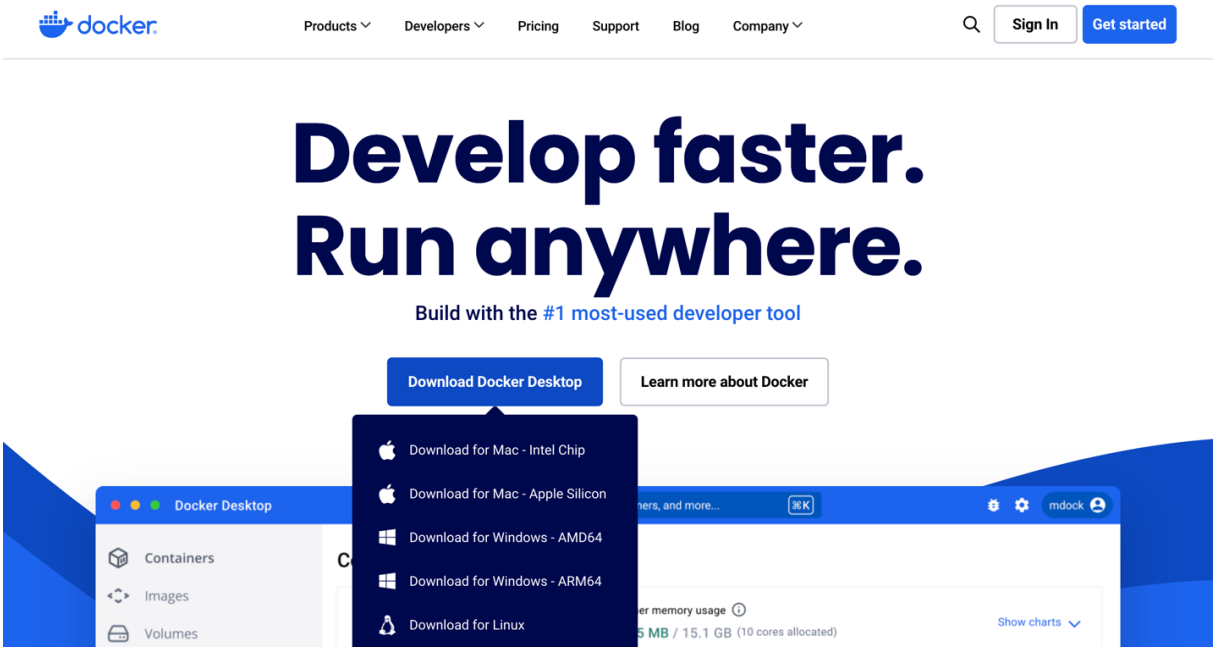

Figure 1. The Docker website

Click on the link to download a Docker Desktop operating system version: Windows, macOS, or Linux. Follow all instructions on the screen until installation of Docker is completed. In case Docker requires some additional components to be installed give the permission to do so. When installation was completed, the following command was executed to verify that Docker was properly installed:

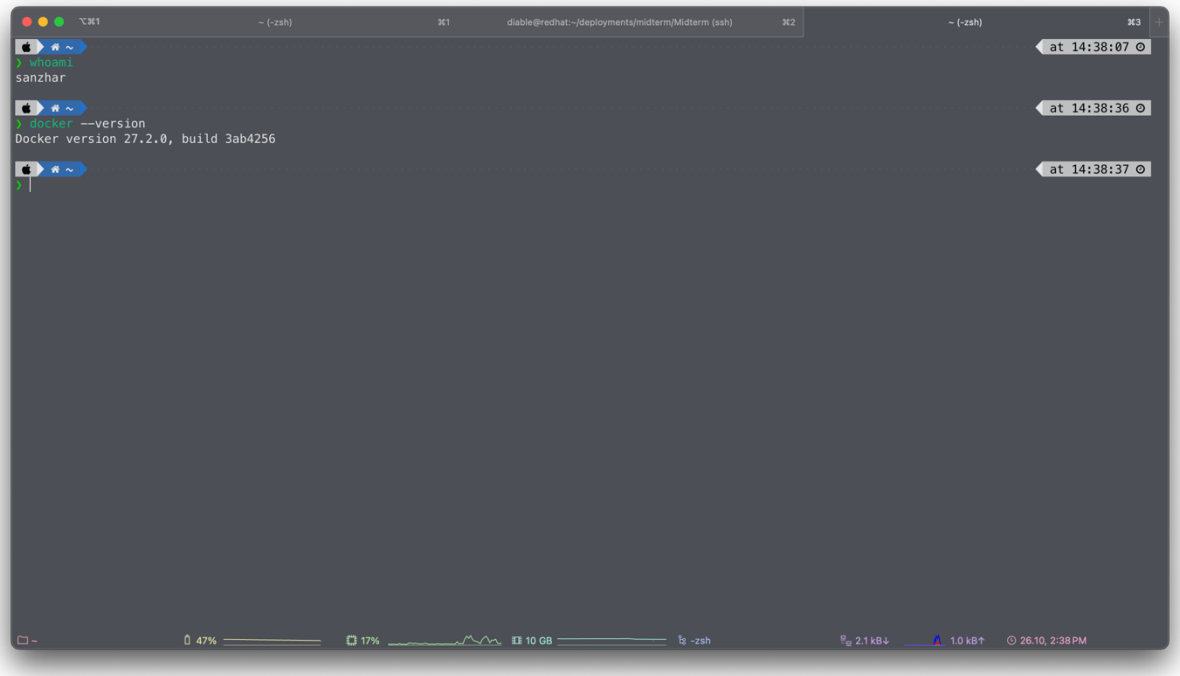

Figure 2. Docker version output

The status of Docker was supposed to be further asserted if it worked as anticipated through the running of a "Hello World" container with the following command:

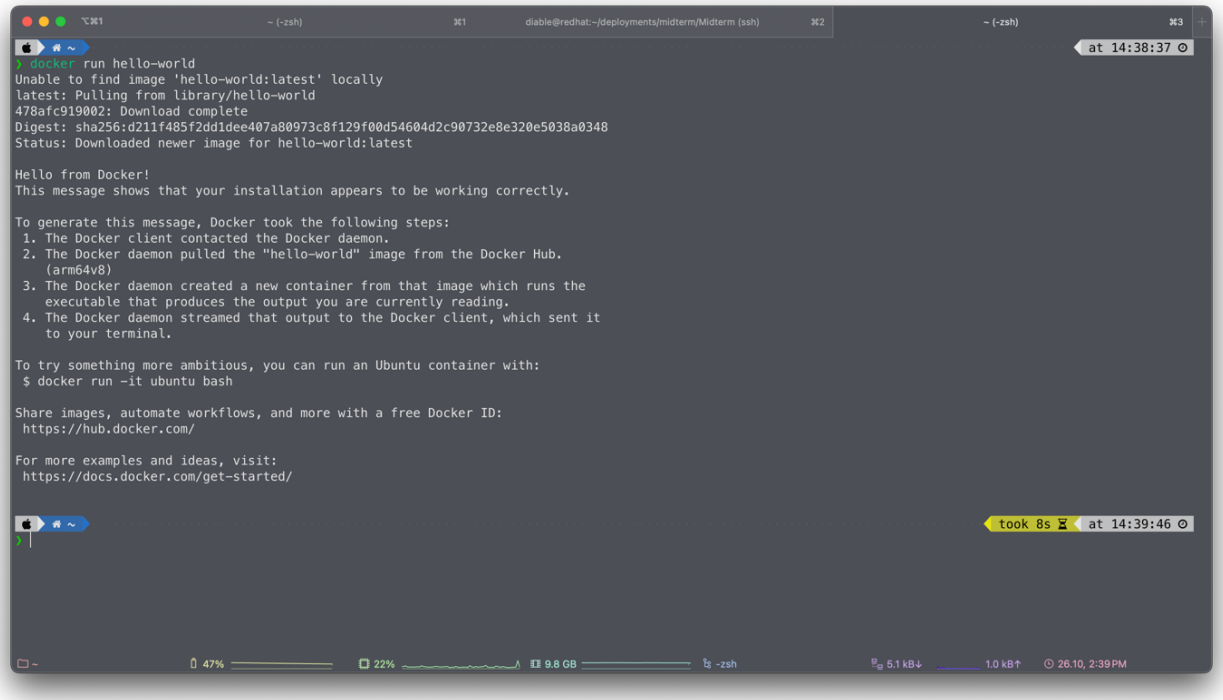

Figure 3. Docker hello world output

The command does a pull for the hello-world image from Docker Hub, creates a container from it, and then runs the container. If Docker was installed correctly, the container outputs a message that confirms Docker works properly.

The Docker installation itself had already verified that everything was working fine by running these initial containers. Docker will provide the basis for most of the remaining parts of the project since Docker Compose will smoothly orchestrate the Django application, PostgreSQL database, and Redis caching service. Once installed, the focus will shift to containerizing the task management app through the creation of a Dockerfile to further ease the deployment process.

Creating a Dockerfile

This section will focus on the Dockerfile, which is an important configuration in containerizing the Django-based task management application. A Dockerfile is a syntax of instructions for building a Docker image. In this project, I will be using a multistage build to create an optimized container that reduces image size by separating the build process from the final runtime environment. That way, I copy only the needed files and dependencies will make it into the production container, and it will be lightweight and efficient.

```
FROM python:3.12-slim AS builder

ENV PYTHONDONTWRITEBYTECODE=1
ENV PYTHONUNBUFFERED=1

RUN apt-get update && apt-get install -y --no-install-recommends \
    build-essential \
    libpq-dev \
    && rm -rf /var/lib/apt/lists/*

WORKDIR /app

COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

COPY . .

FROM python:3.12-slim   Sanzhar, Yesterday • added: Midterm

ENV PYTHONDONTWRITEBYTECODE=1
ENV PYTHONUNBUFFERED=1

RUN apt-get update && apt-get install -y --no-install-recommends \
    netcat-openbsd \
    && rm -rf /var/lib/apt/lists/*

WORKDIR /app

COPY --from=builder /usr/local/lib/python3.12/site-packages /usr/local/lib/python3.12/site-packages
COPY --from=builder /usr/local/bin /usr/local/bin

COPY --from=builder /app /app

EXPOSE 8000

RUN chmod +x /app/todo/scripts/entrypoint.sh

ENTRYPOINT ["/app/todo/scripts/entrypoint.sh"]

up-to-date  Blame: Sanzhar 25.10.2024, 18:10  18:22  LF  UTF-8  4 spaces  Python 3.12 (Midterm)
```

Figure 4. Dockerfile of project.

