KBTU KAZAKH-BRITISH TECHNICAL UNIVERSITY

Web application development

Student: Almabekuly Aibek

Date of submission: 10.11.2024

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

Almaty, 2024

Table of Contents

1. **Executive Summary……………………………………………………….3**
2. **Introduction 3**
3. **Code Snippets ....…4**
4. **Results 7**
5. **Conclusion 10**
6. **References 10**

1. Executive Summary

This project focuses on developing a functional blog application using the Django web framework. The blog project serves as an exploration of core Django components Models, Views, and Templates which are fundamental for creating dynamic, database-driven web applications. This assignment requires implementing these elements, documenting the development process, and submitting a report and code repository.

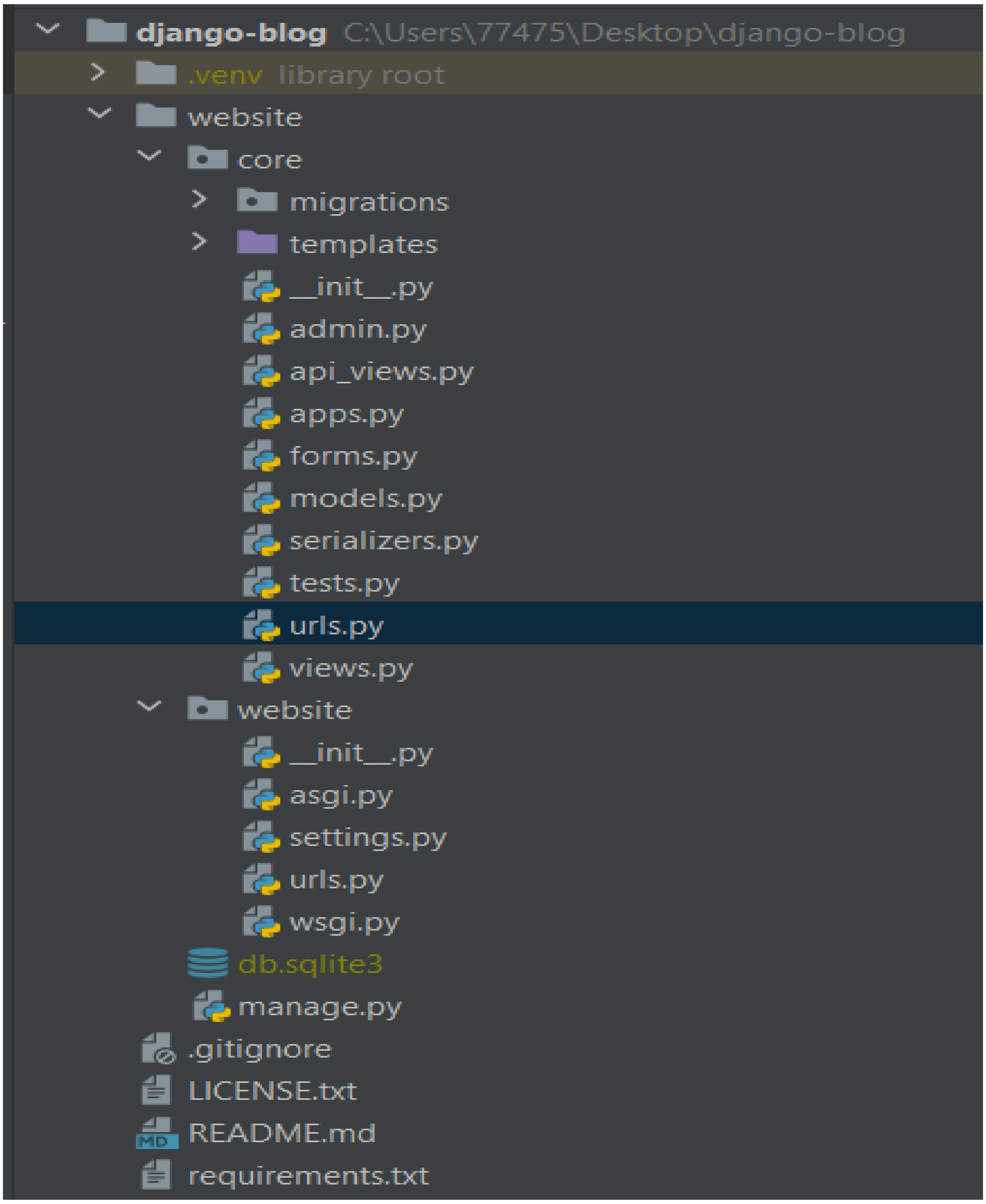
2. Introduction

This project focuses on building a fully functional blog application using the Django web framework, a popular tool for creating robust, scalable web applications. Django is based on the Model-View-Template architectural pattern, which separates an application's data, user interface, and business logic, making it an efficient and structured framework for web development. This project serves as an in-depth exercise in implementing core components of Django Models, Views, and Templates which form the backbone of any Django application. In modern web development, applications are expected to handle complex data relationships, provide an intuitive interface for users, and offer dynamic content that can be easily managed. With Django, developers can quickly set up these capabilities through reusable and maintainable code. This project was designed to explore these functionalities in a structured way, following a series of exercises to develop a simple but feature-rich blog application. The application allows users to view, categorize, and comment on blog posts, offering a practical example of Django's power and flexibility.

