Web development

Assignment 4: Building a RESTful API with Django Rest Framework (DRF) and Advanced Features with Django Rest Framework

Bitanov Assanali (23MD0392)

01.12.2024

Table of Contents

[Introduction 3](#_Toc183433946)

[Building a RESTful API with Django Rest Framework 4](#_Toc183433947)

[Advanced Features with Django Rest Framework 10](#_Toc183433948)

[Conclusion 13](#_Toc183433949)

[References 14](#_Toc183433950)

# Introduction

In assignment 4, I built a blog API using Django Rest Framework (DRF) and deployed it with Docker. The goal was to create a backend system where users can manage posts and comments through API endpoints. I also focused on adding security, scalability, and documentation to the project.

First, I created models in Django to organize the data for posts and comments. Models define the structure of the data, like the title, content, author, and timestamp, making it easier to manage and retrieve information in the app.

Then, I built serializers using DRF to handle the conversion between Python objects and JSON format. Serializers allowed the app to send and receive data in a format that external clients could understand, making the API easy to use.

After that, I wrote views using DRF’s ModelViewSet to process requests like creating, reading, updating, and deleting posts and comments. I also added permissions to control who can perform certain actions. For example, anonymous users could only view posts, but only authenticated users could create or edit them.

I set up URL routing to make sure all API endpoints were reachable and well-organized. I also added versioning to the API so that I could support future changes without affecting older versions.

Testing was a key part of the project. I wrote unit tests to check if the API endpoints worked as expected. These tests helped me find and fix issues and made sure the app behaved correctly in different situations.

Finally, I used Docker to package the application so it could run consistently in any environment. I created a Dockerfile to define the app’s setup and a docker-compose.yml file to manage the web app and database services. This made deployment simple and efficient.

This project helped me improve my skills in creating RESTful APIs, securing them with permissions, and deploying them using Docker. It also gave me hands-on experience with key tools for building and managing scalable backend applications.

# Building a RESTful API with Django Rest Framework

First, I created a new directory for the project. Assignment4 is the name of the main directory. In this folder I created the Django project using <django-admin startproject blog\_project\_v2>. In the Assignment4 folder I created blog folder. It’s an application inside the project. There I created the models.py file and defined the Post and Comment models. The models are shown in Figure 1.



Figure 1. The Post and Comment models in the model.py file

The Post class has title, content, author and timestamp. The comment class has post (as Foreign Key), author, content and timestamp also. We need these models to save data in the database and work with it, like creating, reading, updating, or deleting posts and comments.

Then I created the serializers.py file to define the CommentSerializer and PostSerializer. Serializers are used to convert complex data types, such as Django model instances, into Python data types that can be easily rendered into JSON, XML, or other content types[1]. The file is shown in Figure 2.



Figure 2. The CommentSerializer and PostSerializer in the serializers.py file

Both classes inherit the serializers.ModelSerializer. It automatically generates fields and validation rules based on a model. Both classes handle how to post data like title, content, and author sent to and from the API.

Next, I created the views.py file where I defined the PostViewSet and CommentViewSet classes. We create views to handle how our API interacts with the data in the models. The PostViewSet and CommentViewSet classes are shown in Figure 3.



Figure 3. The PostViewSet and CommentViewSet classes

These classes inherit ModelViewSet. It combines the functionality of multiple views like listing, creating, updating, retrieving, and deleting objects into one class [2].

In the urls.py file of the blog app, I set up the routing for the API endpoints using Django REST Framework's DefaultRouter. The router automatically generates the necessary URL patterns for the PostViewSet and CommentViewSet views. The urls.py file of the blog app is shown in Figure 4.

