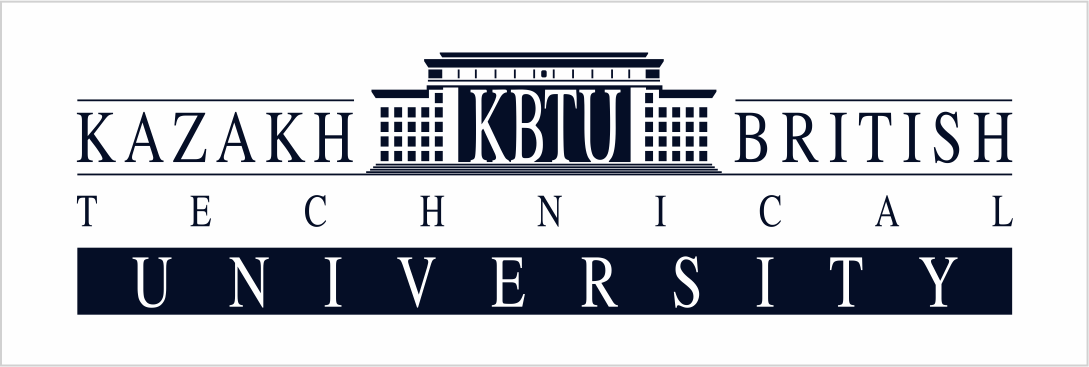
****

**Assignment 4**

**Building a RESTful API with Django rest framework.**

**Prepared by Azimkhanov Bauyrzhan**

**Almaty, 30.11.2024**

**Executive summary**

Key findings of Django RESTful API:

1. Ease of use: DRF simplifies the process of building RESTful APIs in Django. It provides a straightforward way to create serializers and views.
2. Serialization: One of the most crucial aspects of DRF is its serialization capability, converting complex data types like querysets and model instances to native Python datatypes that can then be easily rendered into JSON or XML.
3. Authentication and permissions: DRF offers robust authentication and permission classes that ensure secure API access.
4. Browsable API: DRF comes with a built-in, human-friendly web-browsable API interface that aids in development and testing.
5. Validation: DRF supports data validation through serializers, which helps maintain data integrity.

Implementations and utilities from Django RESTful API:

1. Serializers: Create serializers to transform Django models into JSON format and vice versa.
2. Views: Use function-based views (FBVs) or class-based views (CBVs) to define the logic for handling HTTP requests.
3. Routing: Define URL patterns for the API endpoints using Django’s routing system.
4. Authentication: Implement various authentication schemes like Token, Session, or OAuth for secure access.
5. Permissions: Set up permission classes to control who can access different parts of the API.
6. Testing: Write unit tests for serializers, views, and URL configurations to ensure the API works correctly.

With these findings and implementations, DRF helps simplify the creation of robust and secure RESTful APIs in Django, making it an good choice for developing backend services.

**Table of contents**

Executive summary 2

Table of contents 3

Introduction 4

Building a RESTful API with Django Rest Framework 5

Project setup 5

Data models 6

Serializers 7

Views and endpoints 8

URL routing 8

Authentication and permissions 9

Advanced features with Django Rest Framework 10

Nested serializers 10

Versioning 10

Rate limiting 10

Deployment 11

Testing and documentation 12

API testing 12

API documentation 14

Challenges and solutions 16

Conclusion 16

Recommendations 17

References 18

**Introduction**

In web development, RESTful APIs are important because they allow different systems to communicate with each other over the internet. They use standard methods like GET, POST, PUT, and DELETE, making it easier to interact with web services. RESTful APIs are flexible, scalable, and can be used with various programming languages.

The purpose of this report is to explain why RESTful APIs and Django Rest Framework are significant in web development. It will cover the benefits of using RESTful APIs, such as flexibility and scalability, and how DRF helps developers create and manage these APIs more efficiently. The scope includes an overview of key features of DRF, its role in web development, and practical examples of its use.

My main motivation for making a task manager using Docker and Django was the desire to master a new stack of relevant technologies and the desire to get a good grade at the end of the semester.

**Building a RESTful API with Django Rest Framework**

**Project Setup**

Initialize the project exactly as we did in all previous assignments on this subject.

