KBTU KAZAKH-BRITISH TECHNICAL UNIVERSITY

Prepared by: Serzhan Yerasil

# Assignment 4

Web Development

30.11.2024

# Table of Contents

## Introduction

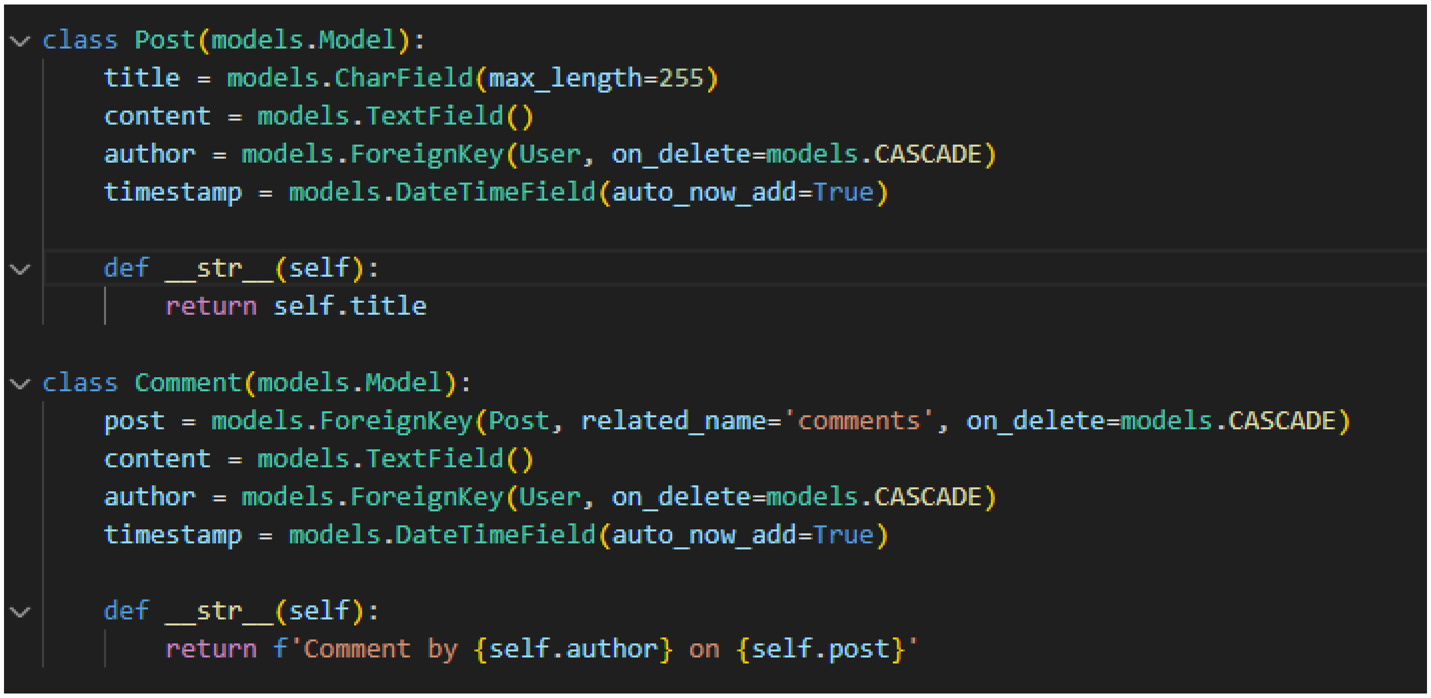
This project demonstrates the use of core DRF features such as serializers, viewsets, permissions, and routers, alongside advanced functionalities like nested serializers, API versioning, and rate limiting. The API is secured with JWT-based authentication and custom permissions, ensuring robust access control. Comprehensive documentation and automated testing further enhance the usability and reliability of the API.

## Project Setup

The project was initialized using Django and Django Rest Framework (DRF). Dependencies such as djangorestframework, drf-spectacular, and djangorestframework-simplejwt were installed to enable API functionality, documentation, and token-based authentication. The project includes a blog app to manage core features like posts and comments.

## Data Models

Post Represents blog posts with fields for title, content, author (linked to User), and timestamp. Each post is associated with a user through a ForeignKey relationship. Comment Represents comments with fields for post (linked to Post), content, author (linked to User), and timestamp. Comments are related to specific posts and users, enabling nested relationships.

