KBTU KAZAKH-BRITISH TECHNICAL UNIVERSITY

Web application development

Student: Almabekuly Aibek

Date of submission: 1.12.2024

Git: https://github.com/Aibek2201/Web_app_dev_MD.git

Almaty, 2024

Table of Contents

1. **Executive Summary……………………………………………………3**
2. **Introduction 3**
3. **Building a RESTful API with Django Rest Framework.......................…4**
4. **Advanced Features with Django Rest Framework………………………...8**
5. **Results 9**
6. **Conclusion 10**
7. **References 11**

1. Executive Summary

This project provides an overview of the process of building a RESTful API using Django Rest Framework. The goal of the project was to create a fully functional and scalable API for a simple blogging platform, enabling operations such as creating, retrieving, updating, and deleting blog posts and comments. Throughout the implementation, we focused on applying key features of DRF, including model serialization, authentication, and permissions.

2. Introduction

This report aims to explore the process of building a RESTful API using DRF by developing a simple blogging platform. The application allows users to manage blog posts and comments, with endpoints to create, retrieve, update, and delete posts, as well as add and retrieve comments. Additionally, the report covers advanced features such as nested serializers, API versioning, rate limiting, and WebSocket integration for real-time functionality.

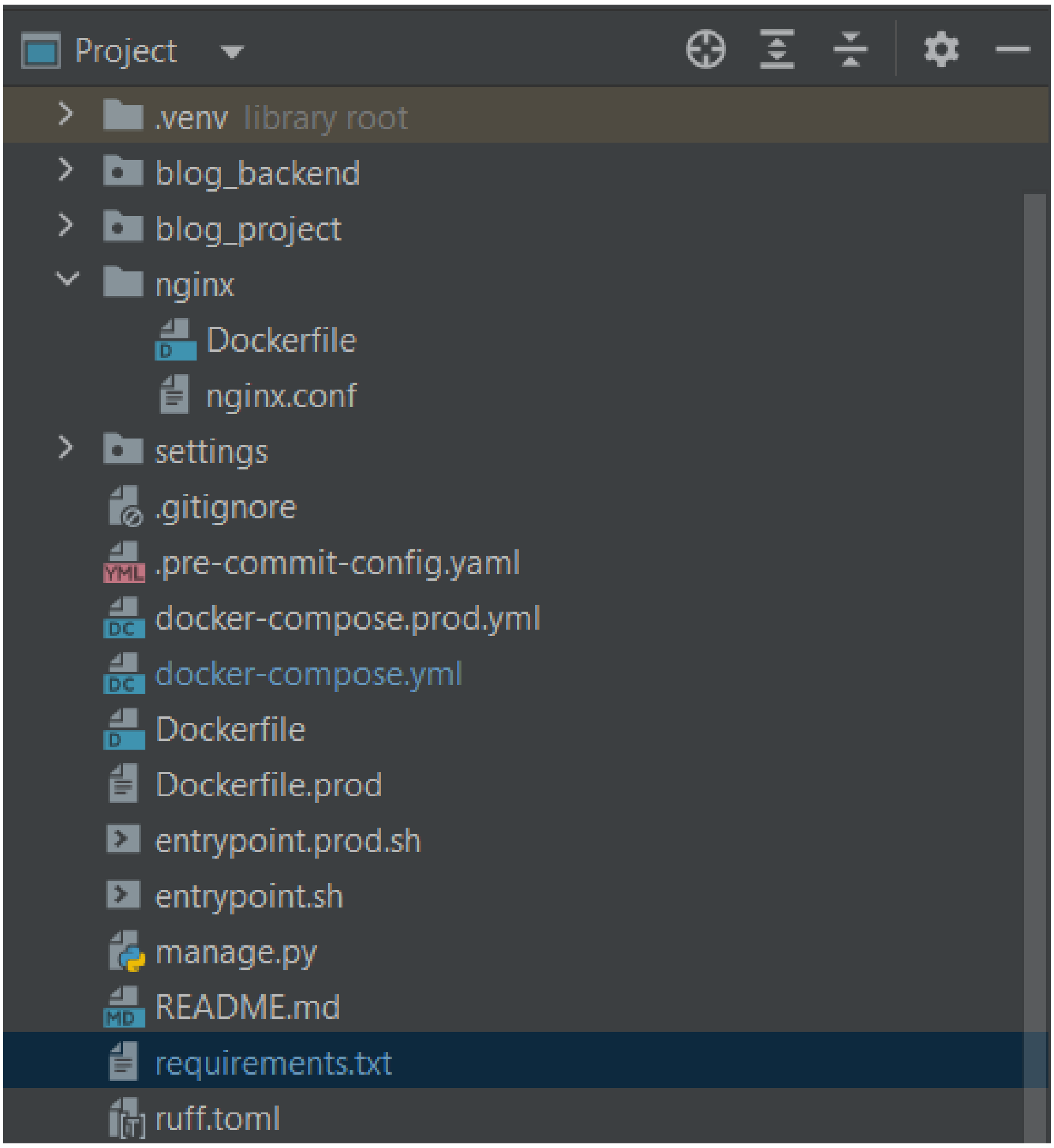
Django Rest Framework simplifies the development of RESTful APIs by providing tools and features that handle common tasks such as serialization, authentication, and routing, all while maintaining Django's flexibility and scalability. Its popularity in the developer community is attributed to its simplicity, ease of use, and ability to integrate seamlessly with Django-based applications.

