Title: “Building a RESTful API with Django Rest Framework”

Issayev Abzal

01.12.2024

KBTU

GitHub: https://github.com/Karisbala/WebDev2024/tree/main/Assignment4

Executive Summary

This project focused on developing a robust RESTful API for a blogging platform using Django Rest Framework (DRF). DRF was chosen for its comprehensive toolkit that streamlines the creation of scalable and maintainable APIs within the Django ecosystem. The development process began with setting up the Django project and application, ensuring that all necessary dependencies were installed and configured correctly.

Central to the application's functionality were the Post and Comment models, which were meticulously designed to capture essential data attributes and establish clear relationships between blog posts and their corresponding comments. These models were then paired with serializers that facilitate the seamless conversion of complex model instances into JSON format, enabling efficient data exchange between the server and clients.

Views and viewsets were implemented to handle various API endpoints, providing capabilities such as listing, creating, retrieving, updating, and deleting posts and comments. URL routing was configured using DRF's DefaultRouter, which automatically generated clean and intuitive URL patterns, enhancing the API's accessibility and usability.

A significant aspect of the project was the integration of authentication and permissions. Token-based authentication was employed to secure the API, ensuring that only authenticated users could perform write operations. Custom permission classes were developed to restrict certain actions, such as editing or deleting posts, exclusively to their respective authors. This approach not only bolstered the API's security but also maintained the integrity of user-generated content.

Advanced features were incorporated through the use of nested serializers, allowing comments to be embedded within post responses. This enhancement reduced the need for multiple API calls, thereby improving the efficiency and performance of data retrieval. Additionally, Docker was utilized to containerize the application, promoting consistency across different deployment environments and simplifying the deployment process.

Comprehensive testing was conducted using Django's testing framework, ensuring that all API endpoints functioned as intended and adhered to the specified requirements. Interactive API documentation was generated using Swagger via the drf-yasg package, providing developers with a user-friendly interface to explore and interact with the API's functionalities.

Throughout the development lifecycle, challenges such as authentication issues within the Swagger UI and model visibility in the Django admin panel were effectively addressed through targeted solutions, including proper configuration of authentication tokens and registering models with the admin interface.

Table of Contents

Introduction

RESTful APIs are vital in modern web development, providing a standardized method for web applications to communicate over HTTP protocols. Django Rest Framework (DRF) is a powerful toolkit that simplifies creating robust and scalable RESTful APIs using the Django web framework. This report outlines the development of a RESTful API for a blog application using DRF, covering project setup, data modeling, serialization, view creation, URL routing, authentication and permissions, nested serializers, deployment with Docker, and testing and documentation.

Building a RESTful API with Django Rest Framework

Project Setup

The project, named blog_project, was initialized using Django's startproject command. An application called blog_app was created within the project to handle the blog functionalities. Dependencies, including Django and Django Rest Framework, were installed using pip:

- django-admin startproject blog_project
- cd blog_project
- python manage.py startapp blog_app
- pip install django djangorestframework

In the settings.py file, the INSTALLED_APPS list was updated to include 'rest_framework' and 'blog_app', enabling their functionalities within the project.

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'rest_framework',
    'blog_app',
    'drf_yasg',
    'rest_framework.authtoken',
]
```

Data Models

Two primary data models were defined: Post and Comment. The Post model represents blog posts and includes fields for title, content, author, and timestamp. The Comment model represents comments on posts and includes fields for post (foreign key to Post), author, content, and timestamp.

```
from django.db import models
from django.contrib.auth.models import User

class Post(models.Model):
    title = models.CharField(max_length=200)
    content = models.TextField()
    author = models.ForeignKey(User, related_name='posts',
on_delete=models.CASCADE)
    timestamp = models.DateTimeField(auto_now_add=True)

    def __str__(self):
        return self.title

class Comment(models.Model):
    post = models.ForeignKey(Post, related_name='comments',
on_delete=models.CASCADE)
    author = models.ForeignKey(User, related_name='comments',
on_delete=models.CASCADE)
    content = models.TextField()
    timestamp = models.DateTimeField(auto_now_add=True)

    def __str__(self):
        return f'Comment by {self.author}'
```

The author fields are foreign keys to Django's built-in User model, establishing relationships between users and their posts or comments. The related_name attribute allows reverse lookup from the User model to associated posts and comments.

Serializers

Serializers in DRF handle the conversion of complex data types, like Django models, into native Python data types that can be easily rendered into JSON or other content types. Two serializers were implemented: PostSerializer and CommentSerializer.