The blog project not only highlights the importance of each component within Django's MVT architecture but also demonstrates how they work together to create a dynamic, user-friendly web application. Through this assignment, we gain practical experience with essential Django features, reinforcing concepts like model relationships, view logic, form handling, and template rendering. This knowledge is invaluable for developing more complex and interactive applications in Django, providing a strong foundation for future projects.

3. Code snippets

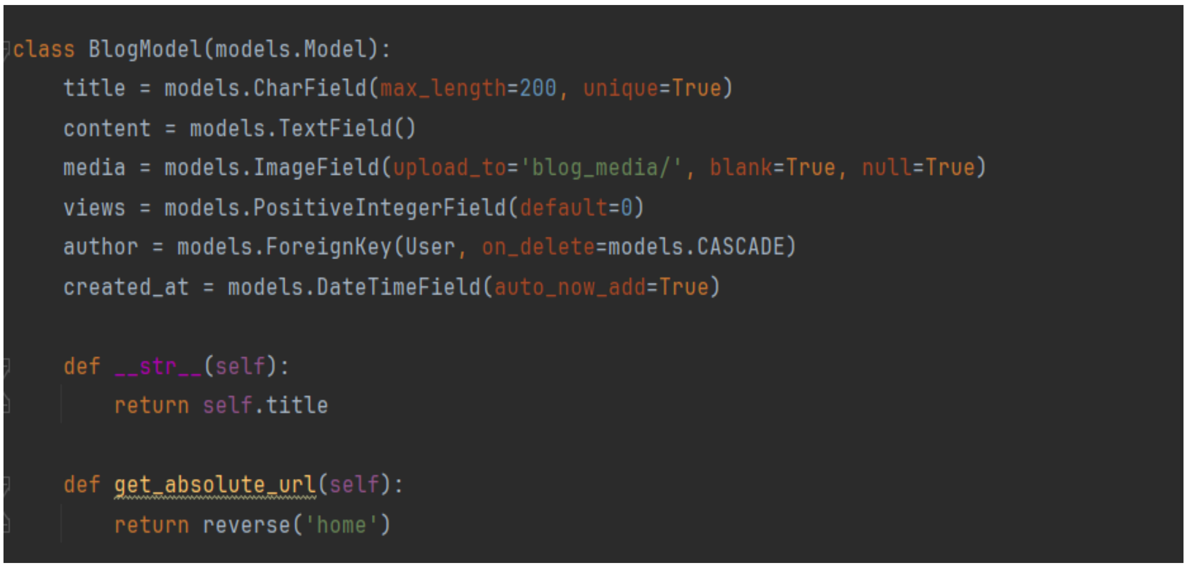
To organize this project firstly I created the main base in Django using default commands *django-admin startproject*. And I have been base of my project. We can see the main apps of my project in figure 1.



```
django-blog C:\Users\77475\Desktop\django-blog
  .venv library root
  website
    core
      migrations
      templates
      __init__.py
      admin.py
      api_views.py
      apps.py
      forms.py
      models.py
      serializers.py
      tests.py
      urls.py
      views.py
    website
      __init__.py
      asgi.py
      settings.py
      urls.py
      wsgi.py
    db.sqlite3
    manage.py
  .gitignore
  LICENSE.txt
  README.md
  requirements.txt
```

