KBTU KAZAKH-BRITISH TECHNICAL UNIVERSITY

Assignment 4

Web Application Development
Building a RESTful API with
Django Rest Framework

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

Almaty, 2024

Table of Contents

Executive Summary

This project focuses on creating a RESTful API for a blogging platform utilizing Django Rest Framework (DRF). The implementation included essential API features such as CRUD operations, JWT-based authentication, and thorough testing methods. To improve usability, security, and scalability, advanced features like real-time notifications with Django Channels, rate limiting for API protection, and detailed documentation using drf-spectacular were also incorporated.

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. This project is application that is scalable, secure, ready for production with monitoring through Grafana combined with Docker.

Introduction

RESTful APIs are essential to contemporary software systems, offering a standardized way for applications to interact with one another. The Django Rest Framework (DRF) is a notable tool that streamlines API development by integrating Django's ORM features with robust functionalities like serializers, authentication, and testing tools. This project centers on developing an API for a blogging platform that includes advanced features such as nested serializers, API versioning, and real-time notifications.

The goal of this project was to utilize DRF's flexibility and strength to create a feature-rich, scalable API. The blogging platform was crafted to facilitate user interactions through WebSocket-based notifications and secure access using JWT authentication. Thorough testing and clear documentation were also key priorities to guarantee reliability and ease of use for both developers and clients.

Building a RESTful API with Django Rest Framework

Project Setup

I start our project by setting set a Django project integrated with REST Framework, along with the PostgreSQLand and Redis. I choose PostgreSQL as backend database for performance and scalability, and Redis as message broker for WebSocket notifications communication. Other libraries such as the djangoreatframework-simplejwt for authentication and drf-yasg for the documentation.

```
INSTALLED_APPS = [
    "daphne",
    "django.contrib.admin",
    "django.contrib.auth",
    "django.contrib.contenttypes",
    "django.contrib.sessions",
    "django.contrib.messages",
    "django.contrib.staticfiles",
    "rest_framework",
    "blog",
    "drf_spectacular",
    "channels",
]

```

Figure 1. Installed Apps.

Configure the application environment to differentiate between development and production phases. Sensitive data such as Database Credentials and JWT keys are stored as environment variables to remain secure. Docker was used to containerize the application, ensuring consistent deployments across environments (or your dev, stage or prod servers).

```
POSTGRES_DB=blog
POSTGRES_USER=user
POSTGRES_PASSWORD=password
POSTGRES_HOST=localhost
POSTGRES_PORT=5432
SECRET_KEY='django-insecure-8f8@d@^noc-&qq)@y3$@^z8fl%q+#%8oqa+-d*w&up9y5d+bar'
DEBUG=True
```

Figure 2. Environment Variables.

This containerization made it easier to have your application deployments on any infrastructure, be it on your local development machine or on cloud platforms. I use the environment variables to store securely secrets and offer flexibility in application.

Data Models

A Blogging Application would generally require two core models - Post and Comment. The Post model represents individual blog entries and supports fields for title, content, author, and timestamp. The Comment model is related to the Post model by a foreign key, so that each post can have multiple comments associate with it.

```python
class Post(models.Model):  15 usages  Sanzhar        Sanzhar, 28.11.2024, 22:48 • added: Assignment 4
    title = models.CharField(max_length=255, verbose_name="Заголовок")
    content = models.TextField(verbose_name="Контент")
    author = models.ForeignKey(User, on_delete=models.CASCADE, verbose_name="Автор")
    timestamp = models.DateTimeField(auto_now_add=True, verbose_name="Дата создания")

    def __str__(self) -> str:  Sanzhar
        return self.title

    def __repr__(self) -> str:  Sanzhar
        return self.title

    class Meta:  Sanzhar
        verbose_name = "Пост"
        verbose_name_plural = "Посты"
        ordering = ["-timestamp"]
```

Figure 3. Post Model.

These models were well-constructed to incorporate additional fields or features in the future without major changes to existing code. Both author fields in the models are tied to the User model enabling powerful user-specific filtering and permisions. An efficient system of different one-to-many relationship is now maintained for clear and useful queries onto the database.

```python
class Comment(models.Model):  9 usages  Sanzhar
    content = models.TextField(verbose_name="Контент")
    author = models.ForeignKey(User, on_delete=models.CASCADE, verbose_name="Автор")
    timestamp = models.DateTimeField(auto_now_add=True, verbose_name="Дата создания")
    post = models.ForeignKey(
        Post, related_name="comments", on_delete=models.CASCADE, verbose_name="Пост"
    )

    def __str__(self) -> str:  Sanzhar
        return f"Комментарий от {self.author} на пост {self.post}"

    def __repr__(self) -> str:  Sanzhar
        return f"Комментарий от {self.author} на пост {self.post}"

    class Meta:  Sanzhar
        verbose_name = "Комментарий"
        verbose_name_plural = "Комментарии"
        ordering = ["-timestamp"]
```

Figure 4. Comment Model.

This data arrangement leads to a healthy, extensive look at the database for a blogging implementation. Django's ORM also allows to interact with the database in a smooth way, pushing out new features rapidly as I wish. APIs of the platform are built on these models, making it easy for SDKs to connect models to serializers and views as well.

Serializers

Serializers in DRF are used to convert model instances into JSON, which is consumable by the clients. In the course of this project, an implementation of serializers was developed to handle validation and transformation of data for the Post and Comment models. The nested serializer was adopted to embed comments inside post responses to reduce the number of calls the client needed to make from the API.

Figure 5. V1 Serializers.

The CommentSerializer has been nested inside the PostSerializer such that one API call can return a whole representation of a post-including comments on it. This is a good practice in that it improves performance and reduces the number of round trips to the server requesting related data. In this example, read-only fields are also used for sensitive data such as usernames and timestamps.

This nesting structure makes the processing at the client side much easier because the client receives everything in one go. Secondly, DRF has some mechanisms for validation that make sure only valid data is processed and stored, hence improving the reliability of the API. These serializers actually come into play as an important bridge between models and views for efficient handling of data in an API.

Views and Enpoints

The API views were developed using the ModelViewSet brought about by DRF to handle, with much ease, CRUD. Viewsets created are PostViewSet and CommentViewSet for handling posts and comments, respectively. The viewsets make use of generic functionality provided by DRF to avoid boilerplate code while keeping flexibility for custom logic.

Figure 6. V1 Views.

Here in the PostViewSet, the overridden perform_create method associates the currently authenticated user with the new posts. Again, this is a good abstraction that keeps the complexity off the client side, as the author field is inferred from the authentication context. Similarly, CommentViewSet here associates' comments with their respective posts.

This design provides a clean, modular structure for managing API endpoints. ModelViewSet class will provide, out of the box, the methods for listing, retrieving, creating, updating, and deleting resources. Where needed, custom logic can be inserted to ensure the API is agile for changing requirements.

URL Routing

URL routing was performed to clearly and version-expose API endpoints. Using DefaultRouter from DRF to generate routes automatically for PostViewSet and CommentViewSet helped to simplify the configuration. Namespaces v1 and v2 were created for API versioning, thus allowing further work on API development without loss of backward compatibility.

