Kazakh British Technical University

Web Application Development

Assignment 3

by Moldir Polat

November 7th 2024

Table of contents

Introduction

In web development with Django, understanding the fundamental components—models, views, and templates—is essential for building dynamic web applications. Each of these components plays a distinct role in Django's Model-View-Template (MVT) architecture, enabling developers to separate data handling, user interface design, and application logic. This assignment provides a structured approach to these essential concepts, guiding us through creating and manipulating models, crafting function- and class-based views, and designing templates that bring these models to life in a user-friendly interface.

The first focus of this assignment is on Django models, where we define a model for blog posts with fields for titles, content, and publication details. This section also introduces model relationships, illustrating how data can be organized through many-to-many and foreign key relationships. Additionally, custom managers are implemented to manage and filter data, demonstrating how Django's ORM allows us to interact with the database in a more intuitive way. Understanding models is crucial, as they form the foundation of data handling in Django applications, representing database tables and used for data retrieval and manipulation.

The next point is Django views, where function-based and class-based views are explored to handle HTTP requests and responses. Function-based views offer a straightforward way to manage basic logic, while class-based views enable code reuse and extensibility. This part of the assignment also includes handling forms, enabling users to interact with the application and submit data. Views act as the bridge between models and templates, ensuring that the right data is sent to the templates for display, thus playing a big role in structuring the application's logic and flow.

Finally, the assignment includes Django templates, where we learn how HTML templates render dynamic content on web pages. Templates are essential for defining the look of a Django application. Using template inheritance, static files, and media files, we can create reusable structures, add styling, and handle media uploads, enhancing the overall user experience. By understanding Django models, views, and templates altogether, this assignment teaches us with the skills necessary to build well-organized, visually good-looking, and interactive web applications.

Django Models

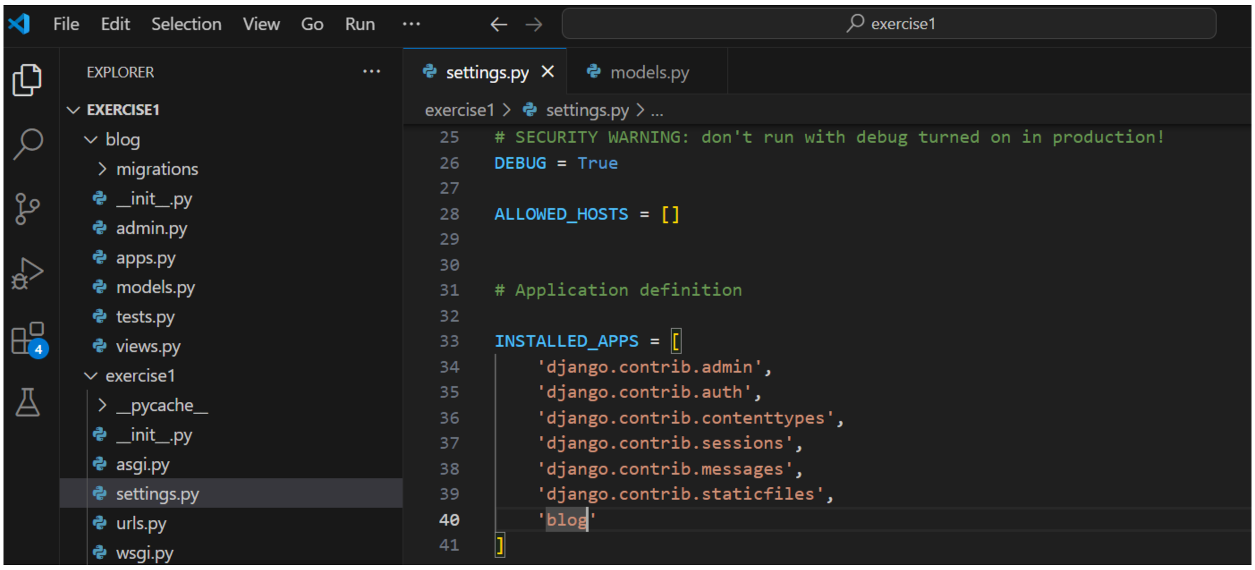
Exercise 1

Exercise title: Creating a Basic Model

Objective: to create a basic model in Djnago called Post within new app, which contains main fields to store information about a blog post. I'll implement a method to provide a string representation for each instance of the Post model.

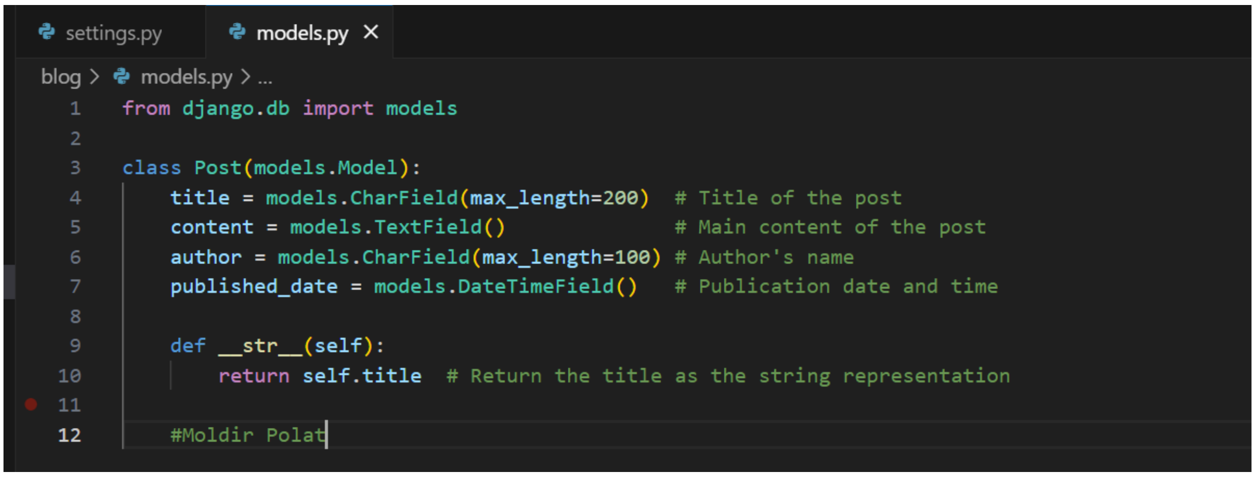
Expected outcome: Post model will be created with fields such as title, author, content, published date, and it will be available for further usage. The class Post will have a method that returns string representation of the post (here the title will be a representation).

Description of the implementation steps: I created new Django project and new Django application, registered new app in settings of the project.



```python
# SECURITY WARNING: don't run with debug turned on in production!
DEBUG = True

ALLOWED_HOSTS = []


# Application definition

INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'blog'
]
```

Then in models.py file I defined the class Post with all properties and methods. Post class has title (stores the title of the post), content (main content), author (author's name), published_date (when the post was published). Each Post is represented by its title, so the method __str__ returns a string representation of the Post, making it easier to identify each post in the Django admin or when displaying in templates.



```python
from django.db import models

class Post(models.Model):
    title = models.CharField(max_length=200)  # Title of the post
    content = models.TextField()              # Main content of the post
    author = models.CharField(max_length=100) # Author's name
    published_date = models.DateTimeField()   # Publication date and time

    def __str__(self):
        return self.title  # Return the title as the string representation

    #Moldir Polat
```

Results: I created the Django project and django app, where I have defined the class Post with all necessary fields, and method to represent each post in the form of its title.

Challenges: no challenges.

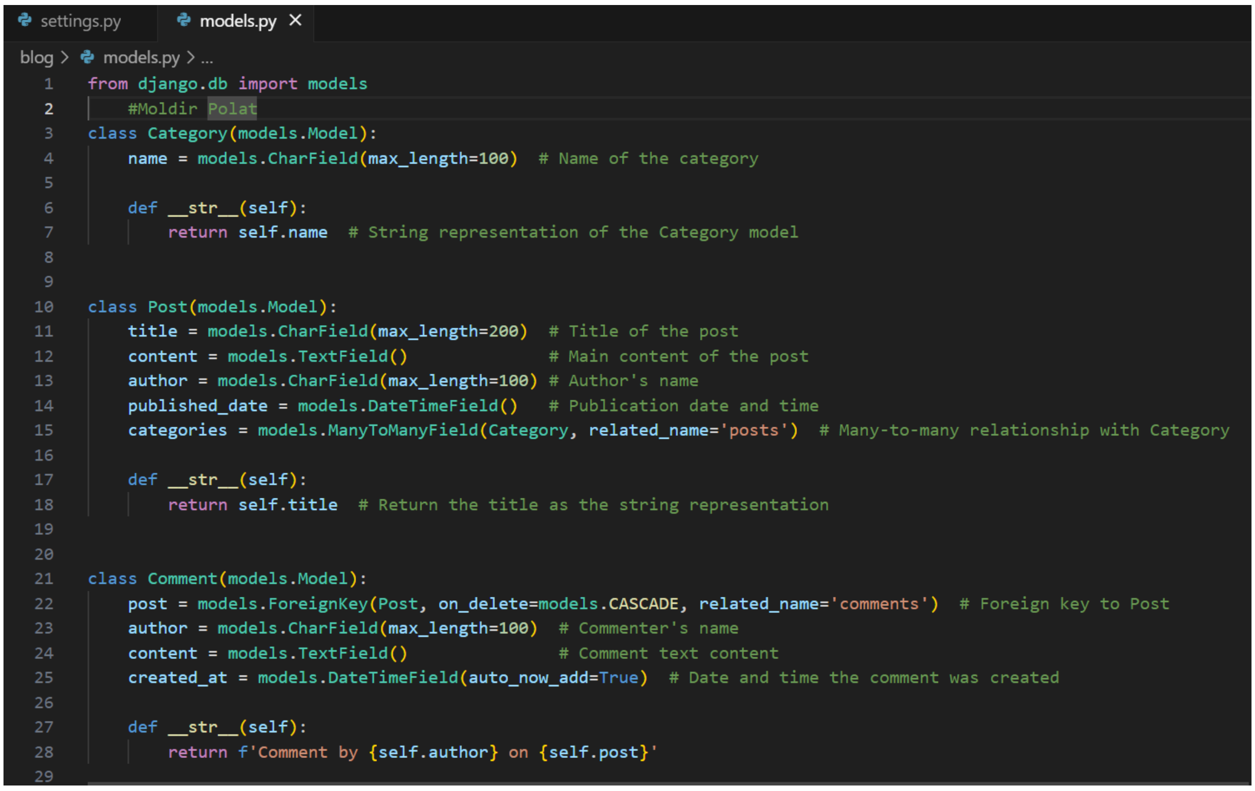
Exercise 2

Exercise title: Model Relationships

Objective: to expand the functionality of the Post model by introducing a Category model with a many-to-many relationship, allowing each post to belong to multiple categories. Additionally, a Comment model will be created, which will have a foreign key relationship with the Post model to store comments connected with each post.

Expected outcome: as an outcome each Post can be associated with multiple Category instances, and each Category can have multiple Post instances. Comments can be added to posts, so each Post will have many Comments what will have foreight key of the corresponding Post.

Description of the implementation steps: I created new model Category with a single field name that stores the name of the category. I added categories to Post model to create many-to-many relationship with Category model, so each Post can belong to several categories. The n I defined Comment model with following fields – author (name of the commenter), content (text content of the comment), created_at(when comment was created), post(foreign key of the Post, one-to-many relationship between Post and Comments).


settings.py   models.py ×

blog > models.py > ...

```
1   from django.db import models
2       #Moldir Polat
3   class Category(models.Model):
4       name = models.CharField(max_length=100)  # Name of the category
5
6       def __str__(self):
7           return self.name  # String representation of the Category model
8
9
10  class Post(models.Model):
11      title = models.CharField(max_length=200)  # Title of the post
12      content = models.TextField()              # Main content of the post
13      author = models.CharField(max_length=100) # Author's name
14      published_date = models.DateTimeField()   # Publication date and time
15      categories = models.ManyToManyField(Category, related_name='posts')  # Many-to-many relationship with Category
16
17      def __str__(self):
18          return self.title  # Return the title as the string representation
19
20
21  class Comment(models.Model):
22      post = models.ForeignKey(Post, on_delete=models.CASCADE, related_name='comments')  # Foreign key to Post
23      author = models.CharField(max_length=100)  # Commenter's name
24      content = models.TextField()               # Comment text content
25      created_at = models.DateTimeField(auto_now_add=True)  # Date and time the comment was created
26
27      def __str__(self):
28          return f'Comment by {self.author} on {self.post}'
29
```

Need to pay attention to Line 15, where *categories* is defined as a *ManyToManyField* pointing to category. The *related name='posts'* means that we can access all posts for a given category using *category.posts.all()*. Similarly, in the Line 22 *post* is ForeignKey to Post, established one-to-many relationship. The option related_name='comments' let us access to a post's comments using *post.comment.all()*.