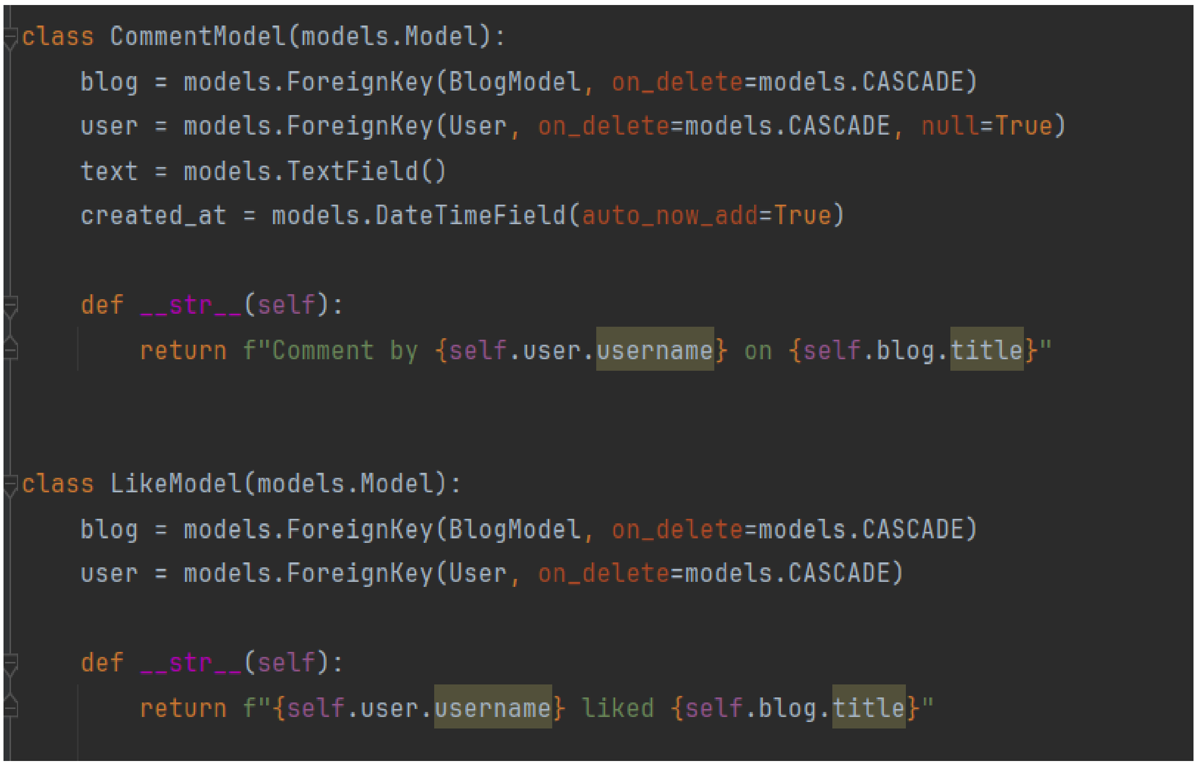
Figure 1. Architecture of blog project

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 BlogModel(models.Model):
    title = models.CharField(max_length=200, unique=True)
    content = models.TextField()
    media = models.ImageField(upload_to='blog_media/', blank=True, null=True)
    views = models.PositiveIntegerField(default=0)
    author = models.ForeignKey(User, on_delete=models.CASCADE)
    created_at = models.DateTimeField(auto_now_add=True)

    def __str__(self):
        return self.title

    def get_absolute_url(self):
        return reverse('home')
```

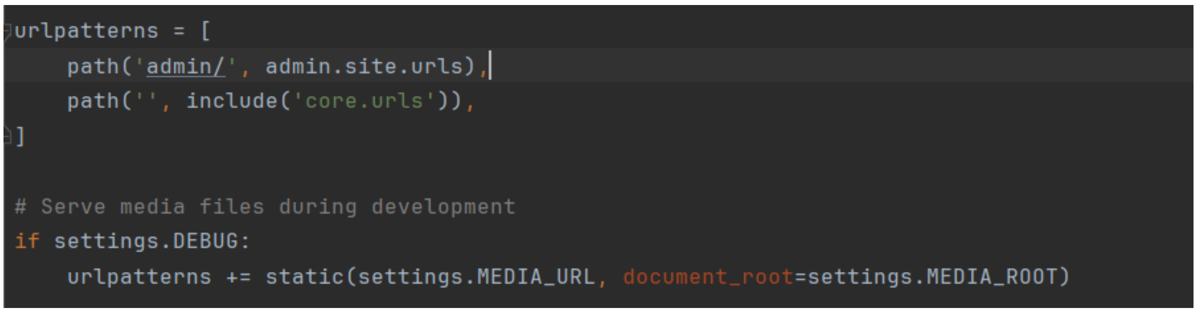
Figure 2. Model for blogs