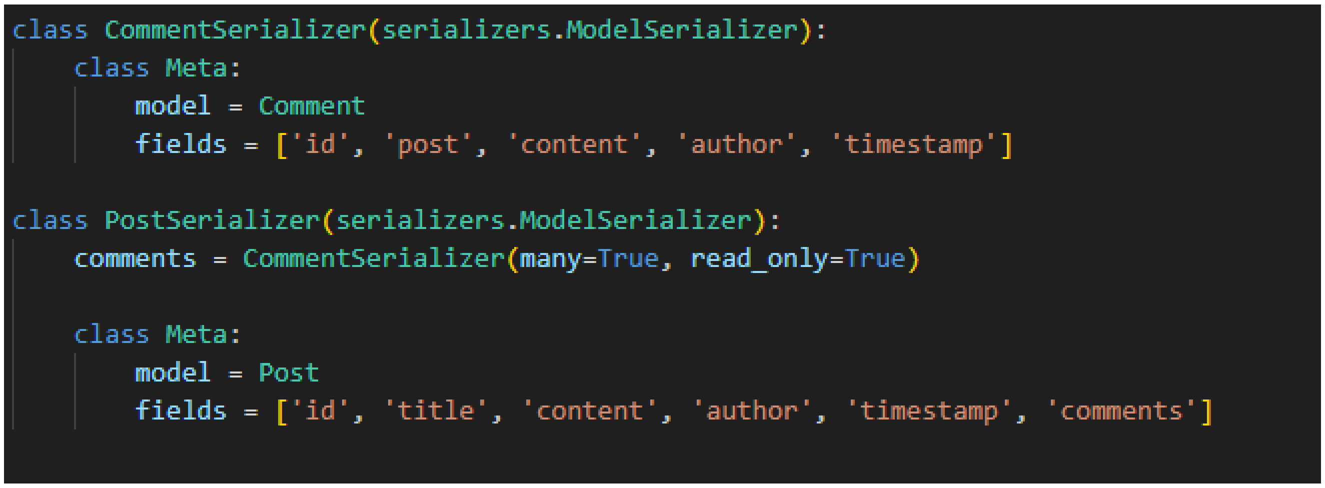


```
class Post(models.Model):
    title = models.CharField(max_length=255)
    content = models.TextField()
    author = models.ForeignKey(User, on_delete=models.CASCADE)
    timestamp = models.DateTimeField(auto_now_add=True)

    def __str__(self):
        return self.title

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

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

The Post model represents the data for a blog post, including the title, content, author (related to the User), and timestamp. If the User is deleted, all of their posts are also deleted. The Comment model stores data for comments that are associated with a specific post via the post field (a one-to-many relationship). Each comment has an author, content, and timestamp. The related_name='comments' field makes it easy to list all the comments for a post. The __str__ methods on both models return a convenient textual representation of the object, such as the post title or "Author's comment on post".

## Serializers

PostSerializer Serializes post data and includes a nested CommentSerializer to provide all comments related to a post in API responses.
CommentSerializer Handles serialization for comment data, supporting creation, update, and retrieval of individual comments. These serializers ensure clean input validation and data transformation for API responses.

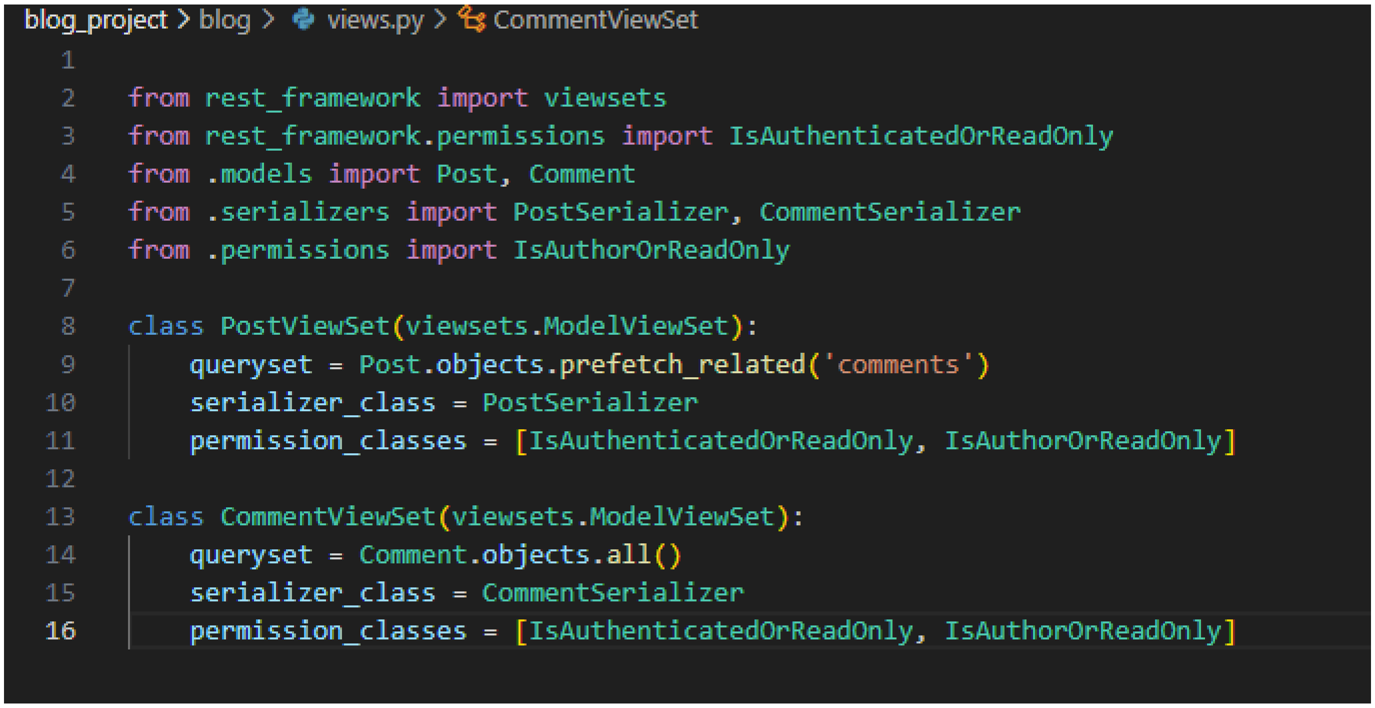


```python
class CommentSerializer(serializers.ModelSerializer):
    class Meta:
        model = Comment
        fields = ['id', 'post', 'content', 'author', 'timestamp']

class PostSerializer(serializers.ModelSerializer):
    comments = CommentSerializer(many=True, read_only=True)

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

The CommentSerializer is responsible for converting the Comment model data into JSON format, including fields such as ID, post, content, author, and creation time. It is used to process individual comment data. The PostSerializer includes comments through a nested CommentSerializer with many=True, which allows it to return a list of related comments for each post. The read_only=True field in the nested serializer setting means that comments are included as read-only and cannot be modified when creating or updating a post. Thus, when requesting post data from the API, the client also receives all related comments in JSON format. This structure simplifies working with data, keeping it related and easy to use.

## Views and Endpoints

PostViewSet view Implements CRUD operations for posts and returns nested comments when retrieving post details. It ensures that each post's data is accessible and manageable through the API. It relates with /posts endpoint CommentViewSet view Provides endpoints for managing comments, including listing, creating, and deleting comments linked to specific posts. It ensures efficient handling of comments through related views. It relates with /comments endpoint



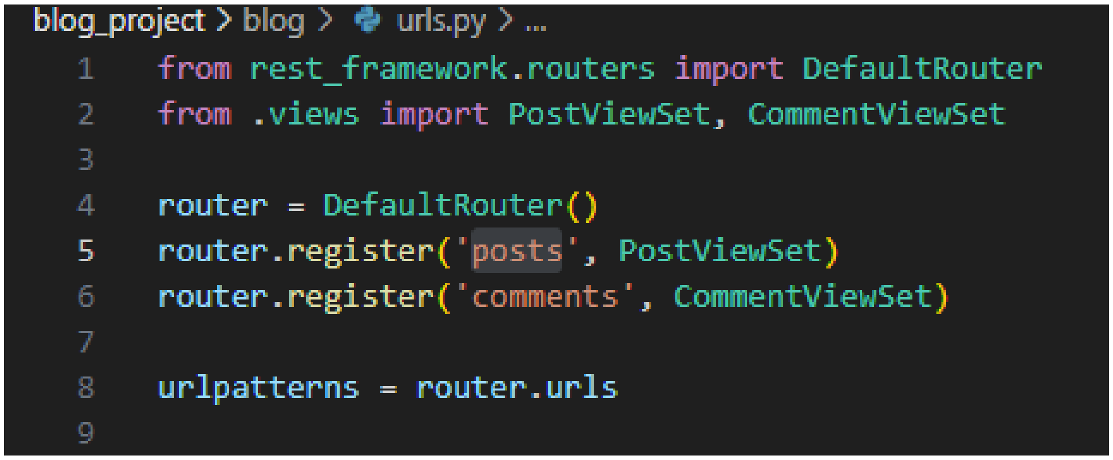
blog_project > blog > views.py > CommentViewSet

```python

from rest_framework import viewsets
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.prefetch_related('comments')
    serializer_class = PostSerializer
    permission_classes = [IsAuthenticatedOrReadOnly, IsAuthorOrReadOnly]

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

PostViewSet and CommentViewSet are classes for managing posts and comments API endpoints using the built-in functionality of ModelViewSet. PostViewSet specifies a queryset that prefetches related comments (prefetch_related('comments')) to optimize database interactions, and a PostSerializer to handle data. CommentViewSet works with comments using CommentSerializer to transform data. Both classes use the IsAuthenticatedOrReadOnly permission to provide read-only access to anonymous users, and a custom IsAuthorOrReadOnly permission to allow only authors to modify or delete their posts. These settings provide a convenient way to manage data through the API while keeping security and performance in mind.

## URL Routing

Urls are configured with DRF’s DefaultRouter to manage endpoint routing automatically. The API supports versioning:
/api/v1/posts/: Provides endpoints for post operations (list, create, retrieve, update, delete).
/api/v1/comments/: Handles CRUD operations for comments. Additionally, API documentation is available via Swagger and ReDoc at /api/schema/swagger-ui/ and /api/schema/redoc/.

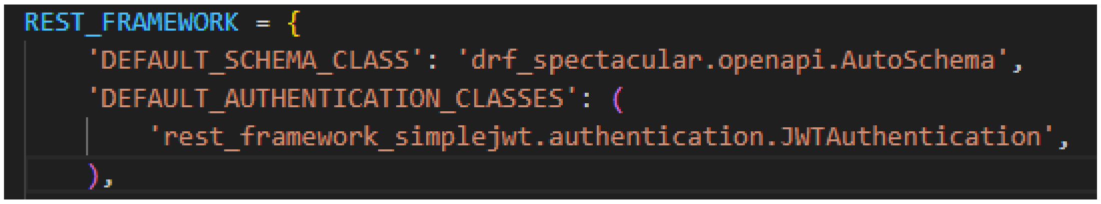

blog_project > blog > urls.py > ...

