KAZAKH-BRITISH TECHNICAL UNIVERSITY

KBTU

**Kazakh-British Technical University**

# Assignment 4

Web Application Development

Done by : Sarsengaliyev Zhaisan

Checked by: Serek Azamat

**Almaty, 2024**

## CONTENT

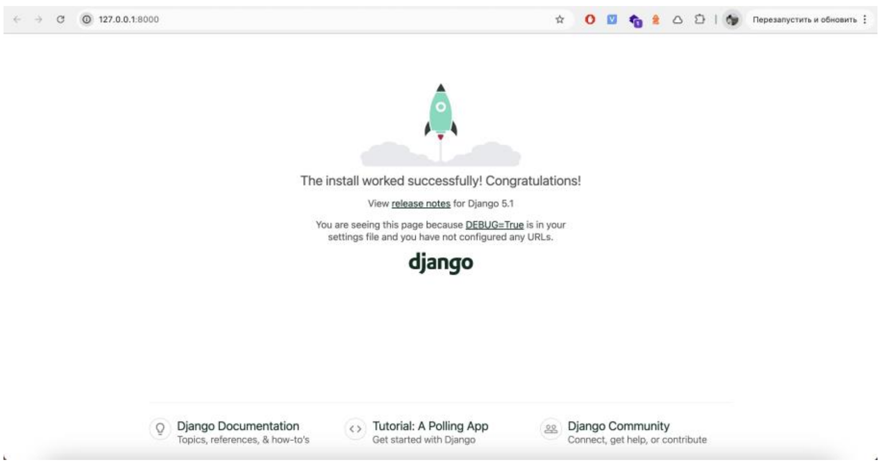
## INTRODUCTION

In this lab assignment I will learn how to build RESTful API using Django Rest Framework. I will create a simple application where users can create, read, update, and delete posts and comments. Also, I will cover concepts like data models, serializers, views, authentication, and deployment with Docker.

## Building a RESTful API with Django Rest Framework (DRF)

### Project Setup

To start project, I need to firstly create virtual environment. Then, installing Django using '*pip install django'* basic command as usual, then installing Django Rest Framework (DRF) using '*pip install djangorestframework'*. Next is starting project using '*django-admin startproject myproj*'. After all, running server using '*python manage.py runserver'*.



The install worked successfully! Congratulations!

View release notes for Django 5.1

You are seeing this page because DEBUG=True is in your settings file and you have not configured any URLs.

django

Django Documentation
Topics, references, & how-to's

Tutorial: A Polling App
Get started with Django

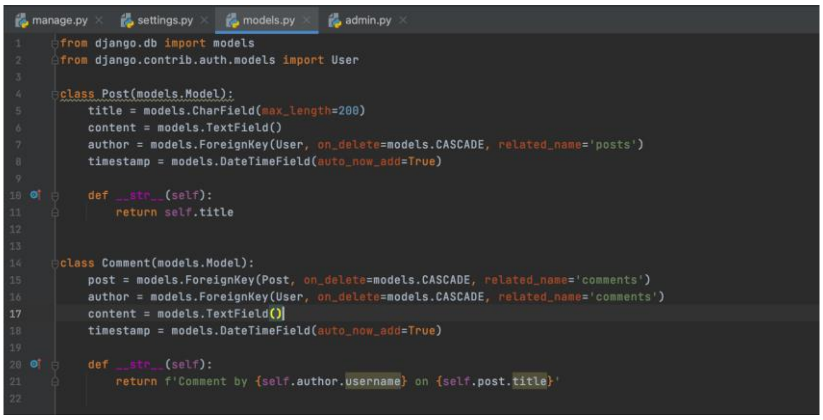
Django Community
Connect, get help, or contribute

Figure 1.1: Running server

In screen 1.1 I verified the setup by running the development server.