Figure 7. V1&V2 URL Patterns.

Each version had a set of routes, serializers, and viewsets in which each could be independently developed and tested. This means that any future updates or breaking changes within newer versions wouldn't affect existing clients using older versions. This would include, for instance, enhancements in the v2 routes such as extra fields in serializers and support for new features.

Coupled with versioned routing, the API remains flexible and scalable. Explicit versioning on requests by clients ensures a smooth experience for users while versions evolve in the API. That, in turn, easily provides an ability to do incremental updates: changes can be tested and deployed within isolated versions.

Authentication and Permissions

djangorestframework-simplejwt was used to implement authentication, as it out-of-the-box provides robust and very secure JWT-based token authentication. This is stateless; every request must provide a token in the Authorization header. Tokens had short lifespans to enhance security, but refresh tokens allowed seamless re-authentication of the user.

```
163  class IsAuthenticatedOrReadOnly(BasePermission):
164      """
165      The request is authenticated as a user, or is a read-only request.
166      """
167
168      def has_permission(self, request, view):
169          return bool(
170              request.method in SAFE_METHODS or
171              request.user and
172              request.user.is_authenticated
173          )
```

Figure 8. IsAuthenticatedOrReadOnly Permission.

Custom permission classes helped in restricting access to a certain endpoint. For instance, through the permission class IsAuthorOrReadOnly, it ensured that others only had read access to such a post while the original author of the post had the modify or delete permission. Permissions would then be set at a viewset level to enforce consistent access control against an API.

```
4  class IsAuthorOrReadOnly(permissions.BasePermission):  6 usages  Sanzhar  Sanzhar, 28.11.2024,
5      def has_object_permission(self, request, view, obj):  Sanzhar
6          if request.method in permissions.SAFE_METHODS:
7              return True
8          return obj.author == request.user
9
```

Figure 9. Custim IsAuthorOrReadOnly Permission.

This will combine JWT authentication with granular permissions on a secure ground for APIs. Users could access only those resources for which they were allowed, while sensitive

endpoints would be protected from unauthorized user actions. It allows the integrity and confidentiality of users' data to be ensured within the system.

Advanced Features with Django Rest Framework

Nested Serializers

The implementation of nested serializers was to make APIs even more usable by embedding relevant objects inside their parent objects. In this project, there was a nested CommentSerializer inside the PostSerializer, which would help in fetching a post and all comments with it at once. This reduces the overall number of API requests that will have to be made from the client-side, which enhances performance and user experience overall.

```
18  class PostSerializerV2(serializers.ModelSerializer):  2 usages  Sanzhar
19      author_full_name = serializers.SerializerMethodField()
20      comments = CommentSerializerV2(many=True, read_only=True)   Sanzhar, 28.11.2024, 22:48
21
22      class Meta:  Sanzhar
23          model = Post
24          fields = ["id", "title", "content", "author_full_name", "timestamp", "comments"]
25
26      @staticmethod  Sanzhar
27      def get_author_full_name(obj: Post) -> str:
28          return f"{obj.author.first_name} {obj.author.last_name}"
29
```

Figure 10. Nested CommentSerializer for comments.

By nesting the comments inside posts, the API provides clients with a full insight into the structure of the data in one go. This greatly simplifies life for many frontend applications, as they do not need to make a few extra requests to fetch related data. DRF natively supports nested serializers in a way that is both easy to implement and robust for validation.

This feature enhances data organization and boosts the productivity of developers. This helps in ensuring related objects are consistently represented each time, thus reducing inconsistencies in data. Moreover, it will enhance nested serializers' efficiency in handling data, particularly for APIs that have very complex relationships.

Versioning

API versioning is one important feature that allows developers to support backward compatibility while adding new features. In this project, the versioning has been implemented at the URL level. That is, different namespaces were defined for each version of the API, along with their serializers and viewsets independent of each other for development and updates.

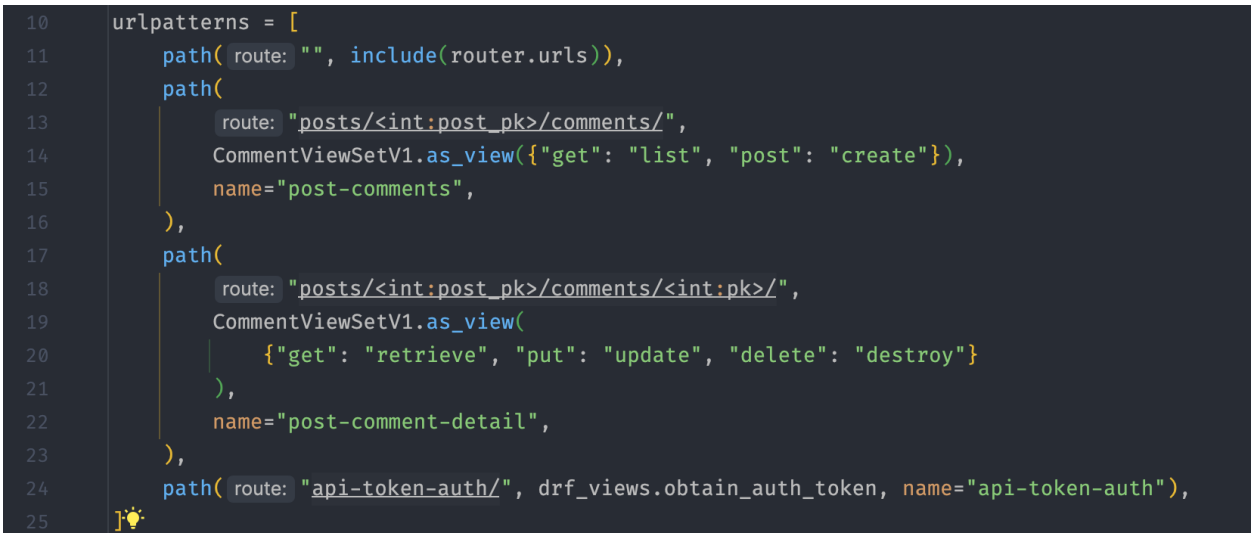

Figure 11. V1 URL Patterns.

v1 was the very first implementation; v2 introduced some additional fields in the PostSerializer and so on. A client could easily access any version by using the prefix of the version in the API URL. This design prevents existing clients from being disrupted due to updates in higher versions.

This will give room for API evolution: a developer can change anything and improve it without breaking anything. This will further ease the testing and debugging processes since each version can be maintained and validated independently. Versioning is about keeping an API both scalable and user-friendly.

Rate Limiting

Rate limiting was implemented to manage and control API traffic to prevent abuse. Here, the built-in throttle classes in DRF - AnonRateThrottle and UserRateThrottle - have been configured to limit both how many requests an anonymous or authenticated user may make within a certain timeframe. This ensures fair usage of your system and prevents it from becoming overwhelmed by excessive traffic.