Results: The Post, Category and Comment models were successfully created with specified relationships. There is many-to-many relationship between Category and Post, then one-to-many relationship between Post and Comment models.

Challenges: no challenges.

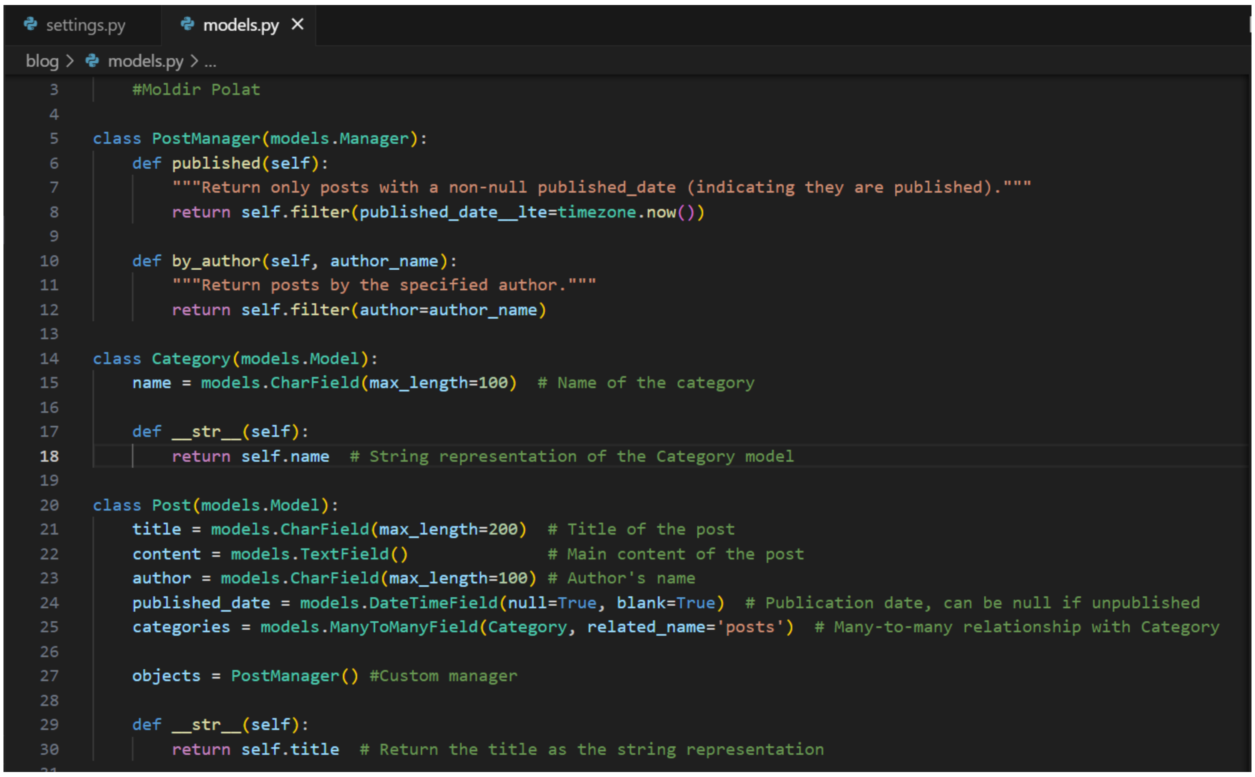
Exercise 3

Exercise title: Custom Manager

Objective: to create a custom manager for the Post model that filters and returns only published posts. Additionally, it will include method to retrieve posts by a author name.

Expected outcome: as an outcome I will have custom manager for Post model, and using *post.objects.published()* I can fetch published posts. Using *Post.objects,by_author('author-name')* I can get all posts of specific author.

Description of the implementation steps: I created new class *PostManager* that inherits from models.Manager. It includes two methods *published()* – returns posts that have been published, *by_author(author_name)* – returns posts that math a certain author's name. In the first method published_date should be set to a date/time that is less then or equal to the current time, and in the second it compares author_name and current post's author's name. I updated Line 24 *published_date* can be null.



```
 3      #Moldir Polat
 4
 5  class PostManager(models.Manager):
 6      def published(self):
 7          """Return only posts with a non-null published_date (indicating they are published)."""
 8          return self.filter(published_date__lte=timezone.now())
 9
10      def by_author(self, author_name):
11          """Return posts by the specified author."""
12          return self.filter(author=author_name)
13
14  class Category(models.Model):
15      name = models.CharField(max_length=100)  # Name of the category
16
17      def __str__(self):
18          return self.name  # String representation of the Category model
19
20  class Post(models.Model):
21      title = models.CharField(max_length=200)  # Title of the post
22      content = models.TextField()              # Main content of the post
23      author = models.CharField(max_length=100) # Author's name
24      published_date = models.DateTimeField(null=True, blank=True)  # Publication date, can be null if unpublished
25      categories = models.ManyToManyField(Category, related_name='posts')  # Many-to-many relationship with Category
26
27      objects = PostManager() #Custom manager
28
29      def __str__(self):
30          return self.title  # Return the title as the string representation
```

Need to pay attention to Line 27, where I attached an instance of PostManager as the custom manager for the Post model, replacing the default one. Now we can use the new filtering methods directly on the Post.

Results: The custom manager was successfully implemented. I defined two methods, that are used to get published posts, in other words where published_date compared to current time; the second is used used to get one author's posts, by judging the current post's author and the argument.

Challenges: how to properly override the default manager – resolved in Line 27; how to define published posts – published_date can be null, and published_date should be set to a date/time that is less then or equal to the current time.

Django Views

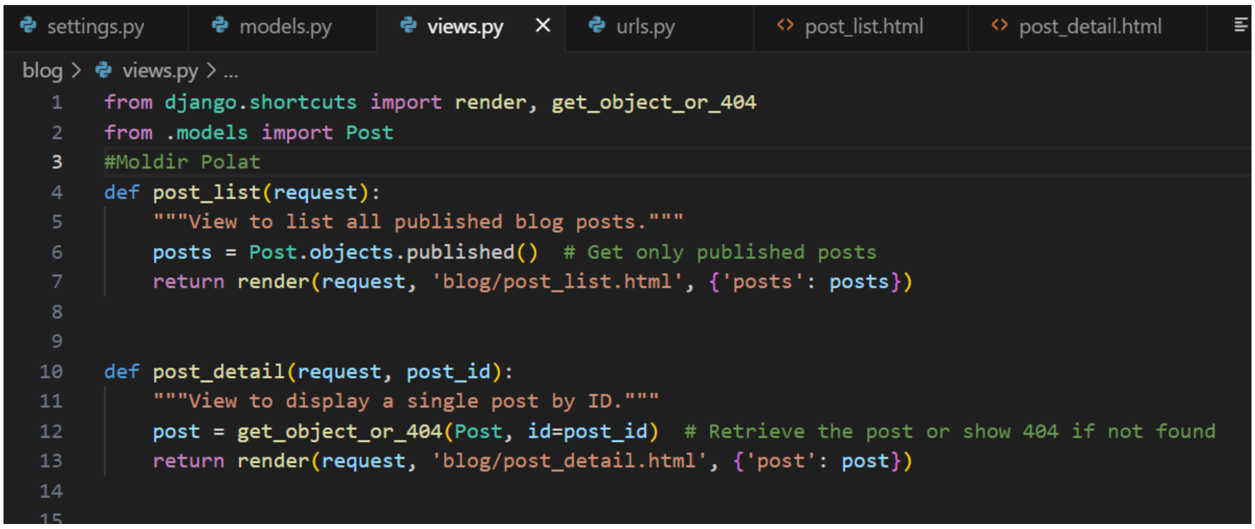
Exercise 4

Exercise title: Function-based views

Objective: to implement function-based views in Django to handle displaying all blog posts in a list format and to display the details of a single post using its ID.

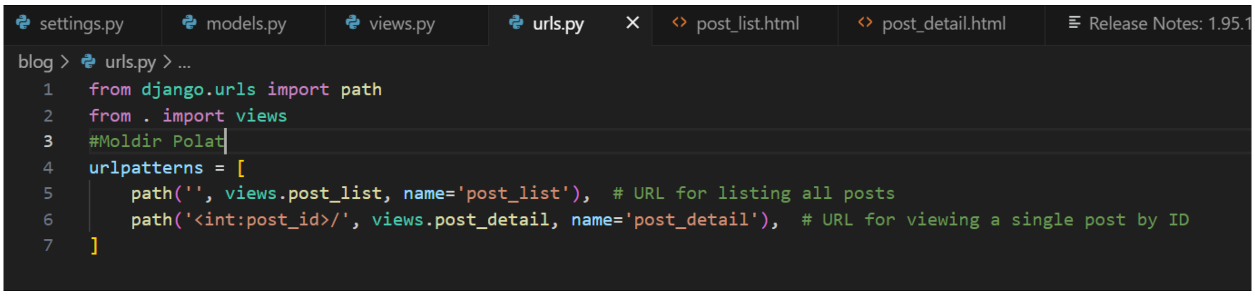
Expected outcome: as an outcome I should be able to get all posts' list with all information(titles, authors, publication dates) through the url. Clicking on the post's title should navigate to post_detail view which will show the full content of the selected post.

Description of the implementation steps: I implemeted a view to list all blog posts, created function-based view called *post_list* in the views.py. I used Post model's custom manager (obket.published()) to retrieve only published posts, then passed the list to a template for rendering. I also created function-based view called *post_detail* where I retrieved Post instance by its ID using *get_object_or_404* to handle cases where the post might not exist. Them passed retrieved posts to a template for rendering.



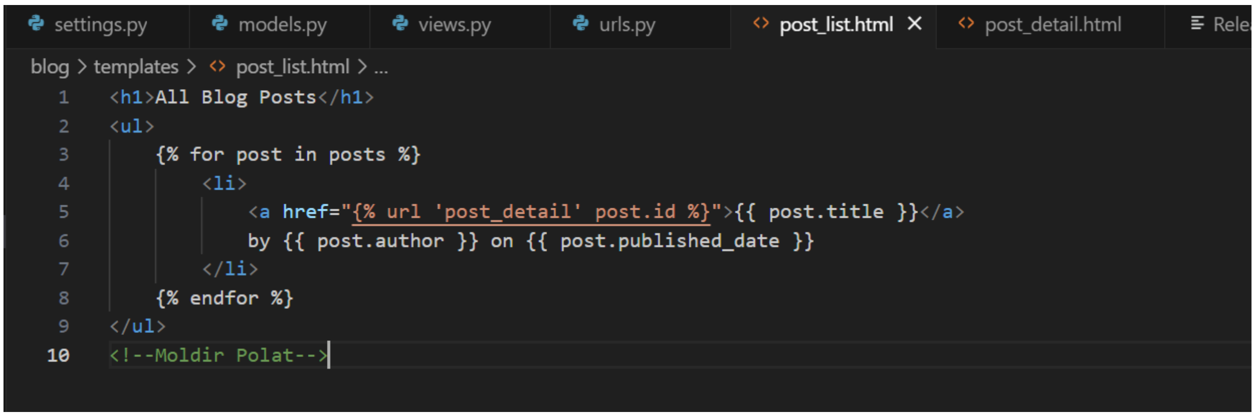
```python
from django.shortcuts import render, get_object_or_404
from .models import Post
#Moldir Polat
def post_list(request):
    """View to list all published blog posts."""
    posts = Post.objects.published()  # Get only published posts
    return render(request, 'blog/post_list.html', {'posts': posts})


def post_detail(request, post_id):
    """View to display a single post by ID."""
    post = get_object_or_404(Post, id=post_id)  # Retrieve the post or show 404 if not found
    return render(request, 'blog/post_detail.html', {'post': post})
```

I set up the urls.py for the blog app, added URL patterns for the post_list and post_detail views. Then I set up a path to display a list of posts and another path to display individual posts by their IDs.