```python
from rest_framework.routers import DefaultRouter
from .views import PostViewSet, CommentViewSet

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

urlpatterns = router.urls

```

This code creates a router to automatically manage API endpoints using the DefaultRouter. The router registers two view sets: PostViewSet under the 'posts' route and CommentViewSet under the 'comments' route. This automatically generates standard RESTful routes such as GET /posts/ to get a list of posts, POST /posts/ to create a new post, GET /posts/{id}/ to get a specific post, and similar routes for comments. This approach simplifies URL configuration and makes the API structured and easy to use.

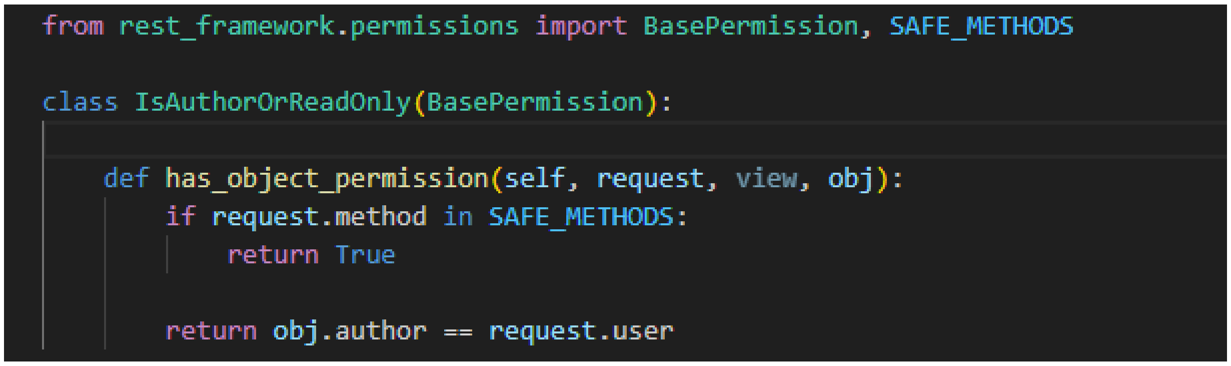
## Authentication and Permissions

Token-based authentication is implemented using rest_framework_simplejwt, ensuring secure access via JWT tokens**.** Anonymous users can view posts and comments, while authenticated users can create, update, or delete them. A custom permission class, IsAuthorOrReadOnly, restricts modifications to posts or comments to their respective authors, ensuring user-specific access control.



```
REST_FRAMEWORK = {
    'DEFAULT_SCHEMA_CLASS': 'drf_spectacular.openapi.AutoSchema',
    'DEFAULT_AUTHENTICATION_CLASSES': (
        'rest_framework_simplejwt.authentication.JWTAuthentication',
    ),
```

Default auth classs specifies the authentication method used in the API. JWT (JSON Web Token) authentication provided by the rest_framework_simplejwt library is enabled here. This allows you to secure API endpoints by requiring a token in the request header to access protected resources.



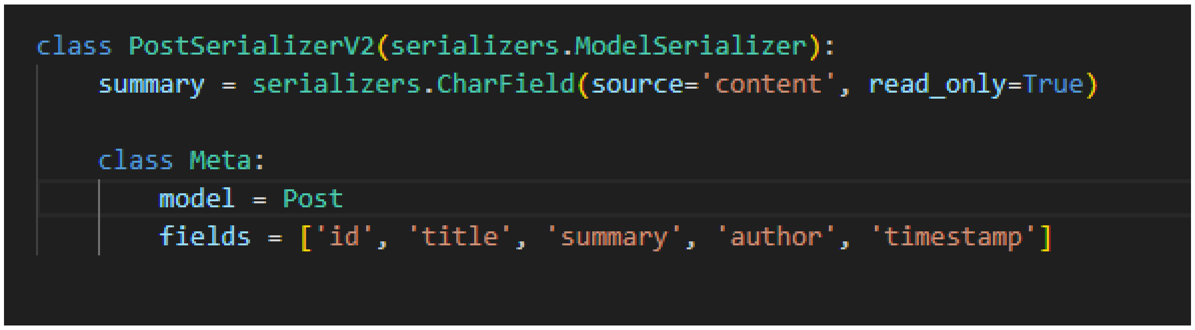
```
from rest_framework.permissions import BasePermission, SAFE_METHODS

class IsAuthorOrReadOnly(BasePermission):

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

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

The IsAuthorOrReadOnly class represents a custom permission for controlling access to objects in the API.

has_object_permission  this method determines whether the user has permission to perform an action on a specific object.
SAFE_METHODS contains read-only methods (GET, HEAD, OPTIONS). If the request method is safe, the method returns True, allowing access to any user.

## Nested Serializers

Nested serializers are implemented in the PostSerializer to include related Comment data directly within post responses. This allows clients to retrieve all associated comments when fetching a post, improving efficiency by reducing the number of API calls needed. For instance, each post's comments field uses CommentSerializer with read_only=True, enabling seamless inclusion of nested data without affecting write operations. This structure is ideal for maintaining a clear relationship between posts and their comments.