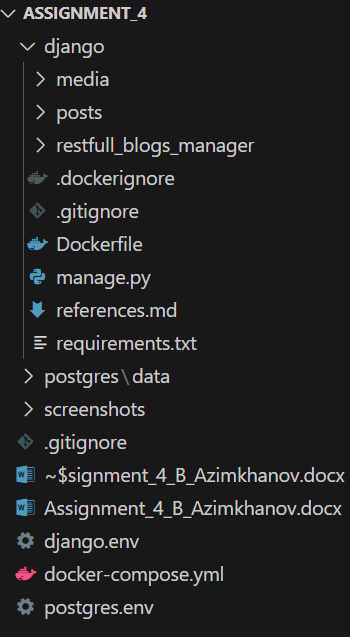


Figure 1. Project structure.

**Data models**

We create models in exactly the same way as we did in all previous assignments on this subject.

from django.db import models

class Category(models.Model):

    name = models.CharField(max\_length=200)

    description = models.TextField()

    def \_\_str\_\_(self):

        return self.name

    class Meta:

        verbose\_name = "Category"

        verbose\_name\_plural = "Categories"

class PostManager(models.Manager):

    def get\_published\_posts(self):

        return self.filter(published\_date\_\_isnull=False)

    def get\_posts\_from\_author(self, author):

        return self.filter(author=author)

class Post(models.Model):

    title = models.CharField(max\_length=200)

    content = models.TextField()

    author = models.CharField(max\_length=70)

    published\_date = models.DateTimeField(auto\_now\_add=True)

    category = models.ManyToManyField(Category)

    image = models.ImageField(upload\_to="post\_images/", blank=True, null=True)

    objects = PostManager()

    def \_\_str\_\_(self):

        return self.title

    class Meta:

        verbose\_name = "Post"

        verbose\_name\_plural = "Posts"

class Comment(models.Model):

    post = models.ForeignKey(Post, on\_delete=models.CASCADE)

    content = models.TextField()

    def \_\_str\_\_(self):

        return self.post.\_\_str\_\_() + " comment"

    class Meta:

        verbose\_name = "Comment"

        verbose\_name\_plural = "Comments"

Figure 2. models.py

**Serializers**

We create serializers using serializers from the rest\_framework library. These are standard serializers for JSON.

Only for the Post model we will override the create and update methods. This was necessary to complete the task and to fix the error associated with incorrect display of categories when calling the serializer of the Post model.

from rest\_framework import serializers

from .models import Category, Post, Comment

class CategorySerializer(serializers.ModelSerializer):

    class Meta:

        model = Category

        fields = ['id', 'name', 'description']

class CommentSerializer(serializers.ModelSerializer):

    post = serializers.PrimaryKeyRelatedField(queryset=Post.objects.all())

    class Meta:

        model = Comment

        fields = ['id', 'post', 'content']

class PostSerializer(serializers.ModelSerializer):

    category = serializers.PrimaryKeyRelatedField(queryset=Category.objects.all(), many=True)

    comments = CommentSerializer(many=True, read\_only=True, source='comment\_set')

    class Meta:

        model = Post

        fields = ['id', 'title', 'content', 'author', 'published\_date', 'category', 'image', 'comments']

    def create(self, validated\_data):

        categories = validated\_data.pop('category')

        post = Post.objects.create(\*\*validated\_data)

        post.category.set(categories)

        return post

    def update(self, instance, validated\_data):

        categories = validated\_data.pop('category', None)

        image = validated\_data.get('image')

        instance.title = validated\_data.get('title', instance.title)

        instance.content = validated\_data.get('content', instance.content)

        instance.author = validated\_data.get('author', instance.author)

        instance.published\_date = validated\_data.get('published\_date', instance.published\_date)

        instance.image = validated\_data.get('image', instance.image)

        if categories is not None:

            instance.category.set(categories)

        instance.save()

        return instance

Figure 3. serializers.py

**Views and endpoints**

We create views in the same way as in all previous tasks on this topic. Only this time we will apply the functionality of the Django RESTful framework library for display in the web interface.

from rest\_framework import viewsets

from rest\_framework.permissions import IsAuthenticated

from .models import Category, Post, Comment

from .serializers import CategorySerializer, PostSerializer, CommentSerializer

from .permissions import IsAuthorOrReadOnly

class CategoryViewSet(viewsets.ModelViewSet):

    queryset = Category.objects.all()

    serializer\_class = CategorySerializer

class PostViewSet(viewsets.ModelViewSet):

    queryset = Post.objects.all()

    serializer\_class = PostSerializer

    permission\_classes = [IsAuthenticated, IsAuthorOrReadOnly]

class CommentViewSet(viewsets.ModelViewSet):

    queryset = Comment.objects.all()

    serializer\_class = CommentSerializer

    permission\_classes = [IsAuthenticated, IsAuthorOrReadOnly]