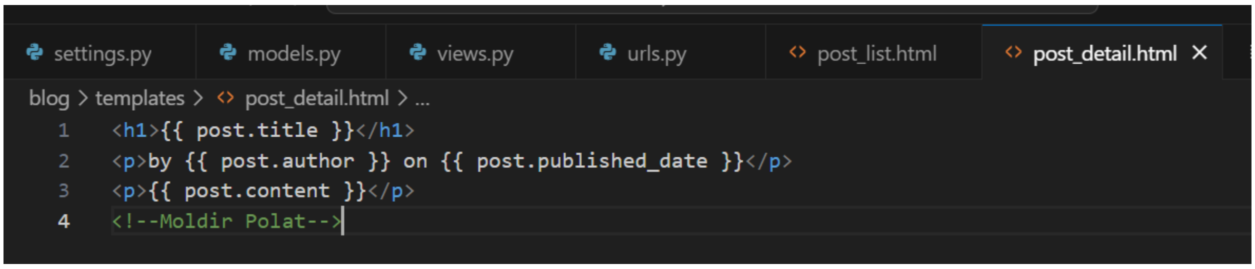


```python
from django.urls import path
from . import views
#Moldir Polat
urlpatterns = [
    path('', views.post_list, name='post_list'),  # URL for listing all posts
    path('<int:post_id>/', views.post_detail, name='post_detail'),  # URL for viewing a single post by ID
]
```

For templates I designed two of them: post_list.html and post_detail.html. The first one contains a list of blog posts with basic details. Another one shows the full content of a single post.

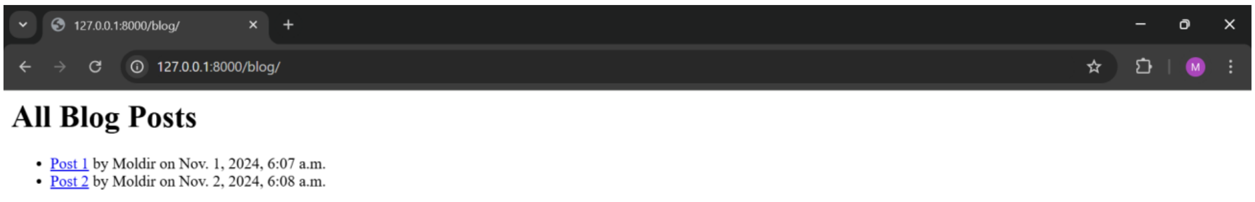


```
<h1>All Blog Posts</h1>
<ul>
    {% for post in posts %}
        <li>
            <a href="{% url 'post_detail' post.id %}">{{ post.title }}</a>
            by {{ post.author }} on {{ post.published_date }}
        </li>
    {% endfor %}
</ul>
<!--Moldir Polat-->
```



```
<h1>{{ post.title }}</h1>
<p>by {{ post.author }} on {{ post.published_date }}</p>
<p>{{ post.content }}</p>
<!--Moldir Polat-->
```

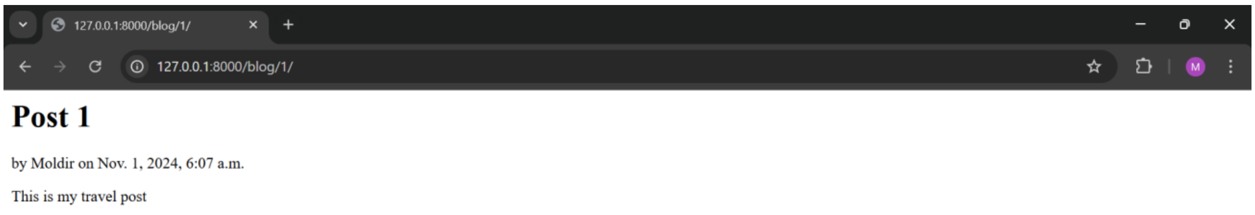
Results: after applying all migrations of models and adding super user credentials, I added some test data through admin page. Then U run server and verified my implementation navigating to the url *http://127.0.0.1:8000/blog/*.



All Blog Posts

- Post 1 by Moldir on Nov. 1, 2024, 6:07 a.m.
- Post 2 by Moldir on Nov. 2, 2024, 6:08 a.m.

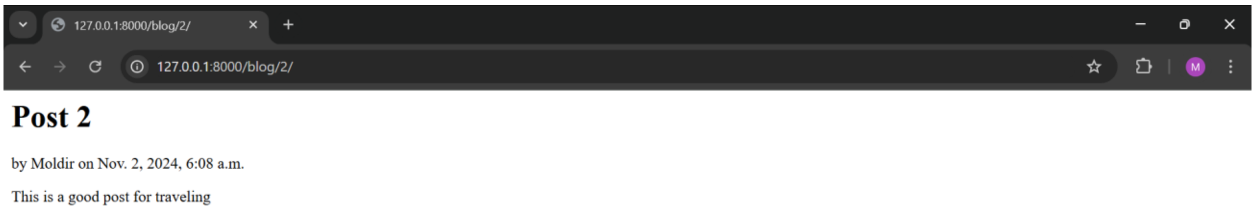
Then clicking on the post, it displayed the detailed content of it.



Post 1

by Moldir on Nov. 1, 2024, 6:07 a.m.

This is my travel post



Post 2

by Moldir on Nov. 2, 2024, 6:08 a.m.

This is a good post for traveling

So the function-based views were implememted successfully. Testing post_list view confirmed that we can get published posts, while post_detail view provided the complete information of the post. Both templates worked as expected with accurate data.

Challenges: the correct path syntax for post_detail in the post_list.html template - *{% url 'post_detail' post.id %}* was a solution.

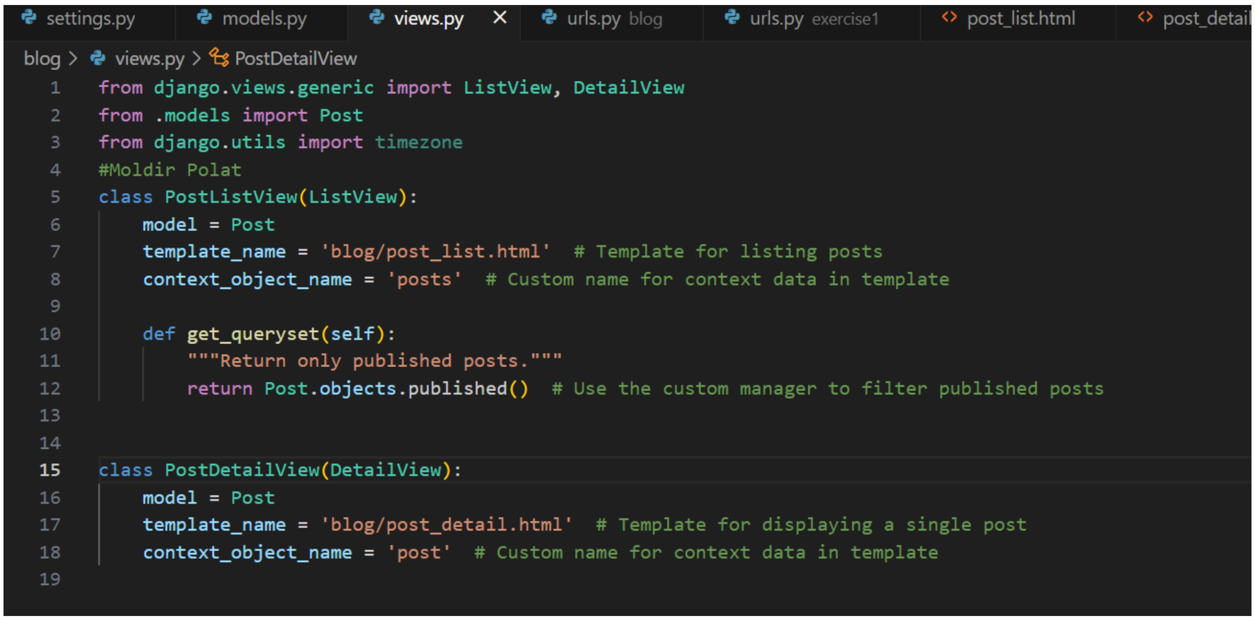
Exercise 5

Exercise title: Class-Based views

Objective: to refactor the function-based views from previous exercise into Django's class-based views using ListView for listing all blog posts and DetailView for displaying single post.

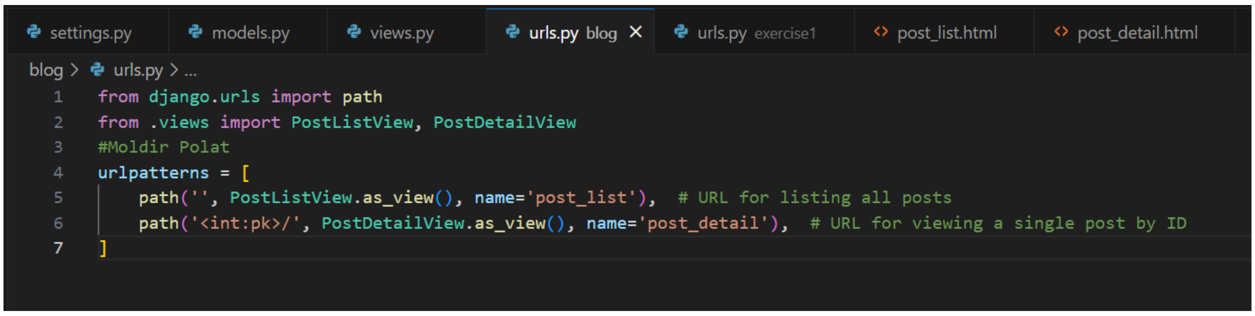
Expected outcome: as an outcome I should be able to get all posts' list with all information(titles, authors, publication dates) through the url. Clicking on the post's title should navigate to post_detail view which will show the full content of the selected post. However, all of this I should get using PostListView and PostDetailView.

Description of the implementation steps: I created new class PostListView that inherits from ListView, set the model attribute to Post and defined a template_name attribute for rendering the list of posts. I used *get_queryset()* to override the default queryset, filtering only published posts using custom manager. Also created PostDetailView class from DetailView. Set the model attribute to Post and defined a template_name.



```python
from django.views.generic import ListView, DetailView
from .models import Post
from django.utils import timezone
#Moldir Polat
class PostListView(ListView):
    model = Post
    template_name = 'blog/post_list.html'  # Template for listing posts
    context_object_name = 'posts'  # Custom name for context data in template

    def get_queryset(self):
        """Return only published posts."""
        return Post.objects.published()  # Use the custom manager to filter published posts


class PostDetailView(DetailView):
    model = Post
    template_name = 'blog/post_detail.html'  # Template for displaying a single post
    context_object_name = 'post'  # Custom name for context data in template
```

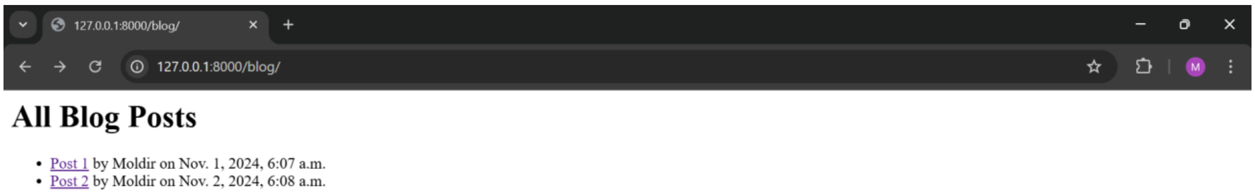
Additionally I updated urls.py to reference the new views. Used the *.as_view()* method to convert each view class into a view function that Django can use.



```python
from django.urls import path
from .views import PostListView, PostDetailView
#Moldir Polat
urlpatterns = [
    path('', PostListView.as_view(), name='post_list'),  # URL for listing all posts
    path('<int:pk>/', PostDetailView.as_view(), name='post_detail'),  # URL for viewing a single post by ID
]
```

Templates from previous exercises remained unchnaged.

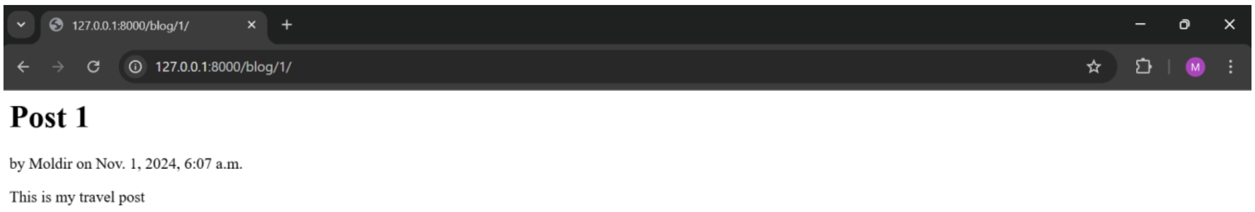
Results: to verify the new implementation I navigated to the url http://127.0.0.1:8000/blog/. The output is the same as in previous exercise, it means new class-based view is working properly.