```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

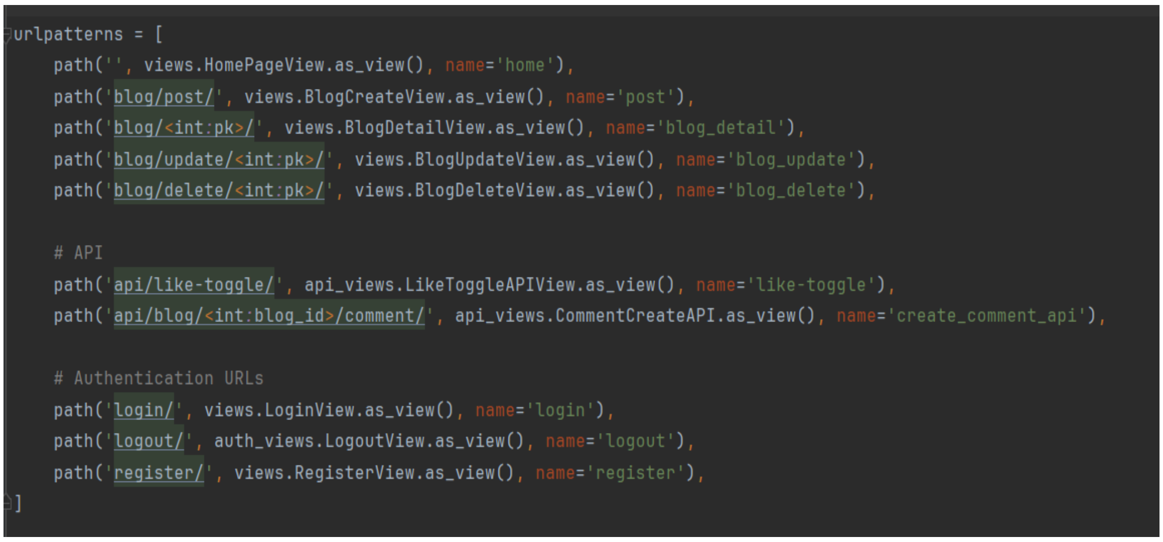
After that I added views to arrange API connections between frontend and backend. This arranging we can see in the next figure.



```python
urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('core.urls')),
]

# Serve media files during development
if settings.DEBUG:
    urlpatterns += static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
```

Figure 4. Core urlpatterns

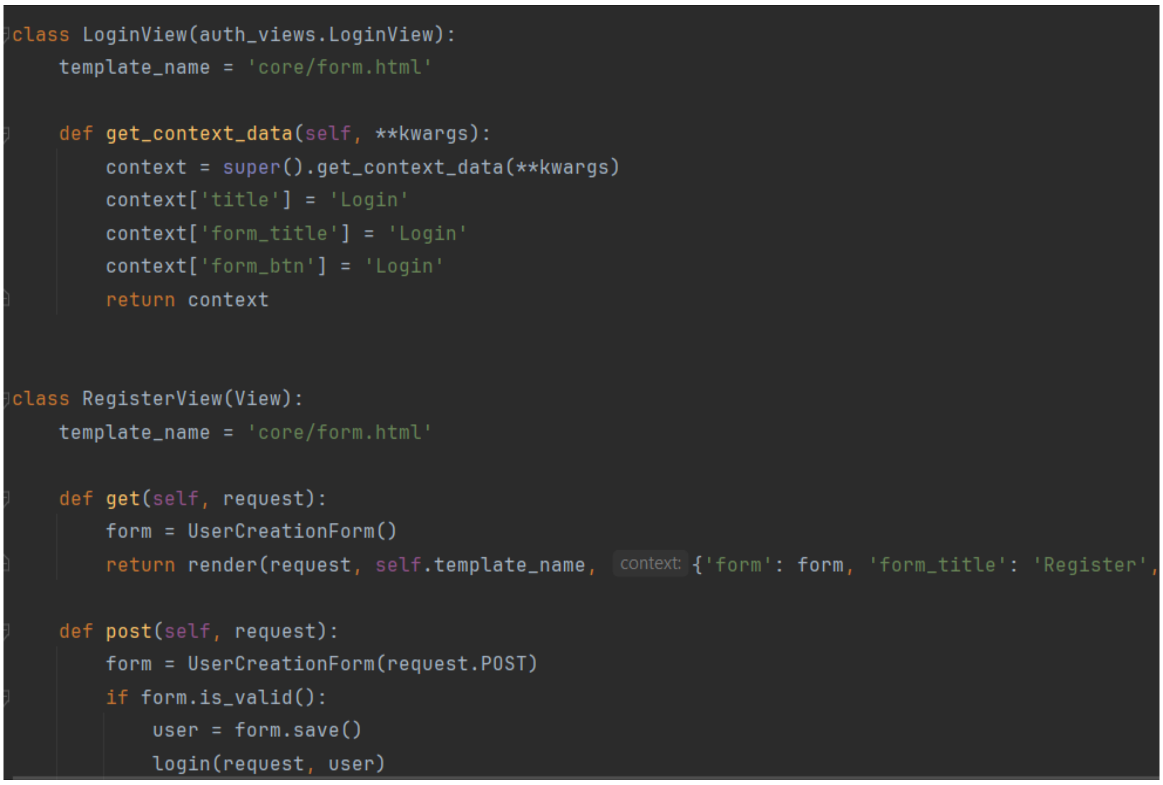
After arranging core urlpatterns, I created also one file for urlpatterns but in this case I put in inside of app. Here I wrote all of url addresses which related with this application.



```
urlpatterns = [
    path('', views.HomePageView.as_view(), name='home'),
    path('blog/post/', views.BlogCreateView.as_view(), name='post'),
    path('blog/<int:pk>/', views.BlogDetailView.as_view(), name='blog_detail'),
    path('blog/update/<int:pk>/', views.BlogUpdateView.as_view(), name='blog_update'),
    path('blog/delete/<int:pk>/', views.BlogDeleteView.as_view(), name='blog_delete'),

    # API
    path('api/like-toggle/', api_views.LikeToggleAPIView.as_view(), name='like-toggle'),
    path('api/blog/<int:blog_id>/comment/', api_views.CommentCreateAPI.as_view(), name='create_comment_api'),

    # Authentication URLs
    path('login/', views.LoginView.as_view(), name='login'),
    path('logout/', auth_views.LogoutView.as_view(), name='logout'),
    path('register/', views.RegisterView.as_view(), name='register'),
]
```