The primary purpose of this project was to demonstrate the capabilities of DRF in creating a functional and maintainable API, while also addressing essential features like security and performance optimization. The scope of the report includes project setup, model design, serializer and view implementation, as well as testing and deployment.

By the end of the report, readers will gain insight into the steps involved in building and enhancing a RESTful API using Django Rest Framework and the best practices followed during development.

3. Building a RESTful API with Django Rest Framework

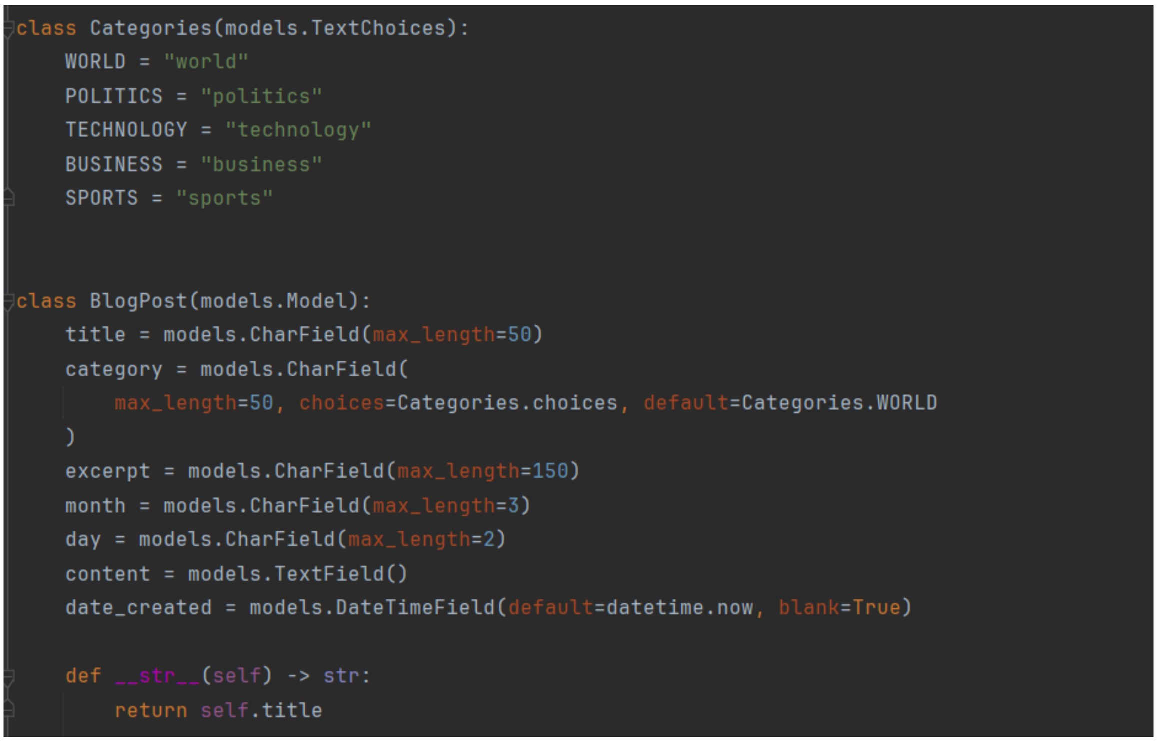
To organize this project firstly I created the main base in Django using default commands *django-admin startproject*. And I have been base of my project. We can see the main apps of my project in figure 1.



```
Project
  > .venv  library root
  > blog_backend
  > blog_project
  v nginx
      Dockerfile
      nginx.conf
  > settings
    .gitignore
    .pre-commit-config.yaml
    docker-compose.prod.yml
    docker-compose.yml
    Dockerfile
    Dockerfile.prod
    entrypoint.prod.sh
    entrypoint.sh
    manage.py
    README.md
    requirements.txt
    ruff.toml
```