Dockerfile consists of two major parts: Build Stage and Final Runtime Stage. It follows a pattern called multi-stage build. Such a design helps in keeping the final image smaller by excluding unnecessary build-time tools and dependencies from the production environment.

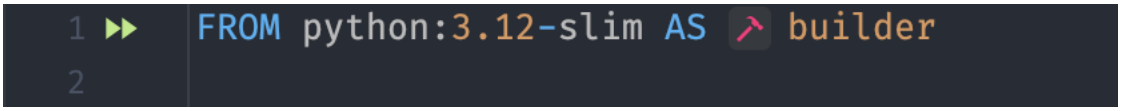

Figure 5. Python:3.12-slim Builder Image

This uses the python:3.12-slim image, which is a minimal version of Python that's optimized for Docker containers. The "slim" variant ensures the image remains lightweight, focusing only on the essential components required to run Python applications. I declare this first stage as builder, which I can refer to later in the final deploy stage.

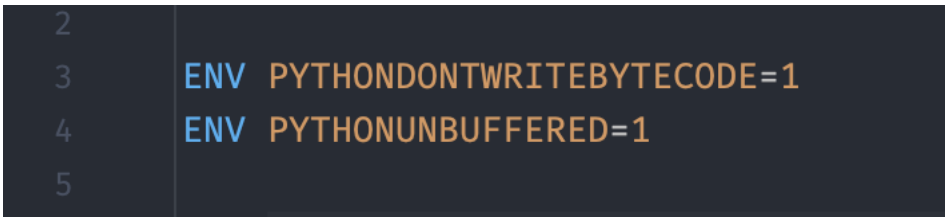

Figure 6. Docker environment variables

The following environment variables optimize the behavior of Python within a container during build:

1. PYTHONDONTWRITEBYTECODE=1: This prevents Python from writing .pyc files to disk and decreases superfluous I/O.
2. PYTHONUNBUFFERED=1: This is so that Python will output the logs straight to the console; this is a prerequisite for real-time logs to appear in Docker.

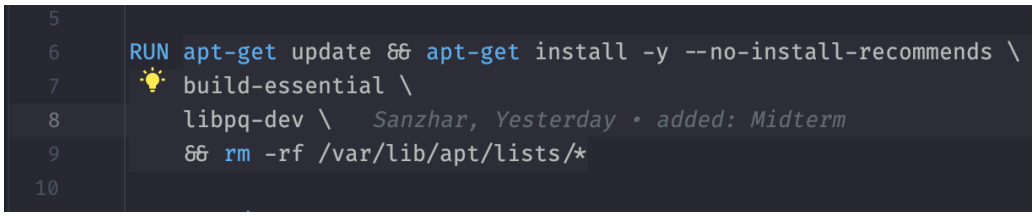

Figure 7. Install necessary linux packages.

Here I update and install packages without any optional dependencies and after that remove cached package lists to free up space:

1. Build-essential: this package contains essential tools like gcc, g++ etc.
2. Libpq-dev: library required for building PostgreSQL development.

```
11  WORKDIR /app
12
13  COPY requirements.txt .
14  RUN pip install --no-cache-dir -r requirements.txt
15
16  COPY . .
```

Figure 8. Install python requirements and copy codebase.

I set the working directory to /app and copy the requirements.txt listing Python dependencies. pip install -r requirements.txt will install the required packages inside the builder stage. After that I copy the source code of the application into the container's /app directory, making the codebase available for the build process.

```
17
18  FROM python:3.12-slim
19
```

Figure 9. Final stage image

The final stage uses the same minimal Python image as in the builder stage but does the exact opposite. It reduces to the bare running of the application without installing any of dependencies that might have been used by the build phase.

```
22
23  RUN apt-get update && apt-get install -y --no-install-recommends \
24      netcat-openbsd \
25      && rm -rf /var/lib/apt/lists/*
26
```

Figure 10. Install netcatd-bsd in final stage.

Here, I install netcat-openbsd, a lightweight utility to check on network connections. This is handy within Docker containers to make sure services like the PostgreSQL database is up and running before the Django app starts.

```
27  WORKDIR /app
28
29  COPY --from=builder /usr/local/lib/python3.12/site-packages /usr/local/lib/python3.12/site-packages
30  COPY --from=builder /usr/local/bin /usr/local/bin
31
32  COPY --from=builder /app /app
```

Figure 11. Copy the packages and codebase to final stage image.

After that, I set workdir to /app. COPY --from=builder commands copy the essential components from the builder stage into the final runtime container such as python libraries and executable bin commands. I am copied to ensure that the runtime environment has all the dependencies. The application code from /app gets copied in to make the Django project available to the final container.

```
33
34  EXPOSE 8000   Sanzhar, Yesterday • added: Midterm
35
```

Figure 12. Expose 8000 port for Django application.

The EXPOSE instruction informs Docker that the application will run on port 8000. This is the default port for the Django development server. So, I can access to Django application from outside.

```
35
36  RUN chmod +x /app/todo/scripts/entrypoint.sh
37
38  ENTRYPOINT ["/app/todo/scripts/entrypoint.sh"]
39
```

Figure 13. Make entrypoint script executable and execute it.

This makes sure that the entrypoint script is executable and would hence run correctly when this container starts. The entrypoint script here takes care of all the start-up tasks, for example, waiting for the database to be ready before launching the Django server.

ENTRYPOINT is an instruction that allows the entry of the container. It specifies a script to be run when a container is started. This, in effect, makes the app self-contained because once started, the app should be able to execute its own start-up operation without additional commands.

One of the immediate benefits I get from using multi-stage build is the ability to take advantage of separating the build environment from the runtime environment. The builder stage consists of all the tools and libraries that must be present during compilation, installing the dependencies that later aren't necessary once the application has been built. The final stage includes only what's necessary for running the Django app; it is smaller and, by implication, more efficient. This approach helps improve security by reducing the attack surface since unnecessary tools are excluded from the final image.

This Dockerfile rapidly and efficiently containerizes the Django-based task management application. Employment of multi-stage builds ensures the container remains lightweight and optimized. Inclusion of netcat and a custom entrypoint script guarantees smooth startup and reliable service operation. With this setup, the application is ready to run consistently from local development to production.

Using Docker Compose

The following section describes the configuration of docker-compose.yaml used to orchestrate multiple services working over a Django task management application. Docker Compose provides an easy way to manage complex multi-container environments by keeping each service-for example, a Django app, a PostgreSQL database, or Redis-isolated while enabling smooth communication across Docker networks. I have different services in my docker-compose file:

1. Django Web Application: Provides the task management application.
2. PostgreSQL Database: Provides data persistence for the application.
3. Redis: Used to improve the performance through caching.

Above that, volumes make sure that data persist when containers restart, and a custom network is used so services could communicate with each other securely. Next, let's cover the structure of the docker-compose.yaml file and of each definition of the services in detail.

```yaml
services:
  web:
    image: diable201/midterm:latest
    volumes:
      - ./static:/app/static
      - ./media:/app/media
      - ./certs/fullchain.pem:/app/certs/fullchain.pem:ro
      - ./certs/privkey.pem:/app/certs/privkey.pem:ro
    ports:
      - "8000:8000"
    env_file:
      - .env
    depends_on:
      - db   Sanzhar, Yesterday • added: Midterm
      - redis
    networks:
      - app-network

  db:
    image: postgres:17-alpine
    environment:
      POSTGRES_USER: ${POSTGRES_USER}
      POSTGRES_PASSWORD: ${POSTGRES_PASSWORD}
      POSTGRES_DB: ${POSTGRES_DB}
      POSTGRES_PORT: ${POSTGRES_PORT}
    ports:
      - "${POSTGRES_PORT}:${POSTGRES_PORT}"
    volumes:
      - postgres_data:/var/lib/postgresql/data/
    networks:
      - app-network

  redis:
    image: redis:6-alpine
    ports:
      - "6379:6379"
    networks:
      - app-network

volumes:
  postgres_data:
  static:
  media:
```

Figure 14. docker-compose of project.

Structure of docker-compose.yaml

Let's think about docker-compose.yaml as having logical structure described in items below in order:

1. Services: The application needs containers of hosting services.
2. Volumes: deals with persistent volumes that store data across the restarts of containers.
3. Networks: create an isolated network to communicate with each other securely.

```yaml
services:
  web:
    image: diable201/midterm:latest
    volumes:
      - ./static:/app/static
      - ./media:/app/media
      - ./certs/fullchain.pem:/app/certs/fullchain.pem:ro
      - ./certs/privkey.pem:/app/certs/privkey.pem:ro
    ports:
      - "8000:8000"
    env_file:
      - .env
    depends_on:
      - db
      - redis
    networks:
      - app-network   Sanzhar, Yesterday • added: Midterm

```

Figure 15. Django web service part.

Here I use my diable201/midterm:latest image which contains Django application. The container exposes 8000 port which is mapped to 8000 ports on the host machine, so I can access to Django application. The app contains sensitive data, such as credentials and they are managed using .env file. This app depends on Redis and PostgreSQL, so I ensure that these containers are started first. The app-network network provides communication with other services. Volumes ensure persistence of static and media files, and certificates for SSL connection.

