Kazakh British Technical University

Web Application Development

Assignment 4: Building a RESTful API with Django Rest Framework

by Moldir Polat

December 1st 2024

Executive summary

This report explains how Django Rest Framework (DRF) was used to build a RESTful API for a blog application. The project focused on creating an API that allows users to manage posts and comments. DRF made it easy to create and handle data using serializers and views.

The data models were designed to store posts and their related comments. Custom permissions were added to ensure only the author of a post could edit or delete it. Token-based authentication was used to secure the API, so only authenticated users could access certain features.

Advanced features like nested serializers were added to show comments directly within post details. API versioning was implemented to allow updates to the API without breaking older versions. Throttling was also used to limit how many requests users could make, improving the API's stability.

The application was containerized using Docker, and services were managed with docker-compose. PostgreSQL was used as the database for production, and the application was tested to ensure it worked correctly.

This project showed how DRF can be used to build a reliable and secure RESTful API. It also provided experience in deploying and testing APIs for real-world use.

Table of contents

Introduction

In modern web development, RESTful APIs have become the foundation of communication between client and server applications, enabling smooth data exchange and enhancing the scalability of software systems. A popular architectural style for creating APIs is Representational State Transfer (REST), which offers statelessness, simplicity, and the capacity to manipulate resources using conventional HTTP protocols. With the increasing demand for flexible and interoperable web applications, RESTful APIs are essential in helping developers create reliable, modular, and maintainable systems that support a variety of client platforms, such as web, mobile, and Internet of Things devices.

Django Rest Framework (DRF) is powerful tool for creating RESTful APIs in Python, that offers developers a comprehensive and modular framework for managing serialization, views, authentication, and more. DRF simplifies the process of designing and implementing RESTful APIs by providing a set of features and abstractions, such as class-based views, serializers, and built-in authentication mechanisms. By using DRF, developers can create efficient and secure APIs with minimal effort while adhering to industry standards.

The purpose of this report is to document the process of building a fully functional RESTful API using Django Rest Framework. It explores fundamental and advanced concepts, starting from setting up a Django project to implementing features like nested serializers, API versioning, rate limiting, and authentication. The scope of the report extends to cover the deployment of the API using Docker, emphasizing the importance of containerization in modern software development. This report can be as a practical guide for understanding and applying DRF's capabilities, demonstrating how to create scalable and secure APIs.

Building a RESTful API with Django Rest Framework

Exercise 1

Project setup.

To begin, I initialized a new Django project named blog_api and created an application called blog. This involved running the following commands in my terminal.

```
C:\Users\User\Desktop\KBTU-Moldir-Polat\3-semester\Web-app-dev\Assignment-4>django-admin startproject blog_api

C:\Users\User\Desktop\KBTU-Moldir-Polat\3-semester\Web-app-dev\Assignment-4>cd blog_api

C:\Users\User\Desktop\KBTU-Moldir-Polat\3-semester\Web-app-dev\Assignment-4\blog_api>python manage.py startapp blog
```

Next, I installed Django Rest Framework (DRF) to enable API creation. I added 'rest_framework' to the INSTALLED_APPS section of the settings.py file to integrate DRF into my project. Here is the updated section in settings.py:

```
32
33  INSTALLED_APPS = [
34      'django.contrib.admin',
35      'django.contrib.auth',
36      'django.contrib.contenttypes',
37      'django.contrib.sessions',
38      'django.contrib.messages',
39      'django.contrib.staticfiles',
40      'rest_framework',
41      'blog',
42  ]
43
```

Data Models.

I designed two models for this application: Post and Comment. The Post model represents a blog post and includes fields for the title, content, author, and timestamp. The Comment model represents comments associated with posts and includes fields for content, author, timestamp, and a ForeignKey relationship to the Post model. The models were defined in models.py as follows:

```
blog > models.py > ...
1   from django.db import models
2
3   class Post(models.Model):
4       title = models.CharField(max_length=100)
5       content = models.TextField()
6       author = models.CharField(max_length=50)
7       timestamp = models.DateTimeField(auto_now_add=True)
8
9       def __str__(self):
10          return self.title
11
12
13  class Comment(models.Model):
14      post = models.ForeignKey(Post, related_name='comments', on_delete=models.CASCADE)
15      content = models.TextField()
16      author = models.CharField(max_length=50)
17      timestamp = models.DateTimeField(auto_now_add=True)
18
19      def __str__(self):
20          return f"Comment by {self.author}"
21
22      #Moldir Polat
```

Serializers.