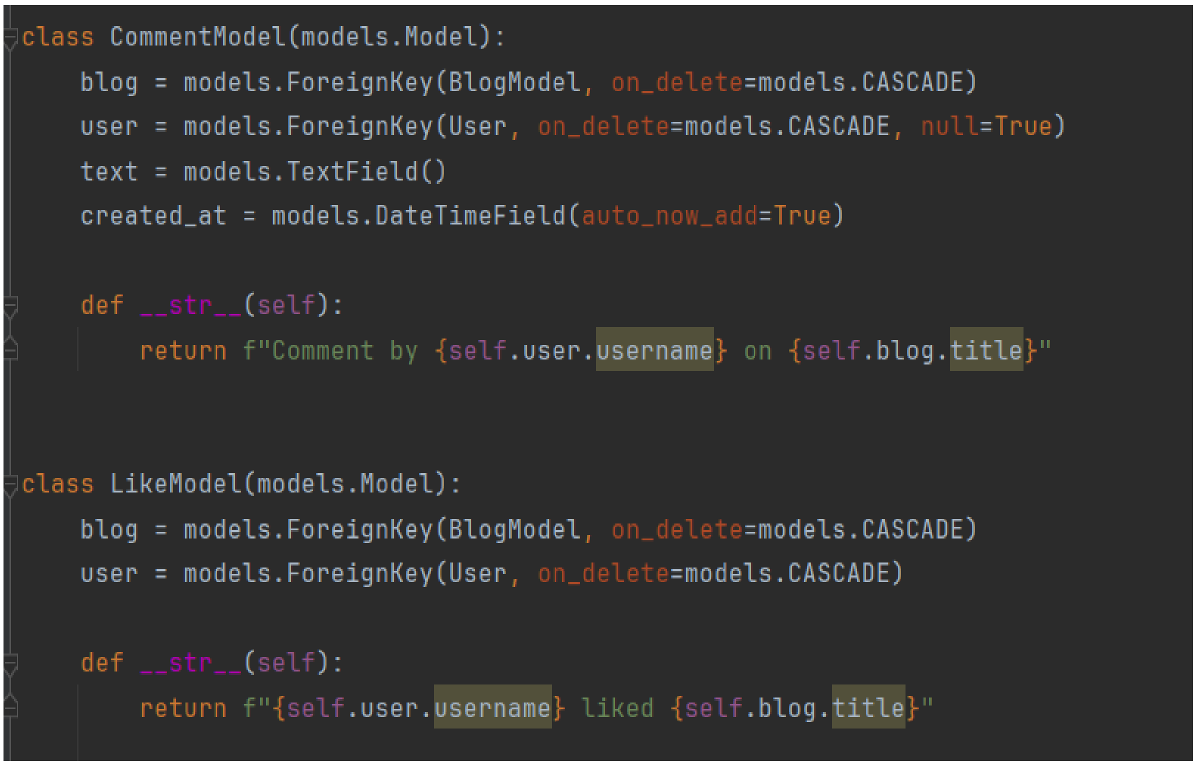
Figure 1. Architecture of blog project

After creating base for project, I started to arrange migration records to database, and created Models in Django inside of models.py. In the next figures we can see models which covered our project.



```python
class Categories(models.TextChoices):
    WORLD = "world"
    POLITICS = "politics"
    TECHNOLOGY = "technology"
    BUSINESS = "business"
    SPORTS = "sports"


class BlogPost(models.Model):
    title = models.CharField(max_length=50)
    category = models.CharField(
        max_length=50, choices=Categories.choices, default=Categories.WORLD
    )
    excerpt = models.CharField(max_length=150)
    month = models.CharField(max_length=3)
    day = models.CharField(max_length=2)
    content = models.TextField()
    date_created = models.DateTimeField(default=datetime.now, blank=True)

    def __str__(self) -> str:
        return self.title
```