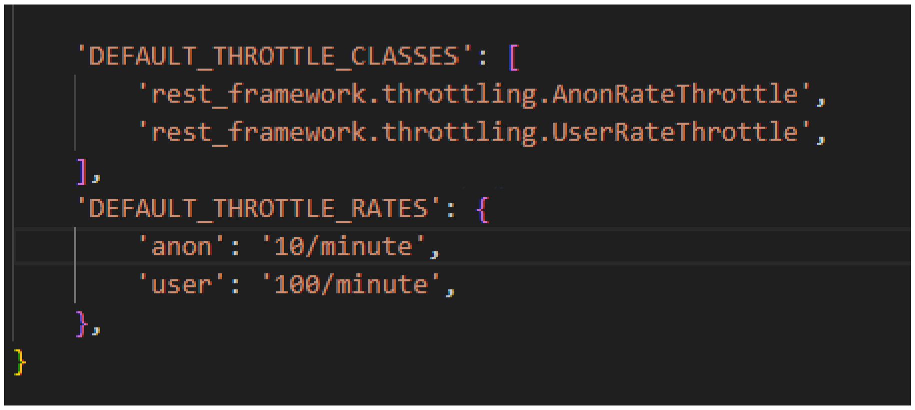


```python
class PostSerializerV2(serializers.ModelSerializer):
    summary = serializers.CharField(source='content', read_only=True)

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

The PostSerializerV2 class is a serializer for version 2 of the API that adds a summary field that displays the post's content as read-only. It includes fields such as id, title, summary, author, and timestamp, while preserving the standard functionality of the Post model. This approach allows changes to the API structure to be made without affecting previous versions.

## Rate Limiting

Rate limiting is configured using DRF's throttling classes, providing separate rates for anonymous and authenticated users. For example, anonymous users are restricted to 10 requests per minute (AnonRateThrottle), while authenticated users can make up to 100 requests per minute (UserRateThrottle). This prevents abuse of API resources while ensuring fair access for all users. These settings are defined in the REST_FRAMEWORK configuration and help maintain API performance and reliability.

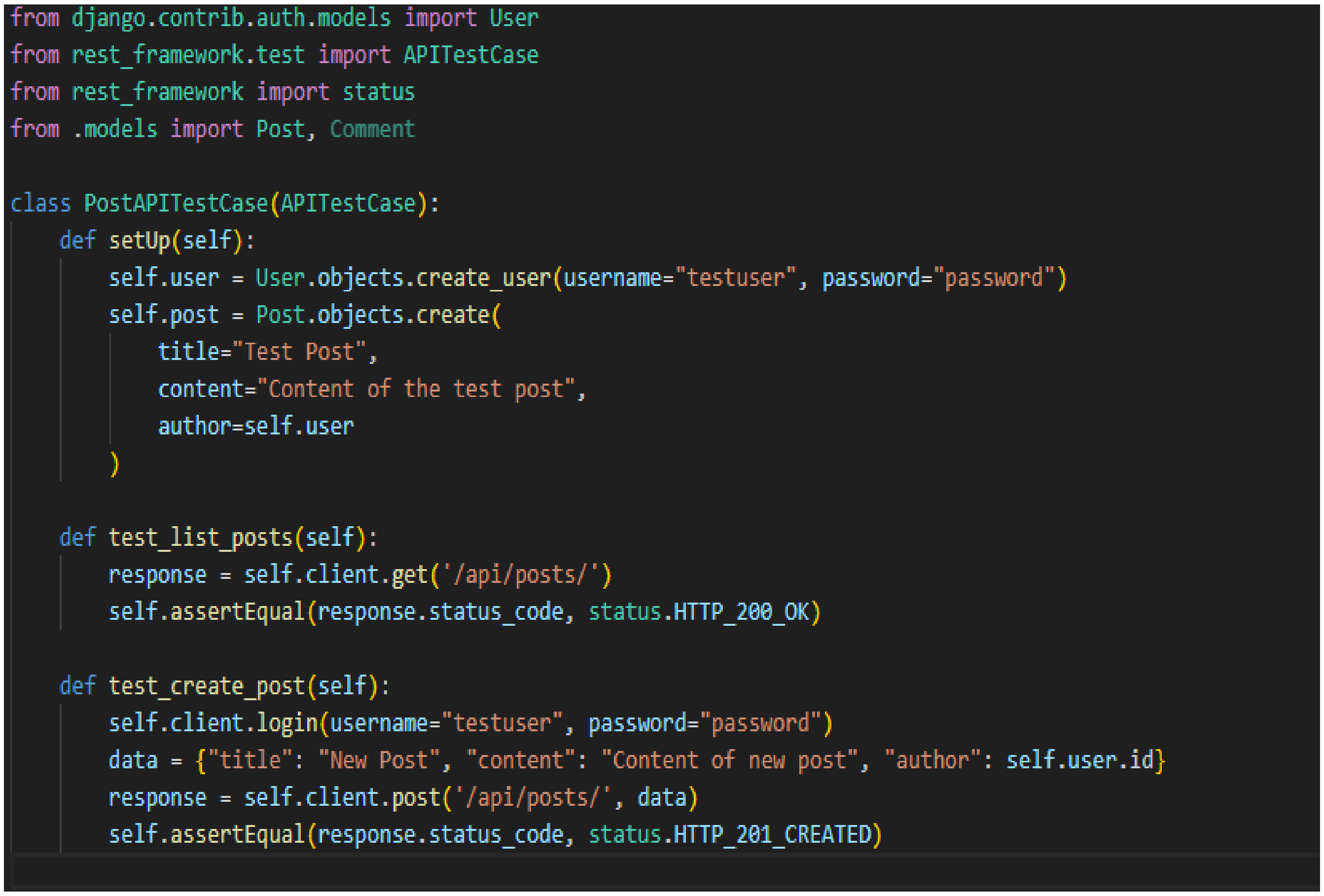