I implemented serializers for the Post and Comment models using serializers.ModelSerializer. These serializers handle the conversion of model instances into JSON format and vice versa. The serializers.py file contains the following code:

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

class CommentSerializer(serializers.ModelSerializer):
    class Meta:
        model = Comment
        fields = '__all__'


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

    class Meta:
        model = Post
        fields = '__all__'
#Moldir Polat
```

The PostSerializer includes a nested representation of related comments for each post.

Views and Endpoints.

I created views using DRF's viewsets.ModelViewSet to handle CRUD operations for both Post and Comment models. Here is the code in views.py:

```python
from rest_framework.permissions import IsAuthenticated
from .models import Post, Comment
from .serializers import PostSerializer, CommentSerializer
from .permissions import IsAuthorOrReadOnly  # Custom permission

class PostViewSet(viewsets.ModelViewSet):
    queryset = Post.objects.all().order_by('-timestamp')
    serializer_class = PostSerializer

    # Apply permissions
    permission_classes = [IsAuthenticated, IsAuthorOrReadOnly]


class CommentViewSet(viewsets.ModelViewSet):
    queryset = Comment.objects.all()
    serializer_class = CommentSerializer

    # Only authenticated users can access comments
    permission_classes = [IsAuthenticated]

#Moldir Polat
```

The following endpoints were implemented:

1. GET /posts/ – List all posts
2. POST /posts/ – Create a new post
3. GET /posts/{id}/ – Retrieve a single post
4. PUT /posts/{id}/ – Update a post
5. DELETE /posts/{id}/ – Delete a post
6. GET /comments/ – List all comments
7. POST /comments/ – Create a comment for a post

URL Routing.

I set up URL routing using DRF's DefaultRouter. The URLs were configured in urls.py in the screenshot below. This approach ensures that all endpoints for Post and Comment are accessible.

```
blog > urls.py > ...
from django.urls import path, include
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)),
]
#Moldir Polat
```

Obtain a Token: To enable token retrieval for users, I added the obtain_auth_token endpoint to urls.py. Users can send a POST request with their username and password to /api-token-auth/ to receive their token.

```
blog > urls.py > ...
from django.urls import path, include
from rest_framework.routers import DefaultRouter
from .views import PostViewSet, CommentViewSet
from rest_framework.authtoken.views import obtain_auth_token

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

urlpatterns = [
    path('', include(router.urls)),
    path('api-token-auth/', obtain_auth_token, name='api_token_auth'),
]
#Moldir Polat
```

Authentication and Permissions.

I implemented token-based authentication using DRF's TokenAuthentication. After installing the necessary package (pip install djangorestframework-authtoken), I added 'rest_framework.authtoken' to INSTALLED_APPS and ran migrations.

In settings.py, I configured the default authentication classes:

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

I also created a custom permission to allow only the author of a post to edit it. Here is the permission class in permissions.py:

```
blog > permissions.py > ...
from rest_framework.permissions import BasePermission

class IsAuthorOrReadOnly(BasePermission):
    def has_object_permission(self, request, view, obj):
        if request.method in ['GET', 'HEAD', 'OPTIONS']:
            return True
        return obj.author == request.user.username
#Moldir Polat
```

This permission class was applied in the PostViewSet:

```python
class PostViewSet(viewsets.ModelViewSet):
    queryset = Post.objects.all()
    serializer_class = PostSerializer
    permission_classes = [IsAuthenticated]
#Moldir Polat
```

Testing API.

I wrote unit tests using Django's TestCase and Django REST Framework's APIClient. These tests were designed to validate the core functionalities of the API endpoints, focusing on authentication, permissions, and edge cases. The goal was to confirm that the API adhered to its expected behavior, including restricting unauthenticated access, allowing authenticated access, and enforcing custom permissions for editing and deleting posts.

First, I created a tests.py file in the blog app directory. In the setUp method, I initialized the test environment by creating two users and generating tokens for both of them using DRF's Token model. I also created a sample post authored by one of the users to test permissions effectively. Below is the setUp code:

```python
#Moldir Polat
class BlogPermissionsTestCase(TestCase):
    def setUp(self):
        # Create two users
        self.user1 = User.objects.create_user(username="user1", password="password1")
        self.user2 = User.objects.create_user(username="user2", password="password2")

        # Generate tokens for users
        self.token1 = Token.objects.create(user=self.user1)
        self.token2 = Token.objects.create(user=self.user2)

        # Create a post by user1
        self.post = Post.objects.create(
            title="Test Post",
            content="This is a test post.",
            author="user1",
        )

        # Initialize API client
        self.client = APIClient()
```

I tested unauthenticated access by attempting to retrieve the list of posts using the GET /api/posts/ endpoint without providing any authentication token. The expected behavior was for the API to return a 401 Unauthorized status. The test confirmed this behavior:

```python
    def test_unauthenticated_access(self):
        # Attempt to get the list of posts without authentication
        response = self.client.get('/posts/')
        self.assertEqual(response.status_code, status.HTTP_401_UNAUTHORIZED)
```

To test authenticated access, I added the Authorization header with a valid token for user1 using APIClient.credentials(). I then called the GET /api/posts/ endpoint. The API returned a 200 OK status, confirming that authenticated users could successfully access the endpoint:

```python
    def test_authenticated_user_access(self):
        # Authenticate as user1 using token
        self.client.credentials(HTTP_AUTHORIZATION=f'Token {self.token1.key}')
        response = self.client.get('/posts/')
        self.assertEqual(response.status_code, status.HTTP_200_OK)
```

I tested whether only the author of a post could edit it. For this, I sent a PUT request to PUT /api/posts/{id}/ using both user2's and user1's tokens. When user2 tried to edit the post, the API returned a 403 Forbidden status. When the request was made with user1's token, the post was successfully updated with a 200 OK status.

```python
    def test_only_author_can_edit_post(self):
        # Authenticate as user2 (not the author)
        self.client.credentials(HTTP_AUTHORIZATION=f'Token {self.token2.key}')
        response = self.client.put(f'/posts/{self.post.id}/', {
            "title": "Updated Title",
            "content": "Updated Content",
            "author": "user1"
        }, format='json')
        self.assertEqual(response.status_code, status.HTTP_403_FORBIDDEN)

        # Authenticate as user1 (author)
        self.client.credentials(HTTP_AUTHORIZATION=f'Token {self.token1.key}')
        response = self.client.put(f'/posts/{self.post.id}/', {
            "title": "Updated Title",
            "content": "Updated Content",
            "author": "user1"
        }, format='json')
        self.assertEqual(response.status_code, status.HTTP_200_OK)
```

Finally, I tested the DELETE /api/posts/{id}/ endpoint to ensure only the author of the post could delete it. Using user2's token resulted in a 403 Forbidden status, while the post was successfully deleted with a 204 No Content status when using user1's token:

```python
    def test_only_author_can_delete_post(self):
        # Authenticate as user2 (not the author)
        self.client.credentials(HTTP_AUTHORIZATION=f'Token {self.token2.key}')
        response = self.client.delete(f'/posts/{self.post.id}/')
        self.assertEqual(response.status_code, status.HTTP_403_FORBIDDEN)

        # Authenticate as user1 (author)
        self.client.credentials(HTTP_AUTHORIZATION=f'Token {self.token1.key}')
        response = self.client.delete(f'/posts/{self.post.id}/')
        self.assertEqual(response.status_code, status.HTTP_204_NO_CONTENT)
#Moldir Polat
```

I executed all the tests using Django's test runner with the following command:

python manage.py test

The terminal output confirmed that all test cases passed successfully:

```
C:\Users\User\Desktop\KBTU-Moldir-Polat\3-semester\Web-app-dev\Assignment-4\blog_api>python manage.py test
Found 4 test(s).
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
....
----------------------------------------------------------------------
Ran 4 tests in 3.782s