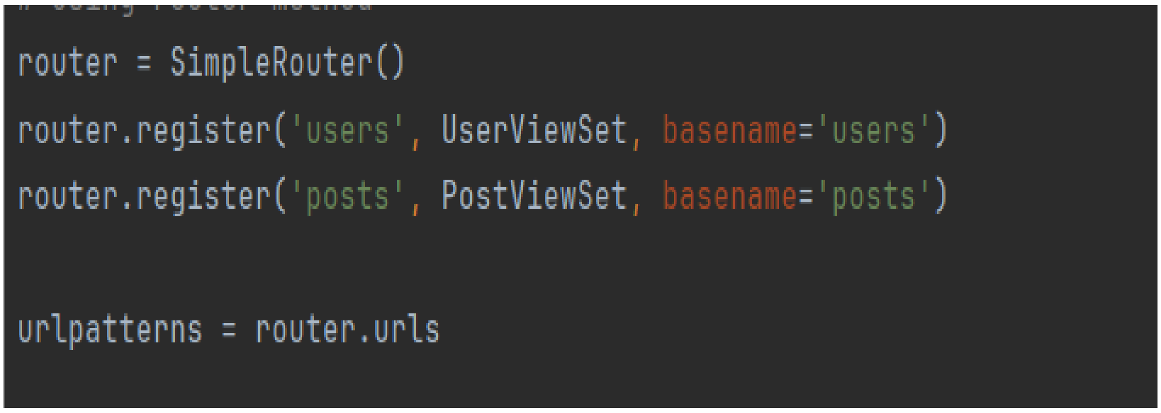
Figure 2. Model for blogs



```python
class CommentModel(models.Model):
    blog = models.ForeignKey(BlogModel, on_delete=models.CASCADE)
    user = models.ForeignKey(User, on_delete=models.CASCADE, null=True)
    text = models.TextField()
    created_at = models.DateTimeField(auto_now_add=True)

    def __str__(self):
        return f"Comment by {self.user.username} on {self.blog.title}"


class LikeModel(models.Model):
    blog = models.ForeignKey(BlogModel, on_delete=models.CASCADE)
    user = models.ForeignKey(User, on_delete=models.CASCADE)

    def __str__(self):
        return f"{self.user.username} liked {self.blog.title}"
```

Figure 3. Model for comments and likes

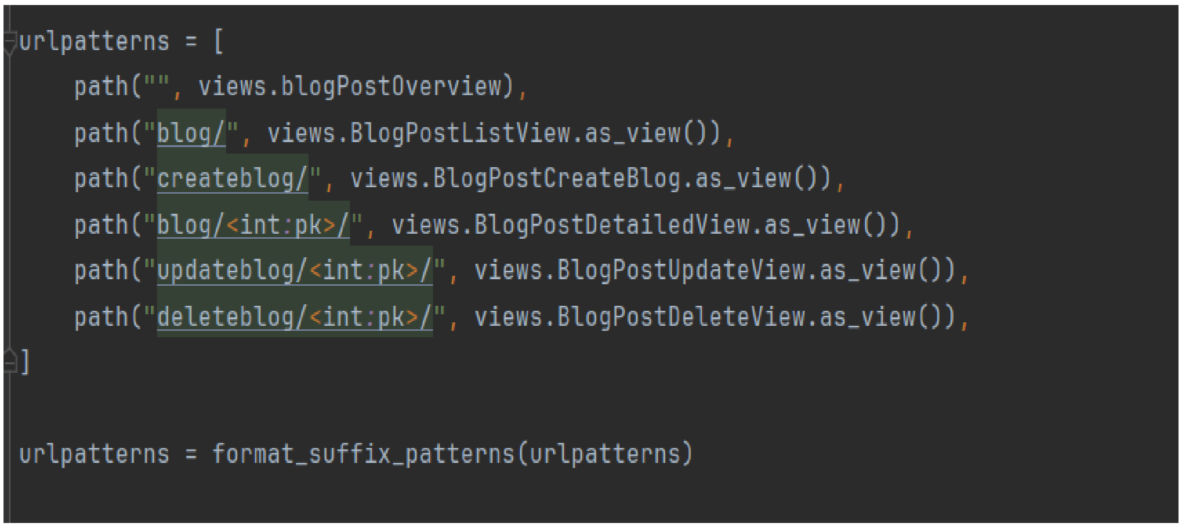
After that I added views to arrange API connections between frontend and backend. This arranging we can see in the next figure. Here I oorganized using routers in DRF.



```
router = SimpleRouter()
router.register('users', UserViewSet, basename='users')
router.register('posts', PostViewSet, basename='posts')

urlpatterns = router.urls
```