```
from rest_framework import serializers
from .models import Post, Comment

class CommentSerializer(serializers.ModelSerializer):
    author = serializers.ReadOnlyField(source='author.username')

    class Meta:
        model = Comment
        fields = ['id', 'author', 'content', 'timestamp']
```

The CommentSerializer handles serialization and deserialization of Comment instances. The author field is read-only and displays the username of the comment's author.

```
class PostSerializer(serializers.ModelSerializer):
    author = serializers.ReadOnlyField(source='author.username')
    comments = CommentSerializer(many=True, read_only=True)
```

```
    class Meta:
        model = Post
        fields = ['id', 'title', 'content', 'author', 'timestamp',
'comments']
```

The PostSerializer handles serialization and deserialization of Post instances. It includes a nested comments field that uses the CommentSerializer to represent related comments.

Views and Endpoints

Views in DRF handle HTTP requests and return responses. Viewsets were used to group related views for the Post and Comment models, providing a set of default actions.

```
from rest_framework import viewsets, status
from rest_framework.decorators import action
from rest_framework.response import Response
from rest_framework.permissions import IsAuthenticatedOrReadOnly
from .models import Post, Comment
from .serializers import PostSerializer, CommentSerializer
from .permissions import IsAuthorOrReadOnly

class PostViewSet(viewsets.ModelViewSet):
    queryset = Post.objects.all().order_by('-timestamp')
    serializer_class = PostSerializer
    permission_classes = [IsAuthenticatedOrReadOnly, IsAuthorOrReadOnly]

    @action(detail=True, methods=['get', 'post'],
permission_classes=[IsAuthenticatedOrReadOnly])
    def comments(self, request, pk=None):
        post = self.get_object()
        if request.method == 'GET':
            comments = post.comments.all()
            serializer = CommentSerializer(comments, many=True)
            return Response(serializer.data)
        elif request.method == 'POST':
            serializer = CommentSerializer(data=request.data)
            if serializer.is_valid():
                serializer.save(post=post, author=request.user)
                return Response(
                    serializer.data, status=status.HTTP_201_CREATED
                )
            return Response(
                serializer.errors, status=status.HTTP_400_BAD_REQUEST
            )
    def perform_create(self, serializer):
        serializer.save(author=self.request.user)

class CommentViewSet(viewsets.ModelViewSet):
    queryset = Comment.objects.all()
    serializer_class = CommentSerializer
    permission_classes = [IsAuthenticatedOrReadOnly]
```

```
    def perform_create(self, serializer):
        serializer.save(author=self.request.user)
```

The PostViewSet provides actions like list, create, retrieve, update, and destroy for posts. It also includes a custom comments action to handle listing and creating comments related to a specific post. The perform_create method assigns the current user as the author when a new post is created.

The CommentViewSet provides similar actions for comments. Both viewsets use the IsAuthenticatedOrReadOnly permission class to allow read-only access to unauthenticated users and restrict write operations to authenticated users.

URL Routing

URL routing maps URLs to views. DRF's DefaultRouter automatically generates URL patterns for viewsets, simplifying the routing process.

```
from django.urls import include, path
from rest_framework.routers import DefaultRouter
from .views import PostViewSet, CommentViewSet

router = DefaultRouter()
router.register(r'posts', PostViewSet)
router.register(r'comments', CommentViewSet)

urlpatterns = [
    path('', include(router.urls)),
]
```

In the project's urls.py, the blog_app URLs are included under the blog/ path.