OK
Destroying test database for alias 'default'...
```

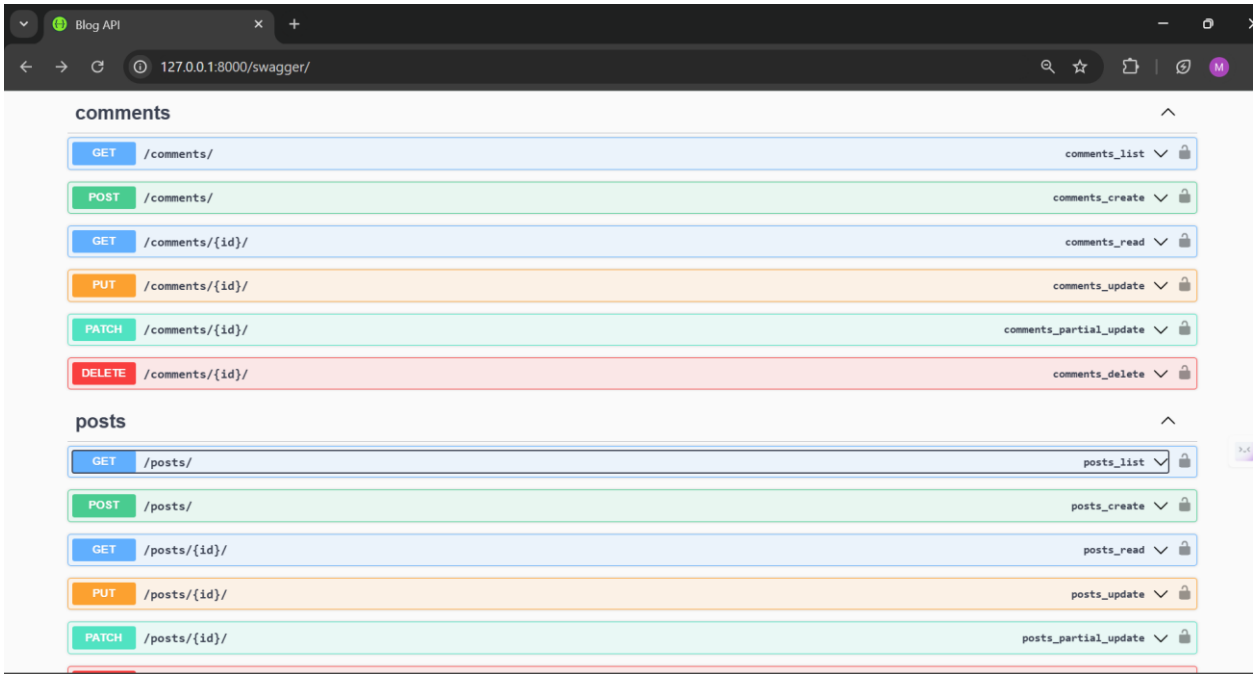
API Documentation

I installed drf-yasg and configured the schema view in the urls.py file. The schema view includes a title ("Blog API"), a description ("API for managing blog posts and comments"), and a version

(v1). The Swagger documentation is accessible via the /swagger/ endpoint.

```python
#Moldir Polat
schema_view = get_schema_view(
    openapi.Info(
        title="Blog API",
        default_version='v1',
        description="API for managing blog posts and comments",
    ),
    public=True,
    permission_classes=[AllowAny],  # Make the Swagger documentation publicly accessible
)

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('blog.urls')),  # Prefix all app routes with 'api/'
     path('swagger/', schema_view.with_ui('swagger', cache_timeout=0), name='schema-swagger-ui'),
]

```

Here the page captures the Swagger documentation UI, showing all endpoints and their details.



Advanced Features with Django Rest Framework

Exercise 2

This part documents the implementation of advanced features in the Django REST Framework (DRF) to enhance the functionality and usability of the API. These features include nested serializers, API versioning, rate limiting, optional WebSocket integration, and deployment preparation.

Nested Serializers

To improve the API's structure and usability, I implemented nested serializers to include comments within the responses for posts. This ensures that when a post is retrieved, its related comments are embedded in the response, providing a more comprehensive representation of the data.

I modified the serializers to include a nested structure. A new CommentSerializer was created to serialize individual comments, and this was used within the PostSerializer to include related comments.

The updated serializers.py:

```
blog > serializers.py > ...
1   from rest_framework import serializers
2   from .models import Post, Comment
3
4   class CommentSerializer(serializers.ModelSerializer):
5       class Meta:
6           model = Comment
7           fields = ['id', 'content', 'author', 'timestamp']
8
9   class PostSerializer(serializers.ModelSerializer):
10      comments = CommentSerializer(many=True, read_only=True)
11
12      class Meta:
13          model = Post
14          fields = ['id', 'title', 'content', 'author', 'timestamp', 'comments']
15  #Moldir Polat
```

The comments field in the PostSerializer retrieves all related comments using the reverse relation (related_name='comments') defined in the Comment model.

I tested the functionality by retrieving a post and ensuring its response included all related comments:

```
C:\Users\User\Desktop\KBTU-Moldir-Polat\3-semester\Web-app-dev\Assignment-4\blog_api>curl -H "Authorization: Token dd04b
cce3ff4a637a15d02cad2d6e5296417f441" http://127.0.0.1:8000/posts/1/
{"id":1,"title":"Post 1","content":"Content 1","author":"molya","timestamp":"2024-11-30T15:18:57.015111Z","comments":[{"
id":1,"content":"Comment 1","author":"User 1","timestamp":"2024-11-30T15:19:19.431044Z"}]}
C:\Users\User\Desktop\KBTU-Moldir-Polat\3-semester\Web-app-dev\Assignment-4\blog_api>
```

This approach reduces the need for multiple API calls, as all related data is retrieved in a single request. It improves efficiency and makes the API more user-friendly for clients.

API Versioning

To support evolving requirements, I implemented API versioning using DRF's built-in support. This ensures backward compatibility for existing clients while enabling new features in future versions.

I configured URL-based versioning in settings.py:

```python
REST_FRAMEWORK = {
    'DEFAULT_VERSIONING_CLASS': 'rest_framework.versioning.URLPathVersioning',

    'DEFAULT_AUTHENTICATION_CLASSES': [
        'rest_framework.authentication.TokenAuthentication',
    ],

    'DEFAULT_PERMISSION_CLASSES': [
        'rest_framework.permissions.IsAuthenticated',
    ],
}
```

Updated urls.py to include versioned routes:

```python
urlpatterns = [
    path('admin/', admin.site.urls),
    path('api/v1/', include('blog.urls')),
    path('api/v2/', include('blog.v2_urls')),
    path('api-token-auth/', obtain_auth_token, name='api_token_auth'),
    path('swagger/', schema_view.with_ui('swagger', cache_timeout=0), name='schema-swagger-ui'),
]
```

Created a new v2_urls.py for version 2 of the API. I also updated serializers and views for version-specific features.

I tested both versions by calling endpoints under /api/v1/ and /api/v2/:

