Assignment 3, web app dev
Django Blog Application

Temirbolat Amanzhol

05.11.2024

Table of Contents

Introduction

This project is about creating a simple Django blog where you can publish posts, add comments to them, and organize them into categories. Django is a powerful web development framework that allows you to quickly create complex applications, manage data, handle user requests, and display beautifully designed pages.

In this project, we learned about three key components of Django:

Models - responsible for storing data in the database, here we define how each post, category, and comment will look.

Views - the "brain" of the application, which processes user requests, prepares data, and chooses what to display on the screen.

Templates - responsible for how our site looks. They allow you to easily add and update the HTML code that is displayed to users.

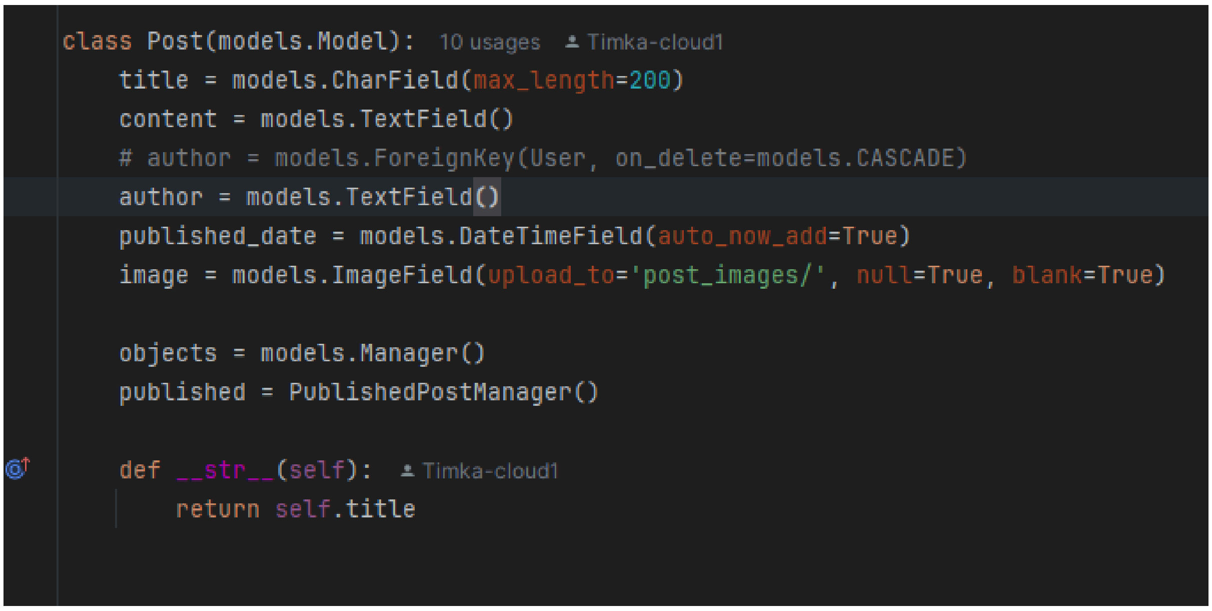
All these components together make Django a convenient tool for creating web applications, and working on this project helped to better understand how they interact.

Creating a Basic Model

Goal: Create a base model for posts that will store information about each published post in the database.

Description: In the first step, we create a Post model, which is a data structure for storing information about each post. This model includes fields such as title (post title), content (post content), author (post author), and published_date (published date). The title field stores the title of the post, content contains its text, author indicates the user who created the post, and published_date stores the date the post was published.