```
from django.contrib import admin
from django.urls import include, path
from rest_framework import permissions
from drf_yasg.views import get_schema_view
from drf_yasg import openapi
from rest_framework.authtoken.views import obtain_auth_token

schema_view = get_schema_view(
   openapi.Info(
      title="Blog API",
      default_version='v1',
      description="API documentation for the Blog app",
   ),
   public=True,
   permission_classes=(permissions.AllowAny,),
)

urlpatterns = [
    path('admin/', admin.site.urls),
    path('blog/', include('blog_app.urls')),
```

```
    path('swagger/', schema_view.with_ui('swagger', cache_timeout=0),
name='schema-swagger-ui'),
    path('api-token-auth/', obtain_auth_token, name='api_token_auth'),
]
```

Authentication and Permissions

Authentication is implemented using DRF's token-based authentication. Users can obtain a token by sending their credentials to the /api-token-auth/ endpoint. The token is then used in the Authorization header of subsequent requests.

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'rest_framework',
    'blog_app',
    'drf_yasg',
    'rest_framework.authtoken',
]
REST_FRAMEWORK = {
    'DEFAULT_AUTHENTICATION_CLASSES': [
        'rest_framework.authentication.TokenAuthentication',
    ],
    'DEFAULT_PERMISSION_CLASSES': [
        'rest_framework.permissions.IsAuthenticatedOrReadOnly',
    ],
}
```

A custom permission class IsAuthorOrReadOnly ensures that only the author of a post can edit or delete it.

```
from rest_framework import permissions

class IsAuthorOrReadOnly(permissions.BasePermission):

    def has_object_permission(self, request, view, obj):
        if request.method in permissions.SAFE_METHODS:
            return True

        return obj.author == request.user
```

Advanced Features with Django Rest Framework

Nested Serializers

Nested serializers were implemented to include related comments within the serialized representation of a post. This allows clients to retrieve a post along with all its comments in a single API call, improving efficiency and reducing the number of requests.

In the PostSerializer, the comments field uses the CommentSerializer with many=True and read_only=True.

```
class PostSerializer(serializers.ModelSerializer):
    author = serializers.ReadOnlyField(source='author.username')
    comments = CommentSerializer(many=True, read_only=True)

    class Meta:
        model = Post
        fields = ['id', 'title', 'content', 'author', 'timestamp',
'comments']
```

Deployment

Steps taken to prepare and deploy the application.

Docker was used to containerize the application, making deployment consistent across different environments.

```
# Use the official Python image
FROM python:3.9-slim
# Set the working directory
WORKDIR /app
# Copy the requirements file
COPY requirements.txt .
# Install dependencies
RUN pip install --no-cache-dir -r requirements.txt
# Copy the project files
COPY . .
# Expose the port
EXPOSE 8000
# Run the server
CMD ["python", "manage.py", "runserver", "0.0.0.0:8000"]
```

Deployment Steps:

1. Build the Docker image: docker build -t blog_project .
2. Run the Docker container: docker run -p 8000:8000 blog_project

The application is accessible at http://localhost:8000/.