### Data Models

I created two models – Post and Comment. Post to represent blog posts, Comment for showing comments made on post.



```python
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, on_delete=models.CASCADE, related_name='posts')
    timestamp = models.DateTimeField(auto_now_add=True)

    def __str__(self):
        return self.title


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

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

Figure 1.2: Models

In screen 1.2 two models are illustrated. To begin with Post, '*title'* field is a char field to store the post title with max length 200 characters. '*content'* field is a text field for the main content of post, '*author'* is a foreign key to the user model setting many to one relationship. '*on_delete=models.CASCADE'* means that if the user is deleted, all their posts will be also deleted. '*related_name='posts''* allows to access a user posts via ‘*user.posts.all()*’. ‘*timestamp*’ is a date time field that records when the post was created. ‘*__str__*’ method returns a string type model which is helpful in admin interface. According to Comment model, everything is almost same.

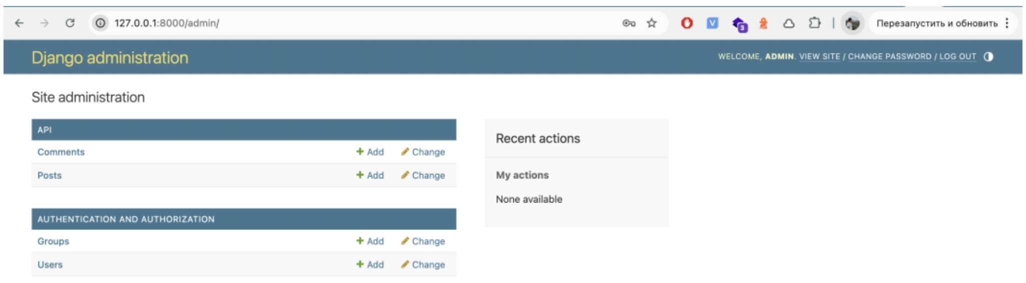
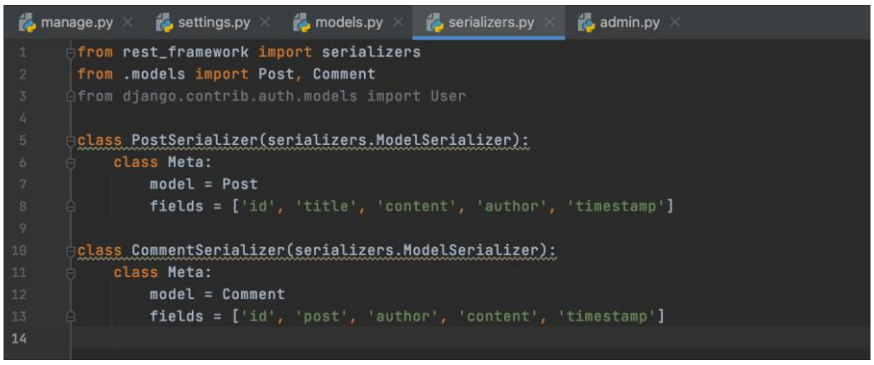


Figure 1.3: Admin panel

To set models in admin panel (1.3 figure), I registered models in admin.py using ‘*admin.site.register(Post)*’, ‘*admin.site.register(Comment)*’. But to access this panel I need to create super users setting it username and password, and only after this I can have access to admin panel.

## Serializers

In Django Rest Framework serializers are for converting complex data types like Django models to native python data types that can be easily converted into json, xml, or other content types. To do this, I created separate file for serializers.



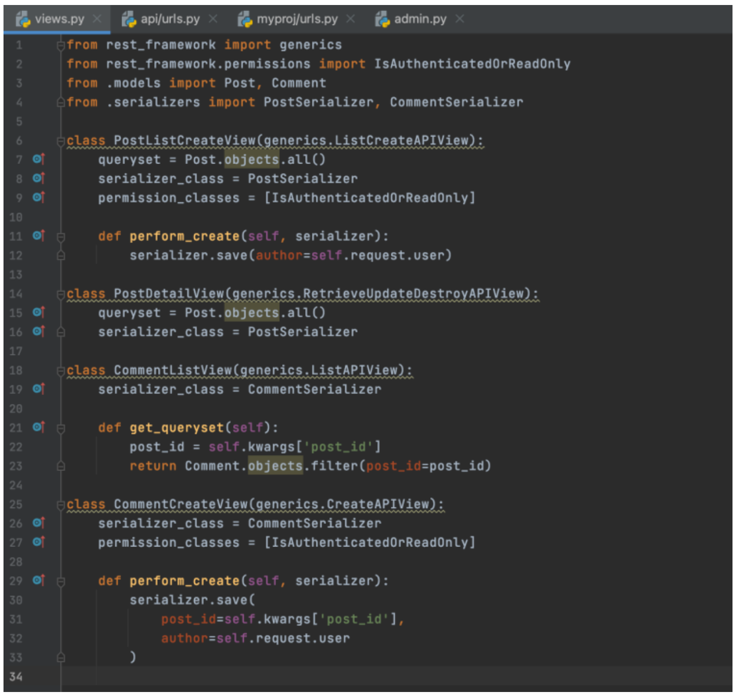
```
from rest_framework import serializers
from .models import Post, Comment
from django.contrib.auth.models import User

class PostSerializer(serializers.ModelSerializer):
    class Meta:
        model = Post
        fields = ['id', 'title', 'content', 'author', 'timestamp']

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

Figure 1.4: Serializers

In serializer file 1.4 screen I imported DRF’s modules for creating serializers. First class inherits from ‘serializers.ModelSerializer’ which generates fields based on the model. The same with second comment serializer. ‘model = Comment’ links the serializer to the comment model.

## Views and Endpoints

Here I started implementing views. ‘generics’ module provides generic class based views for common patterns.



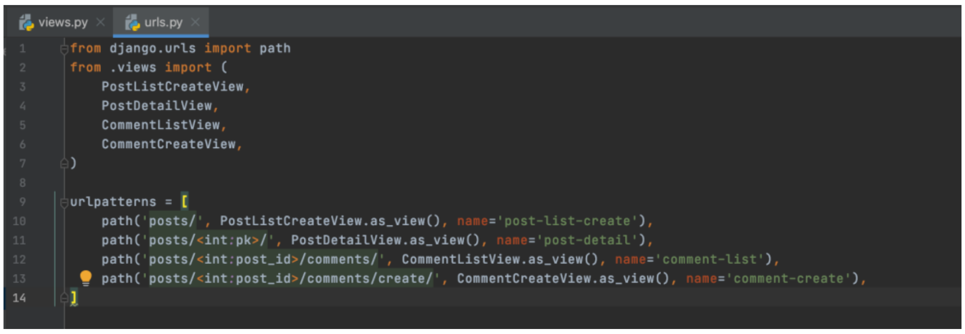
views.py api/urls.py myproj/urls.py admin.py

```python
from rest_framework import generics
from rest_framework.permissions import IsAuthenticatedOrReadOnly
from .models import Post, Comment
from .serializers import PostSerializer, CommentSerializer

class PostListCreateView(generics.ListCreateAPIView):
    queryset = Post.objects.all()
    serializer_class = PostSerializer
    permission_classes = [IsAuthenticatedOrReadOnly]

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

class PostDetailView(generics.RetrieveUpdateDestroyAPIView):
    queryset = Post.objects.all()
    serializer_class = PostSerializer

class CommentListView(generics.ListAPIView):
    serializer_class = CommentSerializer

    def get_queryset(self):
        post_id = self.kwargs['post_id']
        return Comment.objects.filter(post_id=post_id)

class CommentCreateView(generics.CreateAPIView):
    serializer_class = CommentSerializer
    permission_classes = [IsAuthenticatedOrReadOnly]

    def perform_create(self, serializer):
        serializer.save(
            post_id=self.kwargs['post_id'],
            author=self.request.user
        )
```

Figure 1.5: Views

Here ‘*queryset’* defines the list of objects that the view work with. ‘*serializer_class’* sets the serializer to use to validate and serialize input and output. ‘*perform_create’* overrides the method to customize how the instance is saved. Next view ‘*PostDetailView’* is for getting, updating, deleting the post. Next are views for comments. ‘*IsAuthenticatedOrReadOnly’* allows any user to read data, but only authenticated users can create and change the data.