Figure 4. Core routers

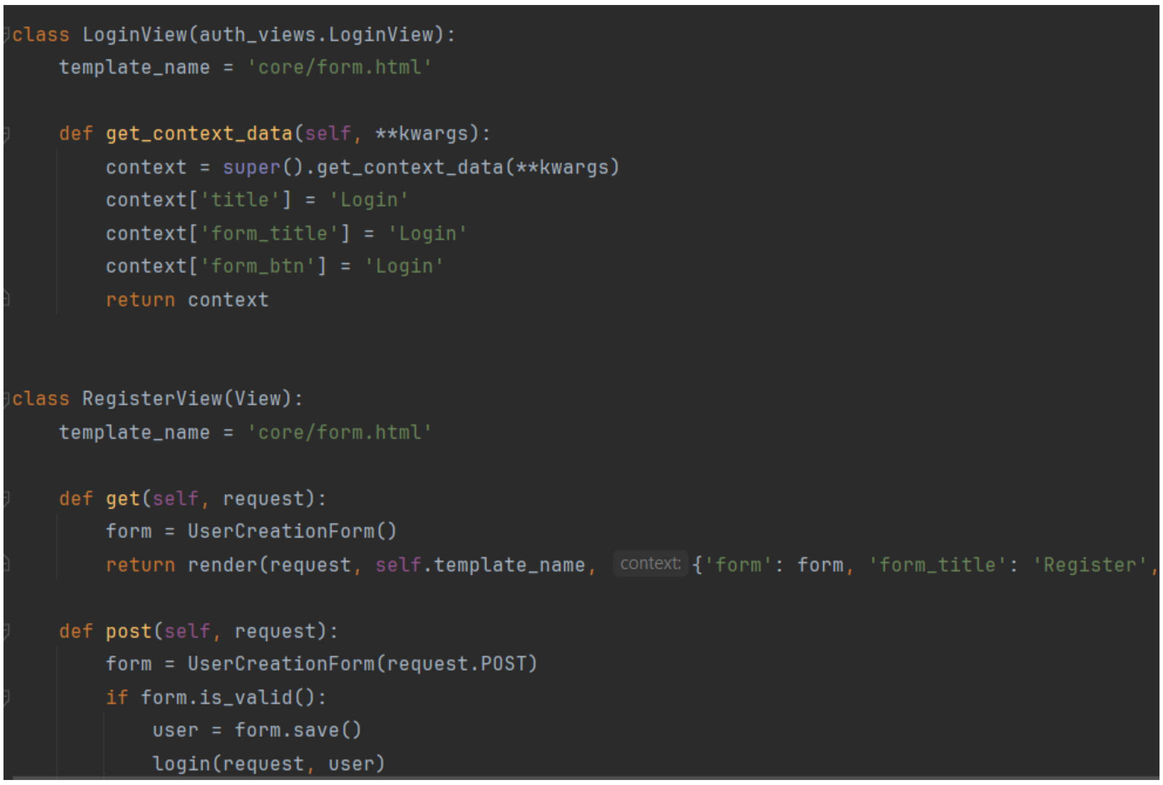
After arranging core routers, I created also one file for urlpatterns but in this case I put in inside of app. Here I wrote all of url addresses which related with this application.



```
urlpatterns = [
    path("", views.blogPostOverview),
    path("blog/", views.BlogPostListView.as_view()),
    path("createblog/", views.BlogPostCreateBlog.as_view()),
    path("blog/<int:pk>/", views.BlogPostDetailedView.as_view()),
    path("updateblog/<int:pk>/", views.BlogPostUpdateView.as_view()),
    path("deleteblog/<int:pk>/", views.BlogPostDeleteView.as_view()),
]

urlpatterns = format_suffix_patterns(urlpatterns)
```

Figure 5. URL addresses related with application

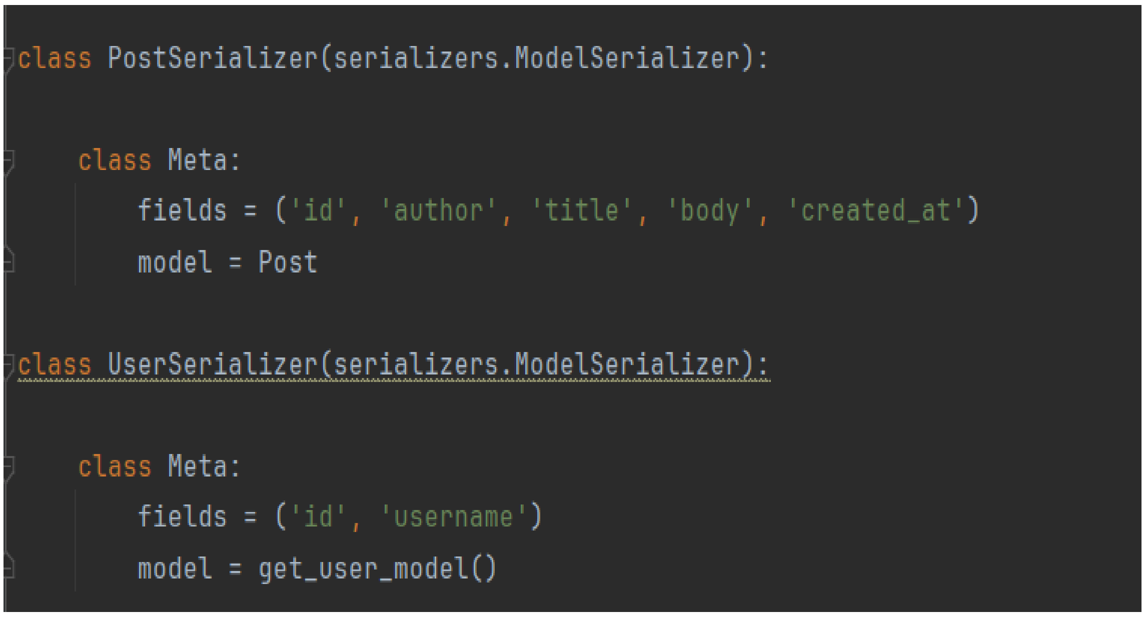
I did a lot of works to arrange blog project, but still it’s not enough to see project in the browser, my next step was arrange views file to request data to frontend side. In this case I used Class based views file. What does it mean? Using this type of views I can put a lot of functions inside of class, and it can be so scalability. I will show several class based views which I used in my project.



```
class LoginView(auth_views.LoginView):
    template_name = 'core/form.html'

    def get_context_data(self, **kwargs):
        context = super().get_context_data(**kwargs)
        context['title'] = 'Login'
        context['form_title'] = 'Login'
        context['form_btn'] = 'Login'
        return context


class RegisterView(View):
    template_name = 'core/form.html'

    def get(self, request):
        form = UserCreationForm()
        return render(request, self.template_name, context: {'form': form, 'form_title': 'Register',

    def post(self, request):
        form = UserCreationForm(request.POST)
        if form.is_valid():
            user = form.save()
            login(request, user)
```