Testing and Documentation

API Testing

Unit tests were written using Django's APITestCase to verify the functionality of API endpoints, including authentication and permissions.

```
from django.urls import reverse
from rest_framework import status
from rest_framework.test import APITestCase
from django.contrib.auth.models import User
from .models import Post, Comment
from rest_framework.authtoken.models import Token

class PostTests(APITestCase):
    def setUp(self):
        self.user = User.objects.create_user(username='testuser',
password='testpass')
        # Generate a token for the user
        self.token = Token.objects.create(user=self.user)
        # Set the token in the client
        self.client.credentials(HTTP_AUTHORIZATION='Token ' +
self.token.key)
        self.post_url = reverse('post-list')

    def test_create_post_authenticated(self):
        data = {'title': 'Test Post', 'content': 'Test Content'}
        response = self.client.post(self.post_url, data, format='json')
        self.assertEqual(response.status_code, status.HTTP_201_CREATED)
        self.assertEqual(Post.objects.count(), 1)
        self.assertEqual(Post.objects.get().title, 'Test Post')

    def test_create_post_unauthenticated(self):
        self.client.credentials()  # Remove any credentials
        data = {'title': 'Test Post', 'content': 'Test Content'}
        response = self.client.post(self.post_url, data, format='json')
        self.assertEqual(response.status_code,
status.HTTP_401_UNAUTHORIZED)

    def test_retrieve_posts(self):
        Post.objects.create(title='Test Post', content='Test Content',
author=self.user)
        response = self.client.get(self.post_url, format='json')
        self.assertEqual(response.status_code, status.HTTP_200_OK)
        self.assertEqual(len(response.data), 1)

class CommentTests(APITestCase):
    def setUp(self):
        self.user = User.objects.create_user(username='testuser',
password='testpass')
        # Generate a token for the user
        self.token = Token.objects.create(user=self.user)
```

```
        # Set the token in the client
        self.client.credentials(HTTP_AUTHORIZATION='Token ' +
self.token.key)
        self.post = Post.objects.create(title='Test Post', content='Test
Content', author=self.user)
        self.comment_url = reverse('post-comments', kwargs={'pk':
self.post.id})

    def test_create_comment_authenticated(self):
        data = {'content': 'Test Comment'}
        response = self.client.post(self.comment_url, data, format='json')
        self.assertEqual(response.status_code, status.HTTP_201_CREATED)
        self.assertEqual(Comment.objects.count(), 1)
        self.assertEqual(Comment.objects.get().content, 'Test Comment')

    def test_create_comment_unauthenticated(self):
        self.client.credentials()  # Remove any credentials
        data = {'content': 'Test Comment'}
        response = self.client.post(self.comment_url, data, format='json')
        self.assertEqual(response.status_code,
status.HTTP_401_UNAUTHORIZED)

    def test_retrieve_comments(self):
        Comment.objects.create(content='Test Comment', author=self.user,
post=self.post)
        response = self.client.get(self.comment_url, format='json')
        self.assertEqual(response.status_code, status.HTTP_200_OK)
        self.assertEqual(len(response.data), 1)
```

API Documentation

Swagger UI was integrated using drf-yasg to provide interactive API documentation. This allows developers to explore and test API endpoints directly from the documentation interface.

```
from django.contrib import admin
from django.urls import include, path
from rest_framework import permissions
from drf_yasg.views import get_schema_view
from drf_yasg import openapi
from rest_framework.authtoken.views import obtain_auth_token

schema_view = get_schema_view(
   openapi.Info(
      title="Blog API",
      default_version='v1',
      description="API documentation for the Blog app",
   ),
   public=True,
   permission_classes=(permissions.AllowAny,),
)

urlpatterns = [
    path('admin/', admin.site.urls),
```

```
    path('blog/', include('blog_app.urls')),
    path('swagger/', schema_view.with_ui('swagger', cache_timeout=0),
name='schema-swagger-ui'),
    path('api-token-auth/', obtain_auth_token, name='api_token_auth'),
]
```

The Swagger UI can be accessed at http://localhost:8000/swagger/, providing a user-friendly interface for interacting with the API.

Challenges and Solutions

Several challenges were encountered during development. The Post and Comment models were not visible in the Django admin interface. Registering these models in admin.py resolved the issue. Tests were failing due to improper authentication setup. Adjusting the test cases to use token authentication instead of session authentication fixed the failures.

Conclusion

Building a RESTful API with Django Rest Framework provides a robust and efficient method for developing web APIs. The project successfully implemented a blog application with features such as data modeling, serialization, viewsets, URL routing, authentication, permissions, nested serializers, deployment with Docker, and testing and documentation. Nested serializers enhanced the API by allowing related data to be included in responses, improving efficiency.