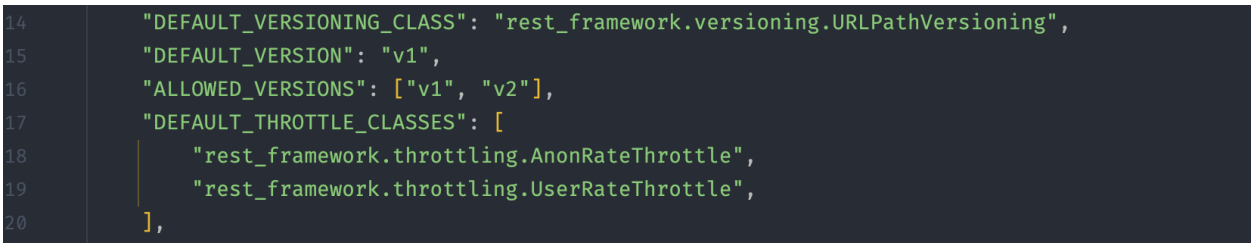

Figure 12. Rate Limiter Settings.

The default throttle rates had been defined within the settings file, using more stringent limits for the anonymous users to prevent abuses. In the view level, this received more granular limits using custom throttle classes: That means, for instance, the endpoint to create a post had

tougher limits to prevent spam.

```
    "DEFAULT_THROTTLE_RATES": {
        "anon": "10/min",
        "user": "100/min",
    },
}
```

Figure 13. Throttle Rate for Anonymous & Registered Users.

This implementation ensures a responsible usage of API resources, which helps in improving system performance and reliability. With global throttling in place, combined with customized per-endpoint limits, the API will be protected against abuse while sustaining a good user experience. Rate limiting assists in the determination and prevention of potential misuse patterns.

WebSocket Integration

Real-time notification is one of the important features in recent applications since it gives fast feedback to users, as well as increasing user interactivity. In this application, real-time notifications have been implemented based on WebSockets by using Django Channels and Redis. Users instantly see that new comments were added to their posts. WebSockets keep the server and client connected, creating the ideal solution for delivering updates in real time with no need for continuous polling over HTTP.

```
class NotificationConsumer(AsyncWebsocketConsumer):  1 usage  Sanzhar  Sanzhar, 28.11
    def __init__(self, *args, **kwargs):  Sanzhar
        super().__init__( *args: *args, **kwargs)
        self.group_name = None

    async def connect(self):  Sanzhar
        if self.scope["user"].is_anonymous:
            await self.close()
        else:
            self.group_name = f'user_{self.scope["user"].id}'
            await self.channel_layer.group_add(self.group_name, self.channel_name)
            await self.accept()

    async def disconnect(self, close_code):  Sanzhar
        if not self.scope["user"].is_anonymous and self.group_name:
            await self.channel_layer.group_discard(self.group_name, self.channel_name)

    async def send_notification(self, event):  Sanzhar
        await self.send(text_data=json.dumps(event["message"]))
```

Figure 14. Notification Consumer.

An in-house WebSocket consumer was implemented; it handled the connections, authenticated users, and issued notifications to the relevant recipient. It would validate the user's JWT token at the time of connection and subscribe him to a channel group unique to his user ID. Ensuring that the notifications are delivered securely and exclusively to their intended recipients.

To handle efficiency in channel communications and scaling of the system for concurrent users, Redis was used as the message broker.

```
const wsProtocol = window.location.protocol === 'https:' ? 'wss://' : 'ws://';
const wsHost = window.location.host;
// Establish WebSocket connection with the token in query parameters
const socket = new WebSocket(
    `ws://127.0.0.1:8000/ws/notifications/?token=${encodeURIComponent(token)}`
);

socket.onmessage = function(e) {
    const data = JSON.parse(e.data);
    const notificationsList = document.getElementById('notifications');
    const newNotification = document.createElement('li');
    newNotification.textContent = data.message;
    notificationsList.appendChild(newNotification);
};

socket.onopen = function() {
    console.log('WebSocket connection opened.');
};

socket.onclose = function(event) {
    console.log('WebSocket connection closed:', event.code, event.reason);
    if (event.code === 4401) {
        alert('Unauthorized: Please log in.');
        window.location.href = '/login/';
    }
};
```

Figure 15. Notifications HTML Page.

I established a WebSocket connection using JavaScript on the client side; it will listen for notifications coming back. The connection string contained as a query parameter the user JWT token to ensure only authenticated users could connect to the websocket, and notifications would show in an appending-in-list dynamic way.

```
CHANNEL_LAYERS = {
    "default": {
        "BACKEND": "channels_redis.core.RedisChannelLayer",
        "CONFIG": {
            "hosts": [
                (os.getenv("REDIS", "localhost"), 6379)
            ],
        },
    },
}
```

Figure 16. Redis Channel Layer.

The immediate advantages of using WebSockets over traditional HTTP polling include but are not limited to lower server load and latency due to the use of one connection, hence avoiding

the creation of requests constantly via HTTP. Having Redis as the message broker, the system will scale well for high traffic and multiple concurrent connections.

This feature makes the blogging platform a more interactive application. Users get immediate notifications about new comments; thus, the level of involvement is improved, and people can hold discussions in real-time. Because of Django Channels and Redis, the system will be scalable and resilient and may easily be expanded with functionality like notifications for likes, mentions, or replies.

Deployment

Deployment is a very important stage in the software development lifecycle, which ensures that the application is available to users in a stable and secure environment. In this project, the application was containerized using Docker, and GitHub Actions were utilized for a smooth CI/CD pipeline. The application was deployed on a DigitalOcean server, which provided robust infrastructure for production deployment.

Docker Configuration

This section initiates the beginning of the deployment process, including creating a Dockerfile, which encapsulates the environment that this application is working with. Further optimizations included multistage builds into one Docker image by separating build and runtime. Finally, Docker-Compose is used for orchestrating services, namely the Django application, PostgreSQL database, and Redis for WebSocket messaging.

```
FROM python:3.12-slim AS builder   Sanzhar, 28.11.2024, 22:48 • added: Assignment 4

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

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
```

Figure 17. Dockerfile Configuration.

Docker Compose was to be configured with service definitions for the web application, database, and Redis, along with volume mappings for static and media files. Handle environment variables securely with the use of a .env file so no credentials would need to be hardcoded into an application.

CI/CD Pipeline

Continuous Integration/Deployment Pipeline GitHub Actions was used in setting up the pipeline which would handle automated tests, building, and deployment. Set it to be triggered every push to the master branch. This way, changes would always get deployed automatically. Using docker/build-push-action, build Docker images and then push to Docker Hub for easy pulling by the Production server.

The deployment on the DigitalOcean server was accomplished by cleaning up old configurations through SSH, transferring updated files, and restarting the Docker Compose stack. The result is this pipeline, which will minimize downtime and provide predictable deployment processes that reduce human errors.