Figure 4. views.py

**URL routing**

To create routing, we use the built-in functionality of the Django RESTful framework library and all the developments used from previous work on storing media files.

from django.urls import path, include

from rest\_framework.routers import DefaultRouter

from .views import CategoryViewSet, PostViewSet, CommentViewSet

from django.conf.urls.static import static

from django.conf import settings

router = DefaultRouter()

router.register(r'categories', CategoryViewSet)

router.register(r'posts', PostViewSet)

router.register(r'comments', CommentViewSet)

urlpatterns = [

    path('', include(router.urls)),

] + static(settings.MEDIA\_URL, document\_root=settings.MEDIA\_ROOT)

Figure 5. urls.py

**Authentication and permissions**

Authentication and permissions are specified in the corresponding views.py (see Figure 4 in the permission\_classes field) and permissions.py files.

from rest\_framework import permissions

from .models import Post

class IsAuthorOrReadOnly(permissions.BasePermission):

    def has\_object\_permission(self, request, view, obj):

        if request.method in permissions.SAFE\_METHODS:

            return True

        if isinstance(obj, Post):

            return obj.author == request.user.username

        return False

Figure 6. permissions.py

**Advanced features with Django Rest Framework**

**Nested serializers**

Nested serializers in Django Rest Framework allow you to include related objects within a single serialized representation. This means you can represent a relationship between models directly in the API response. For example, we have a Comment model that has a foreign key to an Post model, you can nest the Post serializer inside the Comment serializer (Figure 2 and Figure 3).

Benefits:

1. Simplifies API design: With nested serializers, the API consumer doesn't need to make multiple requests to get related data. Everything is included in one response.
2. Cleaner code: It keeps the code clean and organized by defining how related data should be serialized within the parent serializer.
3. Improved performance: It can improve performance by reducing the number of API calls needed to fetch related data.

**Versioning**

API versioning is about managing changes to your API without disrupting the clients that depend on it. In Django Rest Framework (DRF), you can set up versioning in different ways.

Importance of API Versioning:

1. Backward compatibility: allows you to introduce changes or new features without breaking the existing client implementations.
2. Flexibility: enables clients to choose when to upgrade to a new API version, providing them with a smoother transition.
3. Clear Evolution: makes it easy to track changes and improvements over time, aiding in better API management and documentation.
4. Error Reduction: reduces the risk of errors and downtime by isolating changes to specific versions.

**Rate limiting**

Rate limiting is a feature that helps control the number of requests a user can make to an API in a given period. It protects the server from being overwhelmed by too many requests and ensures fair usage by all users.

Features:

1. Request limit: sets a maximum number of requests a user can make in a specific time frame (e.g., 100 requests per minute).
2. Time window: defines the period in which the request limit applies.
3. User identification: uses user credentials or IP address to track request limits.

Benefits:

1. Prevents abuse: stops users from making too many requests too quickly, which could overload the server.
2. Improves performance: helps maintain server performance by managing the load.
3. Fair usage: ensures all users have fair access to the API without being affected by excessive usage from others.

**Deployment**

I didn't come up with anything new for deployment. I just adapted all the code and functionality related to Docker from previous tasks. Docker Compose was used. Then all the other parts can be easily wrapped in the necessary entities for deployment to various hostings or clouds. However, for high-quality implementation of the microservice architecture of horizontal scaling of the project, it is necessary to make significant changes to the project code and get rid of Docker Compose.

FROM python:3.11

RUN apt-get update \

    && apt-get install -y --no-install-recommends \

        postgresql-client \

    && rm -rf /var/lib/apt/lists/\*

WORKDIR /usr/src/app

COPY requirements.txt ./

RUN pip install -r requirements.txt

COPY . .

EXPOSE 8000

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

Figure 7. Dockerfile for Django

services:

  django:

    build: django/.

    ports:

      - 8000:8000

    env\_file:

      - ./django.env

    depends\_on:

      postgres:

        condition: service\_healthy

        restart: true

    networks:

      - net

    volumes:

      - ./django:/usr/src/app

  postgres:

    image: postgres:16.4-bullseye

    ports:

      - 5432:5432

    env\_file:

      - ./postgres.env

    healthcheck:

      test: ["CMD-SHELL", "pg\_isready -U bauyrzhan -d assignment4"]

      interval: 10s

      retries: 3

      start\_period: 30s

      timeout: 10s

    networks:

      - net

    volumes:

      - ./postgres/data:/var/lib/postgresql/data

networks:

  net:

    driver: bridge

Figure 8. docker-compose.yml

**Testing and documentation**

**API testing**

Testing API endpoints is essential to make sure they work correctly. There are several methods used to test API endpoints in Django Rest Framework:

1. Unit tests: these tests check individual parts of the API to make sure they work as expected. They focus on testing a single component in isolation.
2. Integration tests: these tests check how different parts of the API work together. They ensure that various components interact correctly.
3. End-to-End tests: these tests simulate real user scenarios. They check the entire flow from the client to the server and back.