Figure 6. Class based views for login and register page

After creating Class based view models we must organize serializers in our project. Serializers work like validators. So and it can convert some data from dictionary in python to JSON form which use more applications for relating each other.



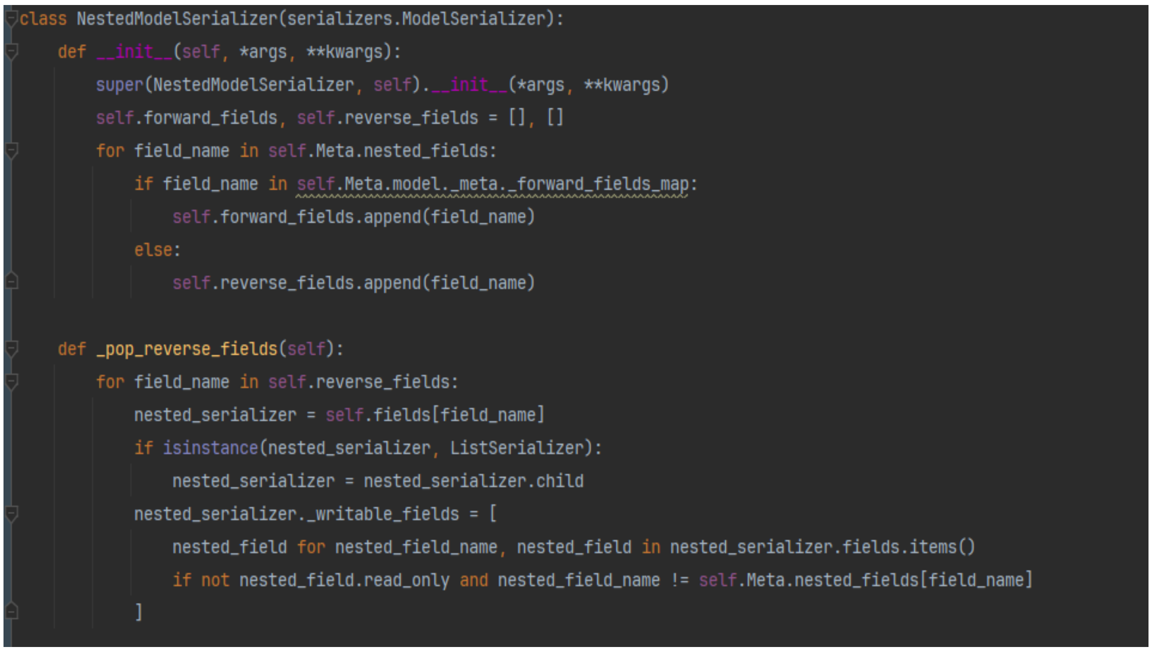
```
class PostSerializer(serializers.ModelSerializer):

    class Meta:
        fields = ('id', 'author', 'title', 'body', 'created_at')
        model = Post

class UserSerializer(serializers.ModelSerializer):

    class Meta:
        fields = ('id', 'username')
        model = get_user_model()
```

Figure 7. Class based Homepage view

Here we finished main part of our project. Next part we can see in the next chapter.

4. Advanced Features with Django Rest Framework

To organize nested serializers we can create it inside of *serializers.py file*. Because nested serializers its also serializers with extended functionality. So here we can see the sample base of nested serializer. And I organized this project using this code. Instance we can see in the next figure.