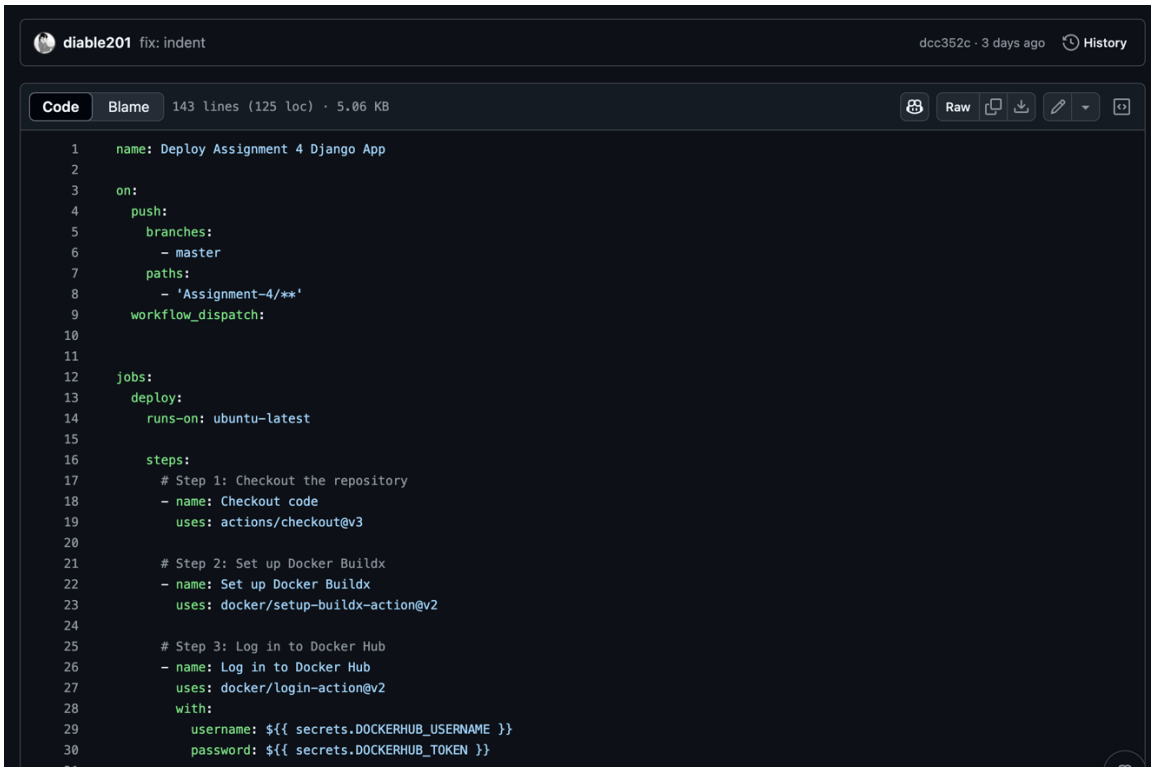

Figure 18. CI/CD Pipeline.

This integration of testing into the CI/CD pipeline made sure that if something went wrong in the codebase, it would be identified as early as possible and resolved before it was shipped. The pipeline also sends Telegram notifications on successful or failed deployments, making the development team more transparent and responsible.

Now, automated deployments made deployments faster and reliable for feature updates to be rolled out quickly and in rapid iteration. Such an approach does sound state-of-the-art in today's DevOps practices: providing a modern approach to handling a scalable application maintainable.

Testing and Documentation

API testing and documentation shall be of a nature such that it should work predictably, with developers finding it very easy to use. Testing: Automated testing of API endpoints for proper responses, rate limiting, authentication, and permissions. Interactive: The API uses drf-spectacular to generate the documentation-an interface where one can view and test the API.

API Testing

The test api by Django's TestCase, DRF's APIClient were used to run tests covering views for creating, retrieving, editing and deleting posts and comments. Much attention was given to corner cases: unauthorized access, rate limit violation and submitting of invalid data.

```
32
33 ▶    def test_create_post(self):  Sanzhar
34          self.authenticate()
35          data = {"title": "New Post", "content": "New Content"}
36          url = reverse(f"{self.api_namespace}:post-list")
37          response = self.client.post(url, data, format="json")
38          self.assertEqual(response.status_code, status.HTTP_201_CREATED)
39          if self.api_namespace == "api_v1":
40              self.assertEqual(response.data["author_username"], self.user.username)
41          else:
42              self.assertEqual(
43                  str(response.data["author_full_name"]).strip(),
44                  self.user.get_full_name(),
45              )
46          post_id = response.data["id"]
47          post = Post.objects.get(id=post_id)
48          self.assertEqual(post.author, self.user)
49
```

Figure 19. Create Post Testing.

For example, the test_create_post case has to do with authenticated users being able to create new posts. This test ensures that only valid JWT could access the endpoint, and the right response is returned on creation.

The developed features included the rate limit for which more and more tests had to be written to test things like "requests over the limit should return 429 Too Many Requests" status. The purpose of such a test is, it replicates real use; thus, the system is strong and reliable.

Documentation Using DRF-Spectacular

drf-spectacular was used to generate complete API documentation. It provides an interface to interact with the endpoints using Swagger UI and ReDoc. All the endpoints were listed, the request methods, parameters required for each request, and the response formats. Also, it states which of the endpoints are protected by authentication.

Figure 20. Swagger URL's for V1 and V2 API.

Both v1 and v2 maintained their schema so that a client would see the proper documentation, based on which version they chose. Also included were examples of valid request payloads, with expected responses to help developers test and debug integrations.

It acted like a single source of truth in understanding the API and hence reduced the learning curve for new developers. This was generated automatically using drf-spectacular so that the documentation remained in cadence with the codebase on each new feature addition or feature update.

Challenges and Solutions

The development of this blogging platform was therefore not without its challenges: from real-time features down to deployment automation, the project had its challenges that needed innovative and practical solutions.

1. Real-Time Notifications: This turned out to be quite a big deal to build-a real-time notification system for user interactions. The greatest challenge here was how to securely deliver the notification to the user. With Django Channels and Redis integration, this challenge was surmounted. In this respect, Redis is a very reliable message broker while at the same time doing custom middleware authenticating WebSocket connections by using JWT tokens. It therefore ensured efficiency and security in the delivery of notifications.

2. Rate Limiting: The API had to be protected from abuse, from excessive traffic originating from unauthenticated users. In principle, rate limiting was easy to implement using built-in throttle classes provided by DRF, but in practice it took some careful tuning to reach a balance between usability and security. Setting up limits for anonymous and authenticated users separately allowed the system to remain accessible yet secure.

3. Automating Deployment: This tends to be a tedious process, full of mistakes when manually done. Automation through GitHub Actions' CI/CD pipeline was one steep learning curve that paid off. Such a pipeline ensured that for every push to the master branch, a chain of automated actions was fired: testing code, building Docker images, and deploying the application onto the production server. The outcomes were both timesaving and reduced the risk of errors from deployment.

Each of these challenges had something over it that was learned in the process of building systems that are scalable and secure. The solutions adopted during this project will be used to solve such problems in the future.

Conclusion

This project showed, pretty effectively, the strength of Django Rest Framework when developing a feature-rich API-from CRUD operations to advanced features involving real-time notification and API versioning, the blogging platform achieved just that for which it was set up: a robust backend at which users' every interaction is taken care of. Adding Django Channels along with Redis integrated into this gave the application a modern feel, with a real-time layer quite engaging and responsive.