```
    'DEFAULT_THROTTLE_CLASSES': [
        'rest_framework.throttling.AnonRateThrottle',
        'rest_framework.throttling.UserRateThrottle',
    ],
    'DEFAULT_THROTTLE_RATES': {
        'anon': '10/minute',
        'user': '100/minute',
    },
}
```

DEFAULT_THROTTLE_CLASSES uses different classes to limit request AnonRateThrottle for anonymous users and UserRateThrottle for authenticated users.
DEFAULT_THROTTLE_RATES  sets limits: anonymous users can make up to 10 requests per minute, and authenticated users can make up to 100.
This prevents abuse of API resources by ensuring fair load distribution

## API Testing

API endpoints are tested using Django's TestCase and DRF's APIClient. Test cases validate endpoint functionalities, including success scenarios, edge cases, and permission restrictions. For instance, creating a post is tested to ensure only authenticated users can perform the action, while retrieving a list of posts verifies that it is accessible to all users. Automated tests ensure API reliability and correctness with every code change.

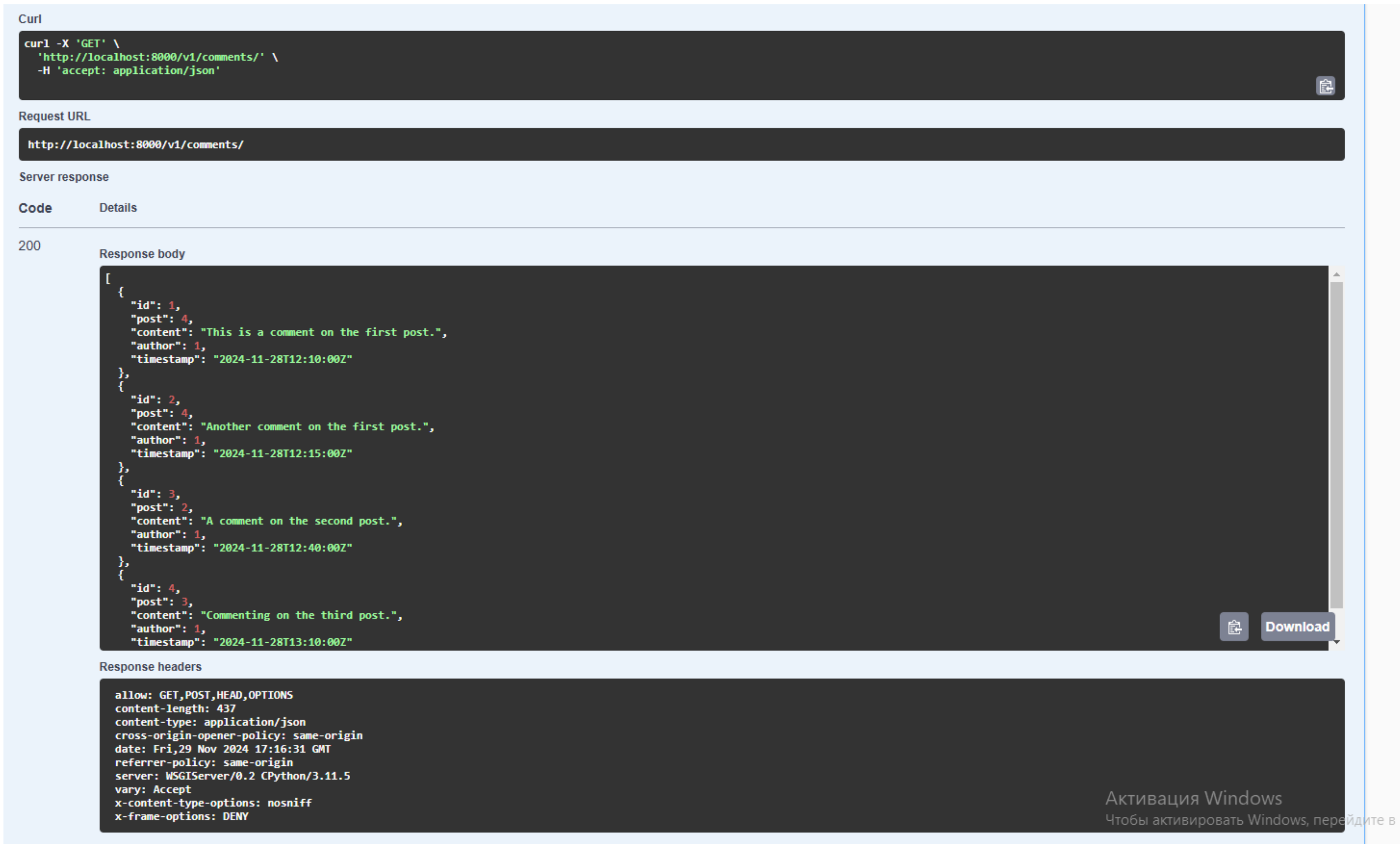


```python
from django.contrib.auth.models import User
from rest_framework.test import APITestCase
from rest_framework import status
from .models import Post, Comment

class PostAPITestCase(APITestCase):
    def setUp(self):
        self.user = User.objects.create_user(username="testuser", password="password")
        self.post = Post.objects.create(
            title="Test Post",
            content="Content of the test post",
            author=self.user
        )

    def test_list_posts(self):
        response = self.client.get('/api/posts/')
        self.assertEqual(response.status_code, status.HTTP_200_OK)

    def test_create_post(self):
        self.client.login(username="testuser", password="password")
        data = {"title": "New Post", "content": "Content of new post", "author": self.user.id}
        response = self.client.post('/api/posts/', data)
        self.assertEqual(response.status_code, status.HTTP_201_CREATED)
```

The PostAPITestCase class is a set of tests for testing post-related API endpoints.
setUp creates a test user and post before running the tests.
test_list_posts verifies that the GET /api/posts/ request returns a status of 200 (success), confirming that the list of posts is available.
test_create_post logs in a test user and sends a POST /api/posts/ request with the data to create a new post. Verifies that the status returned is 201 (successful creation).

## API Documentation

API documentation is generated using drf-spectacular, providing interactive Swagger and ReDoc interfaces. The documentation includes detailed descriptions of all endpoints, request/response formats, and authentication requirements (e.g., Bearer token usage). It automatically reflects changes in the codebase, ensuring accuracy. Components such as schema definitions, parameter details, and security schemes make it easier for developers to understand and interact with the API.



Curl

```
curl -X 'GET' \
  'http://localhost:8000/v1/comments/' \
  -H 'accept: application/json'
```

Request URL