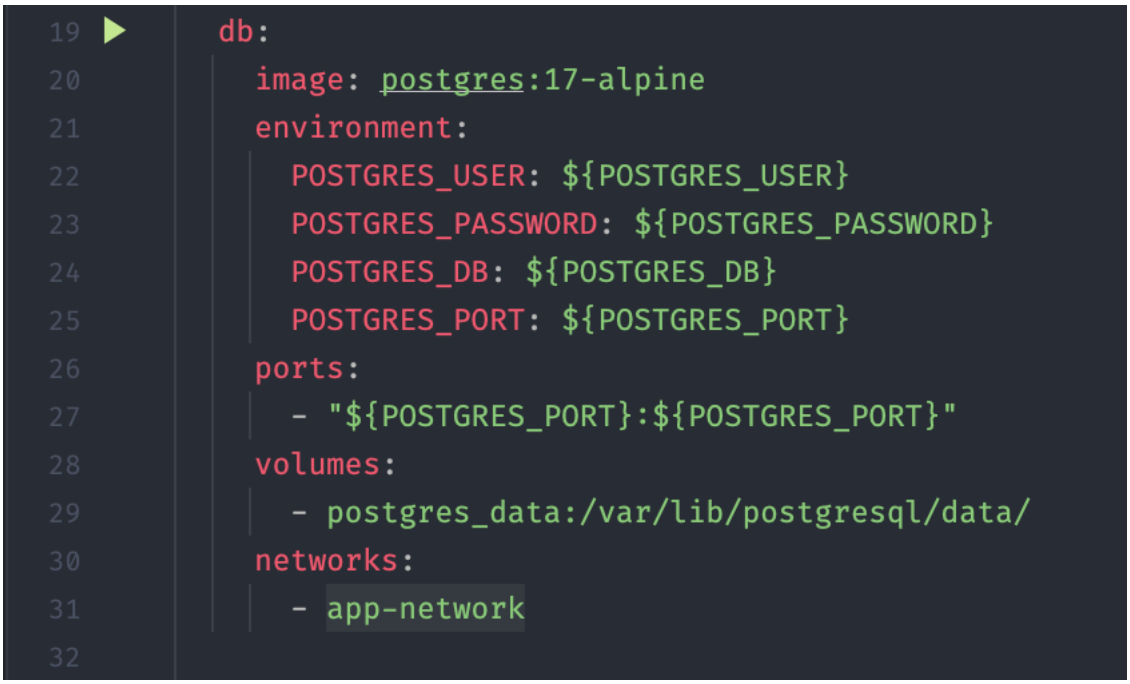

Figure 16. PostgreSQL service part.

Here I use postgres:17-alpine image for database. It is the lightweight version of image for small size and minimal resource usage. Variables such as POSTGRES_USER, POSTGRES_PASSWORD and POSTGRES_DB are taken from .env file just like in Django part of docker-compose. This service is exposed to POSTGRES_PORT (default 5432) while volume postgres_data stores persistent data of database, preventing data loss. It also uses the same app-network for communication.

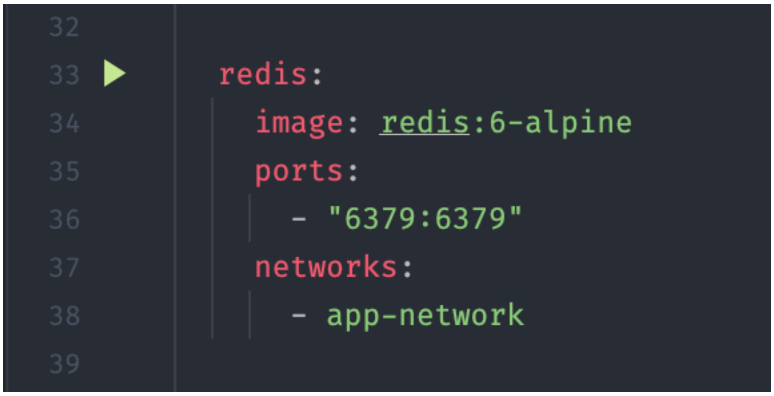

Figure 17. Redis service part.

I use redis:6-alpine image for Redis. Like for database, it is the lightweight version of image. Redis is exposed to standard port 6379 making it accessible for Django application. And it uses the same app-network for secure communication between backend web side, database and Redis itself.

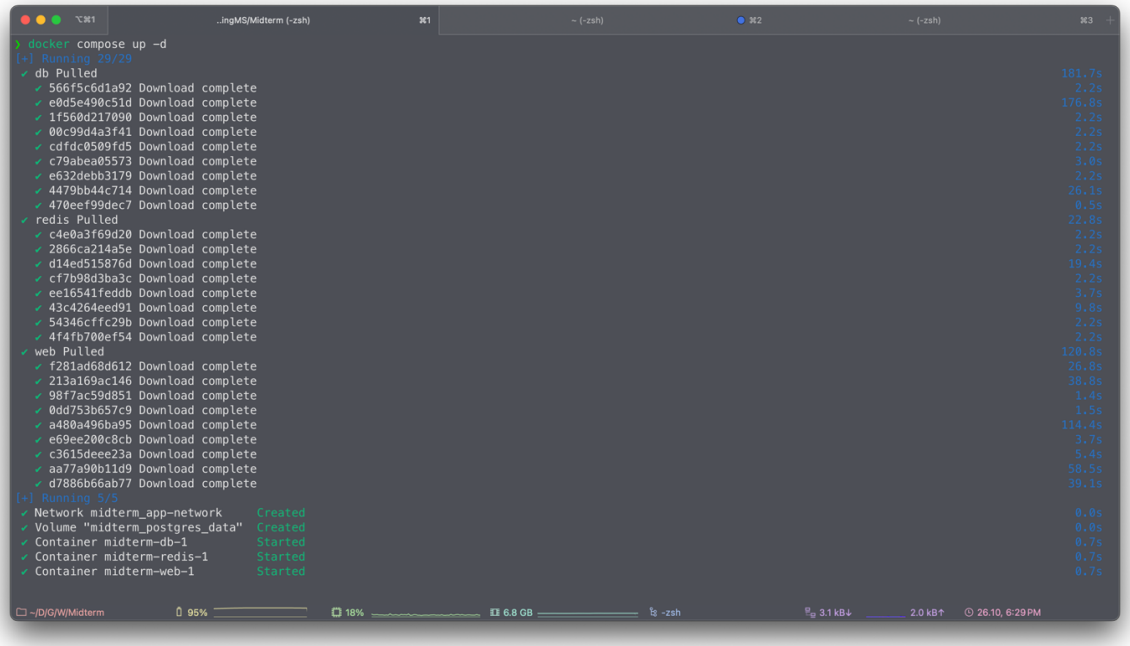

Figure 18. Docker compose building.

Here I build my docker application using docker compose up command. So, I start all the services in one command, knowing they all work properly and can communicate with each other. With volumes, data will persist between restarts, and the bridge network will allow for secure, isolated communication between containers.

Docker Networking and Volumes

In this section, I am going to look at how Docker networking allows the containers to communicate with each other and how volumes could provide persistence for the important data, such as database records and media files. These two concepts networking and volumes are crucial to make the Django-based task management application run robustly and efficiently inside a multi-container environment.

The Docker networking allows containers to communicate with other containers in a secure and efficient manner. This brings all parts of the task management application-Django, PostgreSQL, and Redis-to work as one system.

```
44
45  networks:
46    app-network:
47      driver: bridge
48
```

Figure 19. Docker network.

Here, the app-network is created as a bridge network. This will isolate the application services from outside interference and enable the containers internally to communicate with one another. Every container attached to this network can address other containers by service name instead of being forced to hard-code IP addresses. This becomes extremely important with multi-container deployments because the containers may be restarted or redeployed, which could change their internal IP addresses.

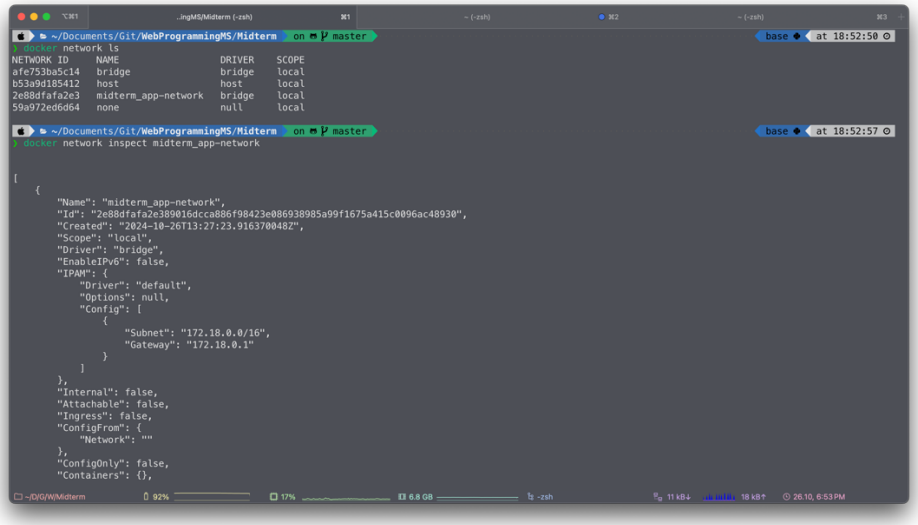

Figure 19. Docker networks and inspection of app-network.

I use command docker network ls to make sure that my network is successfully created. Also, I inspect this network with docker network inspect midterm_app-network command. From the output I can see the details of network such bridge driver, gateway ip, local scope and so on.

The configuration provides a bridge network on which all services - web, db, and redis will connect. This will allow easy communication between containers, exactly that the task management application requires.

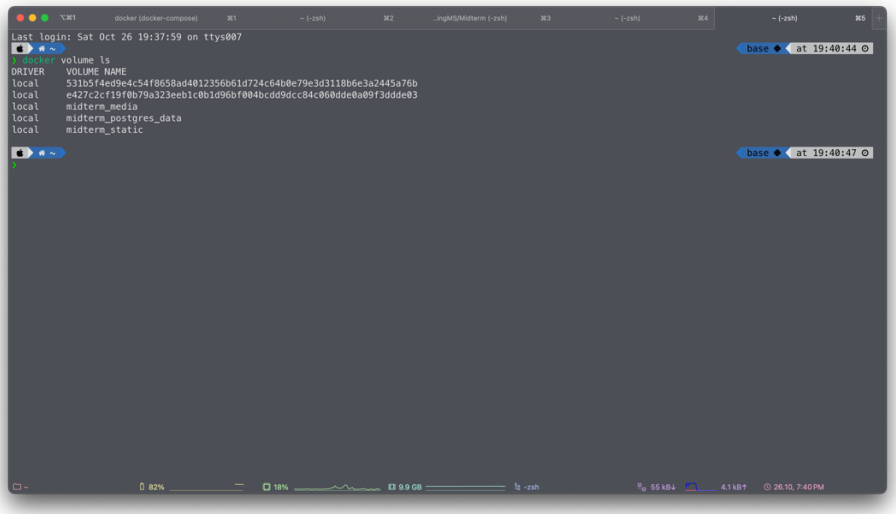

Figure 19. Docker volumes on system.