**URL Routing**

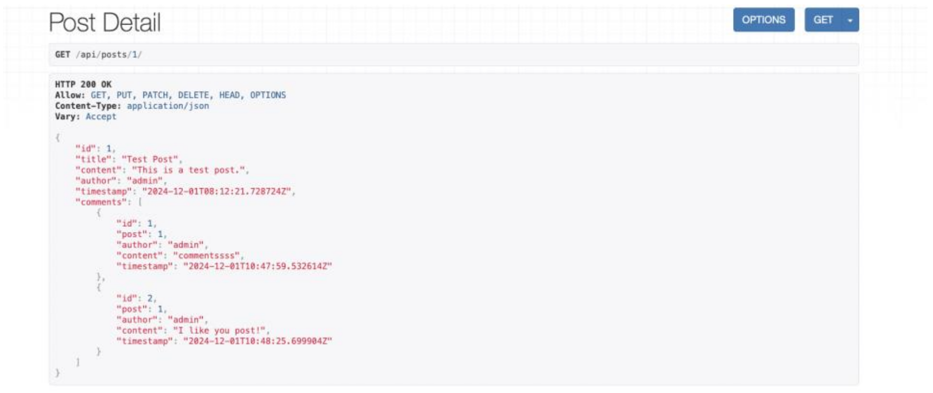


views.py urls.py

```python
from django.urls import path
from .views import (
    PostListCreateView,
    PostDetailView,
    CommentListView,
    CommentCreateView,
)

urlpatterns = [
    path('posts/', PostListCreateView.as_view(), name='post-list-create'),
    path('posts/<int:pk>/', PostDetailView.as_view(), name='post-detail'),
    path('posts/<int:post_id>/comments/', CommentListView.as_view(), name='comment-list'),
    path('posts/<int:post_id>/comments/create/', CommentCreateView.as_view(), name='comment-create'),
]
```

Figure 1.6: URLs

In this URLs file I mapped the views to URLs. First line of url patterns is for listing all posts and creating a new post. Next line is for getting, updating, deleting specific post. Third line is for listing comments of specific post. Fourth line is for creating comment for post.



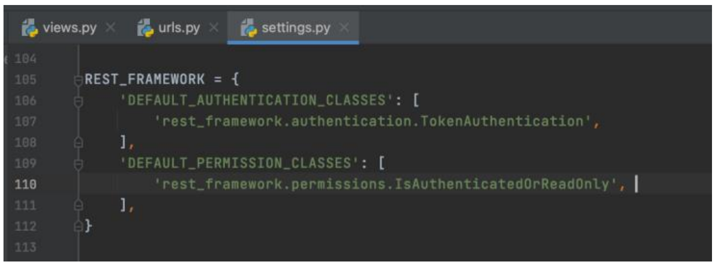
```
GET /api/posts/1/

HTTP 200 OK
Allow: GET, PUT, PATCH, DELETE, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

{
    "id": 1,
    "title": "Test Post",
    "content": "This is a test post.",
    "author": "admin",
    "timestamp": "2024-12-01T08:12:21.728724Z",
    "comments": [
        {
            "id": 1,
            "post": 1,
            "author": "admin",
            "content": "commentssss",
            "timestamp": "2024-12-01T10:47:59.532614Z"
        },
        {
            "id": 2,
            "post": 1,
            "author": "admin",
            "content": "I like you post!",
            "timestamp": "2024-12-01T10:48:25.699904Z"
        }
    ]
}
```

Figure 1.7: GET query

This 1.7 figure illustrates that get endpoint works as planned.

## Authentication and Permissions

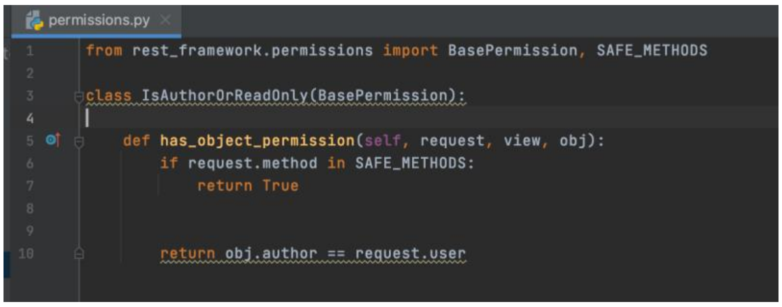
This section is about implementing token-based authentication. Also includes custom permissions to restrict access making sure that only authors can edit or delete own posts.



```python
REST_FRAMEWORK = {
    'DEFAULT_AUTHENTICATION_CLASSES': [
        'rest_framework.authentication.TokenAuthentication',
    ],
    'DEFAULT_PERMISSION_CLASSES': [
        'rest_framework.permissions.IsAuthenticatedOrReadOnly',
    ],
}
```

Figure 1.8: Configurations

'*DEFAULT_AUTHENTICATION_CLASSES'* defines authentication methods, '*TokenAuthentication*' enables token-based auth, '*DEFAULT_PERMISSION_CLASSES'* sets default permission policy.