Docker and GitHub Actions ensured that deployment was easy and repeatable; hence the reason for continuous integration and continuous deployment within modern software development. The project followed the best practices in security, performance, and documentation, and created a solid base for future growth.

This project was ultimately not about delivering a functional application but one of learning, innovating, and solving real-world problems. The experience that will be gained here will therefore contribute a great deal to future projects, especially in scalable and maintainable architectures.

Recommendations

Although the project realized its major objectives, much more can still be added to improve it. Following are some suggestions to further improve the system:

1. Increase Real-Time Features: The current WebSocket integration only supports comment notifications. If this feature could be expanded to include likes, shares, and mentions, this would make the platform even more interactive and engaging.
2. Improve on Security: Token blacklisting helps to add an additional layer of security to invalidate the compromised JWT tokens. It is very useful in such scenarios where a leak of user credentials has happened or a token needs to be revoked.

3. Integrate Third-party APIs: Adding third-party integrations into the platform will let it send emails or push notifications. This would result in greater user engagement and provide further avenues for information delivery.

These recommendations, when implemented, will continue to make the platform grow in functionality and scalability, relevant and useful for its users and developers.

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

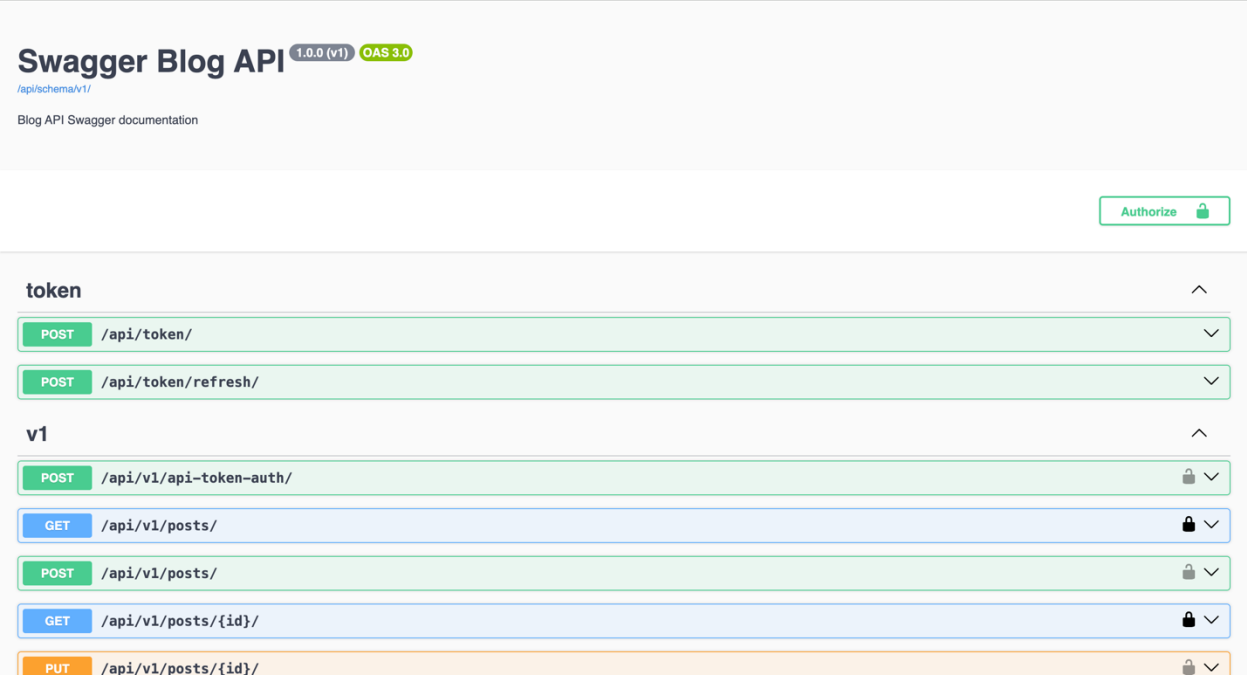

Figure 21. Swagger V1.

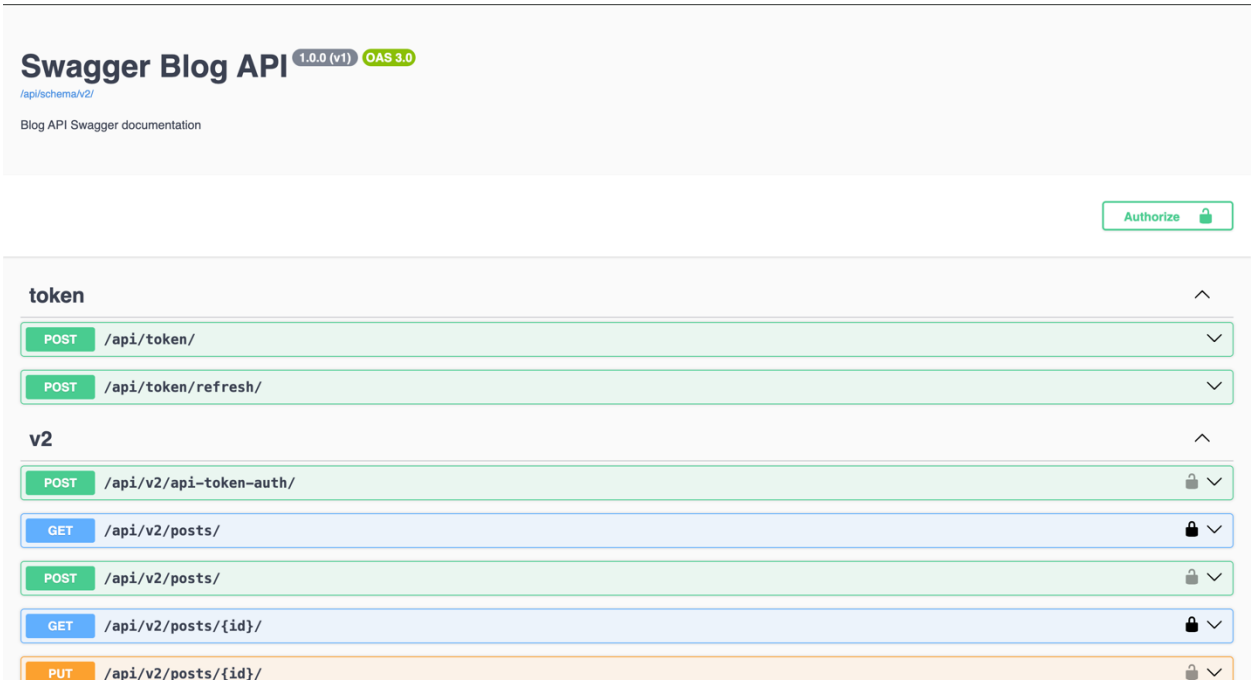

Figure 22. Swagger V2.