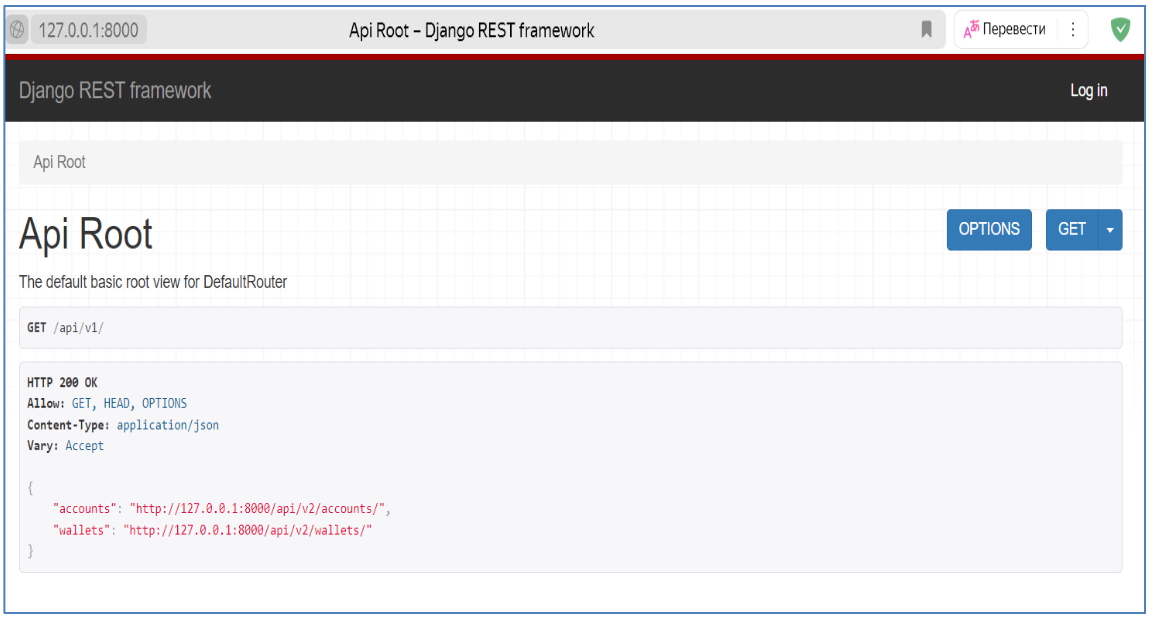
```python
class NestedModelSerializer(serializers.ModelSerializer):
    def __init__(self, *args, **kwargs):
        super(NestedModelSerializer, self).__init__(*args, **kwargs)
        self.forward_fields, self.reverse_fields = [], []
        for field_name in self.Meta.nested_fields:
            if field_name in self.Meta.model._meta._forward_fields_map:
                self.forward_fields.append(field_name)
            else:
                self.reverse_fields.append(field_name)

    def _pop_reverse_fields(self):
        for field_name in self.reverse_fields:
            nested_serializer = self.fields[field_name]
            if isinstance(nested_serializer, ListSerializer):
                nested_serializer = nested_serializer.child
            nested_serializer._writable_fields = [
                nested_field for nested_field_name, nested_field in nested_serializer.fields.items()
                if not nested_field.read_only and nested_field_name != self.Meta.nested_fields[field_name]
            ]
```

Figure 8. Nested serializers of project

Regarding deployment, I believe it's unnecessary for this project because I have structured it solely around the API endpoints, without including a frontend component. This decision simplifies the process, as the focus is entirely on building the backend API, which is fully functional and independent of a frontend interface. Since the project is designed to handle API requests and responses, deploying it to a server or cloud service would be more relevant once a frontend is incorporated or if the API needs to be publicly accessible.

5. Results

In this chapter we can see how works this project, and how organized DRF side. Firstly we must organize urls correctly. And turn on API ROOT page and using this page we can sign any other pages of website



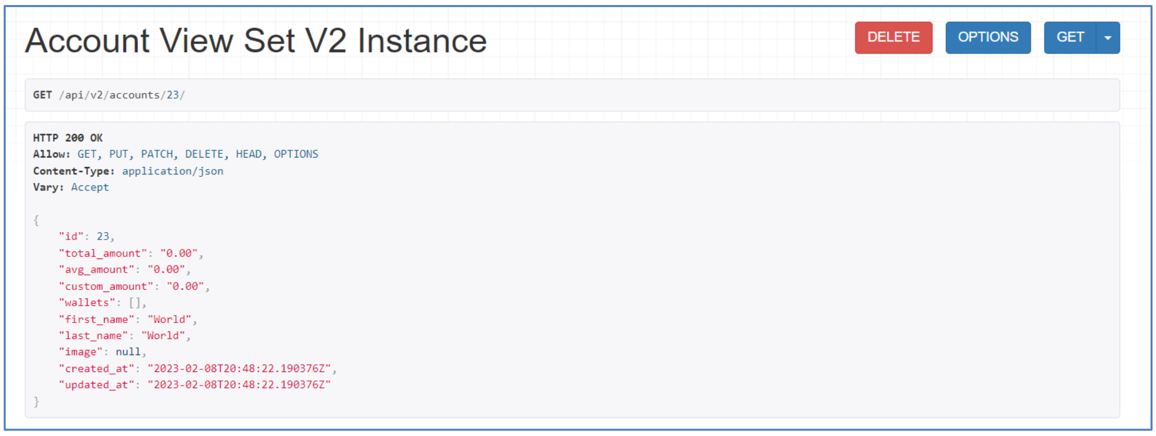
The default basic root view for DefaultRouter

```
GET /api/v1/
```