```
C:\Users\User\Desktop\KBTU-Moldir-Polat\3-semester\Web-app-dev\Assignment-4\blog_api>curl -H "Authorization: Token dd04bcce3ff4a637a15d02cad2d6e5296417f441" http://127.0.0.1:8000/api/v1/posts/
[{"id":3,"title":"Post 3","content":"Content 3","author":"Molya","timestamp":"2024-11-30T15:37:44.548311Z","comments":[]},{"id":2,"title":"Post 2","content":"Content 2","author":"User 1","timestamp":"2024-11-30T15:19:09.039693Z","comments":[{"id":2,"content":"Comment 2","author":"Molya","timestamp":"2024-11-30T15:19:27.681793Z"}]},{"id":1,"title":"Post 1","content":"Content 1","author":"molya","timestamp":"2024-11-30T15:18:57.015111Z","comments":[{"id":1,"content":"Comment 1","author":"User 1","timestamp":"2024-11-30T15:19:19.431044Z"}]}]
C:\Users\User\Desktop\KBTU-Moldir-Polat\3-semester\Web-app-dev\Assignment-4\blog_api>curl -H "Authorization: Token dd04bcce3ff4a637a15d02cad2d6e5296417f441" http://127.0.0.1:8000/api/v2/posts/
[{"id":3,"title":"Post 3","content":"Content 3","author":"Molya","timestamp":"2024-11-30T15:37:44.548311Z","comments":[],"total_comments":0},{"id":2,"title":"Post 2","content":"Content 2","author":"User 1","timestamp":"2024-11-30T15:19:09.039693Z","comments":[{"id":2,"content":"Comment 2","author":"Molya","timestamp":"2024-11-30T15:19:27.681793Z","comment_length":9}],"total_comments":1},{"id":1,"title":"Post 1","content":"Content 1","author":"molya","timestamp":"2024-11-30T15:18:57.015111Z","comments":[{"id":1,"content":"Comment 1","author":"User 1","timestamp":"2024-11-30T15:19:19.431044Z","comment_length":9}],"total_comments":1}]
```

API versioning ensures stability for clients using older versions while allowing continuous improvement. It provides a clear structure for managing breaking changes and introducing new features.

Rate Limiting

To protect the API from abuse and manage traffic, I implemented rate limiting using DRF's throttle classes. Different throttling rates were configured for authenticated and anonymous users.

I configured throttling in settings.py:

```python
    'DEFAULT_THROTTLE_CLASSES': [
        'rest_framework.throttling.AnonRateThrottle',
        'rest_framework.throttling.UserRateThrottle',
    ],
    'DEFAULT_THROTTLE_RATES': {
        'anon': '10/day',  # Anonymous users can make 10 requests per day
        'user': '1000/day',  # Authenticated users can make 1000 requests per day
    },
}
```

I tested the throttling behavior by making multiple requests as both an anonymous and authenticated user. After 11th request for anonymous user there wa a message: "{"detail":"Request was throttled. Expected available in 86379 seconds."}"

```
C:\Users\User\Desktop\KBTU-Moldir-Polat\3-semester\Web-app-dev\Assignment-4\blog_api>curl http://127.0.0.1:8000/api/v1/p
osts/
{"detail":"Request was throttled. Expected available in 86379 seconds."}
```

After 12th request (chnaged to 11 for testing) for authenticated user there was a message "{"detail":"Request was throttled. Expected available in 86379 seconds."}"

```
C:\Users\User\Desktop\KBTU-Moldir-Polat\3-semester\Web-app-dev\Assignment-4\blog_api>curl -H "Authorization: Token dd04b
cce3ff4a637a15d02cad2d6e5296417f441" http://127.0.0.1:8000/api/v1/posts/
{"detail":"Request was throttled. Expected available in 86392 seconds."}
```

Rate limiting helps mitigate denial-of-service (DoS) attacks and ensures fair usage of resources by limiting the number of requests a user can make within a specified period.

Deployment

I started by making the application production-ready. First, I updated the database settings in settings.py to use PostgreSQL. I replaced the environment variable configuration with hardcoded values for simplicity:

```python
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'my_database',
        'USER': 'my_user',
        'PASSWORD': 'my_password',
        'HOST': 'db',  # 'db' is the service name in Docker Compose
        'PORT': '5432',
    }
}
```

To containerize the Django application, I created a Dockerfile in the root directory of the project. This file specified the Python environment, installed dependencies, and configured the Gunicorn server to run the application:

Dockerfile

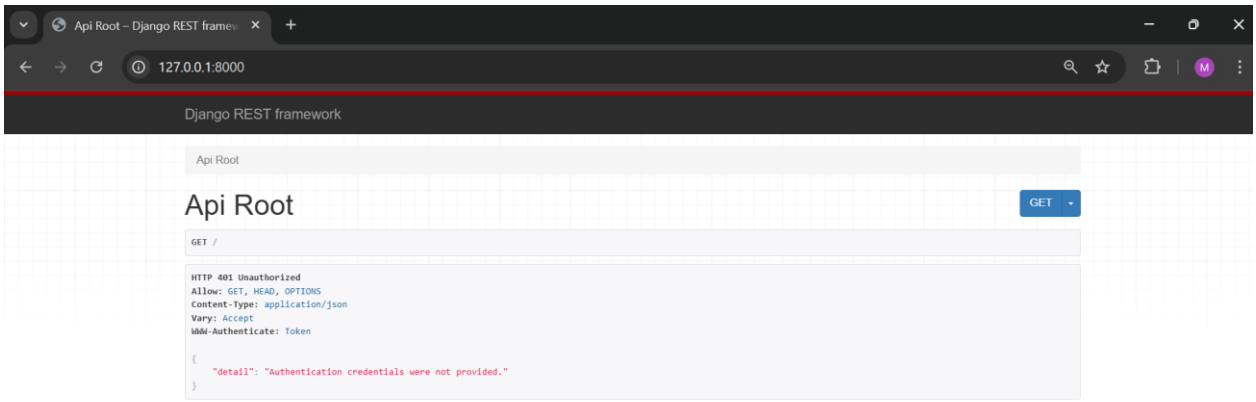
```
FROM python:3.9-slim