Models in Django help manage data in the database, making working with data easier and more understandable. We also added a __str__ method that returns the post title as a text representation of the model, which is convenient when working with data in the admin panel.



```
class Post(models.Model):  10 usages  Timka-cloud1
    title = models.CharField(max_length=200)
    content = models.TextField()
    # author = models.ForeignKey(User, on_delete=models.CASCADE)
    author = models.TextField()
    published_date = models.DateTimeField(auto_now_add=True)
    image = models.ImageField(upload_to='post_images/', null=True, blank=True)

    objects = models.Manager()
    published = PublishedPostManager()

    def __str__(self):  Timka-cloud1
        return self.title
```

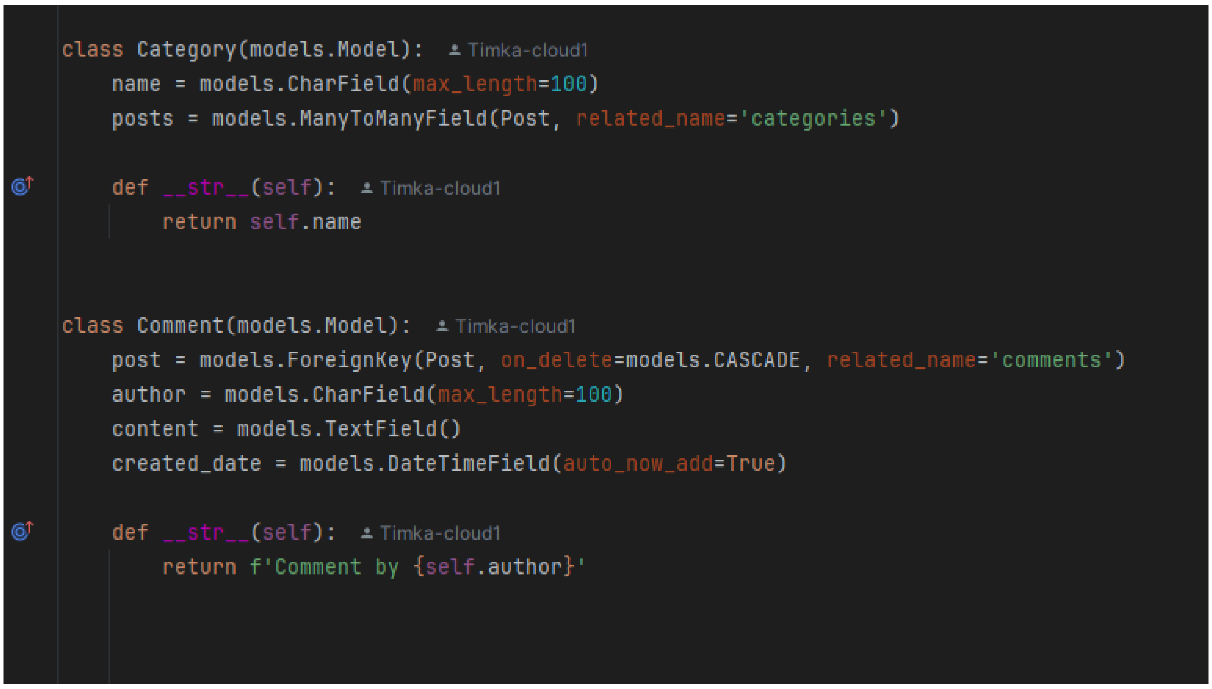
Expected result: We expect that after creating the Post model and running migrations, we will have a table in the database to store posts, and we will be able to add and view posts through the Django Admin.

Model Relationships

Objective: Create relationships between posts, categories, and comments to make the data structure more flexible and manageable.

Description: Here, we extend the `Post` model by adding two additional models: `Category` and `Comment`. The `Category` model allows us to organize posts into multiple categories, so each post can belong to several categories at once. We add a many-to-many relationship between `Post` and `Category`, enabling, for example, a post to be categorized under both "News" and "Analysis."