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

{
    "accounts": "http://127.0.0.1:8000/api/v2/accounts/",
    "wallets": "http://127.0.0.1:8000/api/v2/wallets/"
}
```

Figure 8. API Root page of DRF

So here as we can see 2 url routes which organized with router in urls.py. And after creating this page we can use it as main page.

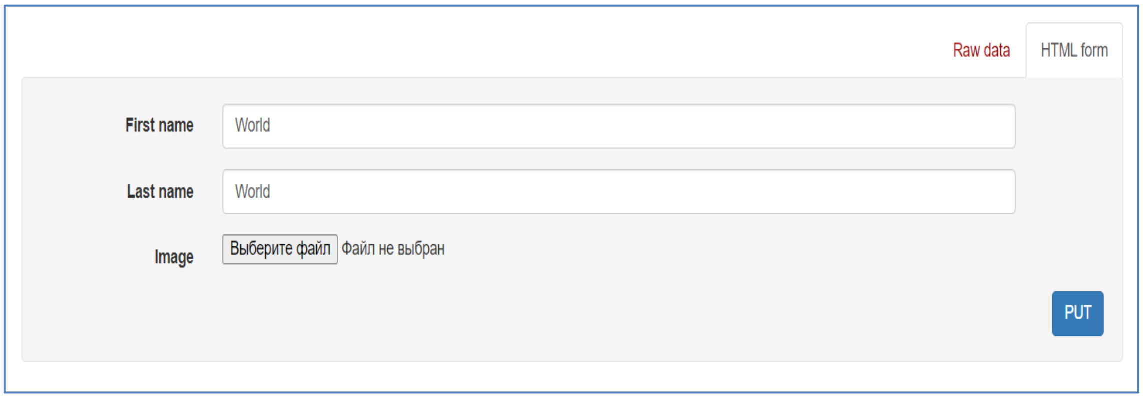


```
GET /api/v2/accounts/23/
```

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

{
    "id": 23,
    "total_amount": "0.00",
    "avg_amount": "0.00",
    "custom_amount": "0.00",
    "wallets": [],
    "first_name": "World",
    "last_name": "World",
    "image": null,
    "created_at": "2023-02-08T20:48:22.190376Z",
    "updated_at": "2023-02-08T20:48:22.190376Z"
}
```

Figure 9. Account details page

And using this page we can request to update some records directly here, To do it we should use in the bottom side of this web page and we will see it.



First name	World
Last name	World
Image	Выберите файл Файл не выбран

Figure 10. POST request instance

Also we can request to update some records using some tools like Postman, CURL, Swagger documentation and etc programs to request head and body of data.

6. Conclusion

In this project, the implementation of a RESTful API using Django Rest Framework (DRF) successfully demonstrated the fundamental principles and best practices of API development. By focusing on creating robust and scalable endpoints, we ensured that the API provides essential functionalities such as creating, reading, updating, and deleting data for a blogging platform. The integration of DRF's features, including serializers, authentication, and custom permissions, allowed for a secure and efficient application design.

Advanced functionalities, such as nested serializers and API versioning, enhanced the API’s usability and future-proofed its structure for potential updates. The inclusion of rate limiting added an additional layer of security, ensuring the API remains resilient under high traffic. Optional features like WebSocket integration were explored for real-time capabilities, showcasing the API's potential for expansion.

Although deployment was considered unnecessary for this specific project due to its backend-only focus, the project is designed to be easily deployable in the future. This flexibility enables seamless integration with frontend applications or external services when needed.

The project highlighted challenges, such as configuring custom permissions and ensuring data validation, which were addressed through careful implementation and testing. Overall, the experience reinforced the importance of modular and maintainable code design, aligning with industry standards for API development.

7. References

1 Docker Documentation: https://docs.docker.com
2 Django Documentation: https://docs.djangoproject.com
3 Official Python Docker Image: https://hub.docker.com/_/python