In Docker, volumes are a way to persist data by storing it outside of the container's writable layer. This way, when a container is stopped, removed, or redeployed, data is not lost. In this project, volumes would be highly instrumental in maintaining database integrity and storing media and static files. In this project, the following volumes are defined:

Figure 20. Docker volumes in docker-compose.yaml.

I have 3 volumes for my application. Postgres_data stores PostgreSQL data. This volume ensures that database for our application is never lost, even if the container is restarted. Static volume stores static files such CSS styling and JS scripts for Django admin page, so it persists across container restarts. Media volume is for user uploaded media files such as PDF files for example. It also ensures persistence of media content during restarts.

Docker networking and volumes ensure the stability, reliability, and performance of the task management application. That is because Docker networking provides the capability for containers to communicate efficiently and share runtime data, while volumes ensure that crucial data-for example, database records, static, and media content-are never lost if a container restarts or gets replaced. On the back of such powerful Docker features, the application gains resiliency and readiness to face all real-world loads and demands without loss in user experience.

Django Application Setup

In this section, I will cover the initialization of the Django project and how to make it work with the PostgreSQL database defined in Docker Compose. Since this project uses Django REST Framework to build APIs, and Gunicorn for serving the application, I'll describe how to set up and configure the entrypoint script so that the application starts properly in a containerized environment.

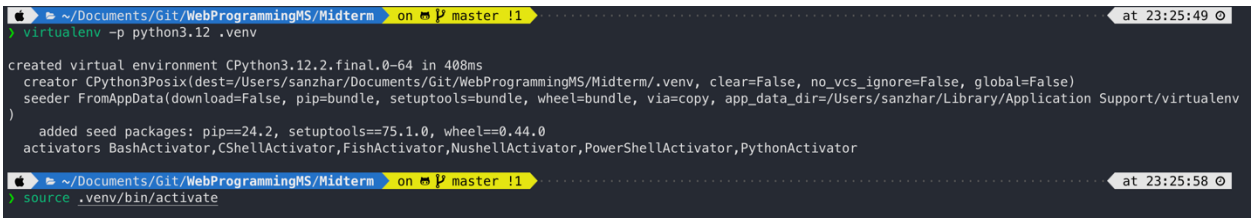

Figure 21. Creating of venv using python3.12

I use virtual environment (venv) for python development since it's the best practice. It creates isolated environment exactly for this project, so I don't need to worry about dependencies issues. I use python3.12 as its fresh and stable version of python.

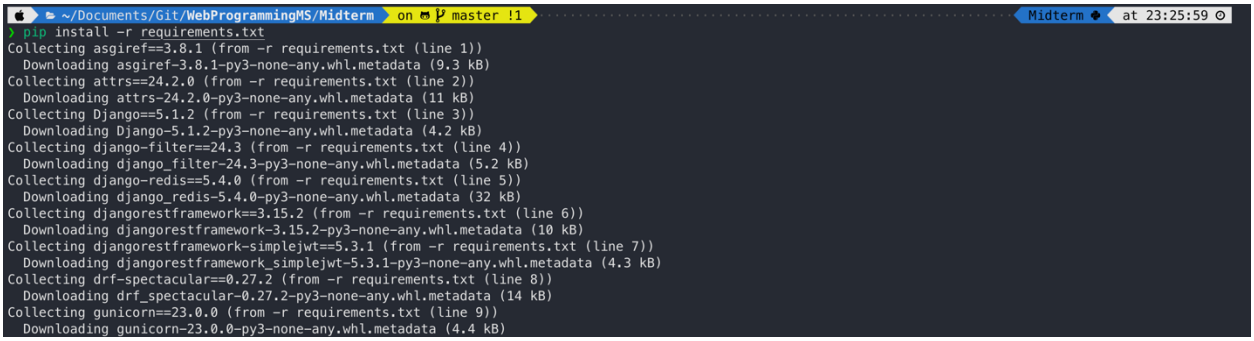

Figure 22. Installing the requirements from requirements.txt

Next step is to install requirements.txt. It contains required libraries such as Django, drf, drf-spectacular, gunicorn, psycopg2-binary and other dependencies. Also, it contains versions of libraries, so I don't need to worry about conflict issues.

Figure 22. Django-admin creation of project and apps.

I create Django application using command django-admin startproject todo. It will create Django application with necessary structure. After that I run Django-admin startapp tasks, Django-admin startapp mixins, Django-admin startapp accounts for necessary parts of project. I showed history of commands in terminal.

Figure 22. Structure of todo directory.

In settings.py I wrote all settings for Django project, including connection to database, redis, rest framework settings, jwt settings etc. I use dotenv library for loading sensitive data such as passwords, host, login from .env file.

```
95  DATABASES = {
96      "default": {
97          "ENGINE": "django.db.backends.postgresql",
98          "NAME": os.getenv("POSTGRES_DB"),
99          "USER": os.getenv("POSTGRES_USER"),
100         "PASSWORD": os.getenv("POSTGRES_PASSWORD"),
101         "HOST": os.getenv("POSTGRES_HOST"),
102         "PORT": os.getenv("POSTGRES_PORT"),
103     }
104 }
105
```

Figure 23. Database configuration.

I use PostgreSQL database as engine since it's perfect choice for production application. I use os.getenv() for getting sensitive information and it can be easily managed without changing the code. This setup enables seamless interaction between Django and database.

```
105
106 CACHES = {
107     "default": {
108         "BACKEND": "django_redis.cache.RedisCache",
109         "LOCATION": os.getenv("REDIS_URL"),
110         "OPTIONS": {
111             "CLIENT_CLASS": "django_redis.client.DefaultClient",
112         },
113     }
114 }
115
116 CACHE_TTL = 60 * 5
117
```

Figure 24. Redis configuration.

The caching is configured using Redis with django-redis library acting as backend to store data for impoved perfomance. The Redis connection URL is managed through an environment variable REDIS_URL. This caching layer speeds up database queries and reducing load on the backend.

```
REST_FRAMEWORK = {
    "DEFAULT_AUTHENTICATION_CLASSES": (
        "rest_framework_simplejwt.authentication.JWTAuthentication",
    ),
    "DEFAULT_PERMISSION_CLASSES": [
        "rest_framework.permissions.IsAuthenticated",
    ],
    "DEFAULT_SCHEMA_CLASS": "drf_spectacular.openapi.AutoSchema",
    "DEFAULT_FILTER_BACKENDS": ["django_filters.rest_framework.DjangoFilterBackend"],
    "DEFAULT_PAGINATION_CLASS": "rest_framework.pagination.PageNumberPagination",
    "PAGE_SIZE": 10,
}

SPECTACULAR_SETTINGS = {
    "TITLE": "Swagger TODO API",
    "DESCRIPTION": "TODO API Swagger documentation",
    "VERSION": "1.0.0",
    "SERVE_INCLUDE_SCHEMA": False,
    "SCHEMA_PATH_PREFIX": "/api/",
    "COMPONENT_SPLIT_REQUEST": True,
    "DEEP_LINKING": True,
}
```

Figure 25. DRF and Swagger configuration.

This section defines settings for DRF, such as JWT authentication to handle token-based authentication. Also, the pagination, filtering, and permissions settings are set and ensure that API requests are handled efficiently, and only authenticated users are allowed to access resources. The use of Spectacular Swagger OpenAPI provides API schema generation, enabling easy API documentation and interaction for development.

This was a deep dive into the setup of a Django project: virtual environment, structure, and settings environment. Best practices in the structure of the project include that the code is modularized into manageable apps and the separation of static and media files have been made correspondingly. Using a virtual environment ensures dependencies are isolated. What's more, with such setup, application is all set to work effectively in both the local and production environments.

Defining Django Models

The following section describes the creation of models used within the Django task management application and the way migrations are applied for the database schema setup. Models define the structure of the data, how it's stored in the database, and the logic associated with it.

```python
class Task(TimestampMixin):  20 usages  Sanzhar
    STATUS_CHOICES = [
        ("PENDING", "Ожидает"),
        ("IN_PROGRESS", "В процессе"),
        ("COMPLETED", "Завершено"),    Sanzhar, 25.10.2024, 18:10 • added: Midterm
        ("ARCHIVED", "В архиве"),
    ]

    PRIORITY_CHOICES = [
        ("LOW", "Низкий"),
        ("MEDIUM", "Средний"),
        ("HIGH", "Высокий"),
        ("CRITICAL", "Критический"),
    ]

    user = models.ForeignKey(
        User,
        on_delete=models.CASCADE,
        related_name="tasks",
        verbose_name=_("Пользователь"),
    )
    title = models.CharField(
        max_length=255, verbose_name=_("Имя задачи"), db_index=True
    )
    description = models.TextField(blank=True, null=True, verbose_name=_("Описание"))
    status = models.CharField(
        max_length=15,
        choices=STATUS_CHOICES,
        default="PENDING",
        verbose_name=_("Статус"),
        db_index=True,
    )
```

Figure 25. Part of Task model.

The main model is Task model that represents what a single task will do, storing title, description, status, priority, due date, and user of the task. This model enables the following key features:

1. User Association: Every single task is assigned to a specific user through a ForeignKey referencing the Django User model.
2. Choice Fields: Tasks have fields for status and priority that utilize predefined choices to normalize values such as PENDING, COMLETED, IN_PROGRESS
3. The model uses methods such as archive(), change_status(), and is_overdue() to embed the key business logic within the model itself.

Once I defined model, I need to create migrations, they are used to translate model into database schema. For this I use commands python manage.py makemigrations && python manage.py migrate. After that I see that migrations are applied for database and new files are

created in migrations folder of apps accordingly. I show migration of Task model.

```python
class Migration(migrations.Migration):  Sanzhar
    initial = True

    dependencies = [
        migrations.swappable_dependency(settings.AUTH_USER_MODEL),
    ]

    operations = [
        migrations.CreateModel(
            name="Task",
            fields=[
                (
                    "id",
                    models.BigAutoField(
                        auto_created=True,
                        primary_key=True,
                        serialize=False,
                        verbose_name="ID",
                    ),
                ),
                (
                    "created_at",
                    models.DateTimeField(
                        auto_now_add=True, verbose_name="Время создания"
                    ),
                ),
                (
                    "updated_at",
                    models.DateTimeField(
                        auto_now=True, verbose_name="Время последнего изменения"
                    ),
                ),
```

Figure 26. Django migration of Task model.