127.0.0.1:8000/blog/

All Blog Posts

- Post 1 by Moldir on Nov. 1, 2024, 6:07 a.m.
- Post 2 by Moldir on Nov. 2, 2024, 6:08 a.m.

Then clicking on the post, it displayed the detailed content of it.


127.0.0.1:8000/blog/1/

Post 1

by Moldir on Nov. 1, 2024, 6:07 a.m.

This is my travel post

So the class-based views were implememted successfully, and both views were rendered the expected templates. The postListView displayed published posts, while PostDetailView accurately rendered each post’s content when accessed by IDs.

Challenges: ensuring the template variable names matched between the views and templates – solution was setting *context_object_name* for each view.

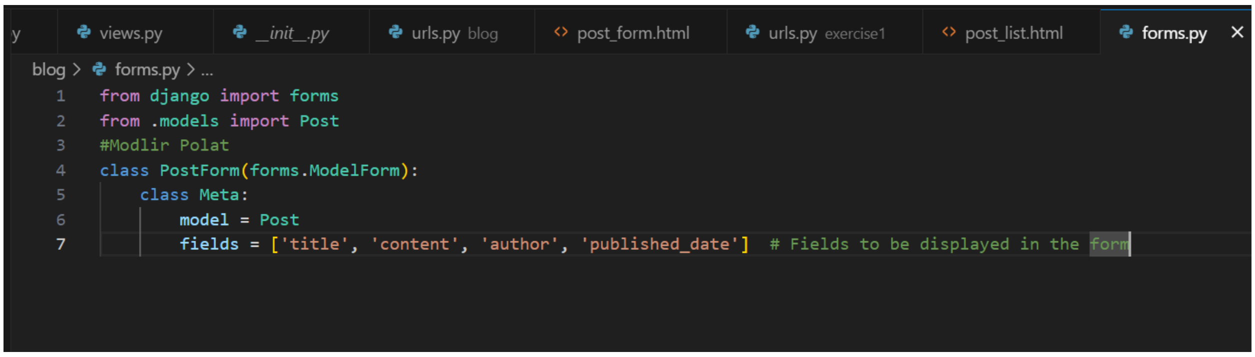
Exercise 6

Exercise title: Handling forms

Objective: to create a form for adding new blog posts using Django’s forms.ModelForm. This form will allow users to submit data for a new blog post, which will be validated and saved to the database.

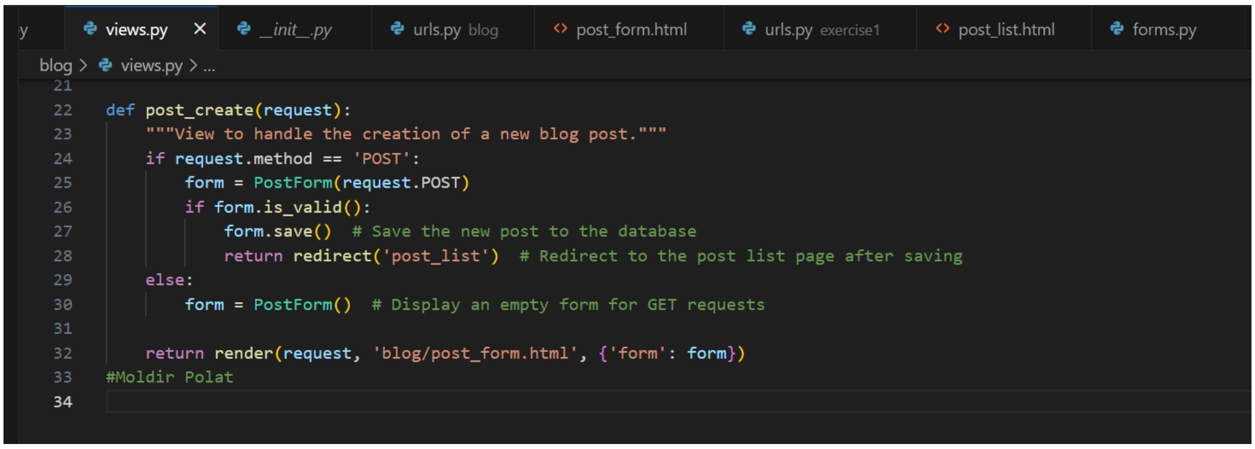
Expected outcome: as an outcome I should be able to add new post navigating to /blog/new/ url. There I should be able to see the form where I can enter all detailsof a new post. Submitting the form with valid data should save the enw post and navigate to lists of all posts. If the form is invalid error message shoud be displayed requiring correct input.

Description of the implementation steps: I created ModelForm for the Post model named PostForm that inherits from forms.ModelForm. I defined the fields in the form that should be displayed and editable by users (title, content, author). I used Django’s Meta class within the form to specify the Post model and the fields to include.



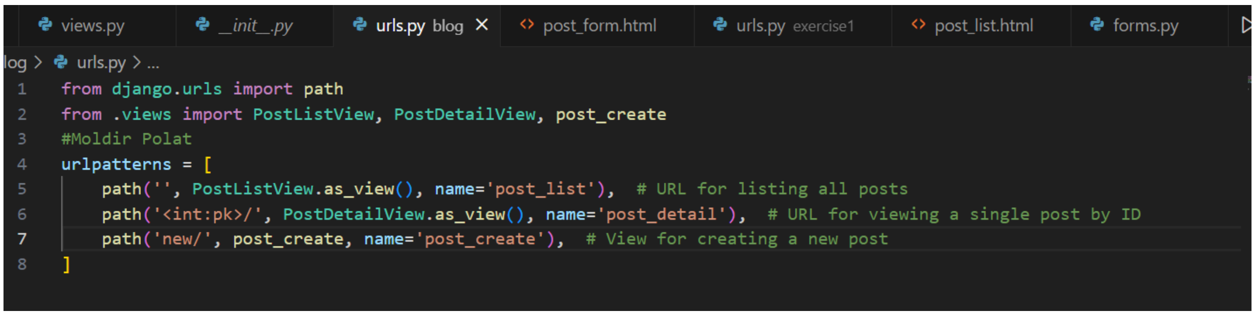
```python
from django import forms
from .models import Post
#Modlir Polat
class PostForm(forms.ModelForm):
    class Meta:
        model = Post
        fields = ['title', 'content', 'author', 'published_date']  # Fields to be displayed in the form
```

In views.py I created a function-based view called *post_create* to handle the display and submission of the form. In the view I checked if the request method is POST. If it is, it validates the form and saves the post if data is valid. If the request method is GET (visiting the form initially), it renders an empty form.



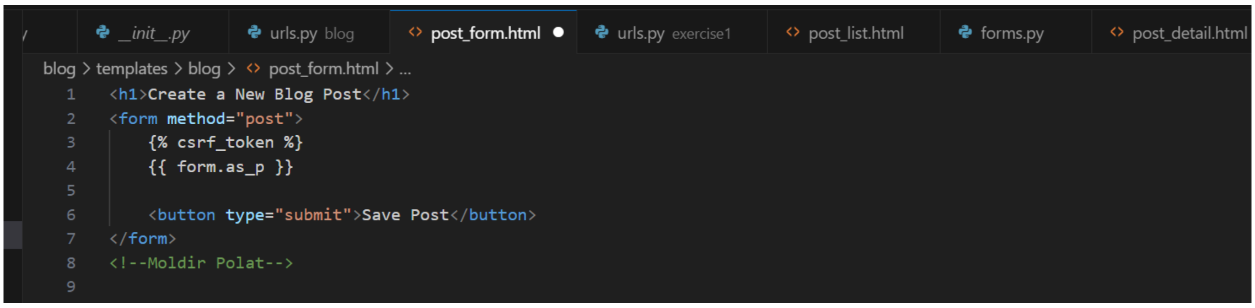
```python
def post_create(request):
    """View to handle the creation of a new blog post."""
    if request.method == 'POST':
        form = PostForm(request.POST)
        if form.is_valid():
            form.save()  # Save the new post to the database
            return redirect('post_list')  # Redirect to the post list page after saving
    else:
        form = PostForm()  # Display an empty form for GET requests

    return render(request, 'blog/post_form.html', {'form': form})
#Moldir Polat
```

I updated IRLs to include the new view (Line 7).



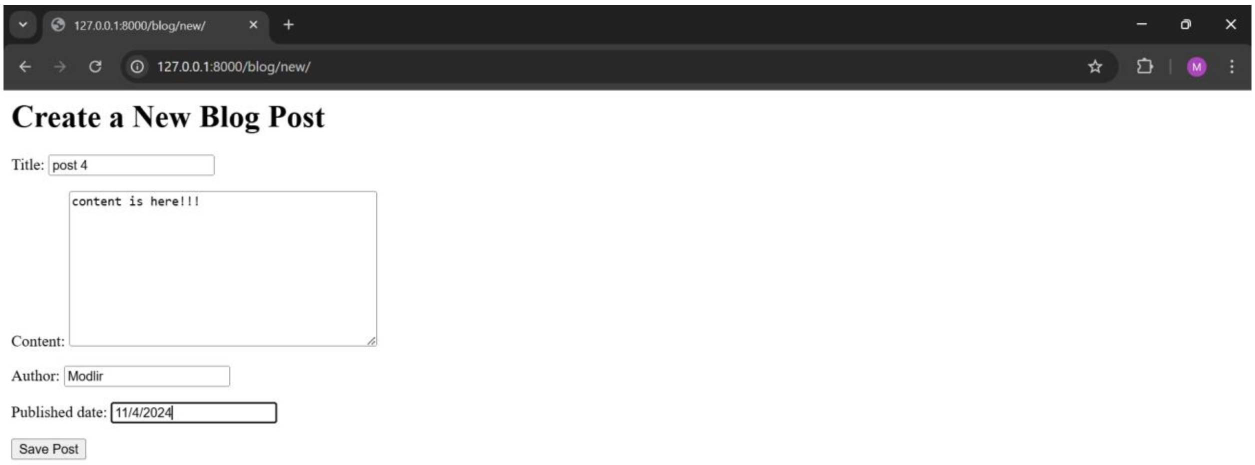
```python
1  from django.urls import path
2  from .views import PostListView, PostDetailView, post_create
3  #Moldir Polat
4  urlpatterns = [
5      path('', PostListView.as_view(), name='post_list'),  # URL for listing all posts
6      path('<int:pk>/', PostDetailView.as_view(), name='post_detail'),  # URL for viewing a single post by ID
7      path('new/', post_create, name='post_create'),  # View for creating a new post
8  ]
```

Additionally I added new template for post form which displays the form, handles error messages, and allows users to submit data. The template renders the form using {{ form.as_p }}which displays each form field as a paragraph, simplifying form layout. The csrf token is included to protect against cross-site request forgery.



```html
<h1>Create a New Blog Post</h1>
<form method="post">
    {% csrf_token %}
    {{ form.as_p }}

    <button type="submit">Save Post</button>
</form>
<!--Moldir Polat-->
```

Results: to verify the new implementation I navigated to the url http://127.0.0.1:8000/blog/*new.* I can see the form where I am able to input all content.


127.0.0.1:8000/blog/new/

Create a New Blog Post

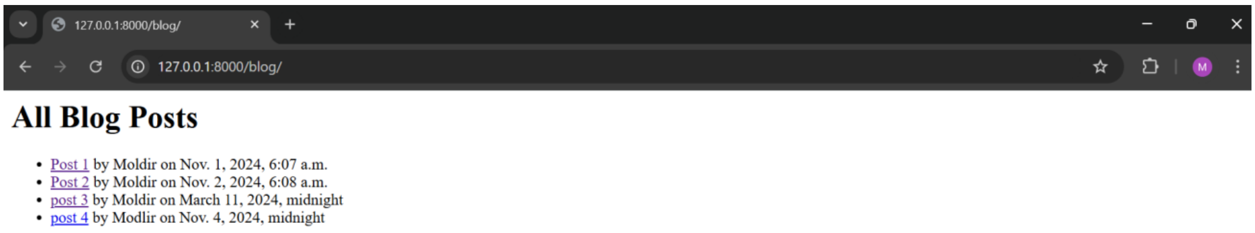
Title: post 4

Content: content is here!!!