Recommendations

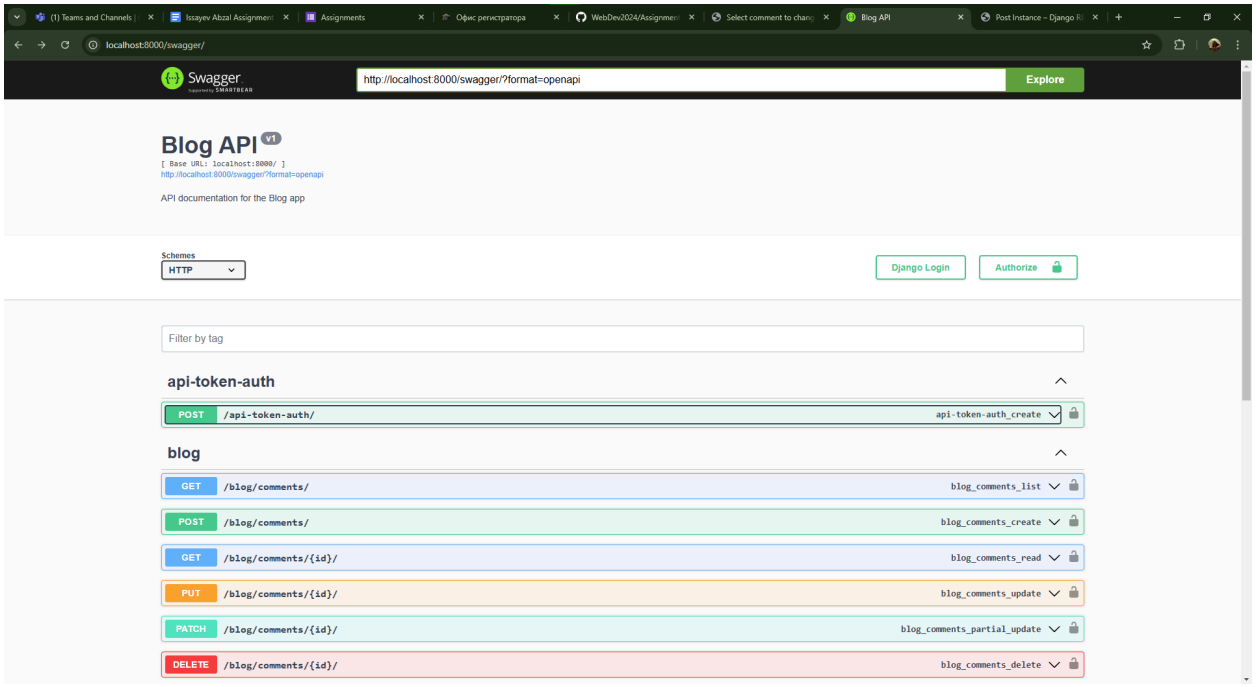
There is plenty of room for further improvements. Implement more comprehensive tests to cover additional functionalities and edge cases. For endpoints returning large datasets, adding pagination would improve performance and user experience. Consider implementing HTTPS and additional security measures for production environments. Set up CI/CD pipelines to automate testing and deployment processes.