WORKDIR /app

COPY requirements.txt /app/
RUN pip install --no-cache-dir -r requirements.txt

COPY . /app/

EXPOSE 8000

CMD ["gunicorn", "--bind", "0.0.0.0:8000", "project_name.wsgi:application"]
```

Next, I created a docker-compose.yml file to define and manage multi-container deployment for the Django application and the PostgreSQL database.

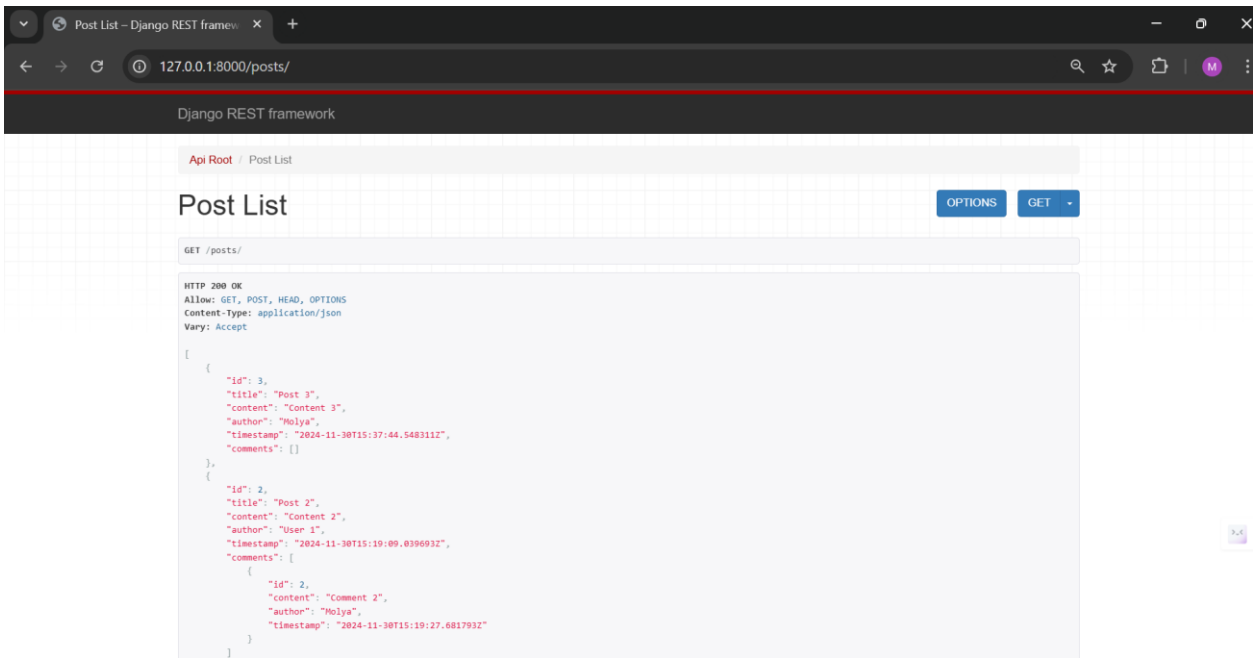
```yaml
services:
  web:
    build:
      context: .
    container_name: django_app
    command: gunicorn --bind 0.0.0.0:8000 project_name.wsgi:application
    ports:
      - "8000:8000"
    volumes:
      - .:/app
    depends_on:
      - db
    environment:
      - DB_NAME=my_database
      - DB_USER=my_user
      - DB_PASSWORD=my_password
      - DB_HOST=db
      - DB_PORT=5432