Author: Modlir

Published date: 11/4/2024

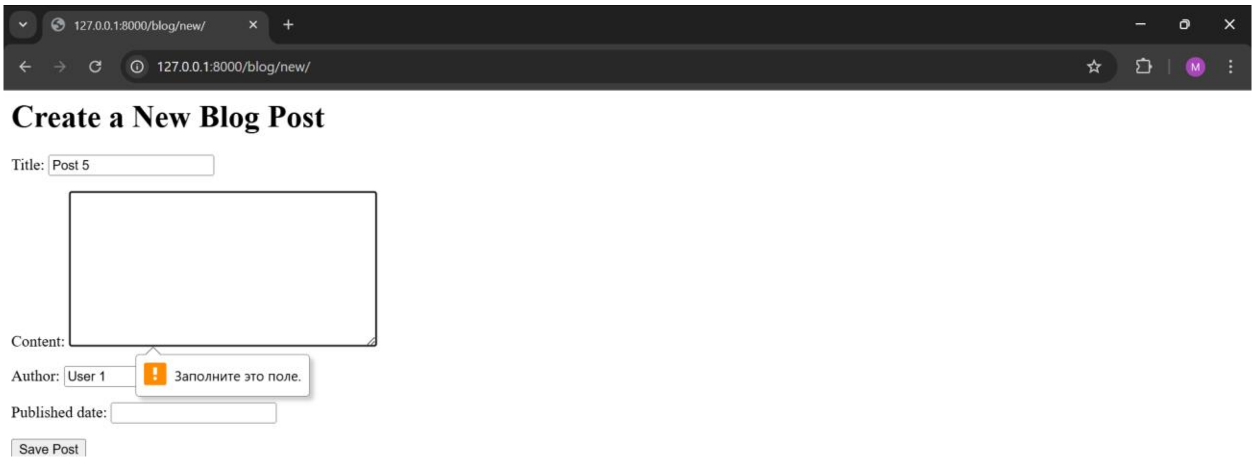
Then clicking on Save Post button, it saves the new post and opens all list of posts.


127.0.0.1:8000/blog/

All Blog Posts

- Post 1 by Moldir on Nov. 1, 2024, 6:07 a.m.
- Post 2 by Moldir on Nov. 2, 2024, 6:08 a.m.
- post 3 by Moldir on March 11, 2024, midnight
- post 4 by Modlir on Nov. 4, 2024, midnight

The validation is also working properly. When content is empty, it requires to fill it up.


127.0.0.1:8000/blog/new/

Create a New Blog Post

Title: Post 5

Content:

Author: User 1

Заполните это поле.

Published date:

So the form and view were successfully implemented. Testing showed that users can access the creation form, fill in the required fields, and submit data. Upon submission, valid data was saved to the databse, and the user was redirested to post list. Invalid data triggered validation errors.

Challenges: redirecting to the list of posts after submission – Django's *redirect()* function was a solution.

Django Templates

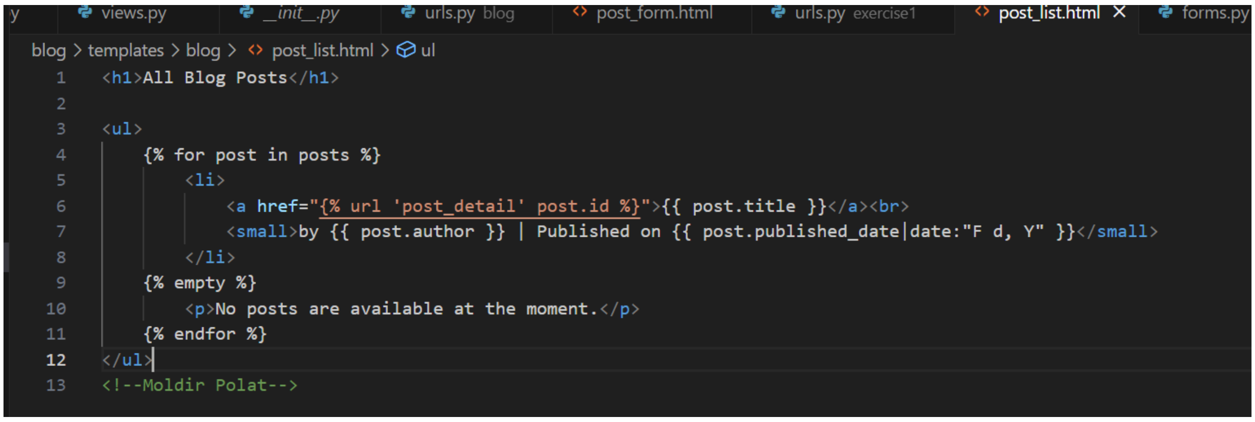
Exercise 7

Exercise title: Basic Template Rendering

Objective: to create a template that displays a list of blog posts, including the title, author, and publication date of each post. Additionally, I will use Django's template tags to format the publication date for a more user-friendly display.

Expected outcome: as an outcome I should be able to visit /blog/ url where all posts will be listed with all information. If there are no published posts, a message saying "No posts are available at the moment" should be displayed.

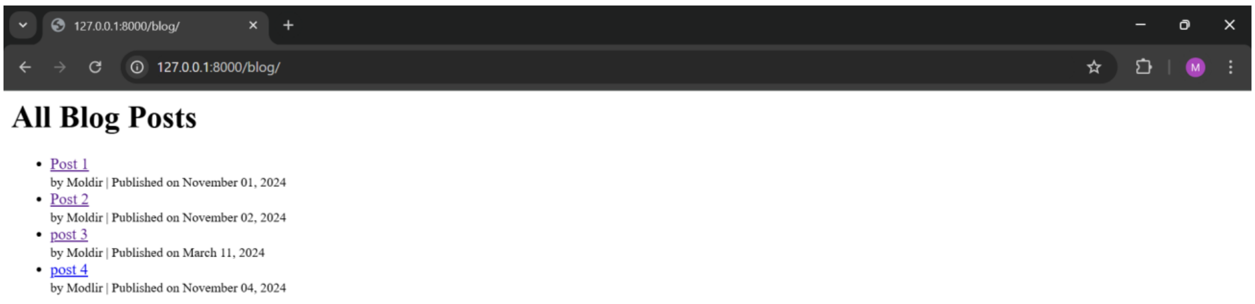
Description of the implementation steps: In the templtes/blog/ directory I modified post_list.html file. Here I have structured the page and looped through each post to display its title, author, and formatted publication date. I used Djnago's date template filter to format the published_date in a readable format.



```
1   <h1>All Blog Posts</h1>
2
3   <ul>
4       {% for post in posts %}
5           <li>
6               <a href="{% url 'post_detail' post.id %}">{{ post.title }}</a><br>
7               <small>by {{ post.author }} | Published on {{ post.published_date|date:"F d, Y" }}</small>
8           </li>
9       {% empty %}
10          <p>No posts are available at the moment.</p>
11      {% endfor %}
12  </ul>
13  <!--Moldir Polat-->
```

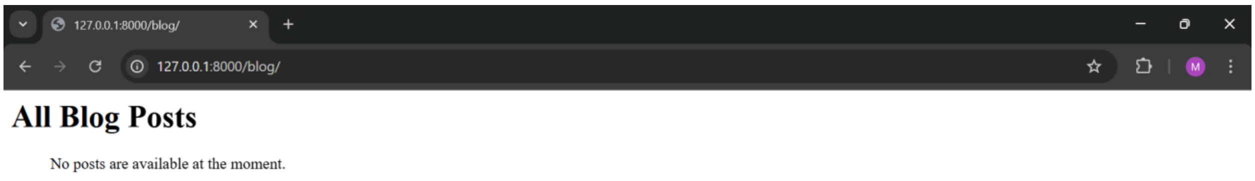
Need to pay attention to Line 4 where I loop through each post in the posts context variable provided by the PostListView. In Line 6 I generate URL for each post's detail page, making them clickable. Date template filter is used in Line 7 to format published_date accordingly. Line 9 means that if there are no posts to display, it will render a part with message.

Results: to verify the new implementation I navigated to the url *http://127.0.0.1:8000/blog/*. I can see the list of all posts with formatted published date. I tried to delete all posts, then it rendered the message saying that no posts are available at the moment.



All Blog Posts

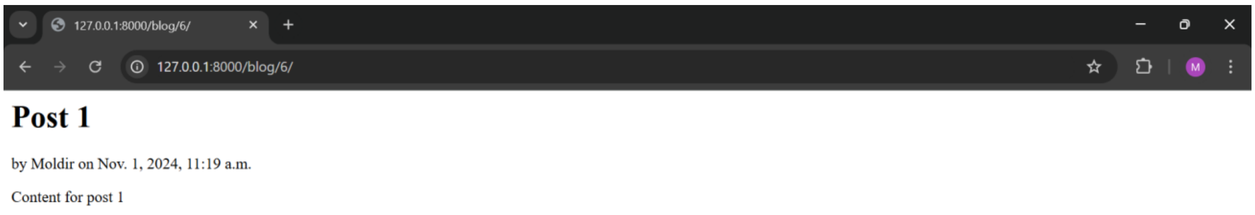
- Post 1
  by Moldir | Published on November 01, 2024
- Post 2
  by Moldir | Published on November 02, 2024
- post 3
  by Moldir | Published on March 11, 2024
- post 4
  by Modlir | Published on November 04, 2024



All Blog Posts

No posts are available at the moment.

When clicking to the post title, it opens the detail page of the post.


127.0.0.1:8000/blog/6/

Post 1

by Moldir on Nov. 1, 2024, 11:19 a.m.

Content for post 1

So the post_list.html template was implemented successfully. Testing showed that all published blog posts were displayed with correctly formatted publication dates. The titles appeared as clickable links that navigate users to each post's detail page. When no posts are in databse, the empty message displayed as expected.

Challenges: no challenges.

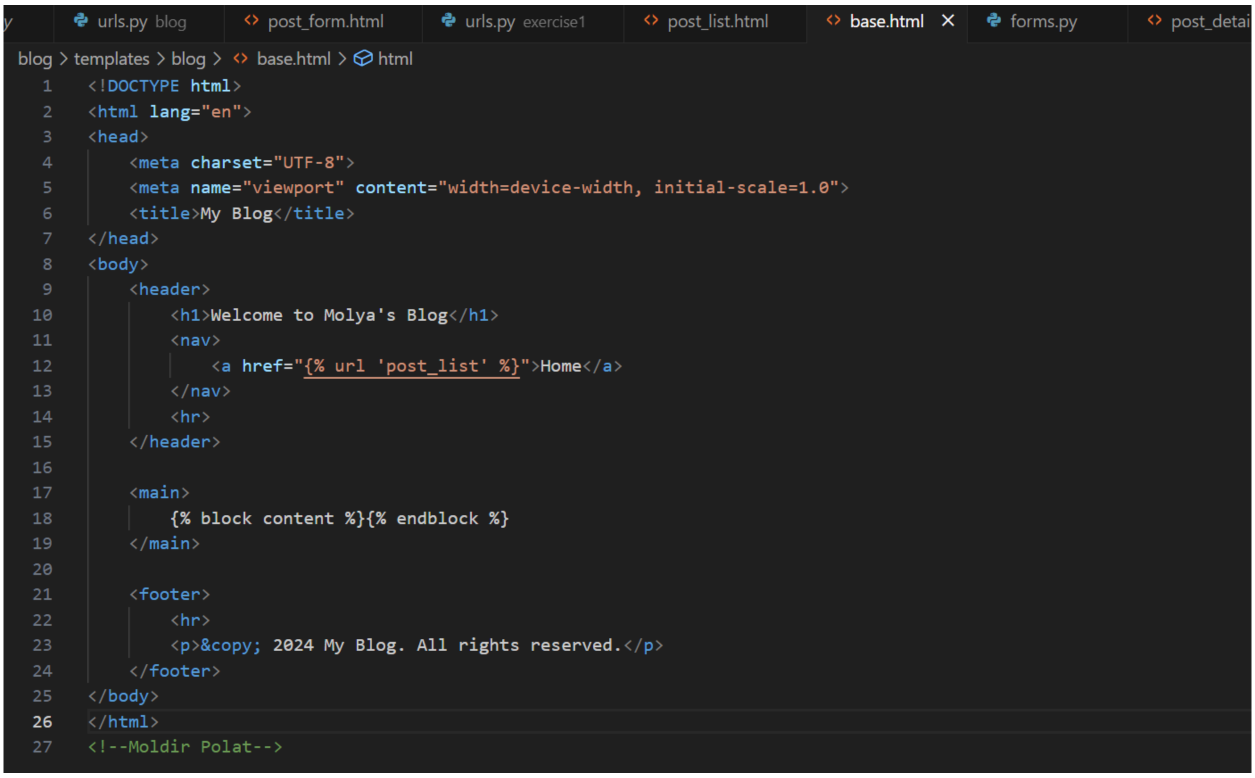
Exercise 8

Exercise title: Template Inheritance

Objective: to create a base template with a header and footer that will be used across multiple pages. I will use Django's template inheritance to extend this base template in the list (post_list.html) and detail (post_detail.html) view templates.