This Django migration file is responsible for creating the Task model in the database. As I said before, migrations are Django's way of translating model definitions into database schema. This migration ensures that the Task table in database is created with the appropriate fields and constraints that I defined in models.py file.

```python
class TimestampMixin(models.Model):  2 usages  Sanzhar
    class Meta:  Sanzhar
        abstract = True

    created_at = models.DateTimeField(
        auto_now_add=True, verbose_name=_("Время создания")
    )
    updated_at = models.DateTimeField(
        auto_now=True, verbose_name=_("Время последнего изменения")
    )
```

Figure 27. Timestamp Mixin abstract model.

Also, I have TimeStampMixin for reusable logic. It is the abstract model that provides automatic tracking of creation and update times. It has fields created_at that records timestamp when instance is created and updated_at that records timestamp when instance is modified. By setting abstract=True this mixin can be inherited by other models without creating a separate database table.

Figure 28. UserProfile model.

This model extends the Django standard User model to store additional information such as bio and location. OneToOneField is linking each profile to a single User instance. In metadata I set verbose names and add an index on the user field for efficient querying. It also has verbose name and verbose name plural for displaying verbose name for example in Django admin page. This model has ordering by user field.

```python
@admin.register(Task)  Sanzhar
class TaskAdmin(admin.ModelAdmin):
    list_display = (
        "title",
        "description",
    )
    search_fields = ("title", "description")
    ordering = ("title",)
    readonly_fields = ("created_at", "updated_at")
    fieldsets = (
        (
            None,
            {
                "fields": (
                    "title",
                    "description",
                )
            },
        ),
        ("Время", {"fields": ("created_at", "updated_at")}),
    )
```

Figure 29. Task admin configuration.

Django gives simple and convenient way for viewing models in admin interface. This is a ModelAdmin class for managing Task objects through the admin page. It has configuration details such as list_display that dispays title and description in admin view list, search based on title and description fields, ordering by title, and makes created_at and updated_at only readable fields. Fieldsets organize field in interface by general fields and time fields.

Views and Serializers

This section describes views and serializers, which are used to handle operations of task management through DRF. Serializers do complex data type conversion-like Django model into JSON, and views handle HTTP requests and, if needed, it will interact with the database using querysets. TaskViewSet exposes a complete interface to manage tasks. The CRUD operations are provided out of the box-with caching-along with other custom actions.

Serializers in are used to transform Django models into JSON responses and validate incoming data for consistency.

```python
class TaskSerializer(serializers.ModelSerializer):  4 usages  Sanzhar
    user = UserSerializer(read_only=True)

    class Meta:  Sanzhar
        model = Task
        fields = "__all__"
        read_only_fields = ["id", "created_at", "updated_at", "user"]
```

Figure 30. Task Serializer.

This serializer converts Task object into JSON and ensures certain fields, such as id, created_at, and user, are read-only to prevent unintended modifications. It uses a nested UserSerializer to provide user information within the task data.

```python
class TaskViewSet(viewsets.ModelViewSet):  9 usages  Sanzhar
    permission_classes = [permissions.IsAuthenticated, IsOwner]
    filter_backends = [
        filters.SearchFilter,
        filters.OrderingFilter,
        DjangoFilterBackend,
    ]
    search_fields = ["title", "description"]
    ordering_fields = ["due_date", "priority", "created_at"]
    filterset_fields = ["status", "priority"]

    @extend_schema(  Sanzhar
        summary="Retrieve all tasks for the current user.",
        description="Retrieve all tasks for the current user.",
    )
    def get_queryset(self) -> QuerySet:
        if not self.request.user.is_authenticated:
            return Task.objects.none()

        cache_key = f"user_tasks_{self.request.user.id}"
        task_ids = cache.get(cache_key)

        if task_ids is None:
            tasks = Task.active_tasks().filter(user=self.request.user)
            task_ids = list(tasks.values_list( *fields: "id", flat=True))
            cache.set(cache_key, task_ids, timeout=settings.CACHE_TTL)

        return Task.objects.filter(id__in=task_ids)
```

Figure 31. Part of Task View Set.

The TaskViewSet extends viewsets.ModelViewSet, so I have all CRUD operations and additional custom actions. It is responsible for querying the database, caching tasks, filtering user-specific data and other custom actions. Here I declared that only authenticated users can access the endpoints. Also, I ensure that users can only manage their own tasks via IsOwner permission that checks request user and task related user. This ViewSet supports filtering by status and priority and allows ordering by due_date, priority and created_at. Also enabled search on title and description. Here I use caching for task queries to improve performance.

```python
    @action(detail=True, methods=["patch"])  1 usage  Sanzhar
    def archive(self, request: Request, pk: int = None) -> Response:
        task: Task = self.get_object()
        task.archive()
        self.invalidate_task_cache(task.id)
        serializer = self.get_serializer(task)
        return Response(serializer.data)
```

Figure 32. Archiving a Task.

In addition to CRUD operations, TaskViewSet defines several custom actions using @action decorator. These actions allow custom operations like archiving tasks and changing status. This function archives the task by id and marks it as archived. The cache is invalidated to reflect the changes.

```python
urlpatterns = [
    /api/tasks/v1/tasks/
    path(
        "tasks/",
        TaskViewSet.as_view({"get": "list", "post": "create"}),
        name="task-list",
    ),
    /api/tasks/v1/tasks/{pk}/
    path(
        "tasks/<int:pk>/",
        TaskViewSet.as_view(
            {
                "get": "retrieve",
                "patch": "partial_update",
                "put": "update",   Sanzhar, 25.10.2024, 18:10 • added: Midterm
                "delete": "soft_delete",
            }
        ),
        name="task-detail",
    ),
    /api/tasks/v1/tasks/{pk}/change-priority/
    path(
        "tasks/<int:pk>/change-priority/",
        TaskViewSet.as_view({"patch": "change_priority"}),
        name="task-change-priority",
    ),
```

Figure 32. URL patterns of tasks app.

I connected TaskViewSet to the Django URL router, allowing each action to be accessed through a corresponding API endpoint. For example, GET /tasks/ retrieve all tasks for authenticated user that belong to him. POST /tasks/ creates a new task. PATCH /tasks/<pk>/change-priority/ update's task priority. PATCH /tasks/<pk>/archive/ achieves task and so on.

```python
from django.urls import path
from .views import RegisterView
from rest_framework_simplejwt.views import TokenObtainPairView, TokenRefreshView

/api/accounts/v1/
urlpatterns = [
    /api/accounts/v1/register/
    path("register/", RegisterView.as_view(), name="register"),
    /api/accounts/v1/login/
    path("login/", TokenObtainPairView.as_view(), name="token_obtain_pair"),
    /api/accounts/v1/token/refresh/
    path("token/refresh/", TokenRefreshView.as_view(), name="token_refresh"),
]

```

Figure 33. URL patterns of accounts app.

I used simplejwt DRF library for authentication and registration endpoints based on JWT-authentication. TokenObtainPairView uses in /login/ endpoint to generate access and refresh token for authenticated requests. Endpoint /token/refresh uses TokenRefreshView to generate a new access token using valid refresh token. Endpoint /register/ allows users to register in application.

The whole creation of views and serializers is an important part of the applications in Django. It is very important that serializers ensure proper serialization and deserialization of tasks and user data to and from JSON format, while the TaskViewSet offers an interface to perform CRUD operations, task filtering, and complex actions like archive and changing priorities. Caching implementation is in place for better performance, while API protection brings in authentication and ownership check to protect integrity and security of the data. The app is strong, scalable, and can manage tasks via a RESTful API.

Swagger Demonstration Flow

This section will show, how authentication and task management can be affected through the Swagger API: user registration, logging in to get the tokens that guarantee access securely, creation of tasks, and listing of tasks using the provided endpoints. Each step is done in such a way that user data is handled in a secure and efficient manner, leveraging JWT authentication.

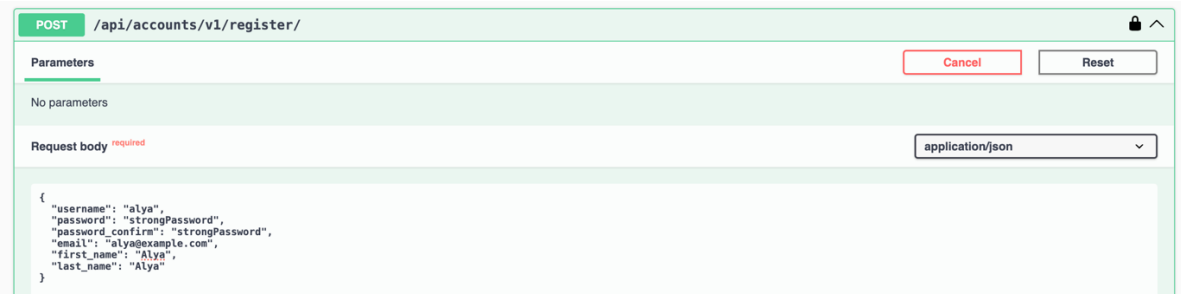

Figure 34. Swagger Register endpoint.

The first step in using the system is to register a new user. This endpoint allows users to create an account by providing essential information such as a username and password. The registered user is then stored in the database.

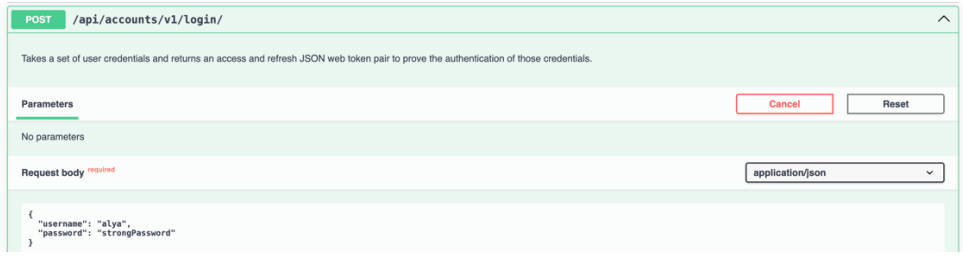

Figure 35. Swagger Login endpoint.

Once the user is registered, they need to log in to obtain an access token and a refresh token. These tokens will allow them to securely interact with the API, such as creating and listing tasks. The refresh token allows users to maintain access without frequent logins.

Figure 36. Swagger Login response.