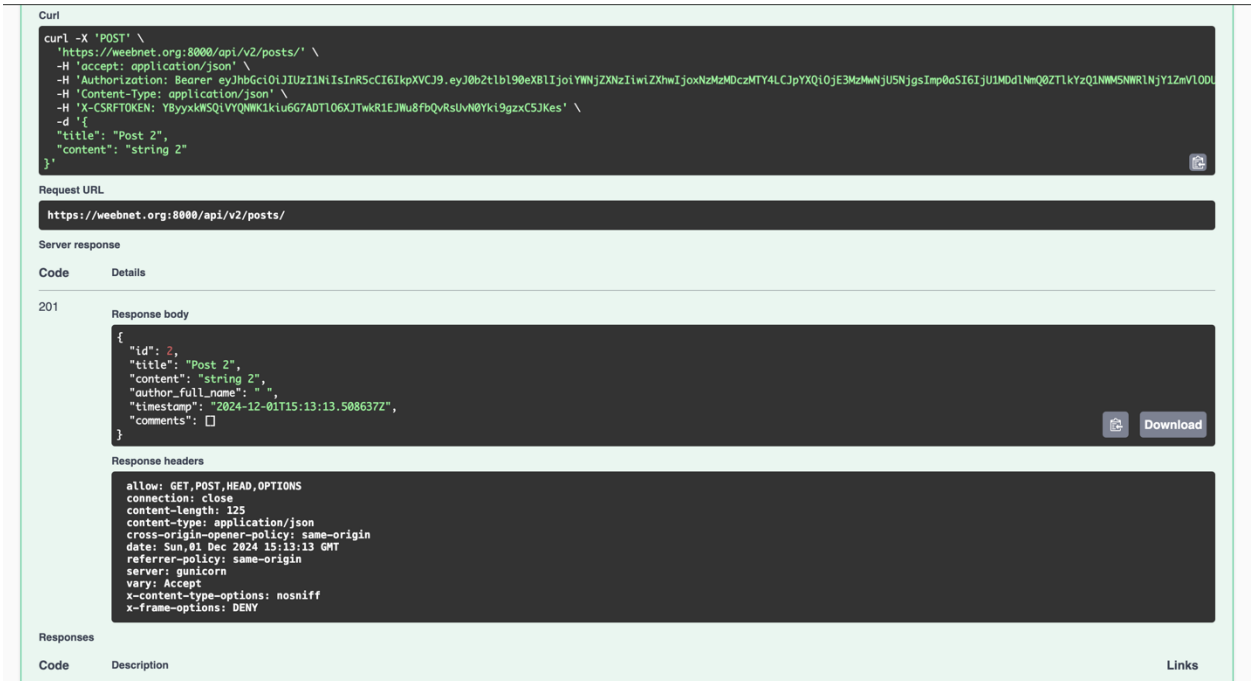

Figure 23. Create Post.

Real-Time Notifications

- New comment on your post "string": string
- New comment on your post "string": string
- New comment on your post "string": Comment

Figure 24. Real Time Notifications Page.

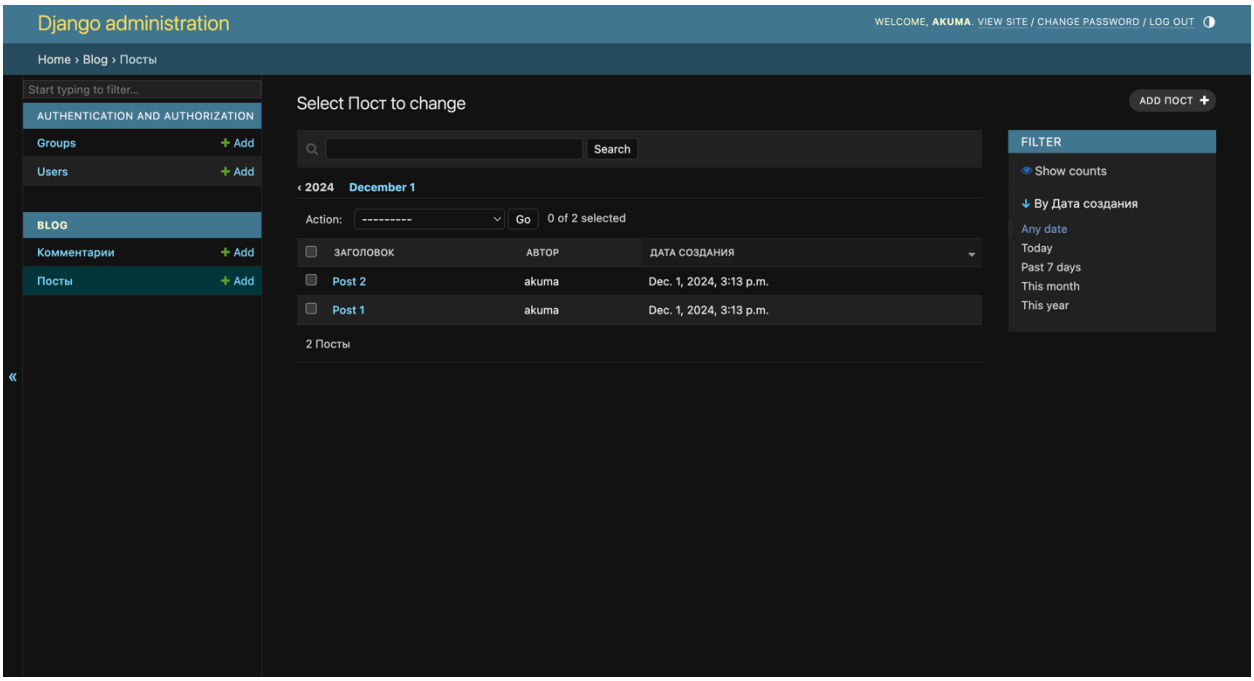

Figure 25. Django Admin Page.