```
http://localhost:8000/v1/comments/
```

Server response

Code Details

200

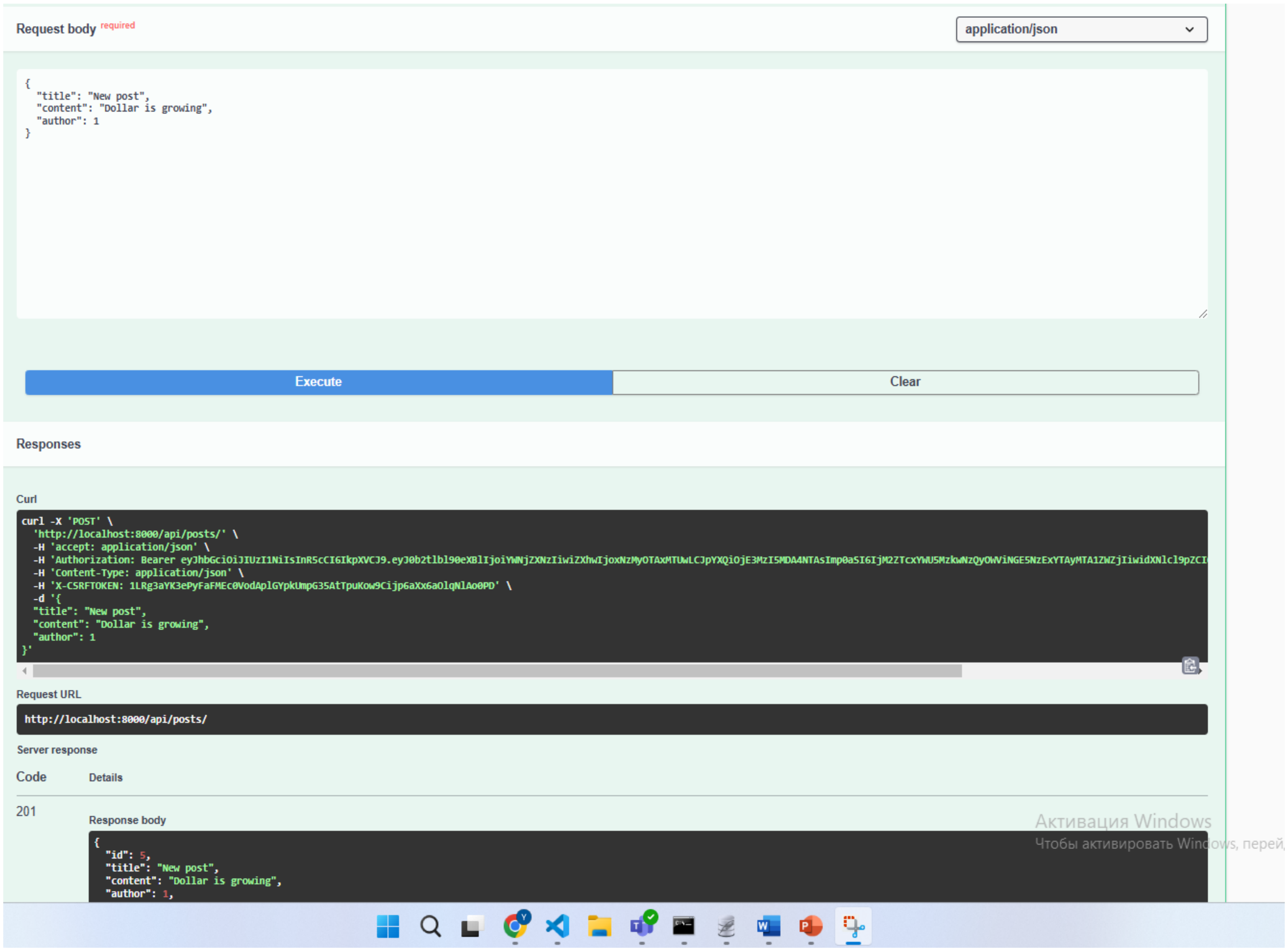
Response body

```
[
  {
    "id": 1,
    "post": 4,
    "content": "This is a comment on the first post.",
    "author": 1,
    "timestamp": "2024-11-28T12:10:00Z"
  },
  {
    "id": 2,
    "post": 4,
    "content": "Another comment on the first post.",
    "author": 1,
    "timestamp": "2024-11-28T12:15:00Z"
  },
  {
    "id": 3,
    "post": 2,
    "content": "A comment on the second post.",
    "author": 1,
    "timestamp": "2024-11-28T12:40:00Z"
  },
  {
    "id": 4,
    "post": 3,
    "content": "Commenting on the third post.",
    "author": 1,
    "timestamp": "2024-11-28T13:10:00Z"
```

Response headers

```
allow: GET,POST,HEAD,OPTIONS
content-length: 437
content-type: application/json
cross-origin-opener-policy: same-origin
date: Fri,29 Nov 2024 17:16:31 GMT
referrer-policy: same-origin
server: WSGIServer/0.2 CPython/3.11.5
vary: Accept
x-content-type-options: nosniff
x-frame-options: DENY
```

Get request for
Posts



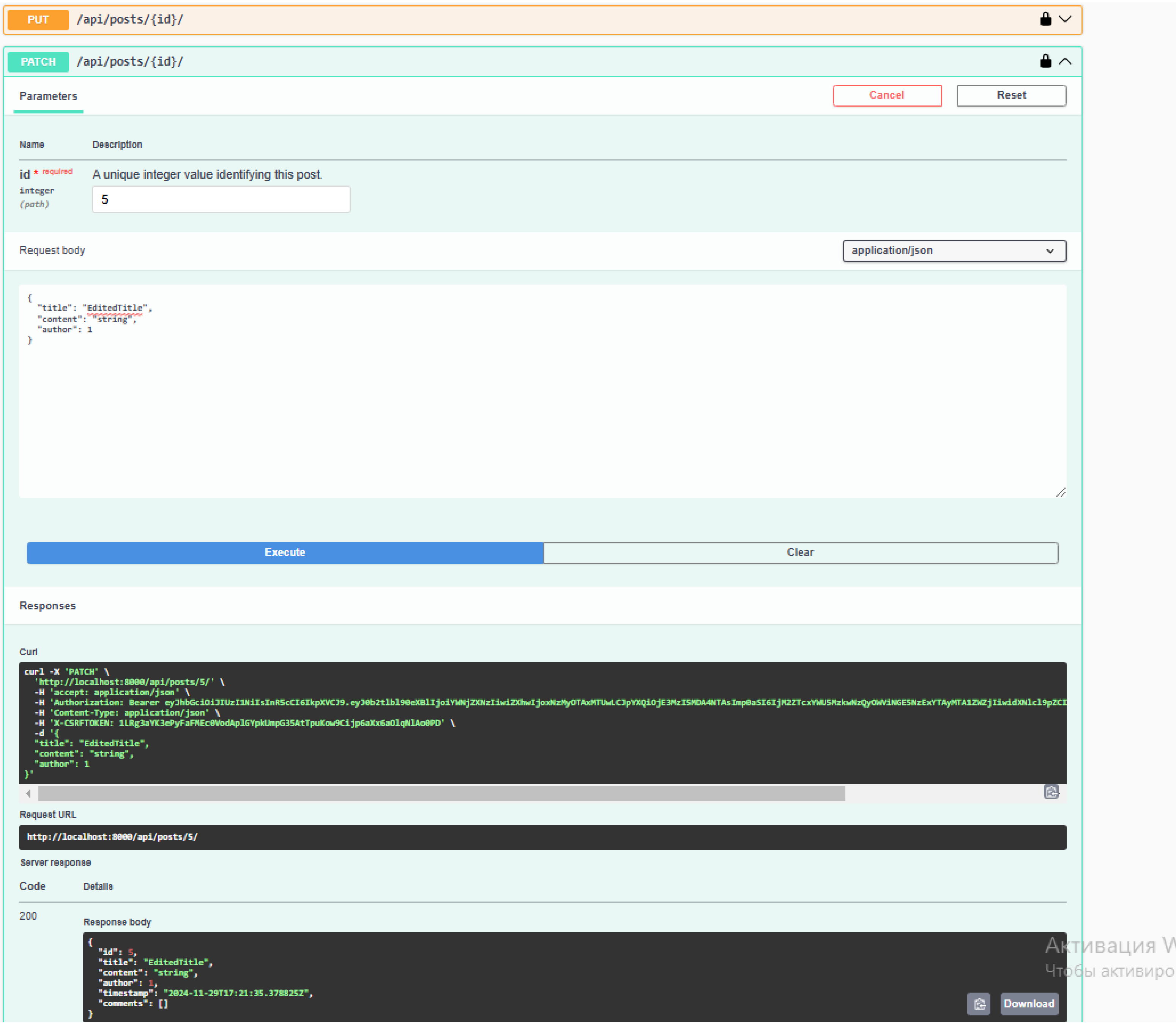
```
{
  "title": "New post",
  "content": "Dollar is growing",
  "author": 1
}
```