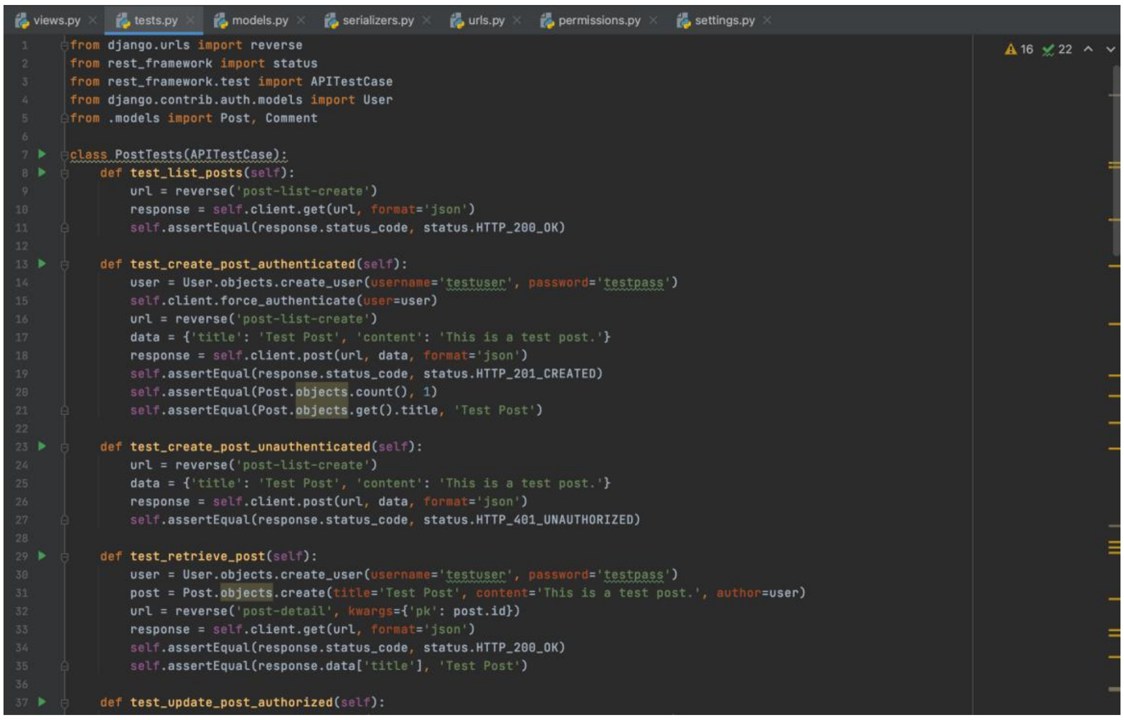
```python
from rest_framework.permissions import BasePermission, SAFE_METHODS

class IsAuthorOrReadOnly(BasePermission):

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


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

Figure 1.9: Permissions

This custom permission to allow only authors to edit or delete their posts.

## Unit Tests

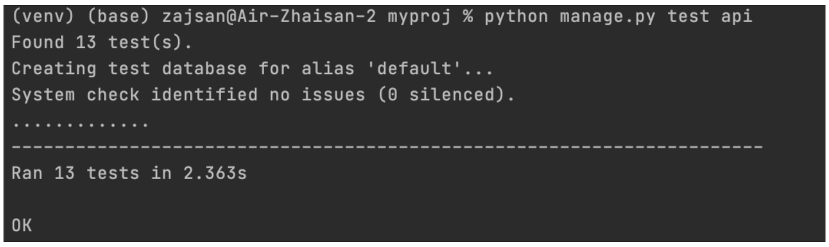
For tests I used Django's built-in '*TestCase*' class and the '*APIClient*' provided by DRF for testing.



```python
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

class PostTests(APITestCase):
    def test_list_posts(self):
        url = reverse('post-list-create')
        response = self.client.get(url, format='json')
        self.assertEqual(response.status_code, status.HTTP_200_OK)

    def test_create_post_authenticated(self):
        user = User.objects.create_user(username='testuser', password='testpass')
        self.client.force_authenticate(user=user)
        url = reverse('post-list-create')
        data = {'title': 'Test Post', 'content': 'This is a test post.'}
        response = self.client.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):
        url = reverse('post-list-create')
        data = {'title': 'Test Post', 'content': 'This is a test post.'}
        response = self.client.post(url, data, format='json')
        self.assertEqual(response.status_code, status.HTTP_401_UNAUTHORIZED)

    def test_retrieve_post(self):
        user = User.objects.create_user(username='testuser', password='testpass')
        post = Post.objects.create(title='Test Post', content='This is a test post.', author=user)
        url = reverse('post-detail', kwargs={'pk': post.id})
        response = self.client.get(url, format='json')
        self.assertEqual(response.status_code, status.HTTP_200_OK)
        self.assertEqual(response.data['title'], 'Test Post')

    def test_update_post_authorized(self):
```

Figure 1.10: Tests

In 1.10 screen '*reverse*' is function to get the URL for the 'post-list-create' endpoint. Sends get request to get all posts. Compares that response code is 200 OK. Next functions checks that only authenticated user can create a new post, and unauthenticated user cannot create a new post. So, in detail, this creates a test user, authenticates with '*force_authenticate*', sends a POST request to create a new post, compares the status code, and verifies that post was created in the database.



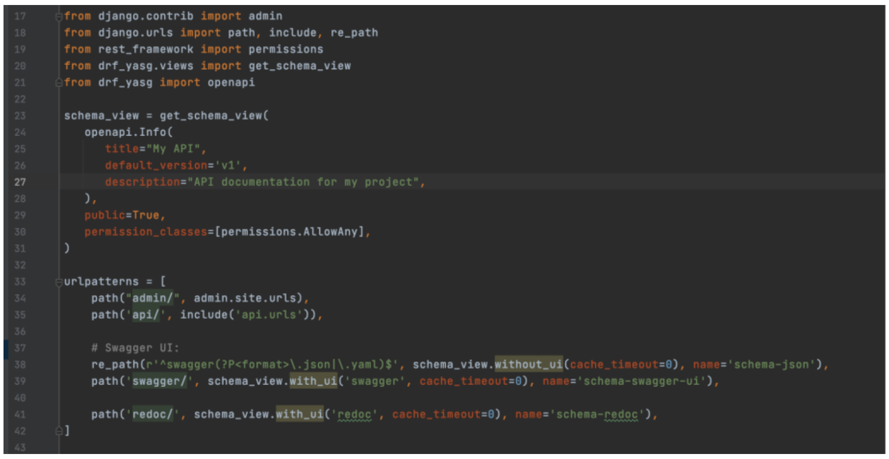
```
(venv) (base) zajsan@Air-Zhaisan-2 myproj % python manage.py test api
Found 13 test(s).
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
.............
----------------------------------------------------------------------
Ran 13 tests in 2.363s