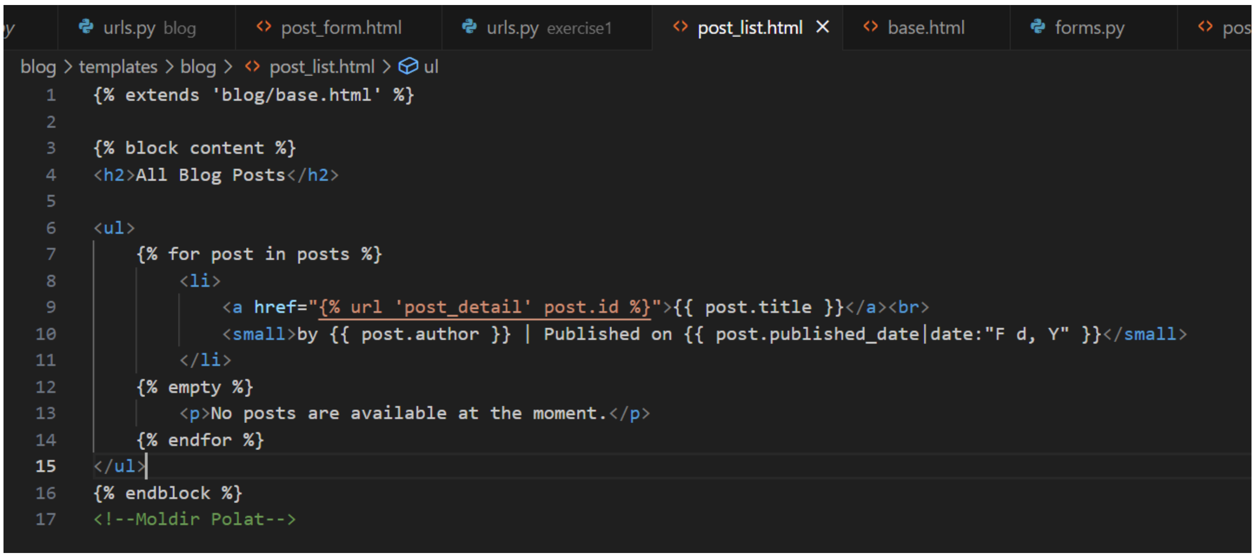
Expected outcome: as an outcome both the list and detail views should display the header, footer defined in base.html. Each page's specific content should be displayed within the base template, under the header and above the footer.

Description of the implementation steps: At first I created base template base.html. I added header, footer and a blocl tag *{% block content %}*. The last tag serves as placeholder where page-specific content will be inserted. Each specific template uses this block *{% block content %}{% endblock %}* to define the main content for that page.

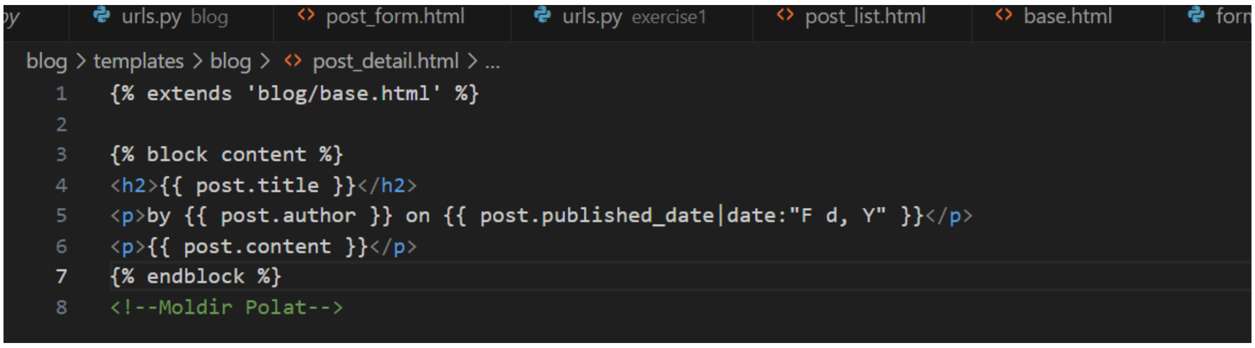

blog > templates > blog > base.html > html

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>My Blog</title>
</head>
<body>
    <header>
        <h1>Welcome to Molya's Blog</h1>
        <nav>
            <a href="{% url 'post_list' %}">Home</a>
        </nav>
        <hr>
    </header>

    <main>
        {% block content %}{% endblock %}
    </main>

    <footer>
        <hr>
        <p>&copy; 2024 My Blog. All rights reserved.</p>
    </footer>
</body>
</html>
<!--Moldir Polat-->
```

Next thing is in post_list,html and post_detail.html I used Django's *{% extends %}* template tag to inherit from base.html. I wrapped the main content of each template in *{% block content %}* and *{% endblock %}* tags so it was inserted into the base template's content block.

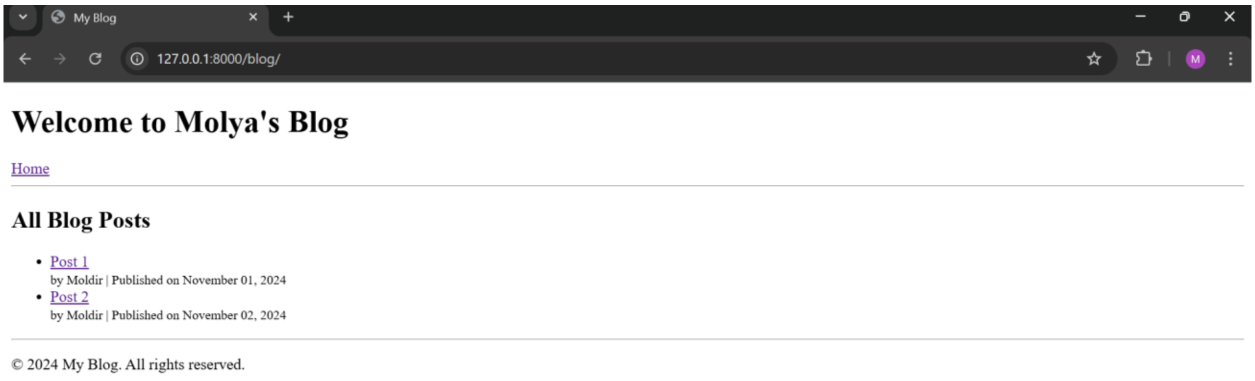


```
{% extends 'blog/base.html' %}

{% block content %}
<h2>All Blog Posts</h2>

<ul>
    {% for post in posts %}
        <li>
            <a href="{% url 'post_detail' post.id %}">{{ post.title }}</a><br>
            <small>by {{ post.author }} | Published on {{ post.published_date|date:"F d, Y" }}</small>
        </li>
    {% empty %}
        <p>No posts are available at the moment.</p>
    {% endfor %}
</ul>
{% endblock %}
<!--Moldir Polat-->
```



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

{% block content %}
<h2>{{ post.title }}</h2>
<p>by {{ post.author }} on {{ post.published_date|date:"F d, Y" }}</p>
<p>{{ post.content }}</p>
{% endblock %}
<!--Moldir Polat-->
```

Results: to verify the new implementation I navigated to the url *http://127.0.0.1:8000/blog/*. I can see the list of all posts with header on the top and and footer on the bottom.



Welcome to Molya's Blog

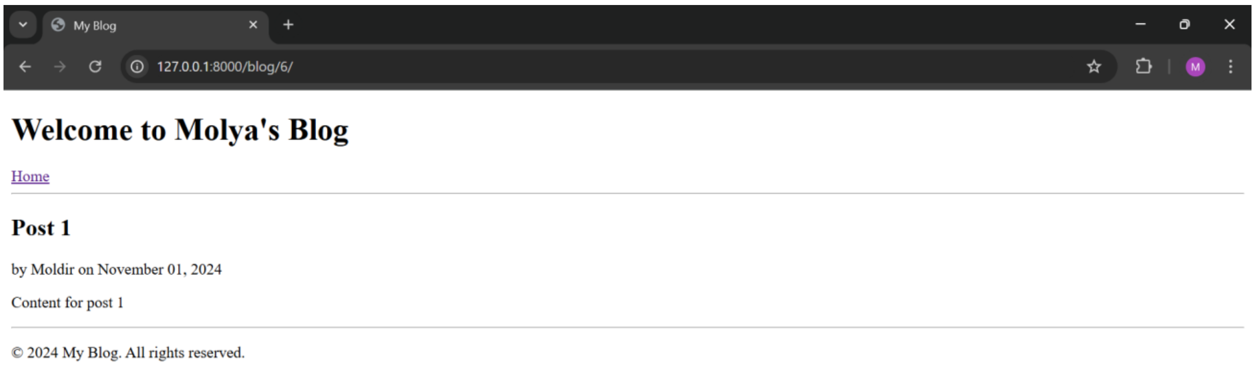
Home

All Blog Posts

- Post 1
  by Moldir | Published on November 01, 2024
- Post 2
  by Moldir | Published on November 02, 2024

© 2024 My Blog. All rights reserved.

Then if I click on separate post, I can see the detail page with header on the top and footer on the bottom.


My Blog

127.0.0.1:8000/blog/6/

Welcome to Molya's Blog

Home

Post 1

by Moldir on November 01, 2024

Content for post 1

© 2024 My Blog. All rights reserved.

So template inheritance was implemented successfully. Testing confirmed that both list and detail views displayed the header and footer from base template, and the main content was rendered correctly within the base template's structure.

Challenges: no challenges.

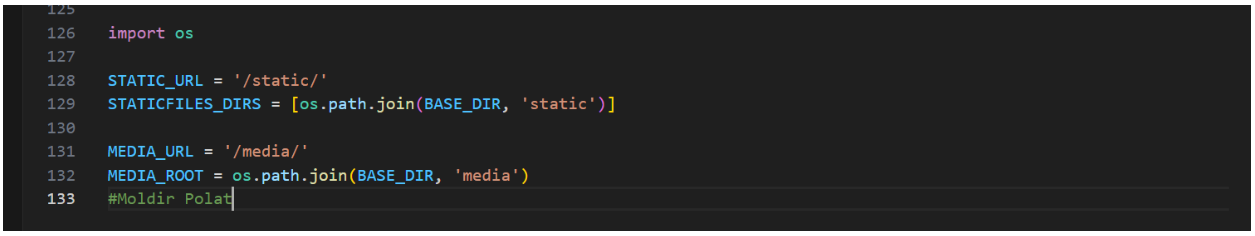
Exercise 9

Exercise title: Static Files and Media

Objective: to enhance the appearance ogf the blog by adding CSS styling using static files and to configure a media folder for user-uploaded images. This will allow users to upload images for each blog post and display those images in the post templates.