References

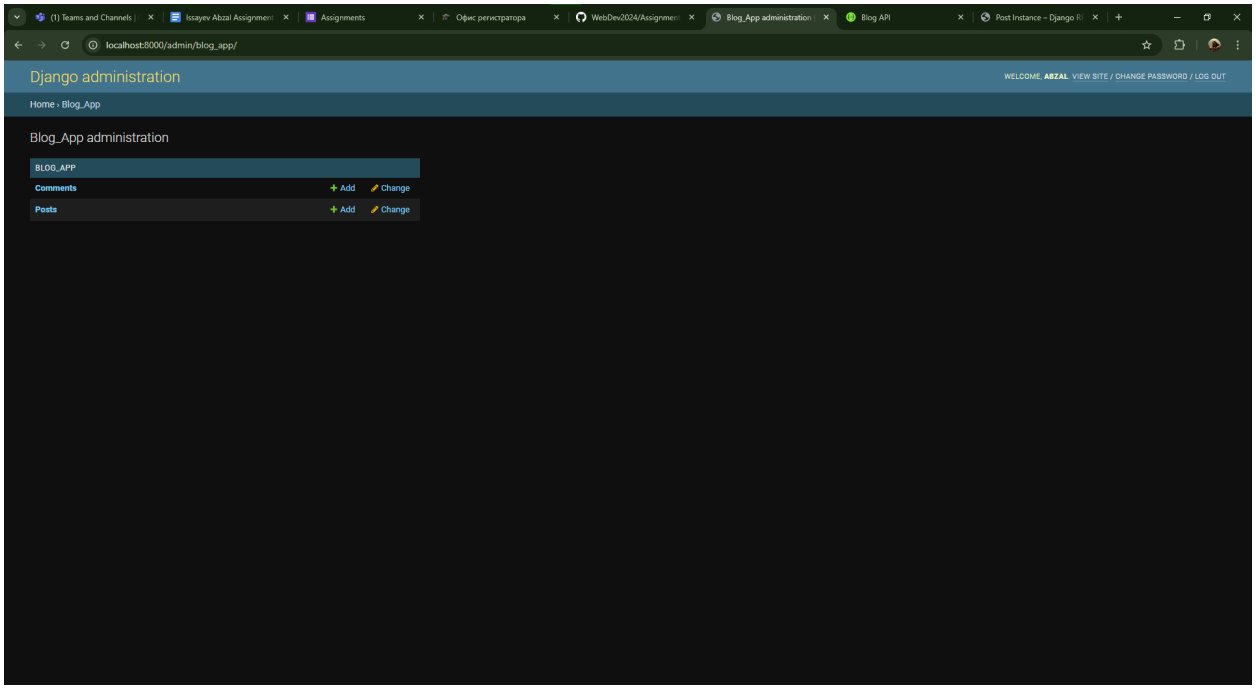
1. Django Documentation: https://docs.djangoproject.com/
2. Django Rest Framework Documentation: https://www.django-rest-framework.org/
3. drf-yasg Documentation: https://drf-yasg.readthedocs.io/
4. Docker Documentation: https://docs.docker.com/

Appendices

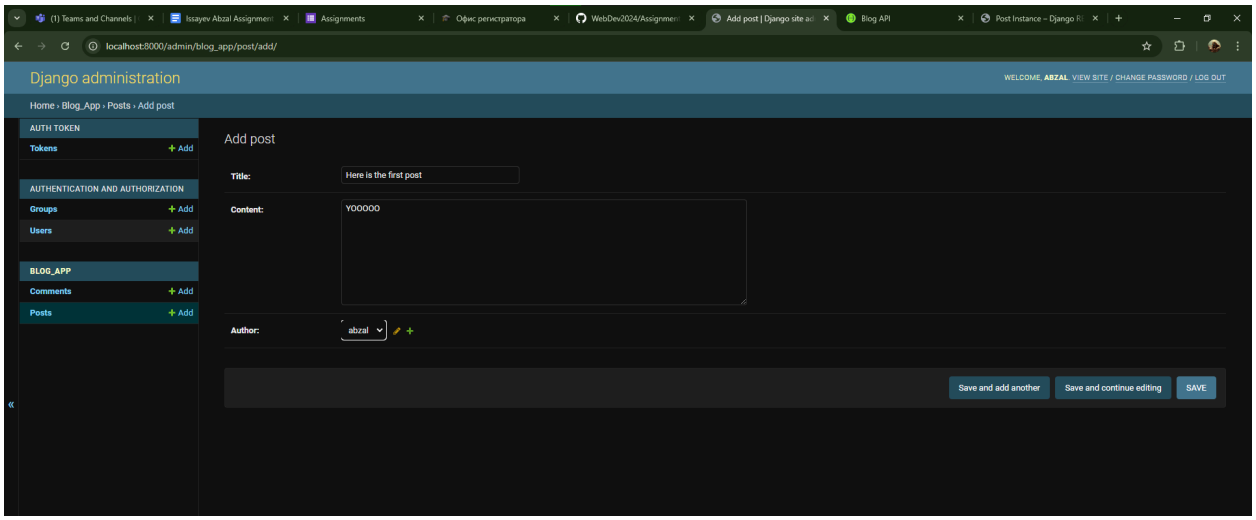
Additional material, such as code snippets, diagrams, or supplementary analysis.