I put into json request body correct data for user and password. So, user “alya” successfully gets access token and refresh token. I will use access token for the next API requests.

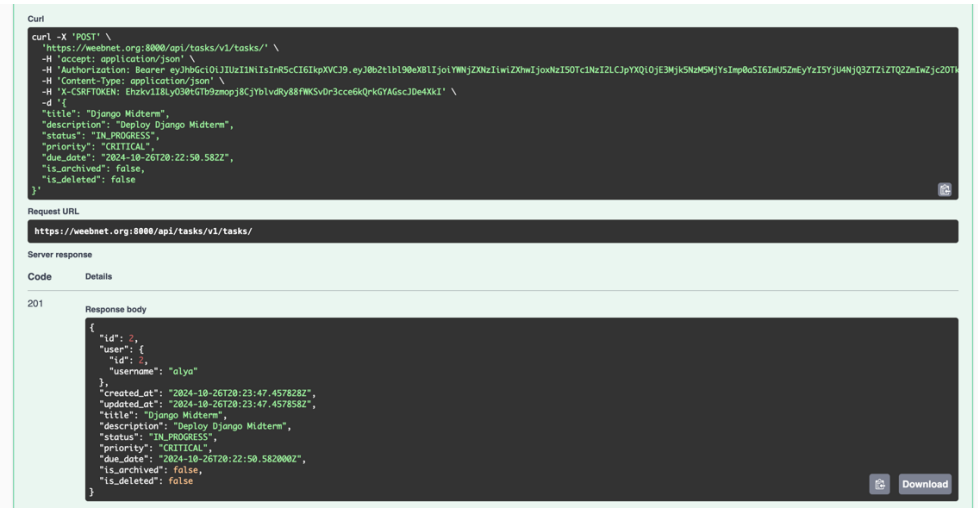

Figure 37. Swagger Create task request and response.

After logging in, I can create a task by sending a POST request with the task details. Each task is associated with the authenticated user, ensuring that only the task creator can manage it. This step allows the user to organize and track their activities.

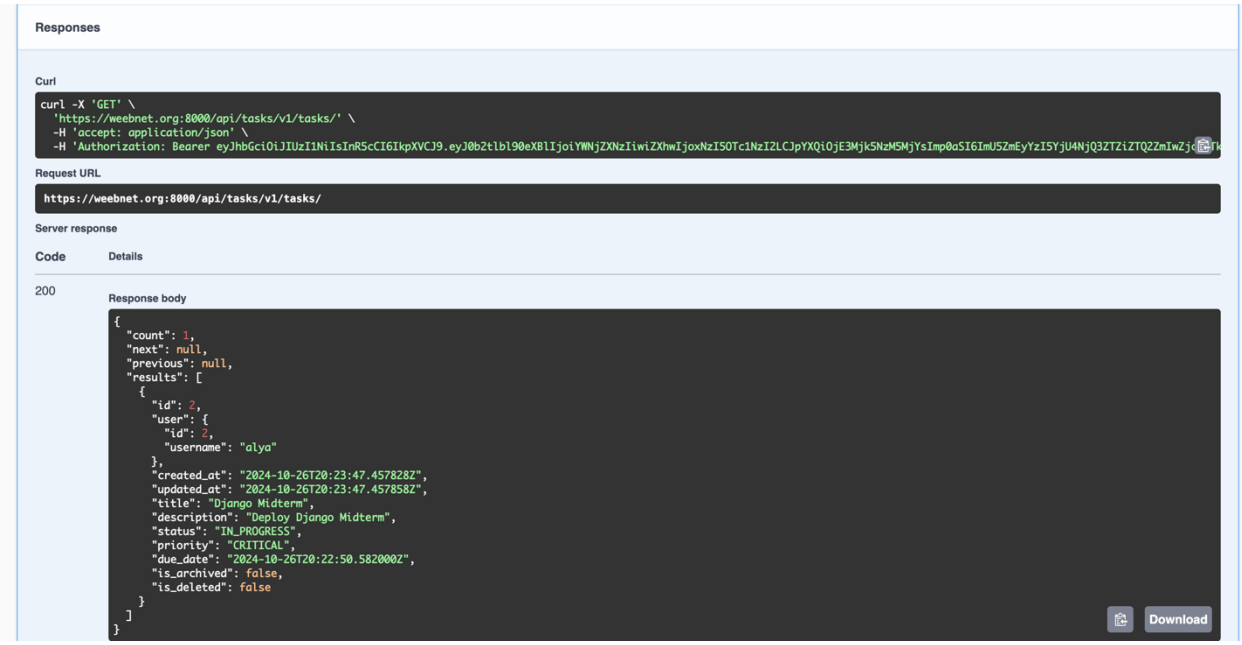

Figure 38. Swagger List tasks request and response.

Next, I can as the authenticated user can retrieve a list of all my tasks. I can also use filter or sort query parameters for tasks based on various criteria. It ensures that only the logged-in user can view their tasks, maintaining privacy and data security.

This flow shows the nice interaction between authentication and task management features of API. The system allows access to resources in a secure way, using JWT tokens, while the endpoints for tasks provide a very convenient way of creating and managing tasks. User registration, login, creation, and listing of tasks and other endpoints give an integrated experience in maintaining their tasks and becoming more productive.

Deploy workflow

For the deploy I use github actions worklow. It automates the deployment of Django application to remote server using Docker and Docker Compose. I will describe the key phases of the deployment process.

```
1  name: Deploy Midterm Django App   Sanzhar, 25.10.2024, 18:52
2
3  on:
4    push:
5      branches:
6        - master
7      paths:
8        - 'Midterm/**'
9
```

Figure 39. Deployment trigger logic.

I make that deployment is triggered only by push in master branch. Also, I specifically targeted changes only in Midterm/ directory. This ensures that deployment only occurs when relevant code or configuration changes are made

```yaml
jobs:
  deploy:
    runs-on: ubuntu-latest

    steps:
      # Step 1: Checkout the repository
      - name: Checkout code
        uses: actions/checkout@v3

      # Step 2: Set up Docker Buildx
      - name: Set up Docker Buildx
        uses: docker/setup-buildx-action@v2

      # Step 3: Log in to Docker Hub
      - name: Log in to Docker Hub
        uses: docker/login-action@v2
        with:
          username: ${{ secrets.DOCKERHUB_USERNAME }}
          password: ${{ secrets.DOCKERHUB_TOKEN }}

      # Step 4: Build and Push Docker Image
      - name: Build and Push Docker Image
        uses: docker/build-push-action@v4
        with:
          context: ./Midterm
          file: ./Midterm/Dockerfile
          push: true
          tags: diable201/midterm:latest

```

Figure 40. Deployment Docker hub pushing logic.

After this, the workflow checks out the repository, providing access to the latest project files needed for deployment. Next, it sets up Docker Buildx, enabling multi-platform builds. After securely logging in to Docker Hub using GitHub secrets, the workflow builds the Docker image from the Midterm Dockerfile and pushes it to Docker Hub under the latest tag. This ensures that the most recent version of the application is available for deployment, ready to be pulled by the server during the next steps.

```yaml
      # Step 6: Transfer docker-compose.yaml via SCP
      - name: Transfer docker-compose.yaml via SCP Action
        uses: appleboy/scp-action@v0.1.3
        with:
          host: ${{ secrets.SERVER_HOST }}
          username: ${{ secrets.SERVER_USER }}
          key: ${{ secrets.SSH_PRIVATE_KEY }}
          port: 45916
          source: "Midterm/docker-compose.yaml"
          target: "${{ secrets.REMOTE_PROJECT_PATH }}/"
          overwrite: true

      # Step 6: Create .env File on Server via SSH
      - name: Create .env File on Server
        uses: appleboy/ssh-action@v1.1.0
        with:
          host: ${{ secrets.SERVER_HOST }}
          username: ${{ secrets.SERVER_USER }}
          key: ${{ secrets.SSH_PRIVATE_KEY }}
          port: 45916
          script: |
            echo "Creating .env file on the server..."
            cat > "${{ secrets.DOCKER_COMPOSE_PATH }}/.env" <<EOF
            DEBUG=${{ secrets.DEBUG }}
            SECRET_KEY=${{ secrets.SECRET_KEY }}
            POSTGRES_DB=${{ secrets.POSTGRES_DB }}
            POSTGRES_USER=${{ secrets.POSTGRES_USER }}
            POSTGRES_PASSWORD=${{ secrets.POSTGRES_PASSWORD }}
            POSTGRES_HOST=${{ secrets.POSTGRES_HOST }}
            POSTGRES_PORT=${{ secrets.POSTGRES_PORT }}
            REDIS_URL=${{ secrets.REDIS_URL }}
            ALLOWED_HOSTS=${{ secrets.ALLOWED_HOSTS }}
            EOF
            echo ".env file created successfully."

      # Step 7: Deploy to Server via SSH and Docker Compose
      - name: Deploy to Server via SSH
        uses: appleboy/ssh-action@v1.1.0
        with:
          host: ${{ secrets.SERVER_HOST }}
          username: ${{ secrets.SERVER_USER }}
```

Figure 41. Server preparation, environment configuration, and deployment.

The workflow connects to the remote server via SSH to clean up any old docker-compose files and transfer the updated docker-compose.yaml using SCP protocol. It then generates a new .env file on the server using sensitive configuration values stored in GitHub secrets (such as database credentials, Redis URL, etc). After setting the environment, the workflow stops any existing containers, pulls the latest Docker image from Docker Hub, and starts the containers in detached mode using Docker Compose. As a final step, Django's collectstatic command is executed inside the container to gather static files, ensuring that the application is production ready.

```yaml
    # Step 8: Send Telegram Notification on Success
    - name: Send Telegram Notification - Success
      if: success()
      uses: appleboy/telegram-action@master
      with:
        to: ${{ secrets.TELEGRAM_CHAT_ID }}
        token: ${{ secrets.TELEGRAM_BOT_TOKEN }}
        message: |
          ✅ *Deployment Successful*
          The Midterm Django App has been deployed successfully! 🎉

    # Step 9: Send Telegram Notification on Failure
    - name: Send Telegram Notification - Failure
      if: failure()
      uses: appleboy/telegram-action@master
      with:
        to: ${{ secrets.TELEGRAM_CHAT_ID }}
        token: ${{ secrets.TELEGRAM_BOT_TOKEN }}
        message: |
          ❌ *Deployment Failed*
          The deployment of the Midterm Django App has failed. Please check the GitHub Actions logs for details. 😞
```

Figure 42. Telegram notifications on success or failure.