Importance: testing ensures that your API endpoints are working correctly and efficiently. It helps catch bugs early, improves code quality, and gives confidence that changes won't break existing functionality.

from django.test import TestCase

from django.urls import reverse

from rest\_framework import status

from rest\_framework.test import APIClient

from .models import Category, Post, Comment

from django.contrib.auth.models import User

class PostAPITestCase(TestCase):

    def setUp(self):

        self.client = APIClient()

        self.user = User.objects.create\_user(username='testuser', password='testpass')

        self.client.force\_authenticate(user=self.user)

        self.category = Category.objects.create(name='Category 1', description='Description 1')

        self.post = Post.objects.create(

            title='Post 1',

            content='Content 1',

            author='testuser',

            published\_date='2024-01-01T00:00:00Z',

            image=None

        )

        self.post.category.set([self.category])

        self.post.save()

    def test\_list\_posts(self):

        response = self.client.get(reverse('post-list'))

        self.assertEqual(response.status\_code, status.HTTP\_200\_OK)

        self.assertEqual(len(response.data), 1)

    def test\_create\_post(self):

        data = {

            'title': 'Post 2',

            'content': 'Content 2',

            'author': 'testuser',

            'published\_date': '2024-01-02T00:00:00Z',

            'category': [self.category.id],

            'image': None

        }

        response = self.client.post(reverse('post-list'), data, format='json')

        self.assertEqual(response.status\_code, status.HTTP\_201\_CREATED)

        self.assertEqual(Post.objects.count(), 2)

    def test\_retrieve\_post(self):

        response = self.client.get(reverse('post-detail', args=[self.post.id]))

        self.assertEqual(response.status\_code, status.HTTP\_200\_OK)

        self.assertEqual(response.data['title'], 'Post 1')

    def test\_update\_post(self):

        data = {

            'title': 'Post 1 Updated',

            'content': 'Content 1 Updated',

            'author': 'testuser',

            'published\_date': '2024-01-01T00:00:00Z',

            'category': [self.category.id],

            'image': None

        }

        response = self.client.put(reverse('post-detail', args=[self.post.id]), data, format='json')

        self.assertEqual(response.status\_code, status.HTTP\_200\_OK)

        self.post.refresh\_from\_db()

        self.assertEqual(self.post.title, 'Post 1 Updated')

    def test\_delete\_post(self):

        response = self.client.delete(reverse('post-detail', args=[self.post.id]))

        self.assertEqual(response.status\_code, status.HTTP\_204\_NO\_CONTENT)

        self.assertEqual(Post.objects.count(), 0)

    def test\_list\_comments\_for\_post(self):

        comment = Comment.objects.create(post=self.post, content='Comment 1')

        response = self.client.get(reverse('comment-list'))

        self.assertEqual(response.status\_code, status.HTTP\_200\_OK)

        self.assertEqual(len(response.data), 1)

    def test\_create\_comment\_for\_post(self):

        data = {

            'post': self.post.id,

            'content': 'Comment 2'

        }

        response = self.client.post(reverse('comment-list'), data, format='json')

        self.assertEqual(response.status\_code, status.HTTP\_201\_CREATED)

        self.assertEqual(Comment.objects.count(), 1)

Figure 9. test.py

**API documentation**

To create the documentation I used the Swagger library.

Swagger allows us to describe the structure of your APIs so that machines can read them. The ability of APIs to describe their own structure is the root of all awesomeness in Swagger. Why is it so great? Well, by reading your API’s structure, we can automatically build beautiful and interactive API documentation. We can also automatically generate client libraries for your API in many languages and explore other possibilities like automated testing. Swagger does this by asking your API to return a YAML or JSON that contains a detailed description of your entire API. This file is essentially a resource listing of your API which adheres to OpenAPI Specification. The specification asks you to include information like:

1. What are all the operations that your API supports?
2. What are your API’s parameters and what does it return?
3. Does your API need some authorization?
4. And even fun things like terms, contact information and license to use the API.

"""

URL configuration for restfull\_blogs\_manager project.