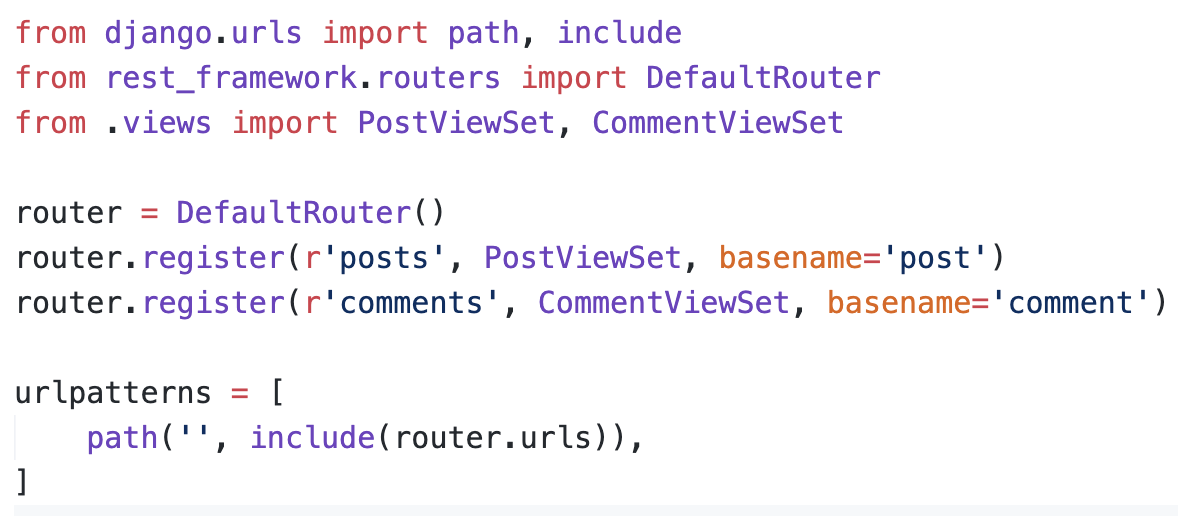


Figure 4. The urls.py file of the blog app

In the urls.py file of the project, I included the routes defined in the app's urls.py file under the api/ prefix. This organizes the API under a single base path. The urls.py file of the project is shown in Figure 5.



Figure 5. The urls.py file of the project

The api/token routes handle JWT-based authentication. The first endpoint allows users to obtain a token, while the second refreshes an expired token. I added authentication to make the API more secure. Only logged-in users can create, update, or delete posts and comments. To implement this I created the permissions.py file where I defined the IsAuthorOrReadOnly class. The class is shown in Figure 6.

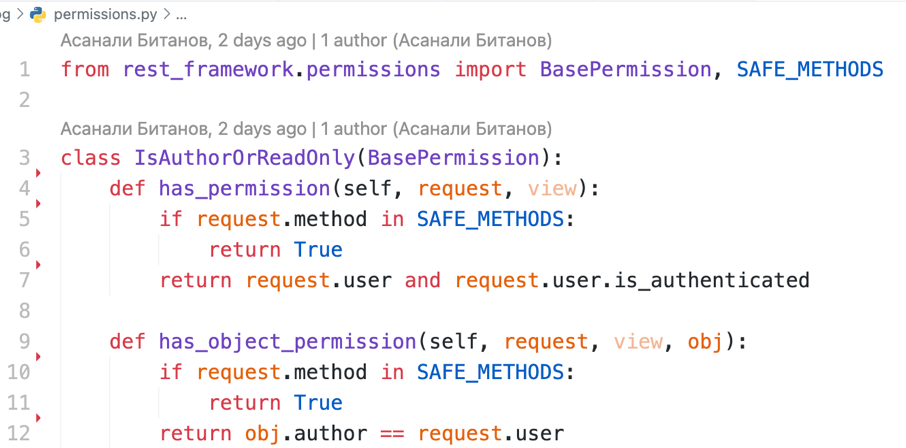


Figure 6. The IsAuthorOrReadOnly class in the permissions.py file

This checks if the user has permission to perform actions on a specific object. For "safe" methods, everyone is allowed. For actions like updating or deleting, it checks if the user is the author of the post or comment.