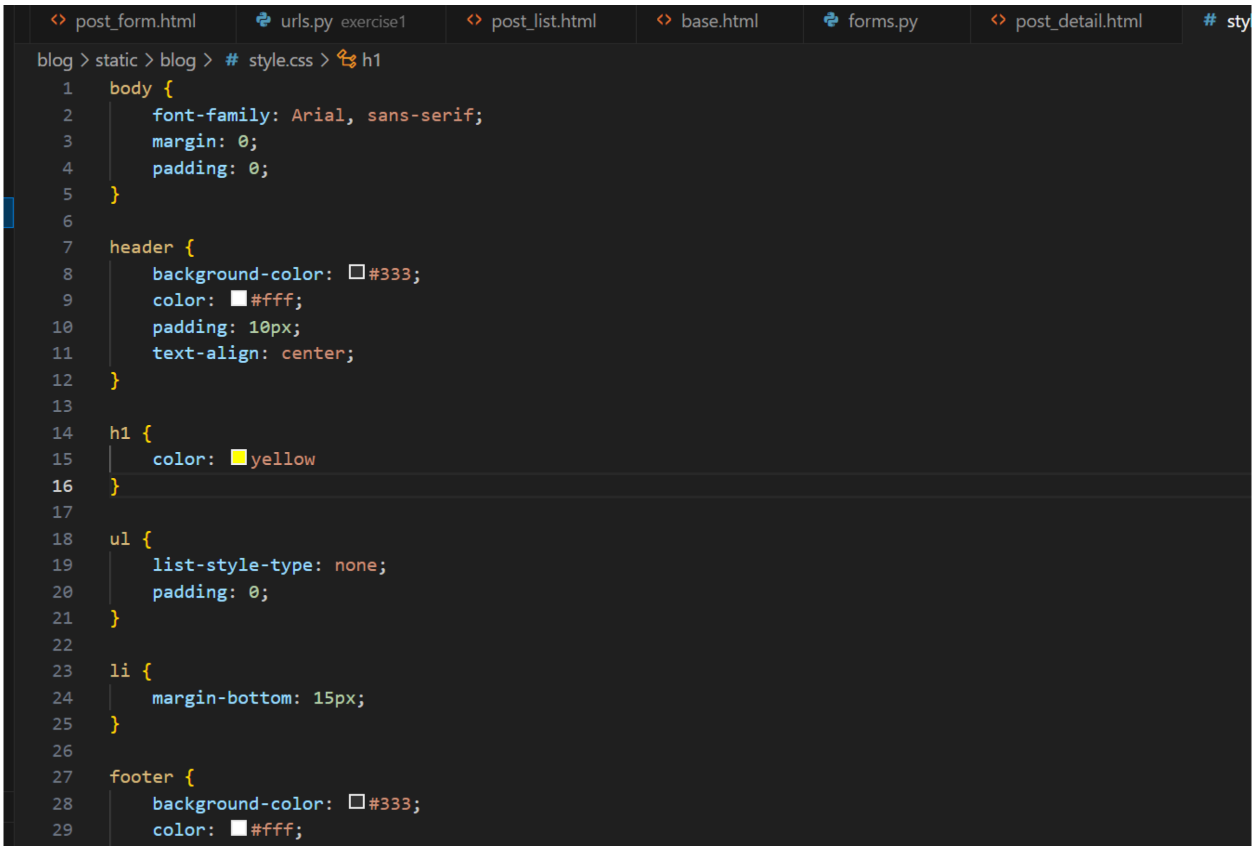
Expected outcome: as an outcome visiting any page should show the styling from style.css. Posts with uploaded images should display those images in both the list and detail views. The images should be stored in the media/post_images/ directory.

Description of the implementation steps: I create a static folder within the blog app directory to store CSS files. In settings.py, I ensured the STATIC_URL and STATICFILES_DIRS settings are configured to serve static files. I also configured the MEDIA_URL and MEDIA_ROOT settings to handle media files.



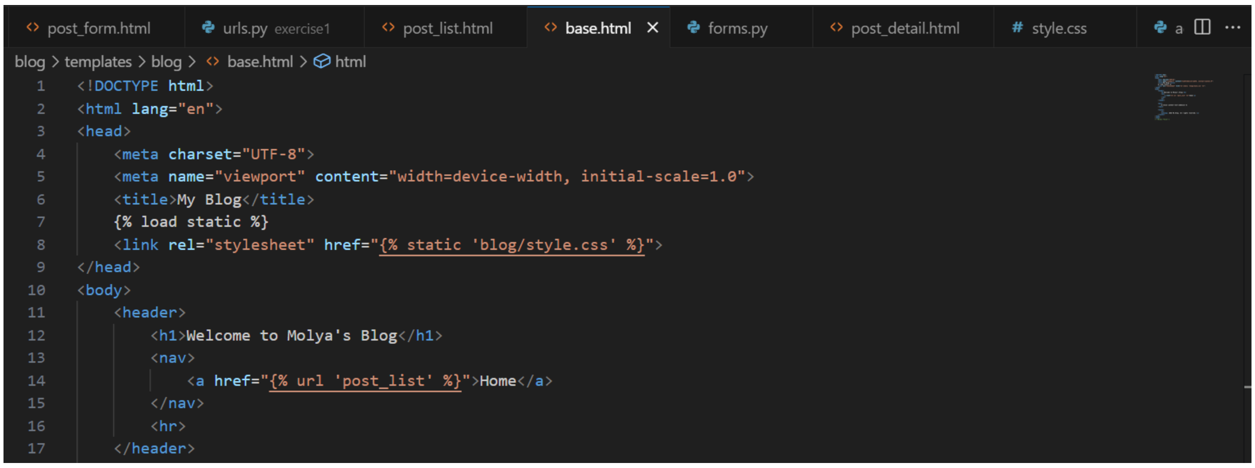
```python
import os

STATIC_URL = '/static/'
STATICFILES_DIRS = [os.path.join(BASE_DIR, 'static')]

MEDIA_URL = '/media/'
MEDIA_ROOT = os.path.join(BASE_DIR, 'media')
#Moldir Polat
```

Then I created a CSS file (style.css) within the static/blog/ folder to define styles for the templates.



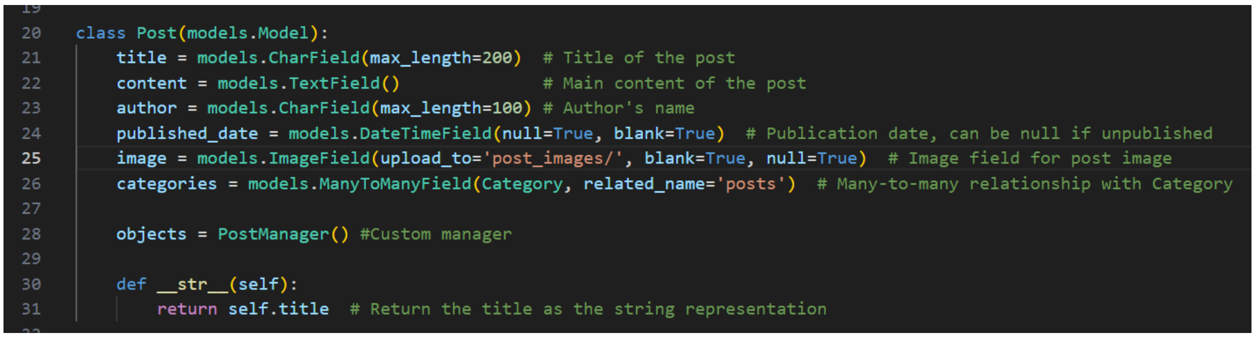
```css
body {
    font-family: Arial, sans-serif;
    margin: 0;
    padding: 0;
}

header {
    background-color: #333;
    color: #fff;
    padding: 10px;
    text-align: center;
}

h1 {
    color: yellow
}

ul {
    list-style-type: none;
    padding: 0;
}

li {
    margin-bottom: 15px;
}

footer {
    background-color: #333;
    color: #fff;
```

In base.html, I used Django's *{% load static %}* template tag to load the CSS file. Then I linked the CSS file in the <head> section of base.html to apply styling across all pages.



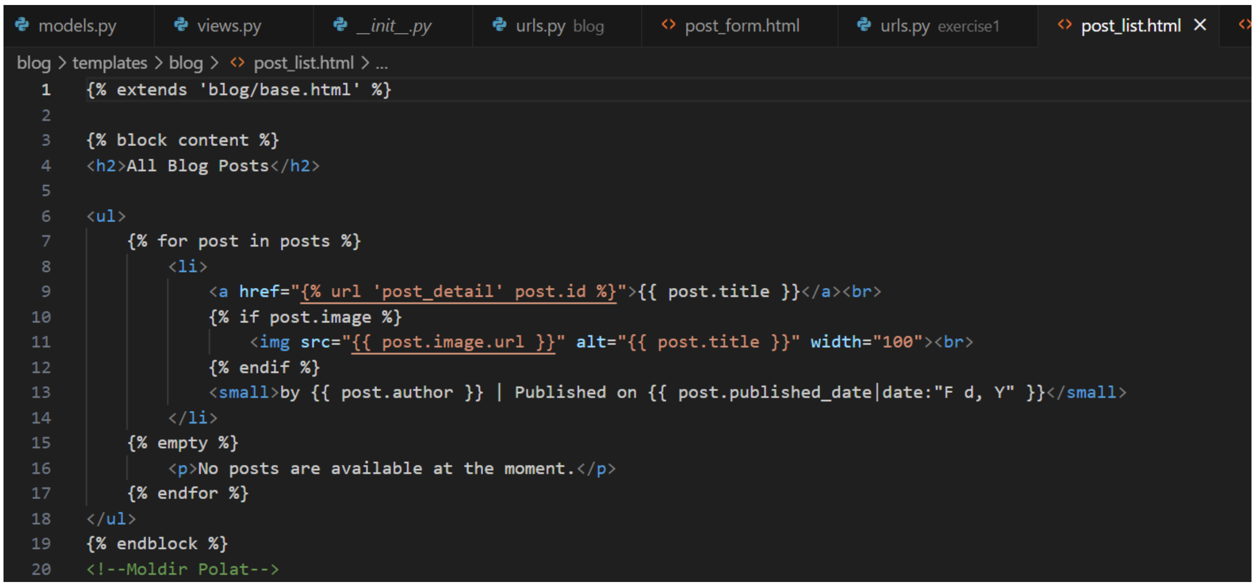
```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>My Blog</title>
    {% load static %}
    <link rel="stylesheet" href="{% static 'blog/style.css' %}">
</head>
<body>
    <header>
        <h1>Welcome to Molya's Blog</h1>
        <nav>
            <a href="{% url 'post_list' %}">Home</a>
        </nav>
        <hr>
    </header>
```

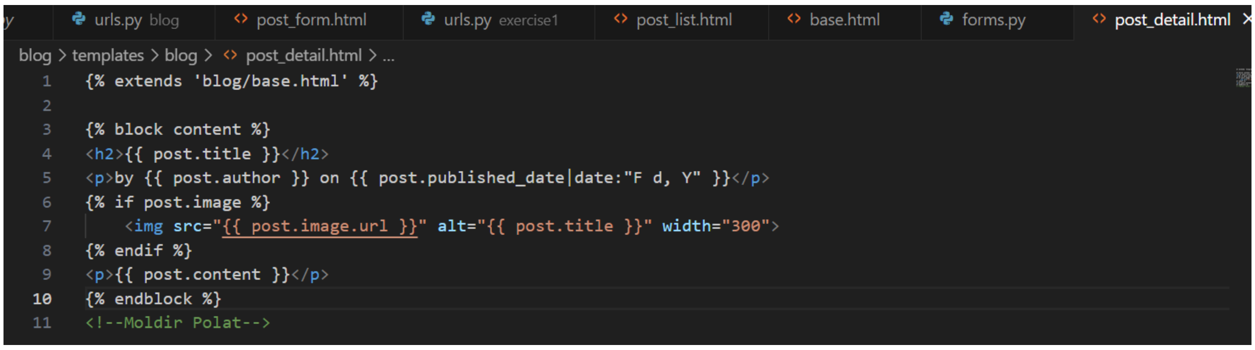
I updated the Post model to include an image field, allowing users to upload an image with each post.