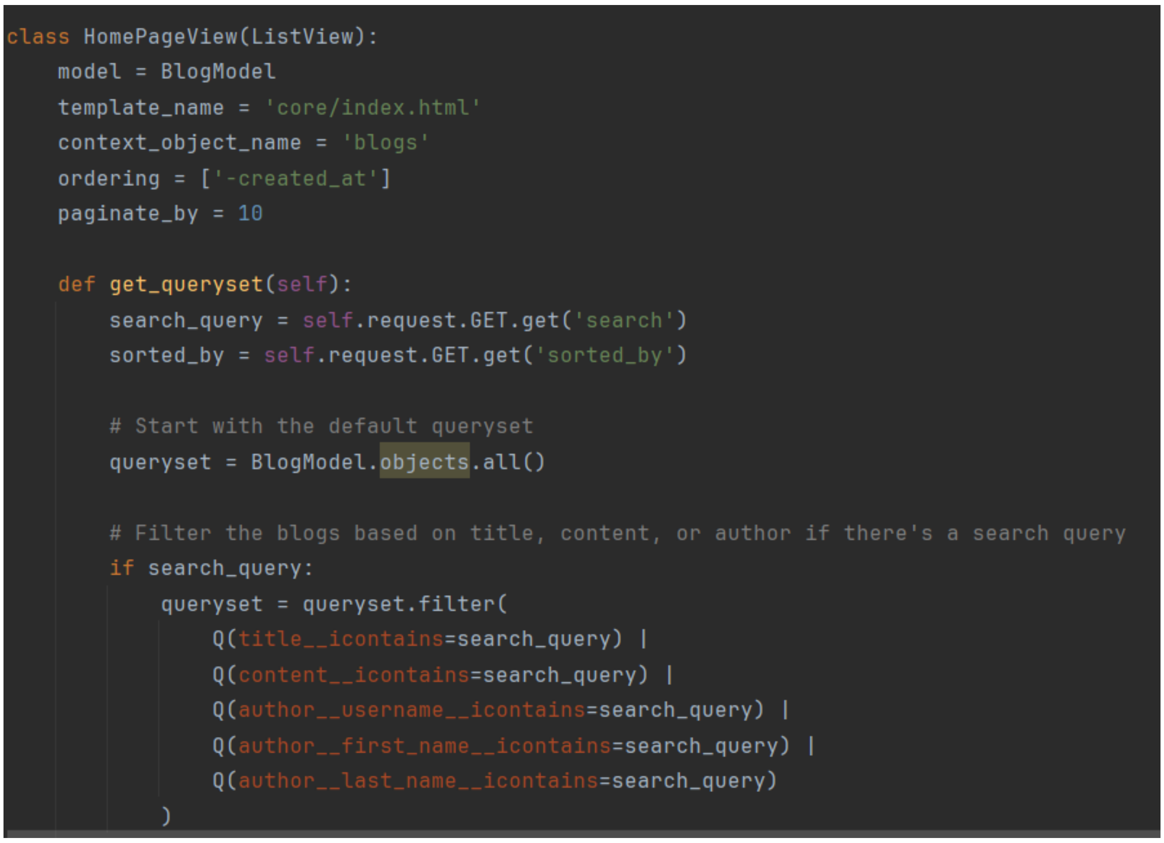
Figure 5. URL addresses related with application