To make sure the API works correctly, I wrote unit tests for the main API endpoints using Django's TestCase. For this I created the test.py file where I defined the APITestCaseWithAuth class. The file is shown in Figure 7.



Figure 7. The APITestCaseWithAuth class in the test.py file

The class checks the API’s functionality, such as creating a post, retrieving all posts, getting a single post, updating a post and deleting a post. To test a post creation I send a POST request with data to create a new post and check if the response returns a status code of 201. To test retrieving all posts, I send a GET request to fetch all posts and check if the response returns a status code of 200. To test retrieving a single post, I send a GET request with the post ID and check if the response returns a status code of 200. To test updating a post, I send a PUT request with updated data for a post and check if the response returns a status code of 200. To test deleting a post, I send a DELETE request for a specific post and check if the response returns a status code of 204. To launch tests I used python manage.py test command. The result of the tests is shown in Figure 8.

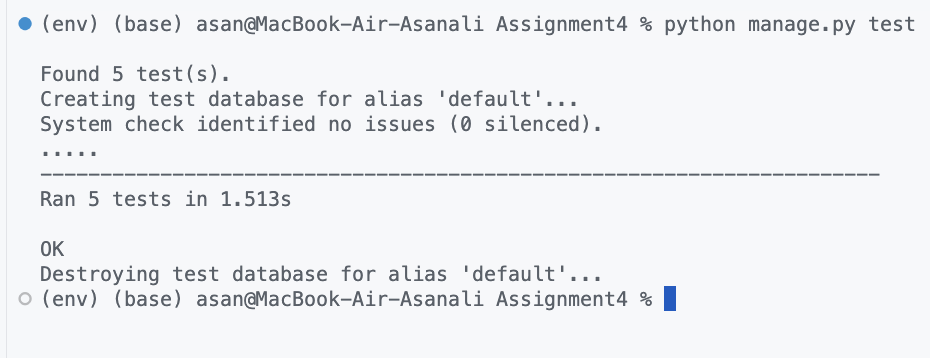


Figure 8. The result of the test. All the five tests are done successfully

In order to create documentation I used the drf-yasg library, which generates Swagger and ReDoc documentation automatically. For that I added the schema\_view in the urls.py (project) file. The file is shown in Figure 9.



Figure 9. The schema\_view in the urls.py file (project)

The openapi.Info section sets up the details of the API, such as the title ("Blog API by W1lden"), version ("v1"), description, and contact information. The swagger page is shown in Figure 10.

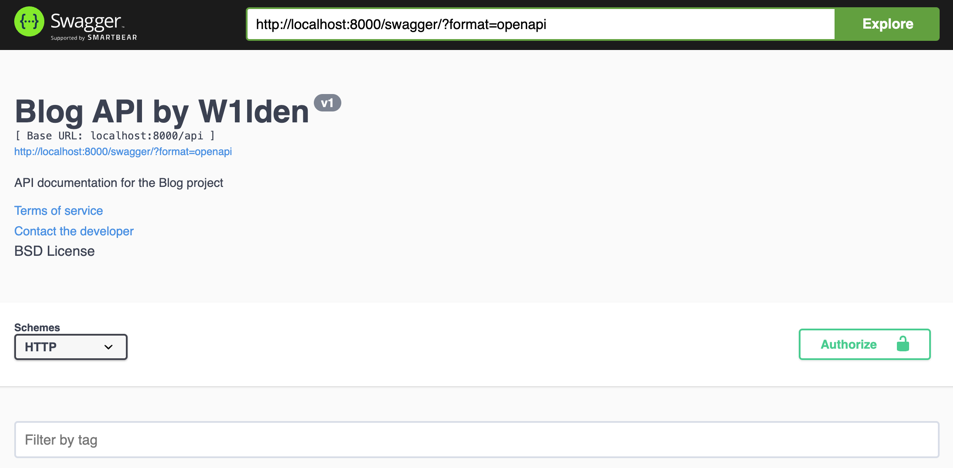


Figure 10. The Swagger page of the project

The Swagger page is an interactive API documentation tool that displays all the available endpoints in the project. Swagger also allows me to test endpoints directly from the browser by entering data and sending requests. There is also the ReDoc page. The page is shown in Figure 11.