OK
```

Figure 1.11: Test output

In this figure I see that unit tests executed successfully.

## API Documentation

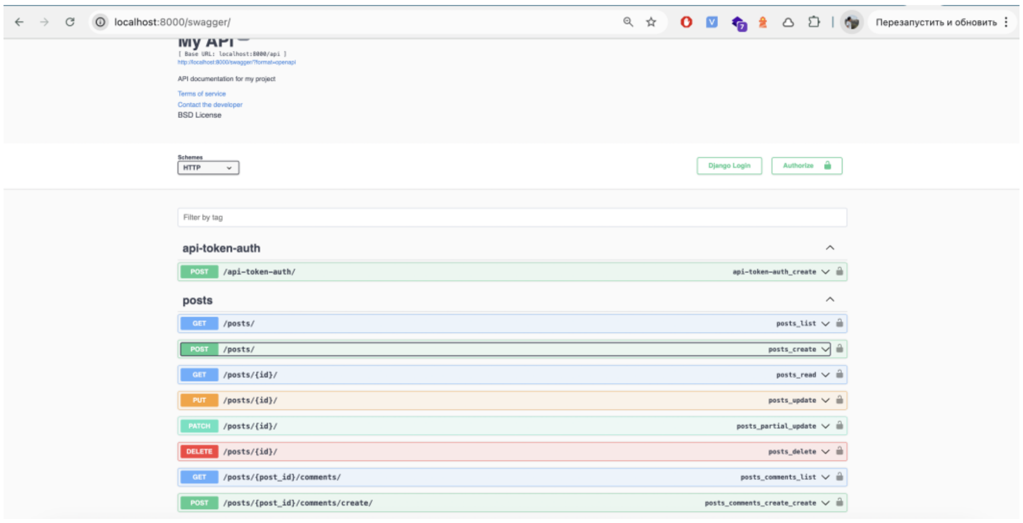
For documentation I use drf-yasg (Django REST Framework - Yet Another Swagger Generator) to generate interactive API doc. Drf-yasg provides swagger UI, also it provides authentication mechanisms, and customizations and extensions. I install it using ‘*pip install drf-yasg*’. Also, I don’t have to forget add it in settings, in installed apps section.



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

schema_view = get_schema_view(
   openapi.Info(
      title="My API",
      default_version='v1',
      description="API documentation for my project",
   ),
   public=True,
   permission_classes=[permissions.AllowAny],
)

urlpatterns = [
    path("admin/", admin.site.urls),
    path('api/', include('api.urls')),

    # Swagger UI:
    re_path(r'^swagger(?P<format>\.json|\.yaml)$', schema_view.without_ui(cache_timeout=0), name='schema-json'),
    path('swagger/', schema_view.with_ui('swagger', cache_timeout=0), name='schema-swagger-ui'),

    path('redoc/', schema_view.with_ui('redoc', cache_timeout=0), name='schema-redoc'),
]
```

Figure 1.12: Configuring documentation

In this 1.12 screen I configure the schema view that will generate the open API schema. ‘*openapi.Info’* contains metadata about API. ‘title’ is title of API, ‘*default_version’* is API version, ‘*description’* is a brief description. Then I added views to URL patterns.



localhost:8000/swagger/

Перезапустить и обновить

My API

[ Base URL: localhost:8000/api ]

http://localhost:8000/swagger/?format=openapi

API documentation for my project

Terms of service

Contact the developer

BSD License

Schemes HTTP

Django Login

Authorize

Filter by tag

api-token-auth

POST /api-token-auth/ api-token-auth_create

posts

GET /posts/ posts_list

POST /posts/ posts_create

GET /posts/{id}/ posts_read

PUT /posts/{id}/ posts_update

PATCH /posts/{id}/ posts_partial_update

DELETE /posts/{id}/ posts_delete

GET /posts/{post_id}/comments/ posts_comments_list

POST /posts/{post_id}/comments/create/ posts_comments_create_create

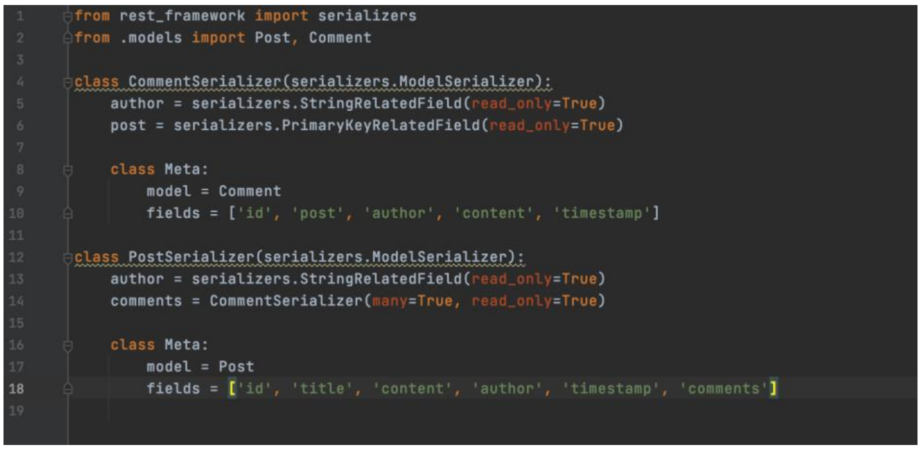
Figure 1.13: Swagger UI

In 1.13 figure is showing all API endpoints with parameters, request, response schemas.

## Advanced Features with Django Rest Framework

### Nested Serializers

Nested serializers allow related data to be included in a single API response making data usage more convenient for client side.