Finally, the workflow sends real-time Telegram notifications. If the deployment is successful, a message confirms that the Midterm Django App is live and deployed. In case of failure, an alert message is sent, prompting an investigation by referring to the GitHub Actions logs. This ensures that any issues are addressed promptly, reducing downtime and ensuring reliability.

Conclusion

In this Midterm Django Todo project, I showed how Django, Docker, and CI/CD automation can work together to build a robust, scalable, and maintainable application. Thanks to orchestration by Docker and Docker Compose, dealing with different services like PostgreSQL and Redis went rather smoothly and created a fluent atmosphere of development and operations.

Furthermore, the implementation of JWT-based authentication brought another layer of security and flexibility to the system. JSON Web Tokens allow the capability for users to log in and gain access to resources in a secured manner without storing session data on the server by giving statelessness and scalability to the system. Indeed, with token-based authentication from JWT, access and refresh tokens were issued and refreshed efficiently, hence providing seamless access to users regarding strong security measures.

This was further enhanced by GitHub Actions to ensure deployment automation, such that any change in the code would reliably build and deploy the pipeline. Real-time feedback is enabled via Telegram notifications, enabling developers to quickly act if something goes wrong with the deployment or even confirm updates. The collectstatic command inside the container ensures all static assets are production ready.

In conclusion, this project lays good foundations for a web application that will be both scalable and maintainable, hence allowing smooth user experiences and efficient workflows.

References

1. Django Documentation - https://docs.djangoproject.com
2. Docker Official Documentation - https://docs.docker.com
3. Django REST Framework Documentation - https://www.django-rest-framework.org
4. GitHub Actions Documentation - https://docs.github.com/en/actions
5. drf-spectacular Documentation - https://drf-spectacular.readthedocs.io
6. Redis Documentation - https://redis.io/

Appendices

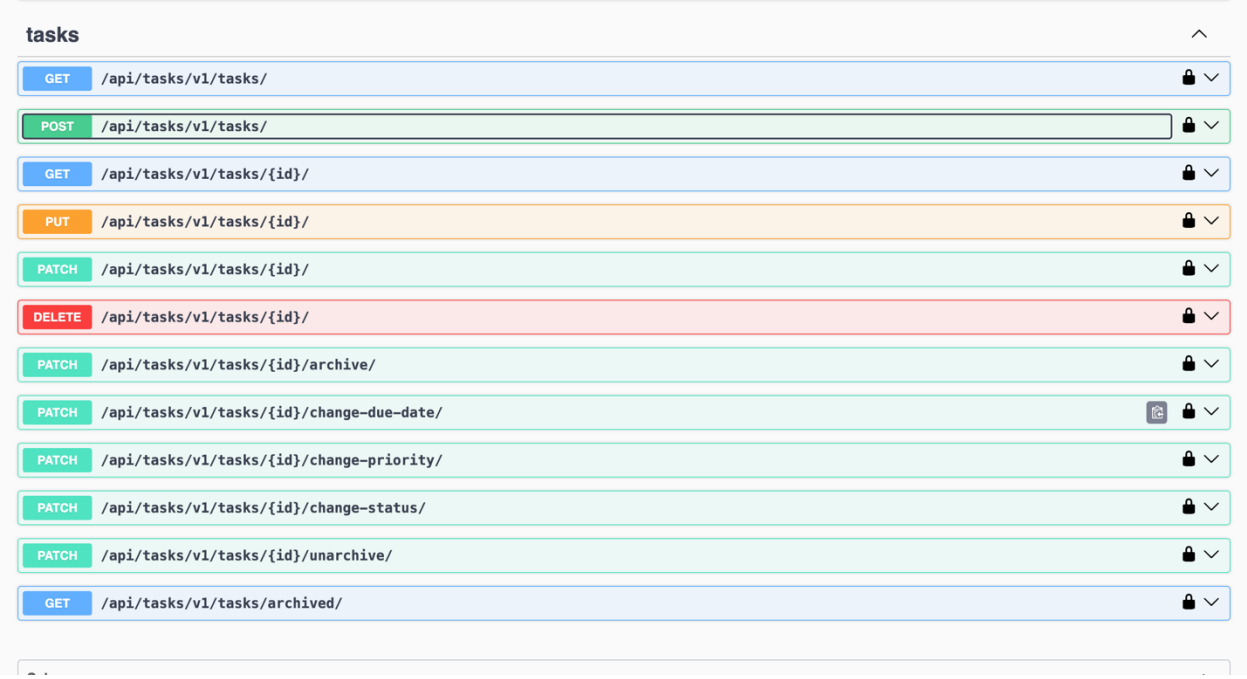

Figure 43. Swagger Tasks API.

Figure 44. Grafana Dashboard.

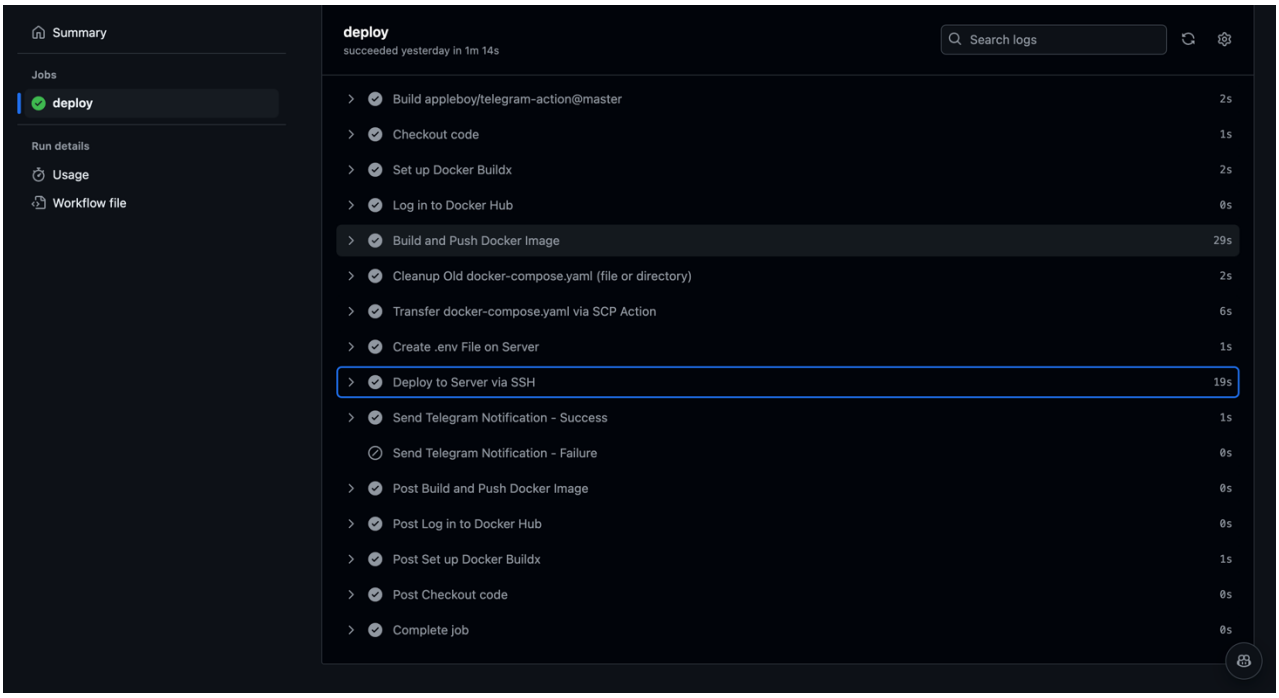

Figure 45. Github Actions Deploy logs.

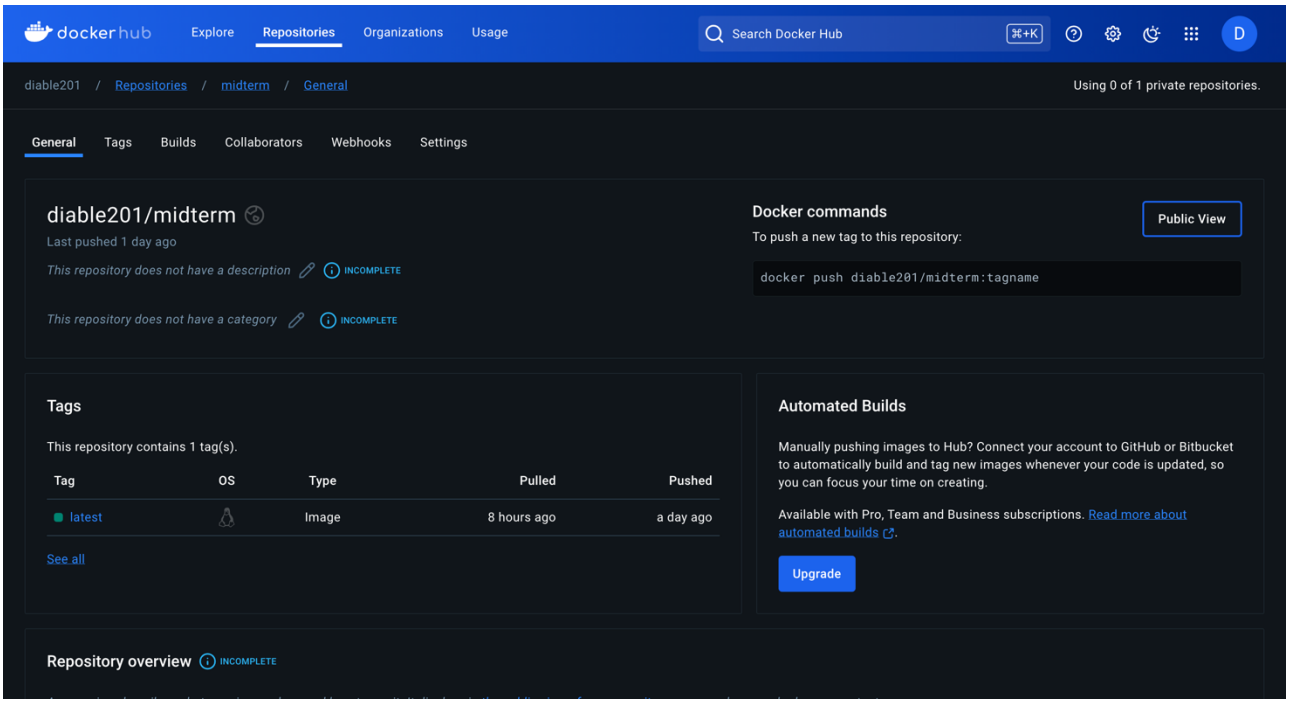

Figure 46. Dockerhub Midterm Image.