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



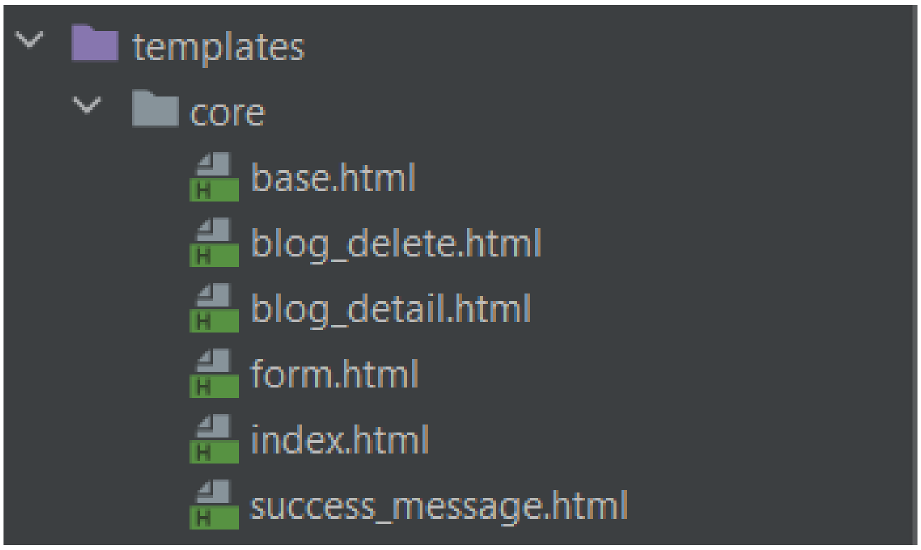
```
class HomePageView(ListView):
    model = BlogModel
    template_name = 'core/index.html'
    context_object_name = 'blogs'
    ordering = ['-created_at']
    paginate_by = 10

    def get_queryset(self):
        search_query = self.request.GET.get('search')
        sorted_by = self.request.GET.get('sorted_by')

        # Start with the default queryset
        queryset = BlogModel.objects.all()

        # Filter the blogs based on title, content, or author if there's a search query
        if search_query:
            queryset = queryset.filter(
                Q(title__icontains=search_query) |
                Q(content__icontains=search_query) |
                Q(author__username__icontains=search_query) |
                Q(author__first_name__icontains=search_query) |
                Q(author__last_name__icontains=search_query)
            )
```

Figure 7. Class based Homepage view

Here we finished backend side of our project. Frontend side we can see in the result chapter.

4. Results

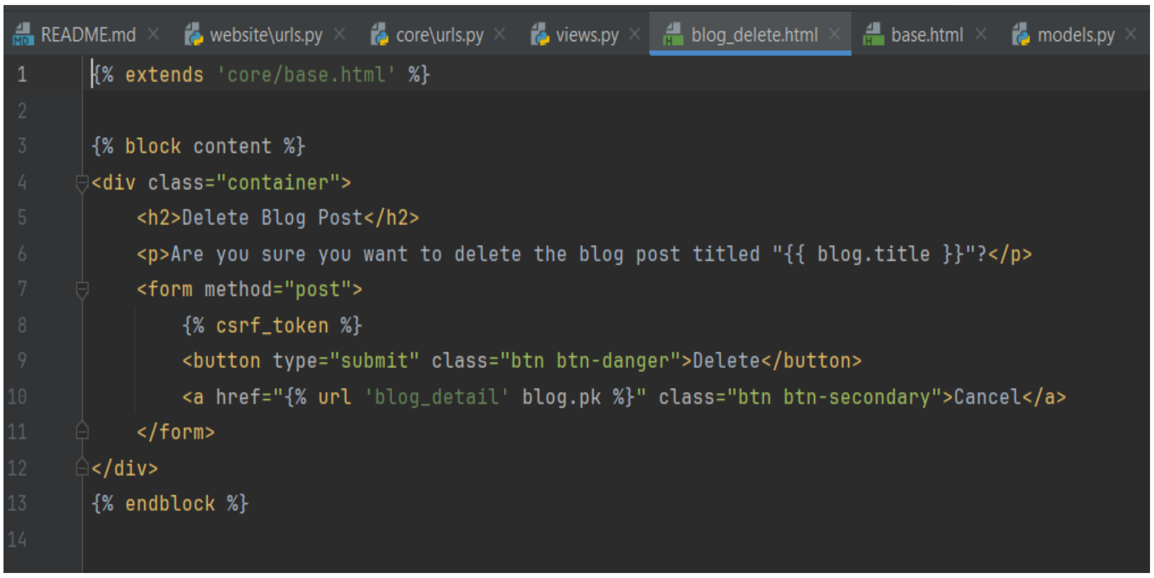
In this chapter we can see how works blog project, and how organized frontend side. Firstly I created templates file.



```
templates
    core
        base.html
        blog_delete.html
        blog_detail.html
        form.html
        index.html
        success_message.html
```