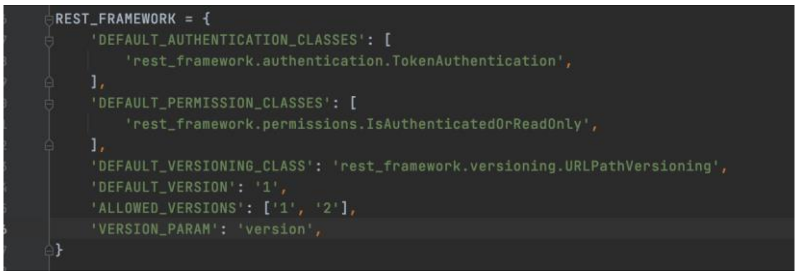
```python
from rest_framework import serializers
from .models import Post, Comment

class CommentSerializer(serializers.ModelSerializer):
    author = serializers.StringRelatedField(read_only=True)
    post = serializers.PrimaryKeyRelatedField(read_only=True)

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

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

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

Figure 2.1: Nested serializers

In the screen, '*author = serializers.StringRelatedField(read_only=True)*' displays the username of author instead of user id. *'post = serializers.PrimaryKeyRelatedField(read_only=True)'* makes sure that post field is read only and represented by primary key. '*comments = CommentSerializer(many=True, read_only=True)*' includes '*many=True'* which shows that there can be many comments.

### Versioning

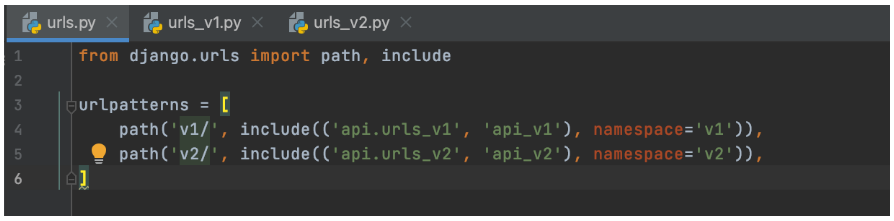
API versioning allows to manage changes in API over time without breaking existing clients. DRF has built-in support for versioning.



```python
REST_FRAMEWORK = {
    'DEFAULT_AUTHENTICATION_CLASSES': [
        'rest_framework.authentication.TokenAuthentication',
    ],
    'DEFAULT_PERMISSION_CLASSES': [
        'rest_framework.permissions.IsAuthenticatedOrReadOnly',
    ],
    'DEFAULT_VERSIONING_CLASS': 'rest_framework.versioning.URLPathVersioning',
    'DEFAULT_VERSION': '1',
    'ALLOWED_VERSIONS': ['1', '2'],
    'VERSION_PARAM': 'version',
}
```

Figure 2.2: Settings

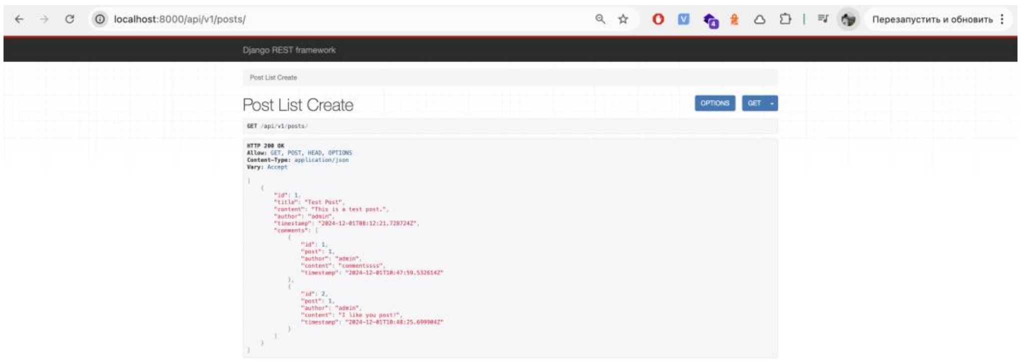
Here I specify versioning scheme using 'URLPathVersioning'.



```python
from django.urls import path, include

urlpatterns = [
    path('v1/', include(('api.urls_v1', 'api_v1'), namespace='v1')),
    path('v2/', include(('api.urls_v2', 'api_v2'), namespace='v2')),
]
```

Figure 2.3: Versions

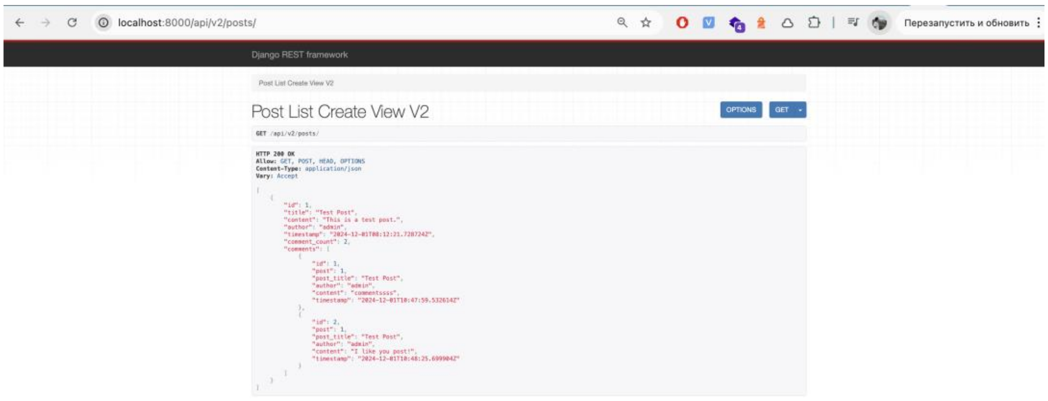
In figure 2.3 In URLs I separated the paths to two versions.



Post List Create

```
GET /api/v1/posts/

HTTP 200 OK
Allow: GET, POST, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