The `Comment` model lets users leave comments on posts. It includes fields like `author` (the comment author), `content` (comment text), and `created_date` (the date the comment was created). The `ForeignKey` relationship links each comment to a specific post, so all comments are tied to individual posts.



```
class Category(models.Model):  Timka-cloud1
    name = models.CharField(max_length=100)
    posts = models.ManyToManyField(Post, related_name='categories')

    def __str__(self):  Timka-cloud1
        return self.name


class Comment(models.Model):  Timka-cloud1
    post = models.ForeignKey(Post, on_delete=models.CASCADE, related_name='comments')
    author = models.CharField(max_length=100)
    content = models.TextField()
    created_date = models.DateTimeField(auto_now_add=True)

    def __str__(self):  Timka-cloud1
        return f'Comment by {self.author}'
```

Expected Outcome: We can now link posts to categories and add comments to posts, making the blog more functional and similar to a real application.

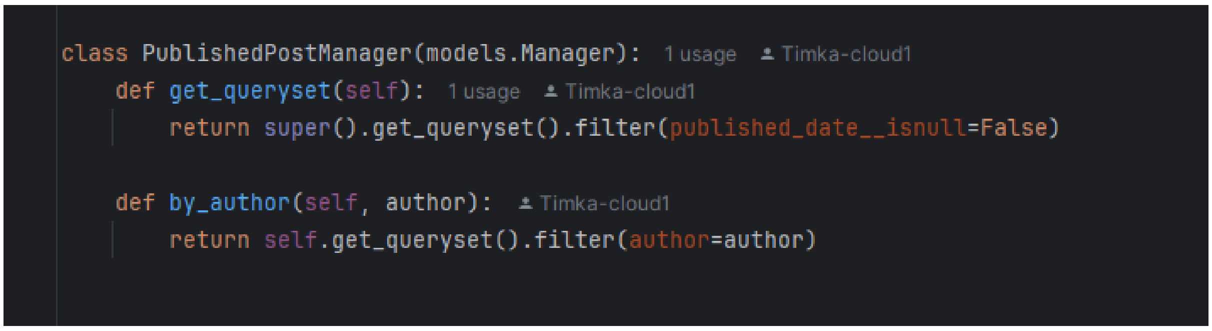
Custom Manager

Objective: Create a custom manager for the `Post` model to easily retrieve only published posts or posts by a specific author.

Description: In this exercise, we create a `Custom Manager` for the `Post` model. A manager in Django is responsible for handling database queries. Creating a custom manager allows us to add methods that simplify data selection. We added a

`PublishedPostManager` with a `get_queryset()` method that returns only published posts (i.e., those with a publication date).

We also added a `by_author` method to filter posts by a specific author. Thanks to this manager, we can easily retrieve only published posts or posts from a specific author by simply calling the manager.



```
class PublishedPostManager(models.Manager):  1 usage  Timka-cloud1
    def get_queryset(self):  1 usage  Timka-cloud1
        return super().get_queryset().filter(published_date__isnull=False)

    def by_author(self, author):  Timka-cloud1
        return self.get_queryset().filter(author=author)
```

Expected Outcome: After implementing the custom manager, we can use it to fetch only published posts and retrieve posts from a specific author.

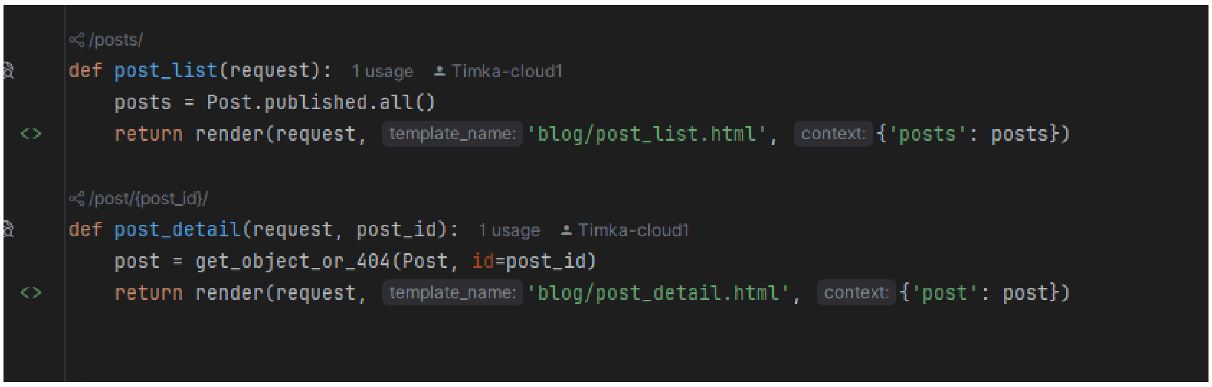
Function-Based Views

Objective**:** Implement function-based views to display a list of all posts and the details of a single post.