  db:
    image: postgres:13
    container_name: postgres_db
    environment:
      POSTGRES_DB: my_database
      POSTGRES_USER: my_user
      POSTGRES_PASSWORD: my_password
    ports:
```

This file described the web service for the Django application and the db service for PostgreSQL. The db service was configured to persist data using a Docker volume.

I tested the deployment to ensure everything was working:



I accessed http://127.0.0.1:8000 in a browser to confirm the application was running. I tested all API endpoints to verify that the application was responding correctly. I restarted the containers to confirm that database data persisted, ensuring the postgres_data volume was working as expected.


Post List – Django REST framew
127.0.0.1:8000/posts/
Django REST framework
Api Root / Post List
Post List
OPTIONS
GET
GET /posts/
HTTP 200 OK
Allow: GET, POST, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept
[
{
"id": 3,
"title": "Post 3",
"content": "Content 3",
"author": "Molya",
"timestamp": "2024-11-30T15:37:44.548311Z",
"comments": []
},
{
"id": 2,
"title": "Post 2",
"content": "Content 2",
"author": "User 1",
"timestamp": "2024-11-30T15:19:09.039693Z",
"comments": [
{
"id": 2,
"content": "Comment 2",
"author": "Molya",
"timestamp": "2024-11-30T15:19:27.681793Z"
}
]

Adding token-based authentication was another challenge. At first, the /api-token-auth/ endpoint returned a 404 Not Found error. After reviewing the configuration, I realized I had not added rest_framework.authtoken to INSTALLED_APPS and hadn't run the migrations for the authentication system. I fixed this by adding 'rest_framework.authtoken' to INSTALLED_APPS and running the migrations with python manage.py migrate.

This allowed me to generate tokens for users and authenticate API requests using headers. I tested the authentication by making requests with and without tokens, verifying that unauthenticated requests were blocked.

For Exercise 2, I faced challenges with implementing throttling and permissions together. Initially, the IsAuthenticated permission class blocked all anonymous requests, preventing me from testing throttling for anonymous users. To resolve this, I temporarily replaced IsAuthenticated with AllowAny in the PostViewSet and confirmed that throttling worked for anonymous users. Afterward, I restored the original permissions.

Setting up API versioning using URLPathVersioning was a challenge. While creating separate serializers and views for version 2, I mistakenly reused the version 1 serializers, causing inconsistencies in the responses. To address this, I created a v2_urls.py file and carefully mapped it to version-specific serializers and views. Testing endpoints under /api/v1/ and /api/v2/ confirmed that versioning worked as expected.

Conclusions

Through the implementation of a RESTful API using Django Rest Framework, I gained valuable lessons of building scalable and secure APIs. I learned how to design efficient data models, use serializers for structured data handling, and create endpoints for CRUD operations. Managing relationships, such as linking comments to posts, taught me the importance of thoughtful database design for real-world applications.

Implementing token-based authentication and custom permissions like IsAuthorOrReadOnly highlighted the significance of securing APIs and controlling access. I also learned how versioning ensures backward compatibility, and nested serializers enhance data presentation by structuring related information in a single API call.

Deploying the application using Docker and docker-compose reinforced the benefits of containerization for consistency across environments. Writing and running test cases helped me validate the functionality and reliability of the API while addressing edge cases.

Overall, this project provided good guide in designing, securing, and deploying RESTful APIs, equipping me with practical skills for real-world backend development.

References

https://docs.djangoproject.com/

https://www.django-rest-framework.org/

https://docs.docker.com/

https://www.postgresql.org/docs/

https://docs.python.org/3/

https://stackoverflow.com/

https://www.youtube.com/user/schafer5

https://chatgpt.com/ (used for issues resolution)