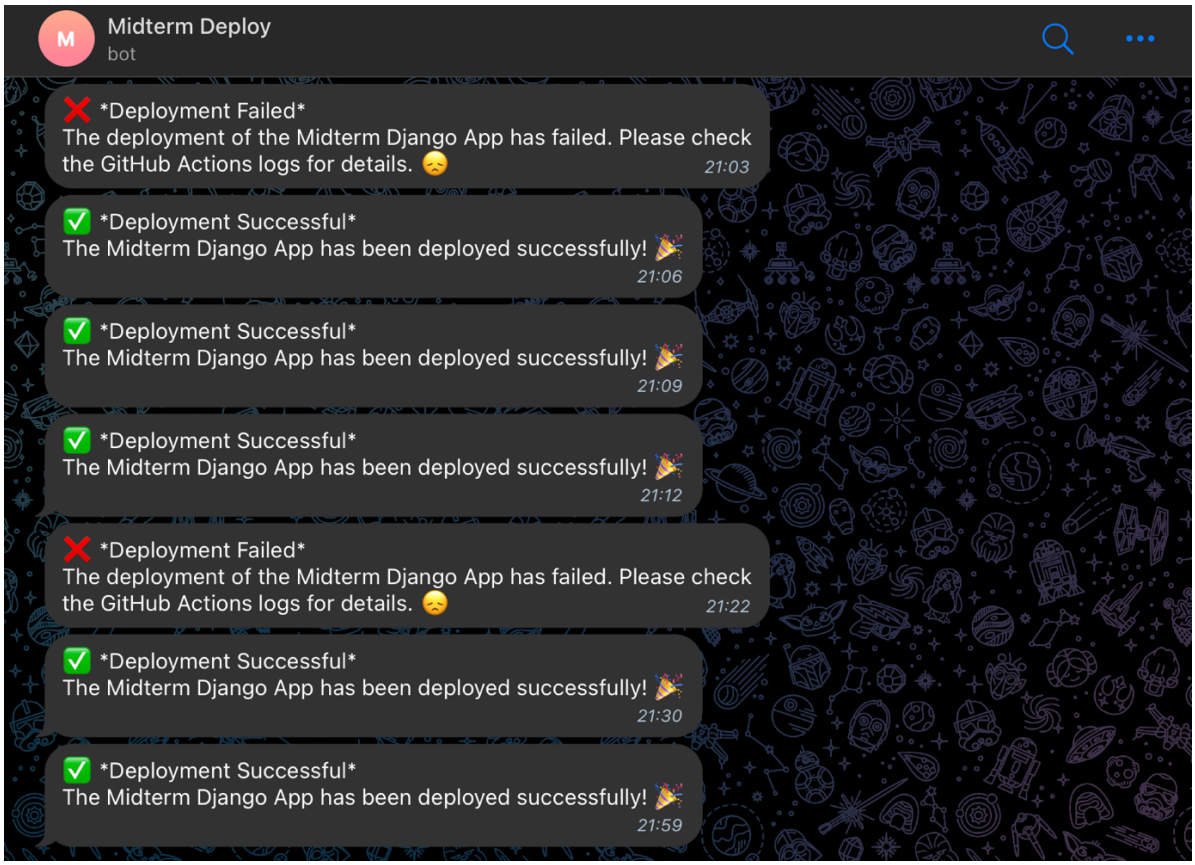

Figure 47. Telegram Notifications.

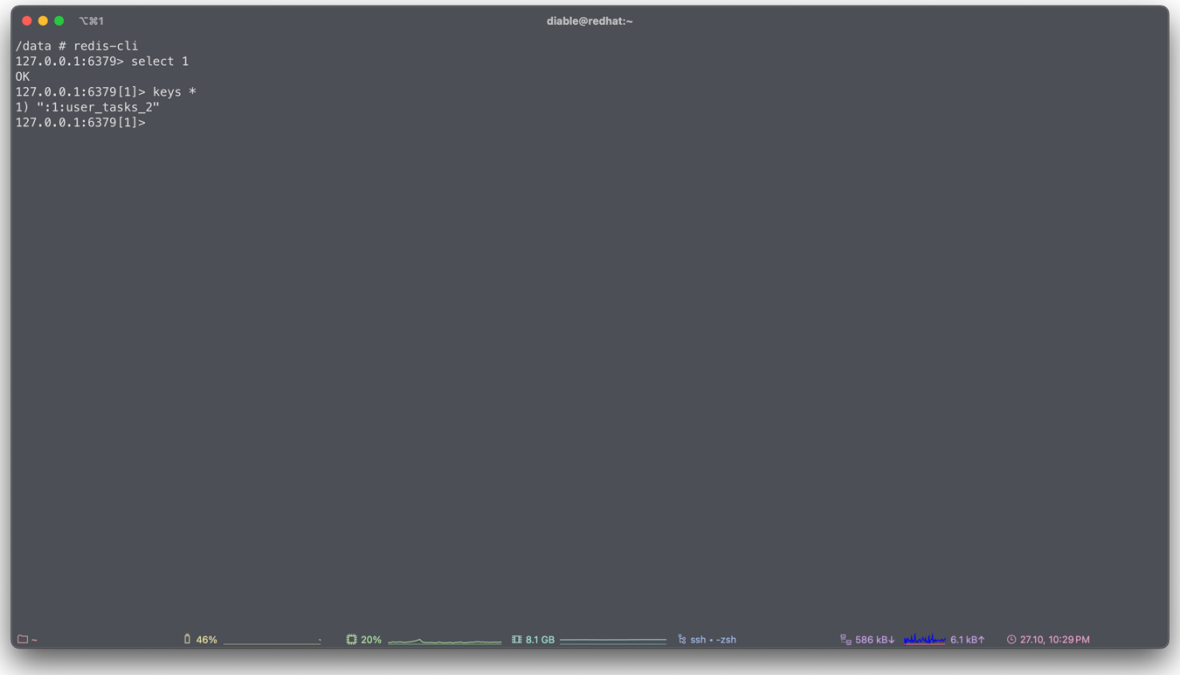

Figure 48. Redis Keys.

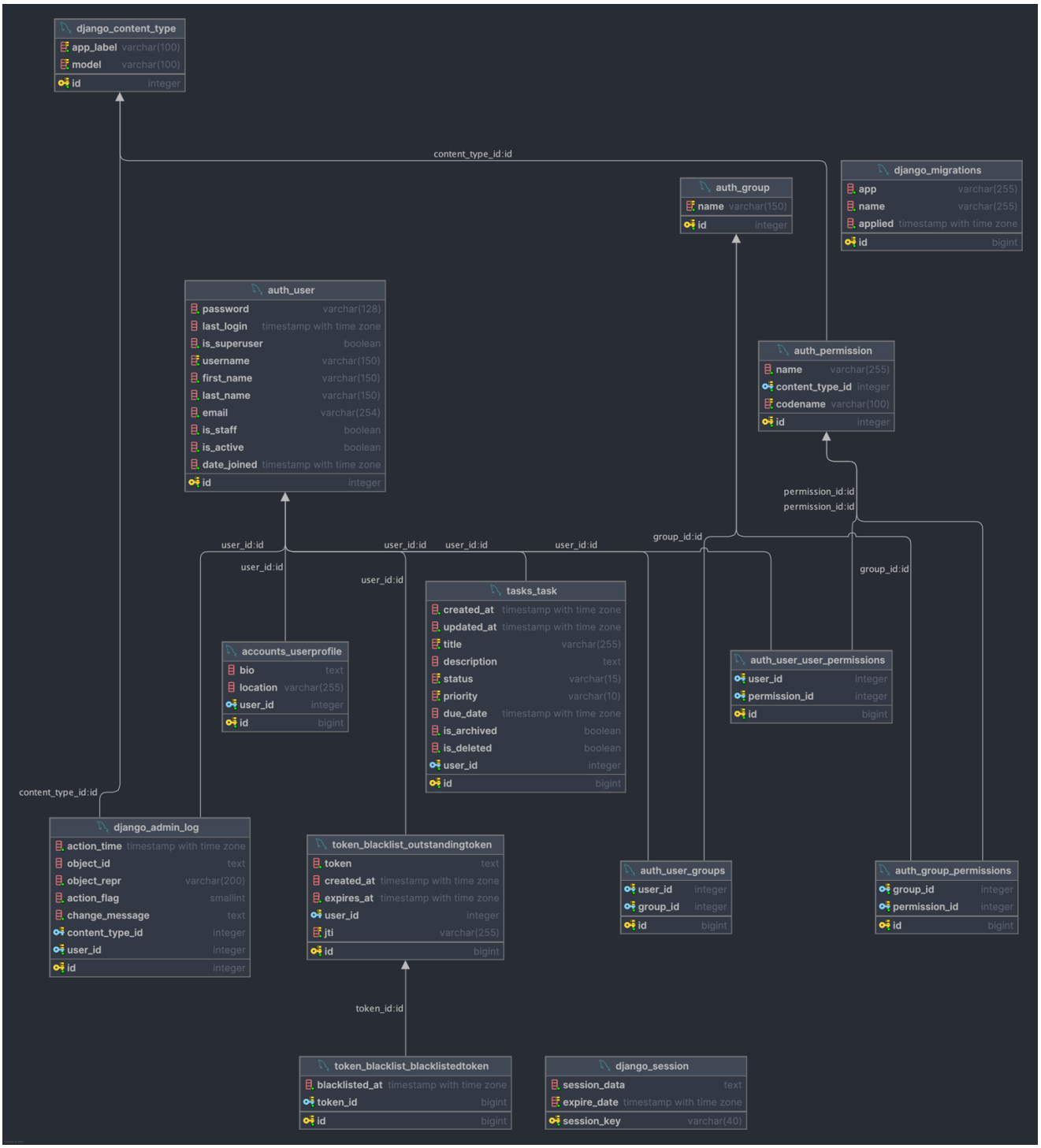

Figure 49. Database Diagram.