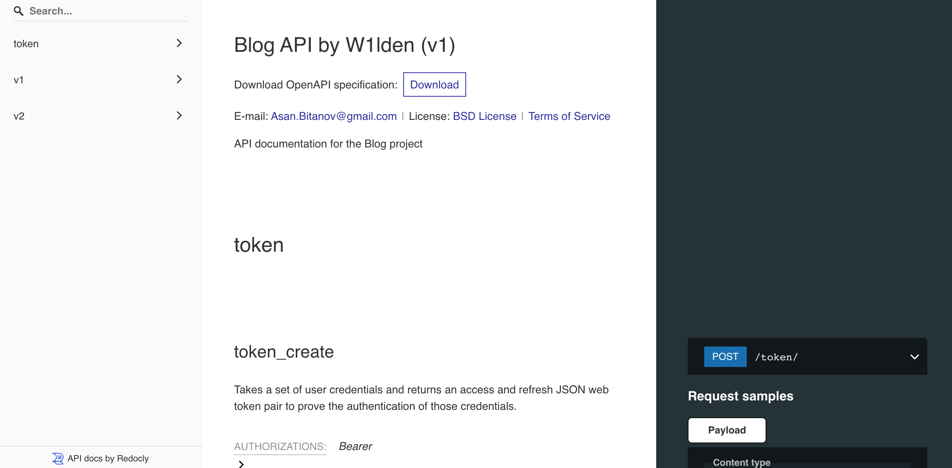


Figure 11. The ReDoc page of the project

The ReDoc page is another API documentation tool that presents the available endpoints in a clean, organized layout. It provides detailed descriptions of each endpoint, including request methods like GET, POST, PUT, DELETE, parameters, and the structure of responses. ReDoc is particularly useful for understanding the API's design, as it displays all endpoints in a structured menu.

# Advanced Features with Django Rest Framework

To implement nested serializers I edited the serializers.py file. I updated the PostSerializer class. The updated class is shown in Figure 12.



Figure 12. The serializers.py file

Here I updated the PostSerializer to include a nested CommentSerializer. This allows each post to display all related comments when retrieved. The comments field in the PostSerializer was set with many=True and read\_only=True to indicate that multiple comments could be associated with a single post, and they are fetched only for display purposes, not for modification through this serializer. This change ensures that when a post is retrieved, its response includes the comments in a structured format.

I added versioning to the API to make it easier to manage changes in the future. I used Django REST Framework's (DRF) built-in URLPathVersioning, which allows specifying the version in the URL. For example, /api/v1/posts/ uses version 1, while /api/v2/posts/ uses version 2. I created separate views and serializers for each version to show how the API can evolve over time.

For version 1, I kept the original PostViewSet and PostSerializer. For version 2, I created PostViewSetV2 and updated it to include more detailed responses or changes as needed. These modifications were done in the views.py file. The file is shown in Figure 13.



Figure 13. The new PostViewSetV2 class in the views.py file

I implemented rate limiting to control how many requests users can make to the API in a given time. To do it I edited the REST\_FRAMEWORK dictionary in the settings.py file. The dictionary is shown in Figure 14.



Figure 14. The REST\_FRAMEWORK dictionary in the settings.py file

Using Django REST Framework's throttle classes, I configured different rates for anonymous and authenticated users. Anonymous users are limited to 5 requests per minute. Authenticated users are allowed 10 requests per minute. This helps prevent abuse and ensures the server can handle requests efficiently.