Figure 8. Architecture of template files

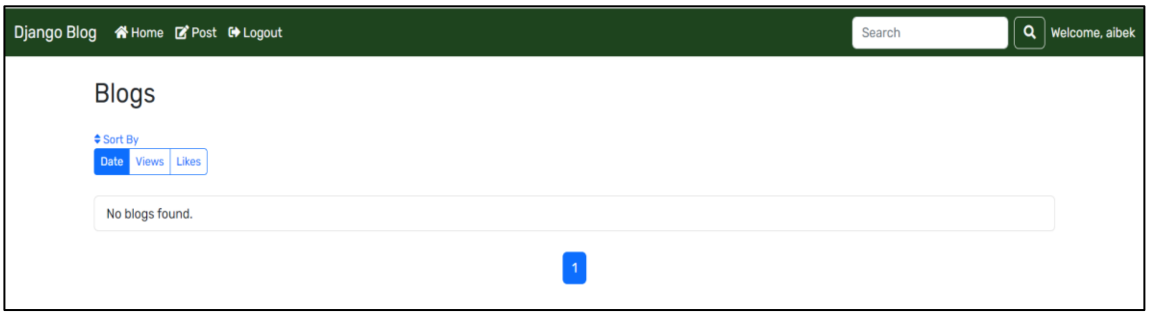
Here we can see base.html file. I used this file in this project like basement. Usign this file we won’t write same code in each file. After organizing base file, I created some files. For instance we can see code in the blog delete file which used base.html file.



```
{% extends 'core/base.html' %}

{% block content %}
<div class="container">
    <h2>Delete Blog Post</h2>
    <p>Are you sure you want to delete the blog post titled "{{ blog.title }}"?</p>
    <form method="post">
        {% csrf_token %}
        <button type="submit" class="btn btn-danger">Delete</button>
        <a href="{% url 'blog_detail' blog.pk %}" class="btn btn-secondary">Cancel</a>
    </form>
</div>
{% endblock %}
```

Figure 9. Blog delete file using base.html

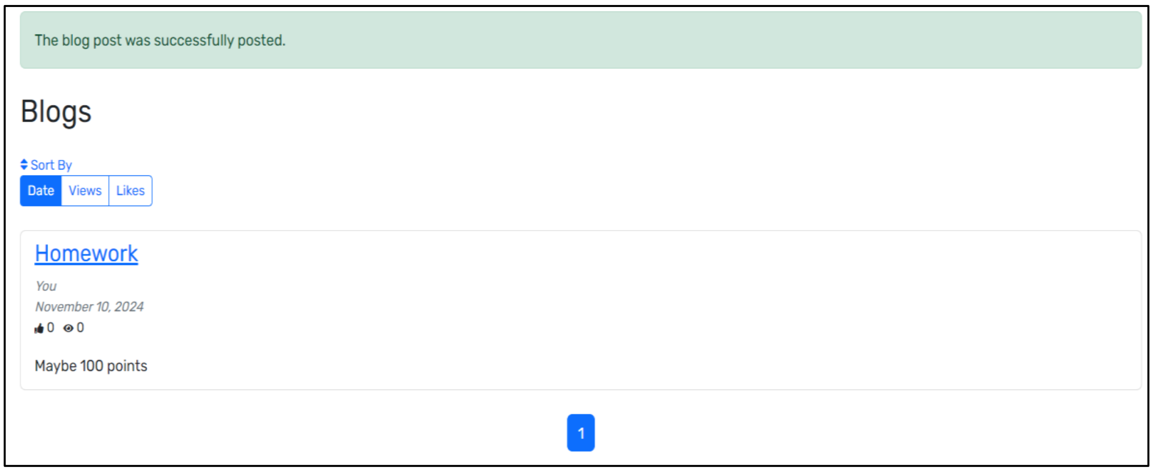
Lets see how works this project in the browser.



Blogs

No blogs found.

Figure 10. Main page of project



The blog post was successfully posted.