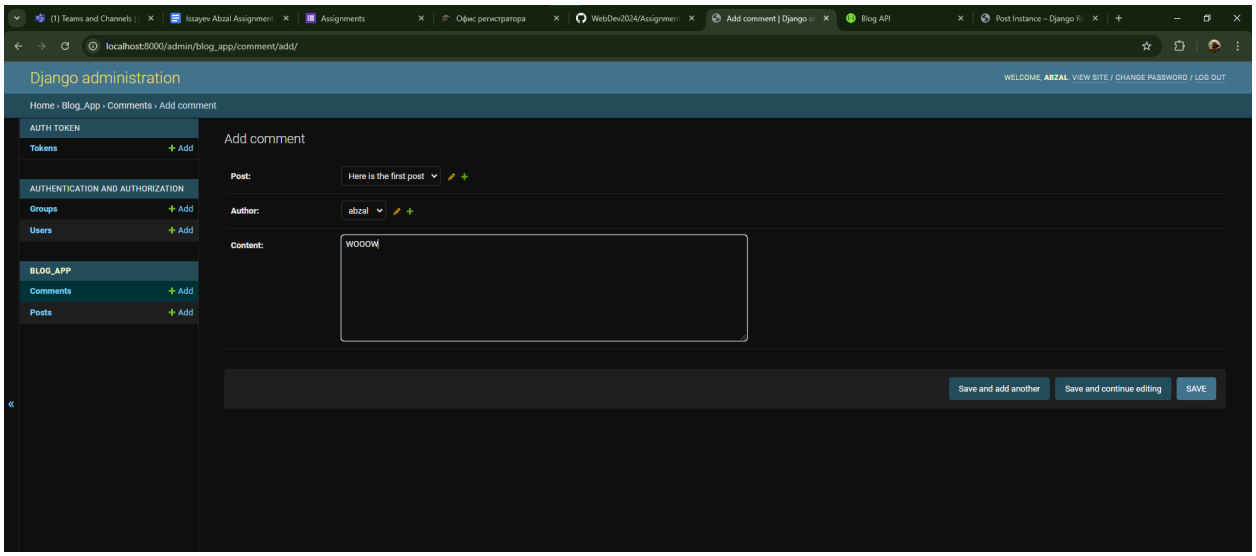
Picture 1. Swagger documentation UI.



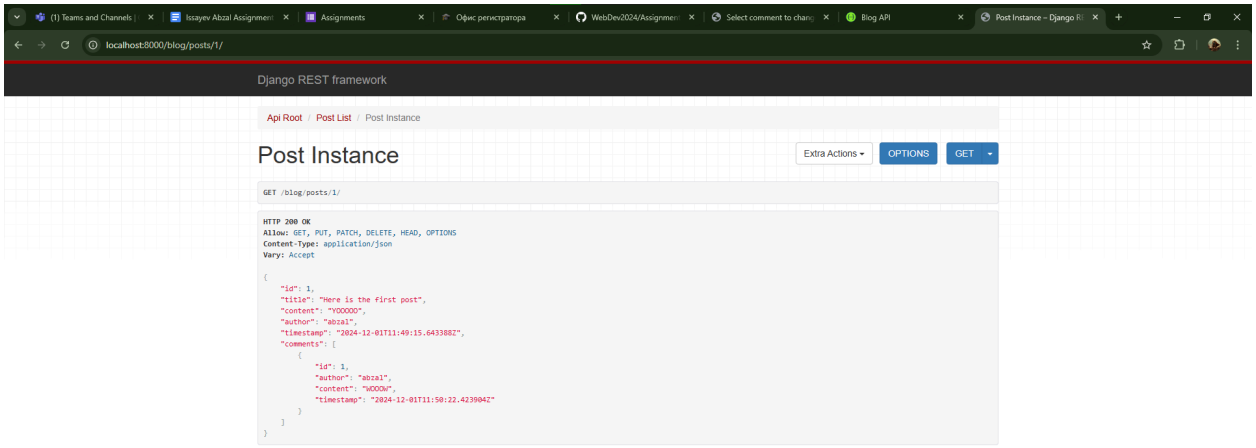
Picture 2. Admin panel view.