```
curl -X 'POST' \
  'http://localhost:8000/api/posts/' \
  -H 'accept: application/json' \
  -H 'Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ0b2tlbl90eXBlIjoiYWNjZXNzIiwiZXhwIjoxNzMyOTAxMTUwLCJpYXQiOjE3MzI5MDA4NTAsImp0aSI6IjM2ZTcxYWU5MzkwNzQyOWViNGE5NzExYTAyMTA1ZWZjIiwidXNlcl9pZCI
  -H 'Content-Type: application/json' \
  -H 'X-CSRFTOKEN: 1LRg3aYK3ePyFaFMEc0VodAplGYpkUmpG35AtTpuKow9Cijp6aXx6aOlqNlAo0PD' \
  -d '{
  "title": "New post",
  "content": "Dollar is growing",
  "author": 1
}'
```

```
http://localhost:8000/api/posts/
```

```
{
  "id": 5,
  "title": "New post",
  "content": "Dollar is growing",
  "author": 1,
```

Post request for comments



| Name | Description |
|---|---|
| id * required integer (path) | A unique integer value identifying this post. 5 |

Request body application/json

```
{
  "title": "EditedTitle",
  "content": "string",
  "author": 1
}
```

Responses

Curl

```
curl -X 'PATCH' \
  'http://localhost:8000/api/posts/5/' \
  -H 'accept: application/json' \
  -H 'Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ0b2tlbl90eXBlIjoiYWNjZXNzIiwiZXhwIjoxNzMyOTAxMTUwLCJpYXQiOjE3MzI5MDA4NTAsImp0aSI6IjM2ZTcxYWU5MzkwMzQyOWViNGE5NzExYTAyMTA1ZWZjIiwidXNlcl9pZCI
  -H 'Content-Type: application/json' \
  -H 'X-CSRFTOKEN: 1LRg3aYK3ePyFaFMEc0VodAplGYpkUmpG35AtTpuKow9Cijp6aXx6aOlqNlAo8PD' \
  -d '{
  "title": "EditedTitle",
  "content": "string",
  "author": 1
}'
```

Request URL

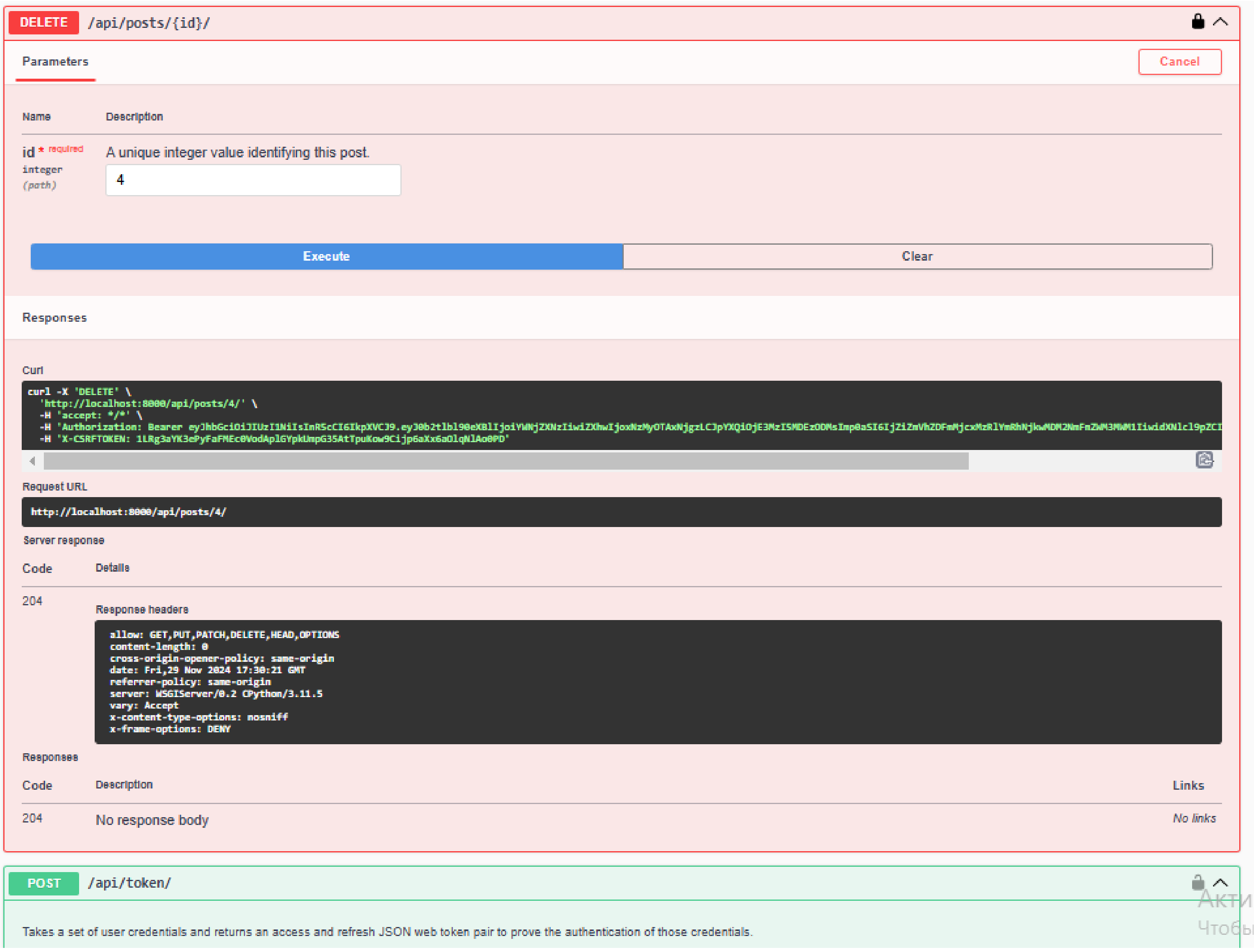
```
http://localhost:8000/api/posts/5/
```

Server response

| Code | Details |
|---|---|
| 200 | Response body |

```
{
  "id": 5,
  "title": "EditedTitle",
  "content": "string",
  "author": 1,
  "timestamp": "2024-11-29T17:21:35.378825Z",
  "comments": []
}
```

Patch request for comments


DELETE /api/posts/{id}/

Parameters

Cancel

| Name | Description |
|---|---|
| id * required<br>integer<br>(path) | A unique integer value identifying this post.<br>4 |

Execute

Clear

Responses

Curl