Description**:** At this stage, we create two function-based views: `post_list` and `post_detail`. In Django, views are functions or classes that take user requests and return responses, usually in the form of HTML pages.

1) `post_list` is a view for displaying a list of all published posts. It retrieves all posts using our custom manager and passes them to the `post_list.html` template, which renders them on the page.
2) `post_detail` is a view for displaying details of a single post. It takes a post ID, retrieves the post from the database, and passes it to the `post_detail.html` template for display.

Expected Outcome:
We now have pages that show a list of all posts and allow users to open a single post to view its details.



```
/posts/
def post_list(request):  1 usage  Timka-cloud1
    posts = Post.published.all()
    return render(request, template_name: 'blog/post_list.html', context: {'posts': posts})

/post/{post_id}/
def post_detail(request, post_id):  1 usage  Timka-cloud1
    post = get_object_or_404(Post, id=post_id)
    return render(request, template_name: 'blog/post_detail.html', context: {'post': post})
```

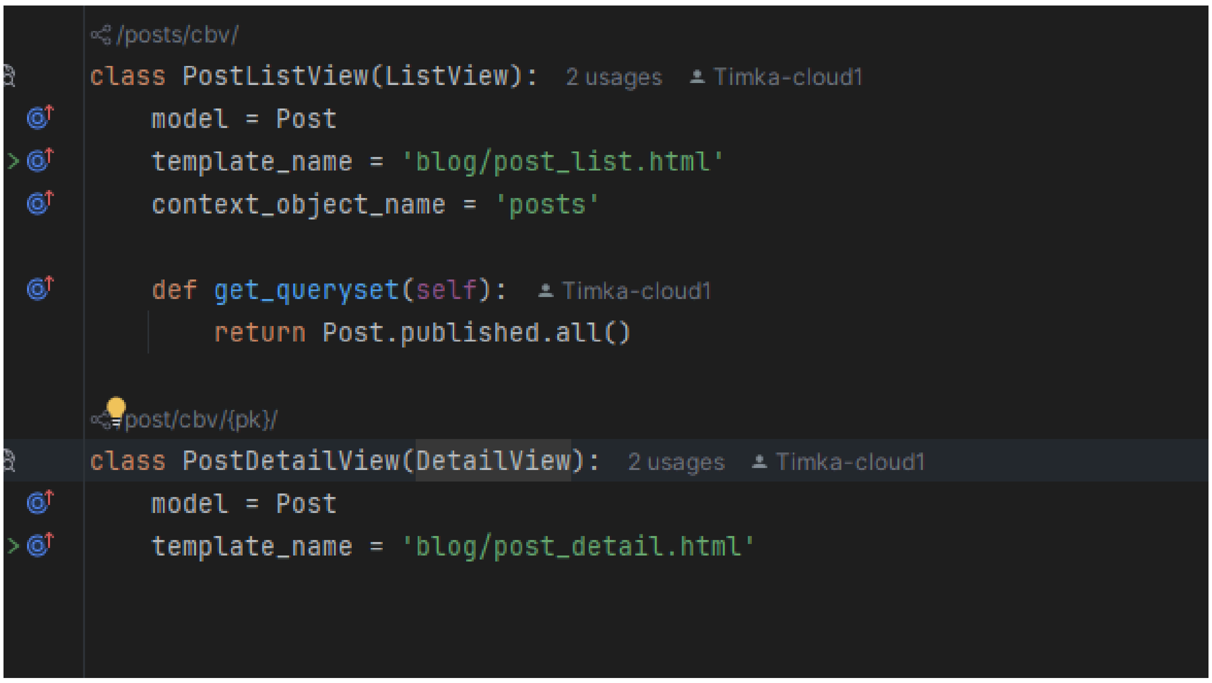
Class-Based Views

Objective**:** Rewrite function-based views using class-based views (CBVs) to make the code more flexible and reusable.