Blogs

Homework

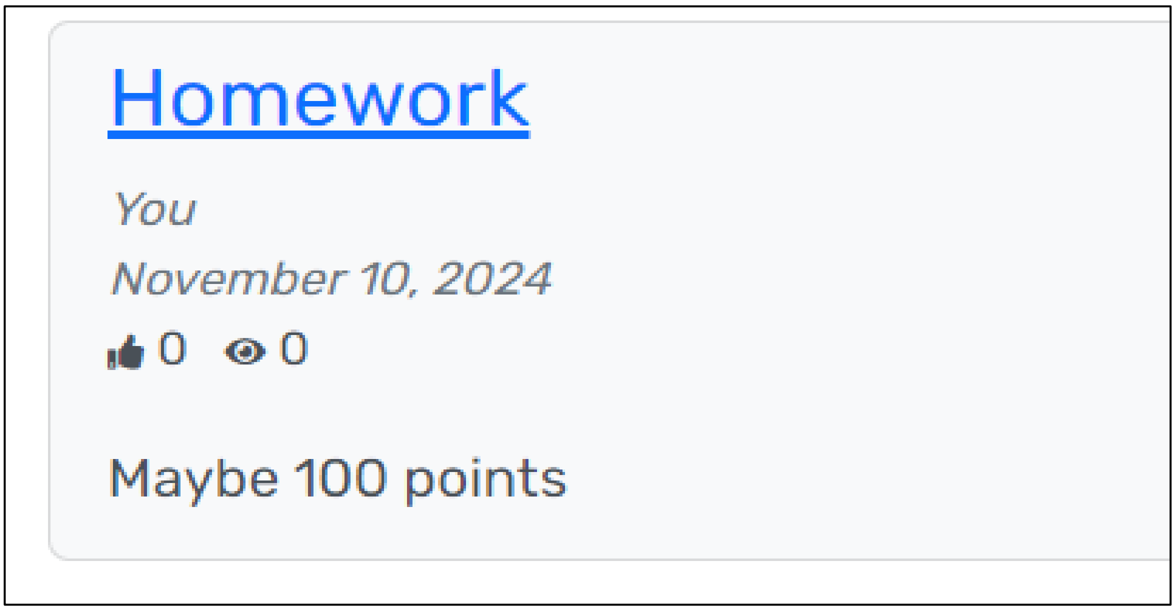
You

November 10, 2024

0 0

Maybe 100 points

Figure 11. Page after creating blog with dialog window


Homework

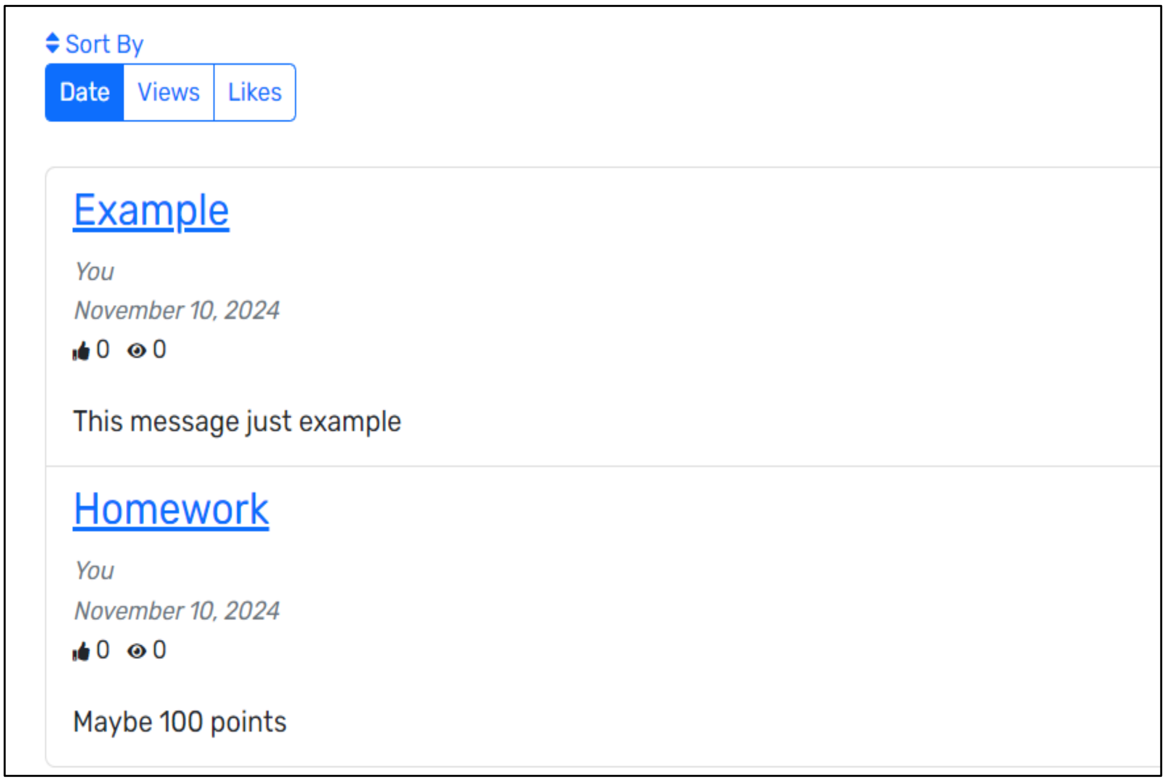
You

November 10, 2024

0 0

Maybe 100 points

Figure 12. Content details with likes and views



Example

You

November 10, 2024

0 0

This message just example

Homework

You

November 10, 2024

0 0

Maybe 100 points

Figure 13. Blog records with sorting

5. Conclusion

This Django blog project has been a comprehensive exploration of the fundamental components involved in developing a modern web application. By implementing models, views, and templates, we have gained hands-on experience with Django’s Model-View-Template architecture, which is essential for creating structured, maintainable, and efficient web applications.

Overall this project has reinforced the significance of the MVT components in Django. By working with models, views, and templates, we have acquired a deep understanding of how these elements work together to produce a complete, functional web application. This foundational knowledge is essential for building more advanced applications in the future, equipping us with the skills to design and implement sophisticated, scalable solutions with Django.

6. 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