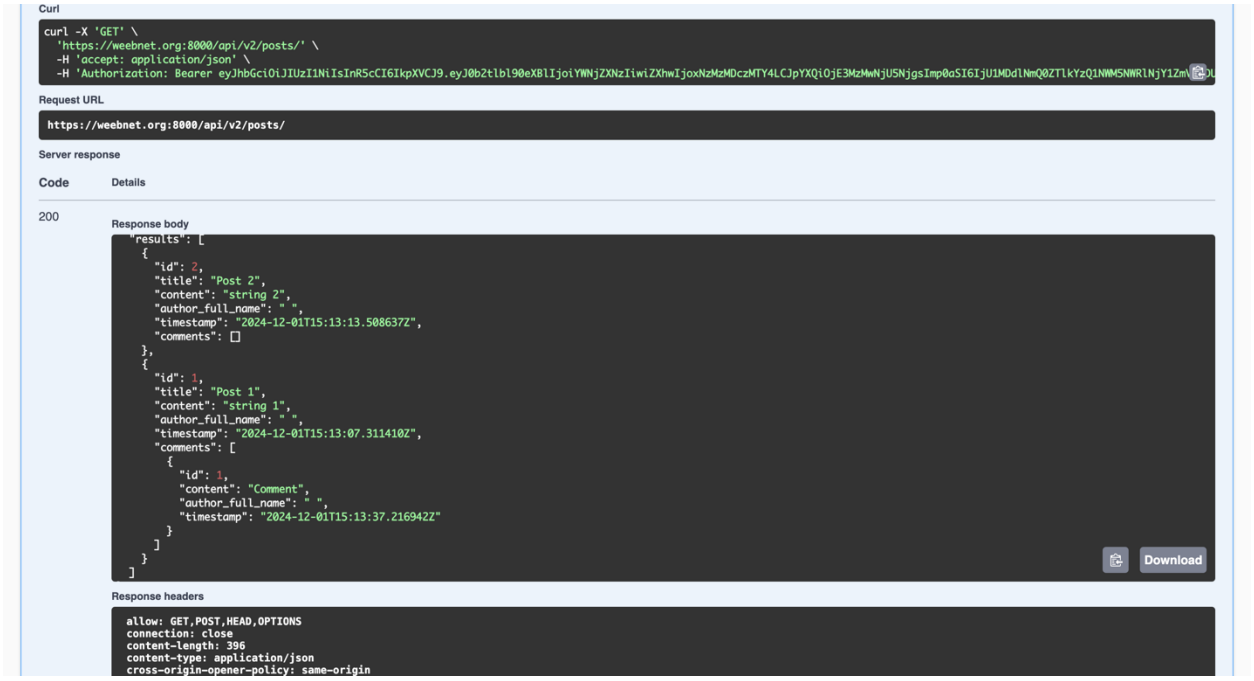

Figure 26. Get All Posts.

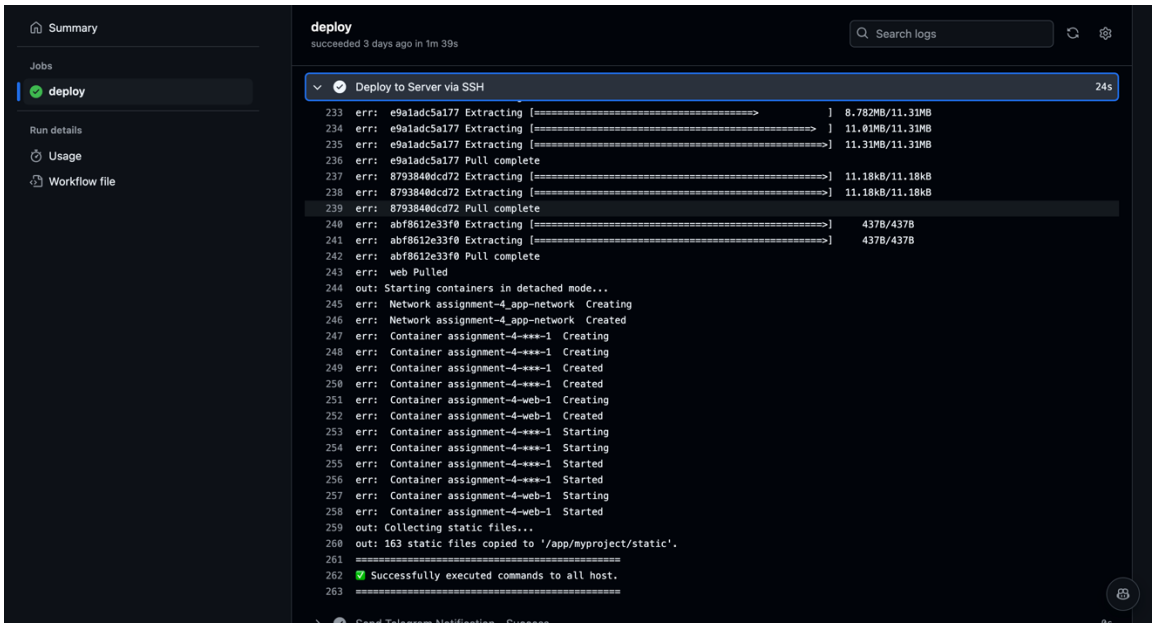

Figure 27. CI/Cd Pipeline logs.

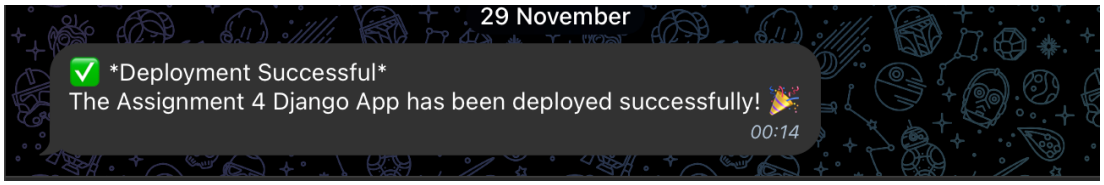

Figure 28. Telegram Notification.

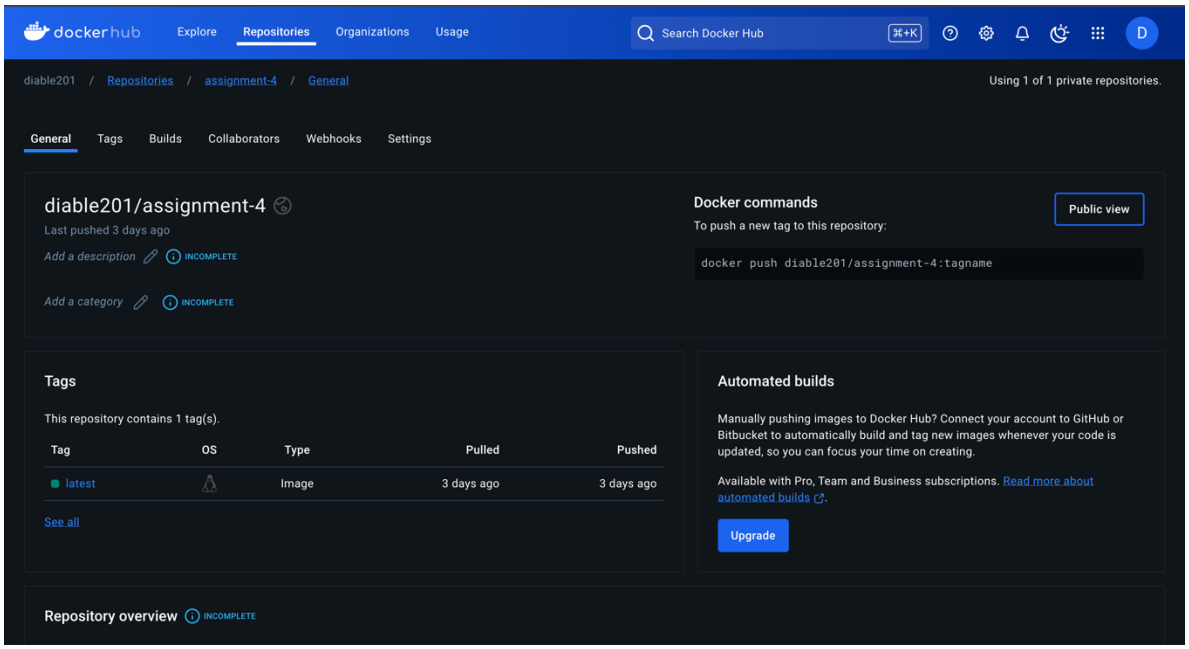

Figure 29. Docker Hub.