```
curl -X 'DELETE' \
  'http://localhost:8000/api/posts/4/' \
  -H 'accept: */*' \
  -H 'Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ0b2tlbl90eXBlIjoiYWNjZXNzIiwiZXhwIjoxNzMyOTAxNjgzLCJpYXQiOjE3MzI5MDEzODMsImp0aSI6IjZiZmVhZDFmMjcxMzRlYmRhNjkwMDM2NmFmZWM3MWM1IiwidXNlcl9pZCI
  -H 'X-CSRFTOKEN: 1LRg3aYK3ePyFaFMEc0VodAplGYpkUmpG35AtTpuKow9Cijp6aXx6aOlqNlAo0PD'
```

Request URL

```
http://localhost:8000/api/posts/4/
```

Server response

| Code | Details |
|---|---|
| 204 | Response headers |

```
allow: GET,PUT,PATCH,DELETE,HEAD,OPTIONS
content-length: 0
cross-origin-opener-policy: same-origin
date: Fri,29 Nov 2024 17:30:21 GMT
referrer-policy: same-origin
server: WSGIServer/0.2 CPython/3.11.5
vary: Accept
x-content-type-options: nosniff
x-frame-options: DENY
```

Responses

| Code | Description | Links |
|---|---|---|
| 204 | No response body | *No links* |

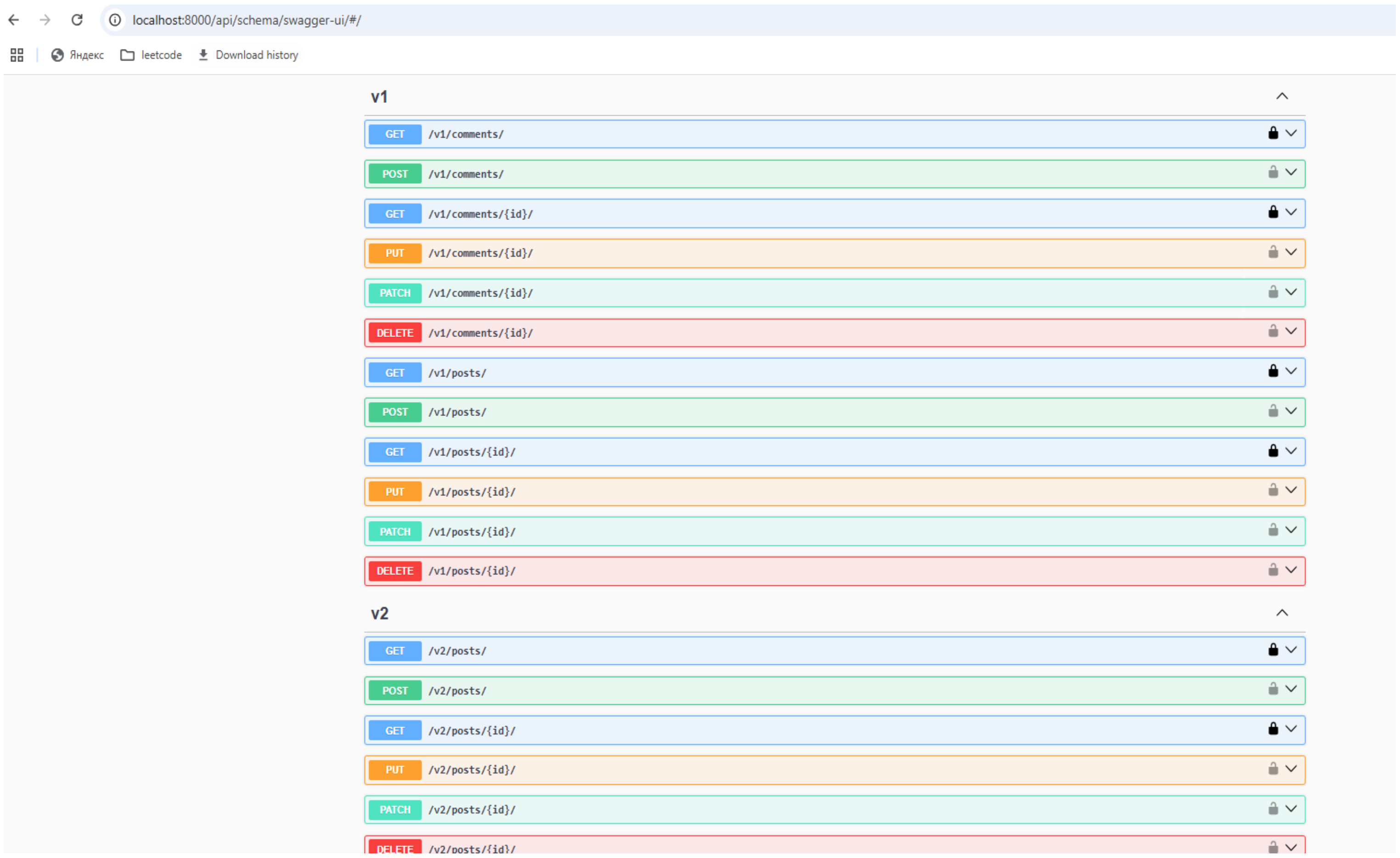
POST /api/token/

Takes a set of user credentials and returns an access and refresh JSON web token pair to prove the authentication of those credentials.

Delete request for comments

## Versioning

API versioning is set up using Django REST Framework's NamespaceVersioning. Each version (e.g., /api/v1/ and /api/v2/) is configured with separate URL namespaces and, if necessary, unique serializers or views. This ensures backward compatibility for older clients while allowing new features and updates in newer versions. Versioning is critical for managing API evolution, enabling developers to introduce changes without breaking existing implementations.



localhost:8000/api/schema/swagger-ui/#/

Яндекс | leetcode | Download history

v1

```
GET     /v1/comments/
POST    /v1/comments/
GET     /v1/comments/{id}/
PUT     /v1/comments/{id}/
PATCH   /v1/comments/{id}/
DELETE  /v1/comments/{id}/
GET     /v1/posts/
POST    /v1/posts/
GET     /v1/posts/{id}/
PUT     /v1/posts/{id}/
PATCH   /v1/posts/{id}/
DELETE  /v1/posts/{id}/
```

v2

```
GET     /v2/posts/
POST    /v2/posts/
GET     /v2/posts/{id}/
PUT     /v2/posts/{id}/
PATCH   /v2/posts/{id}/
DELETE  /v2/posts/{id}/
```

## Conclusion

The API effectively handles CRUD operations for blog posts and comments, providing a structured and secure interface for managing related data. Key features include the use of nested serializers for efficient data representation, versioning to ensure backward compatibility, and rate limiting to prevent misuse. JWT authentication and custom permissions safeguard sensitive operations.