To deploy the Django application, I used Docker to package the project. Docker makes sure the app works the same way on any system. I created a Dockerfile to set up the app's image and a docker-compose.yml file to handle the services. These services include the Django app and the database. The docker-compose file is shown in Figure 15.



Figure 15. The docker-compose file

In the docker-compose.yml file, I set up two services: web for the Django app and db for the PostgreSQL database. For the web service, I used Gunicorn to run the app on port 8000 and added volumes to store the project files and static files. For the database, I set up PostgreSQL with a database name, user, and password, and stored the data in a volume. The running container is shown in Figure 16.

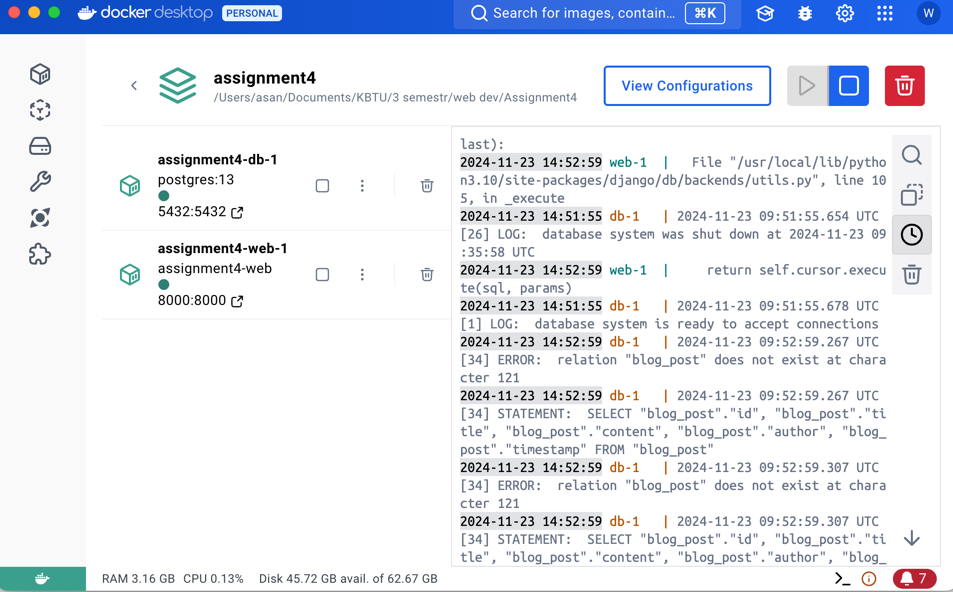


Figure 16. The running container

he web and database services were successfully launched. The web container runs the Django application with Gunicorn, while the db handles the PostgreSQL.

# Conclusion

In assignment 4, I built a blog API using Django Rest Framework (DRF) and deployed it with Docker. This project helped me create a backend system where users can manage posts and comments through API endpoints. I focused on making the API secure, scalable, and easy to use.

I used models to organize data for posts and comments, making it simple to store and access information. With serializers, I converted data between Python objects and JSON, so the API could easily send and receive data. Views built with DRF’s ModelViewSet handled important actions like creating, reading, updating, and deleting posts and comments. I also added permissions to control what users could do, so only authorized users could make changes.

To ensure all endpoints were accessible, I set up URL routing and added versioning to support future updates without breaking older versions. I tested the API using unit tests to make sure it worked correctly and could handle different situations without issues.

Finally, I used Docker to package the app, which made it run smoothly in any environment. I created a Dockerfile and docker-compose.yml to manage the web app and database, making deployment fast and simple.

This project helped me improve my skills in building APIs, securing them, and deploying them with Docker. It gave me practical experience with tools I can use in real-world backend development.

# References

1. Serializers – Django REST Framework [geeksforgeeks] <https://www.geeksforgeeks.org/serializers-django-rest-framework/>
2. ViewSets [Django Rest Framework] <https://www.django-rest-framework.org/api-guide/viewsets/#viewsets>