The `urlpatterns` list routes URLs to views. For more information please see:

    https://docs.djangoproject.com/en/5.1/topics/http/urls/

Examples:

Function views

    1. Add an import:  from my\_app import views

    2. Add a URL to urlpatterns:  path('', views.home, name='home')

Class-based views

    1. Add an import:  from other\_app.views import Home

    2. Add a URL to urlpatterns:  path('', Home.as\_view(), name='home')

Including another URLconf

    1. Import the include() function: from django.urls import include, path

    2. Add a URL to urlpatterns:  path('blog/', include('blog.urls'))

"""

from django.contrib import admin

from django.urls import include, path

from django.conf.urls.static import static

from django.conf import settings

from rest\_framework.authtoken.views import obtain\_auth\_token

from restfull\_blogs\_manager.swagger import schema\_view

urlpatterns = [

    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'),

    path("", include("posts.urls")),

    path('admin/', admin.site.urls),

] + static(settings.MEDIA\_URL, document\_root=settings.MEDIA\_ROOT)

Figure 10. urls.py of the root

**Challenges and solutions**

The most difficult part of the project is working with serializers and models. As I described earlier, there were problems with the correct presentation of comments when calling one or more posts via API.

**Conclusion**

During this project, I learned and practiced building a RESTful API using Django Rest Framework (DRF). I set up a new Django project and app, installed DRF, and added it to the settings. I defined data models for posts and comments, created serializers, and set up views and URL routing for different API endpoints. I also implemented authentication, permissions, and wrote tests to ensure the functionality of the API.

This experience gave me a solid understanding of using DRF to develop robust APIs. The tools and features provided by DRF, like serializers, viewsets, and authentication classes, significantly simplify the development process. They help automate routine tasks and reduce the complexity of creating advanced web applications, making Django and DRF popular choices among developers.

**References**

Links to Django tutorials from official documentation:

1. <https://docs.djangoproject.com/en/5.1/intro/tutorial01/>
2. <https://docs.djangoproject.com/en/5.1/intro/tutorial02/>
3. <https://docs.djangoproject.com/en/5.1/intro/tutorial03/>
4. <https://docs.djangoproject.com/en/5.1/intro/tutorial04/>
5. <https://docs.djangoproject.com/en/5.1/intro/tutorial02/>

Link to official Docker documentation:

* <https://docs.docker.com/>

Link to official Docker image registry where from I took base images for compose:

* <https://hub.docker.com/>

Links to some topics in StackOverflow and other resources:

1. <https://stackoverflow.com/questions/13164048/text-box-input-height>
2. <https://stackoverflow.com/questions/3681627/how-to-update-fields-in-a-model-without-creating-a-new-record-in-django>
3. <https://stackoverflow.com/questions/42614172/how-to-redirect-from-a-view-to-another-view-in-django>
4. <https://stackoverflow.com/questions/3805958/how-to-delete-a-record-in-django-models>
5. <https://stackoverflow.com/questions/14719883/django-can-not-get-a-time-function-timezone-datetime-to-work-properly-gett>
6. <https://stackoverflow.com/questions/3289601/null-object-in-python>
7. <https://stackoverflow.com/questions/11336548/how-to-get-post-request-values-in-django>
8. <https://stackoverflow.com/questions/11714721/how-to-set-the-space-between-lines-in-a-div-without-setting-line-height>
9. <https://stackoverflow.com/questions/3430432/django-update-table>
10. <https://stackoverflow.com/questions/15128705/how-to-insert-a-row-of-data-to-a-table-using-djangos-orm>
11. <https://stackoverflow.com/questions/67155473/how-can-i-get-values-from-a-view-in-django-every-time-i-access-the-home-page>
12. <https://stackoverflow.com/questions/3500859/django-request-get>
13. <https://stackoverflow.com/questions/50346326/programmingerror-relation-django-session-does-not-exist>
14. <https://stackoverflow.com/questions/18713086/virtualenv-wont-activate-on-windows>
15. <https://www.geeksforgeeks.org/swagger-integration-with-python-django/>
16. <https://stackoverflow.com/questions/51182823/django-rest-framework-nested-serializers>
17. <https://www.django-rest-framework.org/api-guide/permissions/>
18. <https://www.django-rest-framework.org/api-guide/versioning/>
19. <https://www.django-rest-framework.org/api-guide/viewsets/>

Link to my GitHub repository (screenshots, logs and SQL structure included):

* <https://github.com/BauyrzhanAzimkhanov/Web-application-development-MSc>

**Appendices**