[
    {
        "id": 1,
        "title": "Test Post",
        "content": "This is a test post.",
        "author": "admin",
        "timestamp": "2024-12-01T08:12:21.728724Z",
        "comments": [
            {
                "id": 1,
                "post": 1,
                "author": "admin",
                "content": "commentssss",
                "timestamp": "2024-12-01T10:47:59.532614Z"
            },
            {
                "id": 2,
                "post": 1,
                "author": "admin",
                "content": "I like you post!",
                "timestamp": "2024-12-01T10:48:25.699904Z"
            }
        ]
    }
]
```

Figure 2.4: V1 endpoint



Post List Create View V2

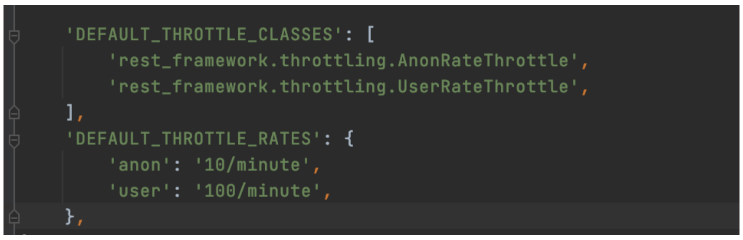
```
GET /api/v2/posts/

HTTP 200 OK
Allow: GET, POST, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

[
    {
        "id": 1,
        "title": "Test Post",
        "content": "This is a test post.",
        "author": "admin",
        "timestamp": "2024-12-01T08:12:21.728724Z",
        "comment_count": 2,
        "comments": [
            {
                "id": 1,
                "post": 1,
                "post_title": "Test Post",
                "author": "admin",
                "content": "commentssss",
                "timestamp": "2024-12-01T10:47:59.532614Z"
            },
            {
                "id": 2,
                "post": 1,
                "post_title": "Test Post",
                "author": "admin",
                "content": "I like you post!",
                "timestamp": "2024-12-01T10:48:25.699904Z"
            }
        ]
    }
]
```

Figure 2.5: V2 endpoint

In these two screenshots I made sure that our queries divided into two versions are working well.

## Rate Limiting

Throttling is a way to limit the number of requests that client can make to a backend within time frame. It helps prevent attack, manage server load. Django REST Framework (DRF) provides built-in feature for throttling.



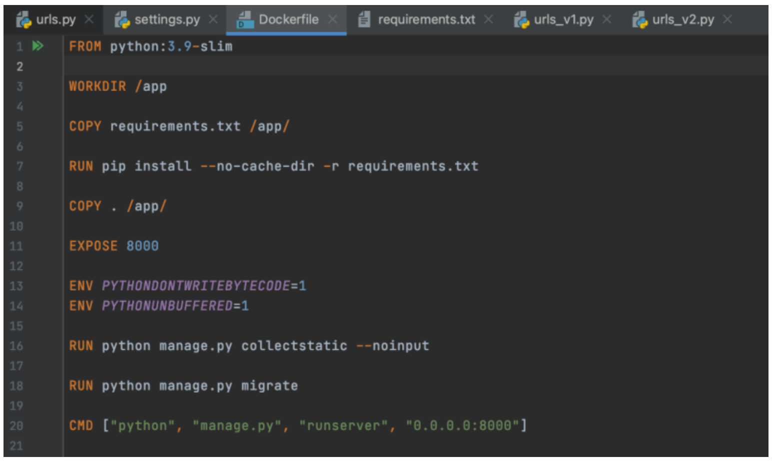
```
'DEFAULT_THROTTLE_CLASSES': [
    'rest_framework.throttling.AnonRateThrottle',
    'rest_framework.throttling.UserRateThrottle',
],
'DEFAULT_THROTTLE_RATES': {
    'anon': '10/minute',
    'user': '100/minute',
},
```

Figure 2.6: Settings for throttling

Here ‘AnonRateThrottle’ limits the rate of requests from anonymous (unauthenticated) users, ‘UserRateThrottle’ limits requests from authenticated users. Here for anonymous users I set limit to 10 requests per minute, for authenticated users limit is 100 requests per minute.

**Deployment**

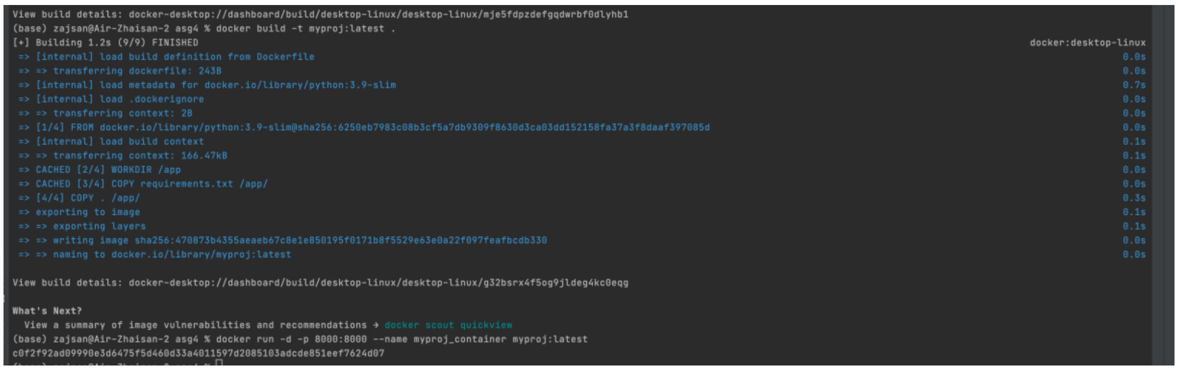
Containerizing the app with Docker ensures consistency in different environments and makes simple the deployment.