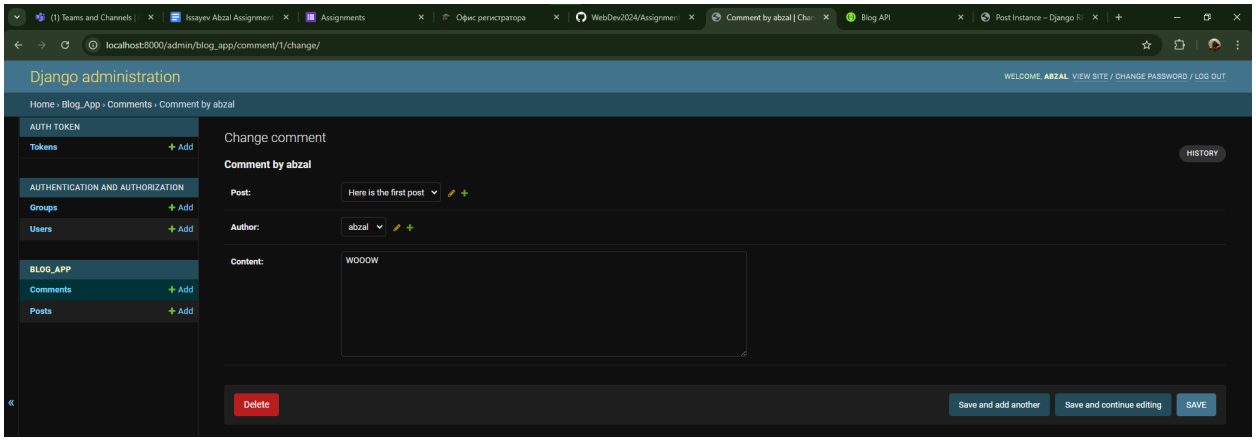
Picture 3. Creating a new post.



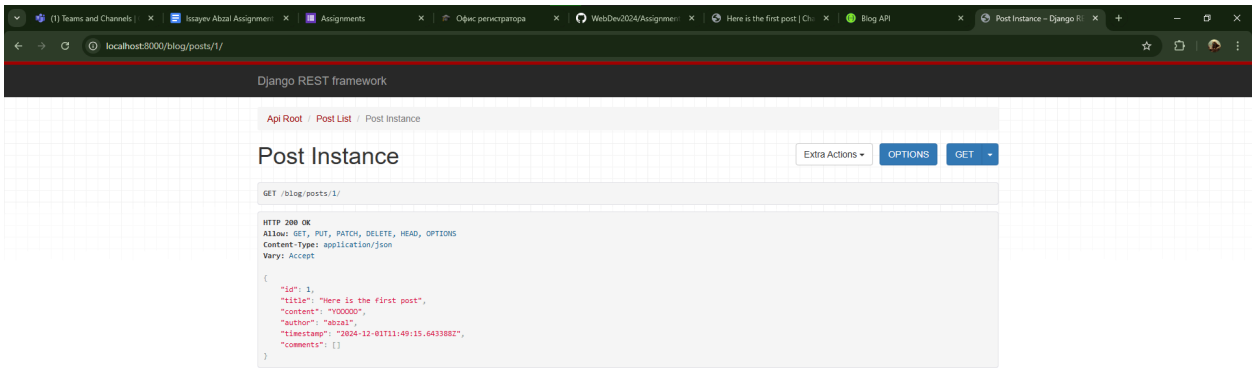
Picture 4. Adding new comments to the post.



Picture 5. Newly created post and comment from get request api endpoint.



Picture 6. Deleting the comment form admin panel.



Picture 7. Post response body after deleting the comment.