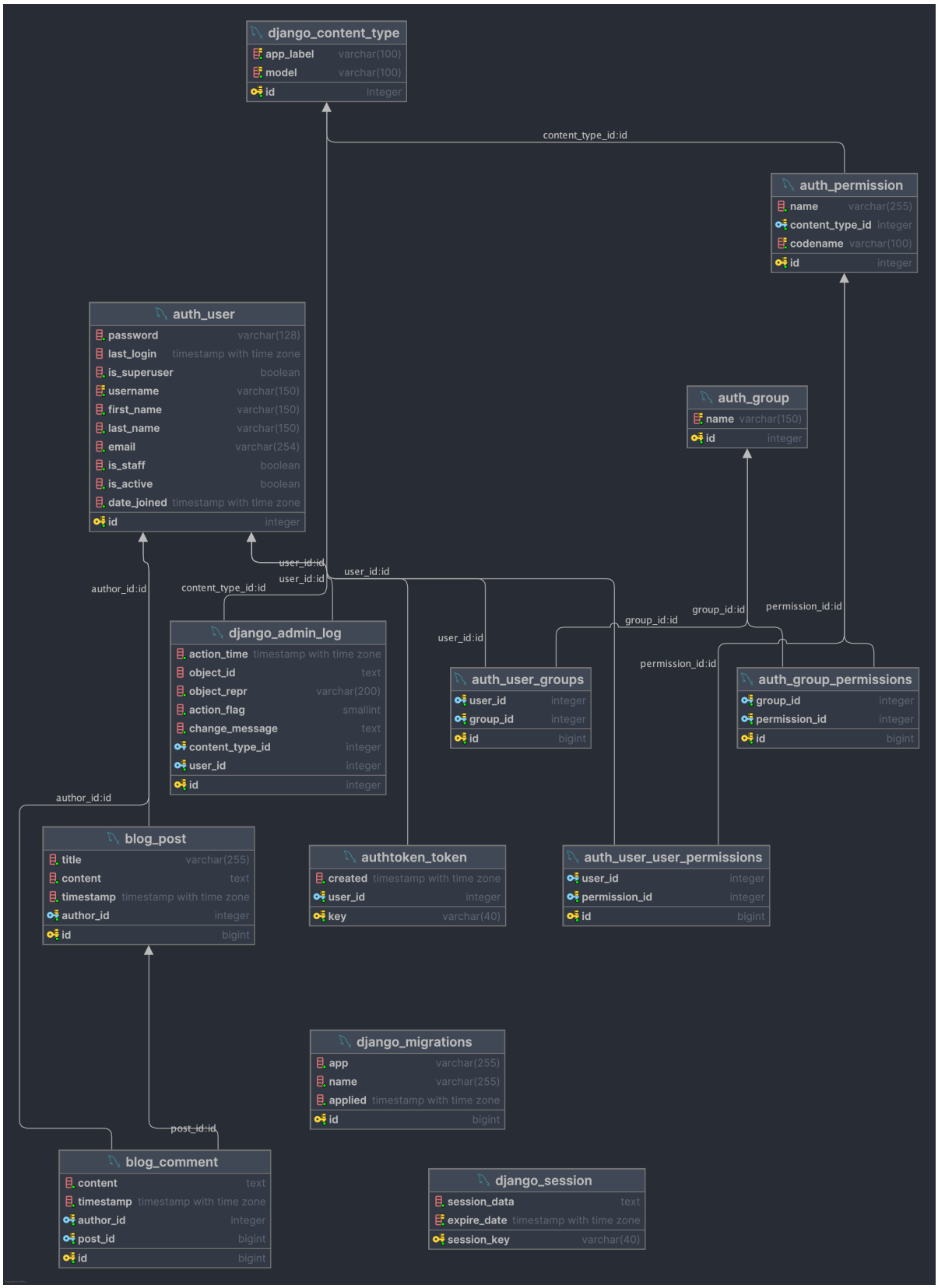

Figure 30. Database Diagram.

Figure 33. Running Tests.

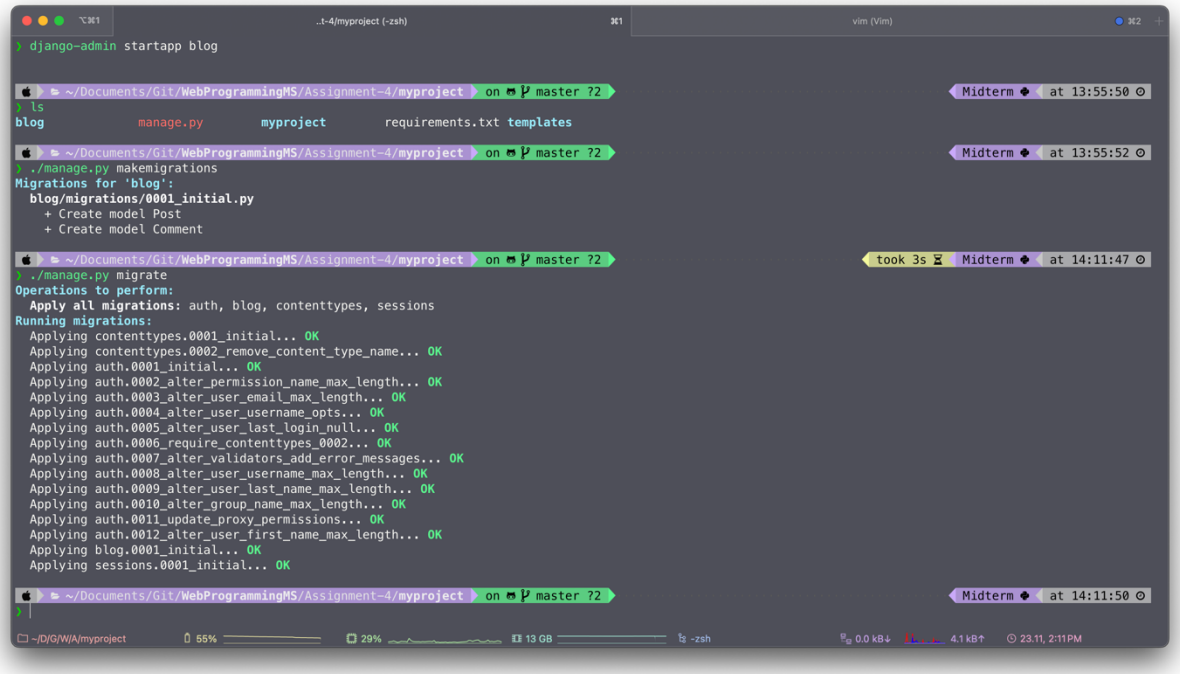

Figure 34. Running Migrations.