```dockerfile
FROM python:3.9-slim

WORKDIR /app

COPY requirements.txt /app/

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

COPY . /app/

EXPOSE 8000

ENV PYTHONDONTWRITEBYTECODE=1
ENV PYTHONUNBUFFERED=1

RUN python manage.py collectstatic --noinput

RUN python manage.py migrate

CMD ["python", "manage.py", "runserver", "0.0.0.0:8000"]
```

Figure 2.7 Dockerfile

In 2.7 my dockerfile to define the docker image. It included steps to install dependencies, collect static files, and run migrations.

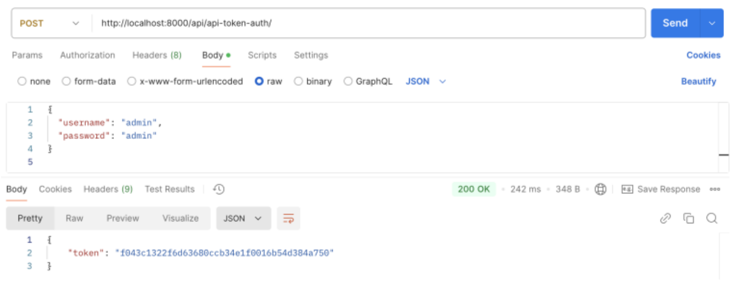


```
View build details: docker-desktop://dashboard/build/desktop-linux/desktop-linux/mje5fdpzdefgqdwrbf0dlyhb1
(base) zajsan@Air-Zhaisan-2 asg4 % docker build -t myproj:latest .
[+] Building 1.2s (9/9) FINISHED                                                                                       docker:desktop-linux
 => [internal] load build definition from Dockerfile                                                                                   0.0s
 => => transferring dockerfile: 243B                                                                                                   0.0s
 => [internal] load metadata for docker.io/library/python:3.9-slim                                                                     0.7s
 => [internal] load .dockerignore                                                                                                      0.0s
 => => transferring context: 2B                                                                                                        0.0s
 => [1/4] FROM docker.io/library/python:3.9-slim@sha256:6250eb7983c08b3cf5a7db9309f8630d3ca03dd152158fa37a3f8daaf397085d               0.0s
 => [internal] load build context                                                                                                      0.1s
 => => transferring context: 166.47kB                                                                                                  0.1s
 => CACHED [2/4] WORKDIR /app                                                                                                          0.0s
 => CACHED [3/4] COPY requirements.txt /app/                                                                                           0.0s
 => [4/4] COPY . /app/                                                                                                                 0.3s
 => exporting to image                                                                                                                 0.1s
 => => exporting layers                                                                                                                0.1s
 => => writing image sha256:470873b4355aeaeb67c8e1e850195f0171b8f5529e63e0a22f097feafbcdb330                                           0.0s
 => => naming to docker.io/library/myproj:latest                                                                                       0.0s

View build details: docker-desktop://dashboard/build/desktop-linux/desktop-linux/g32bsrx4f5og9jldeg4kc0eqg

What's Next?
  View a summary of image vulnerabilities and recommendations → docker scout quickview
(base) zajsan@Air-Zhaisan-2 asg4 % docker run -d -p 8000:8000 --name myproj_container myproj:latest
c0f2f92ad09990e3d6475f5d460d33a4011597d2085103adcde851eef7624d07
```

Figure 2.8: Terminal output

In 2.8 screen I made sure that our container is running.

**API Testing**



```
POST http://localhost:8000/api/api-token-auth/
```

Body: [✓ raw] JSON

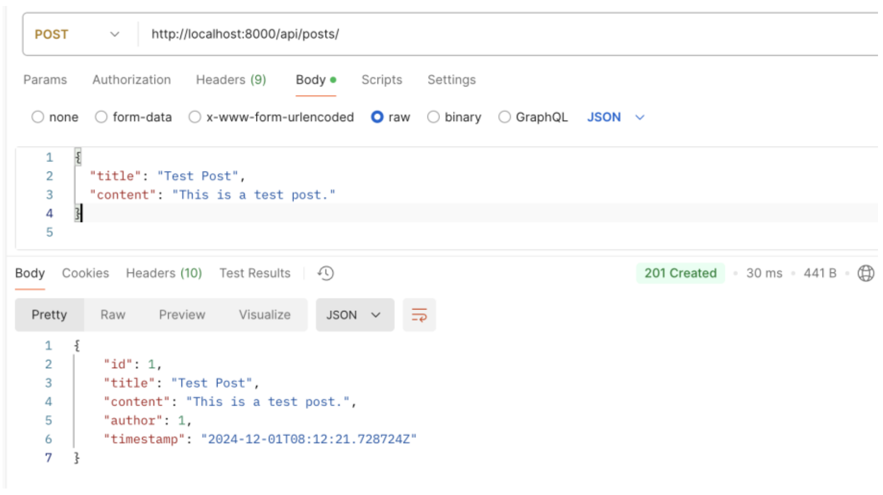
```json
{
  "username": "admin",
  "password": "admin"
}
```

200 OK · 242 ms · 348 B

```json
{
    "token": "f043c1322f6d63680ccb34e1f0016b54d384a750"
}
```

Figure 2.9: Get Token

This endpoint is used to get authentication token for registered user. Token is used to authenticate next requests to secure API endpoints.



POST http://localhost:8000/api/posts/

Body: [✓ raw] JSON

```
{
  "title": "Test Post",
  "content": "This is a test post."
}
```

201 Created · 30 ms · 441 B

```
{
    "id": 1,
    "title": "Test Post",
    "content": "This is a test post.",
    "author": 1,
    "timestamp": "2024-12-01T08:12:21.728724Z"
}
```

Figure 2.10: Create Post using token

This endpoint allows authenticated users to create post with content.

# CONCLUSION

In summary, I successfully built a RESTful API using Django Rest Framework. I created models for posts and comments and used serializers to convert model data to json format for the API. By implementing views and endpoints, I made clients to interact with the API securely adding authentication and permissions. I also added features like nested serializers, versioning and rate limiting, and prepared the application for deployment using Docker.