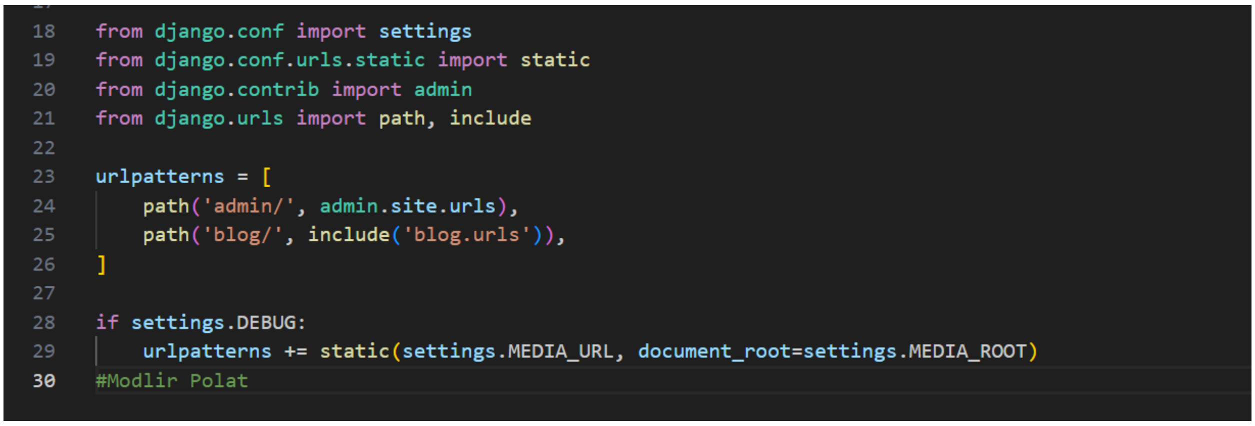


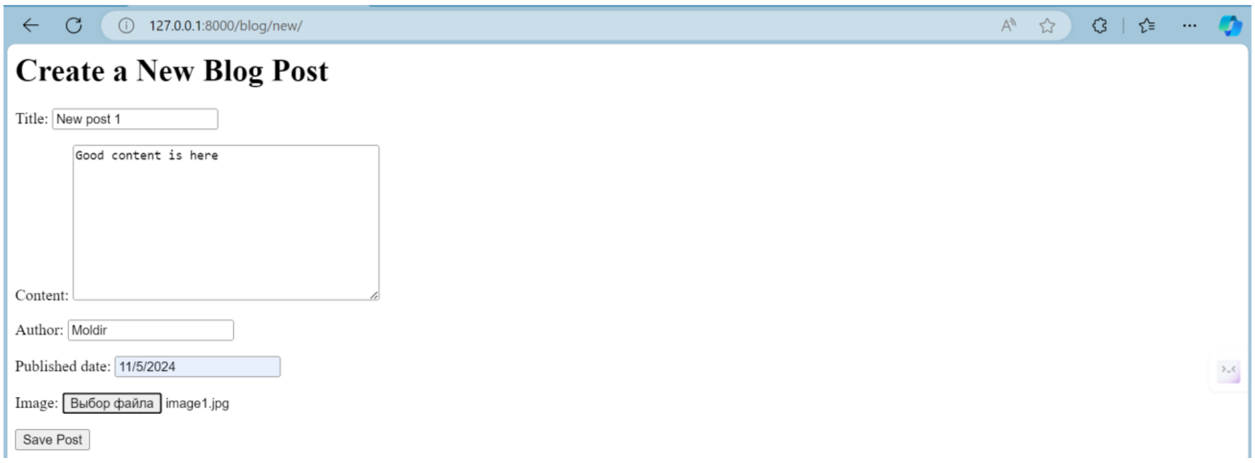
```python
class Post(models.Model):
    title = models.CharField(max_length=200)  # Title of the post
    content = models.TextField()              # Main content of the post
    author = models.CharField(max_length=100) # Author's name
    published_date = models.DateTimeField(null=True, blank=True)  # Publication date, can be null if unpublished
    image = models.ImageField(upload_to='post_images/', blank=True, null=True)  # Image field for post image
    categories = models.ManyToManyField(Category, related_name='posts')  # Many-to-many relationship with Category

    objects = PostManager() #Custom manager

    def __str__(self):
        return self.title  # Return the title as the string representation
```

Next thing is that I updated the PostForm to include the image field. So I modified post_list.html and post_detail.html to display the image for each post if available using *post.image.url*.


```
models.py   views.py   __init__.py   urls.py blog   post_form.html   urls.py exercise1   post_list.html
blog > templates > blog > post_list.html > ...
```
```html
{% extends 'blog/base.html' %}

{% block content %}
<h2>All Blog Posts</h2>

<ul>
    {% for post in posts %}
        <li>
            <a href="{% url 'post_detail' post.id %}">{{ post.title }}</a><br>
            {% if post.image %}
                <img src="{{ post.image.url }}" alt="{{ post.title }}" width="100"><br>
            {% endif %}
            <small>by {{ post.author }} | Published on {{ post.published_date|date:"F d, Y" }}</small>
        </li>
    {% empty %}
        <p>No posts are available at the moment.</p>
    {% endfor %}
</ul>
{% endblock %}
<!--Moldir Polat-->
```


```
urls.py blog   post_form.html   urls.py exercise1   post_list.html   base.html   forms.py   post_detail.html
blog > templates > blog > post_detail.html > ...
```
```html
{% extends 'blog/base.html' %}

{% block content %}
<h2>{{ post.title }}</h2>
<p>by {{ post.author }} on {{ post.published_date|date:"F d, Y" }}</p>
{% if post.image %}
    <img src="{{ post.image.url }}" alt="{{ post.title }}" width="300">
{% endif %}
<p>{{ post.content }}</p>
{% endblock %}
<!--Moldir Polat-->
```
The last step: I updated URLs to Serve Media Files, so in urls.py I configured Django to serve media files during development by adding a urlpatterns entry to handle MEDIA_URL.


```python
from django.conf import settings
from django.conf.urls.static import static
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('blog/', include('blog.urls')),
]

if settings.DEBUG:
    urlpatterns += static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
#Modlir Polat
```
Results: to verify the new implementation I navigated to the url *http://127.0.0.1:8000/blog/new.* It opened for me the form page where I added new Post with image.


127.0.0.1:8000/blog/new/

Create a New Blog Post

Title: New post 1

Content: Good content is here

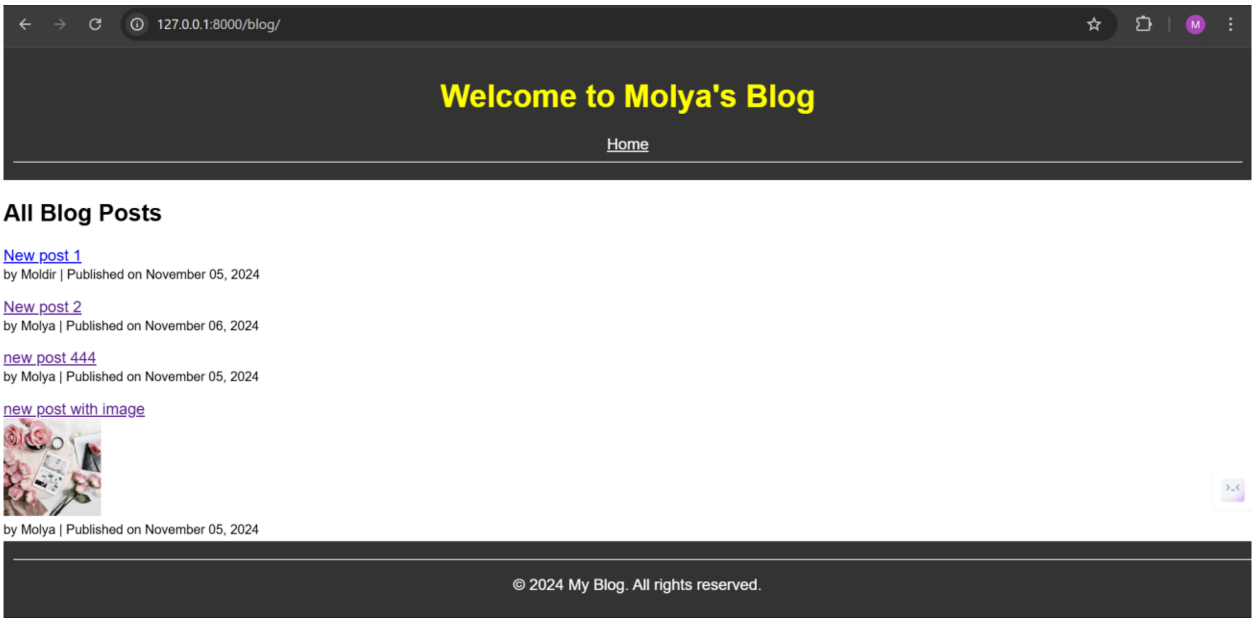
Author: Moldir

Published date: 11/5/2024

Image: Выбор файла image1.jpg

Save Post

So the main page with all posts looks like this. I mostly worked on header's and footer's style.


127.0.0.1:8000/blog/

Welcome to Molya's Blog

Home

All Blog Posts

New post 1
by Moldir | Published on November 05, 2024

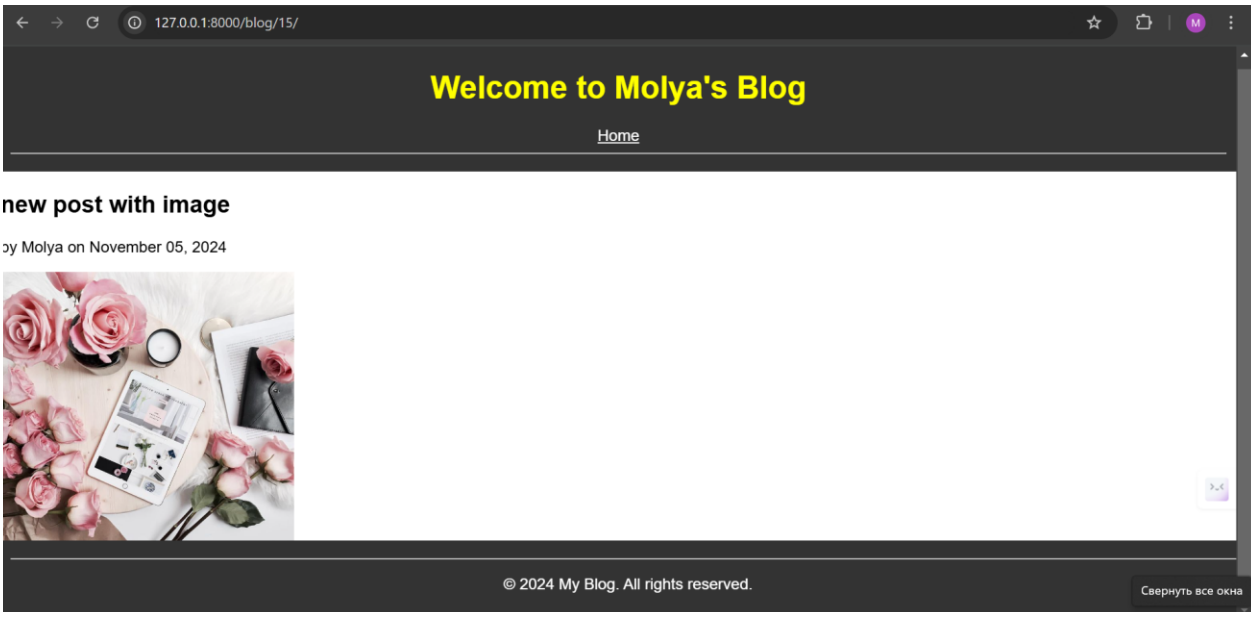
New post 2
by Molya | Published on November 06, 2024

new post 444
by Molya | Published on November 05, 2024

new post with image
by Molya | Published on November 05, 2024

© 2024 My Blog. All rights reserved.

Clicking on the post's title it opens the detail page with image as well.


127.0.0.1:8000/blog/15/

Welcome to Molya's Blog

Home

new post with image

by Molya on November 05, 2024

© 2024 My Blog. All rights reserved.

So the CSS styling was successfully applied, enhancing the appearance of the pages. Testing image upload functionality confirmed that users can upload images with each post, and the images displayed as expected in both list and detail views.

Challenges: correctly configure all folder for static files and folder where all uploaded media will be stored. Initially opened the folder in the wrong place, inside project, then moved media/post_images under general project's directory.

Conclusion

This project provided exercises with Django's core components—Models, Views, and Templates—demonstrating how these elements work together to build dynamic web applications. Each exercise incrementally added functionality to our blog application, highlighting the importance of each component.

Through working with Models, I learned how to structure and manage data, from defining fields to establishing relationships and adding custom managers. Models form the main part of any Django application, allowing us to handle data efficiently. By extending the model with media handling for images, we saw how Django makes it simple to manage user-generated content, which is essential in modern web applications.

Views taught us how to control the logic of an application, respond to user requests, and retrieve relevant data. We explored both function-based views and class-based views, each with its advantages. Function-based views offer straightforward control over application logic, while class-based views provide a reusable and modular approach, which becomes invaluable as applications grow in complexity.

Finally, Templates brought the application to life, enabling us to display data dynamically and in a user-friendly way. With template inheritance, I was able to maintain a consistent layout across pages, and by using static files and media handling, I could further enrich the user interface with styles and images. Templates are critical in defining how users interact with the application, making data and functionality accessible and visually good-looking.

This project demonstrated how the MVT components work together to create a web application. Models handle data, Views manage application logic, and Templates shape the user experience. Knowing these components is essential for developing sophisticated web applications with Django, as they allow developers to create scalable, maintainable, and user-friendly applications that meet the needs of today's web users. This project showed the importance of understanding each layer in Django's framework, providing a solid foundation for future development.

References

https://docs.djangoproject.com/

https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django

https://realpython.com/tutorials/django/

https://stackoverflow.com/

https://chatgpt.com/ (used for issue resolution, when folder for media was wrong)