Description**:** Class-based views (CBVs) provide a more structured way to create views, which is especially useful in more complex applications. We rewrite `post_list` and `post_detail` using the `ListView` and `DetailView` classes.

- `PostListView` is a class inheriting from `ListView`, which automatically takes the `Post` model and renders a list of posts.
- `PostDetailView` is a class inheriting from `DetailView`, which displays the details of a single post.

CBVs simplify the code, as Django handles most of the work for us, and they allow us to easily add new functionality through inheritance.



```
/posts/cbv/
class PostListView(ListView):  2 usages  Timka-cloud1
    model = Post
    template_name = 'blog/post_list.html'
    context_object_name = 'posts'

    def get_queryset(self):  Timka-cloud1
        return Post.published.all()

/post/cbv/{pk}/
class PostDetailView(DetailView):  2 usages  Timka-cloud1
    model = Post
    template_name = 'blog/post_detail.html'
```

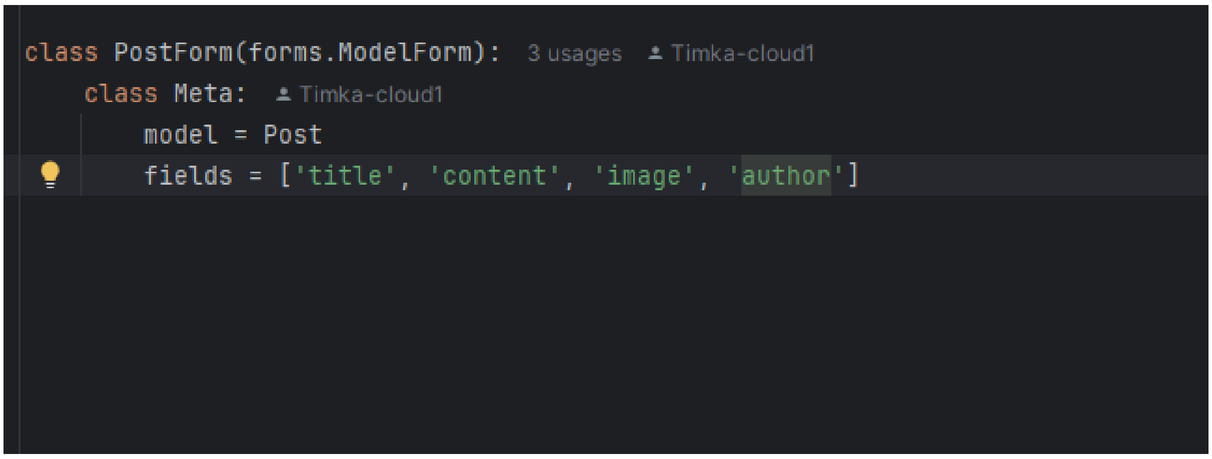
Expected Outcome: Our application now uses classes for displaying the list and details of posts, making the code more flexible and maintainable.

Handling Forms

Objective**:** Create a form to add new posts and implement the logic to handle form submissions.

Description: Django provides `ModelForm`, which simplifies the creation of forms linked to models. We created a form called `PostForm` that includes fields for the title, content, and image. This form is handled in the `post_create` view, where we validate the input data and save a new post if the data is valid.

In the `post_form.html` template, we created a form interface with a “Save” button to submit the data.



```
class PostForm(forms.ModelForm):   3 usages   Timka-cloud1
    class Meta:   Timka-cloud1
        model = Post
        fields = ['title', 'content', 'image', 'author']
```

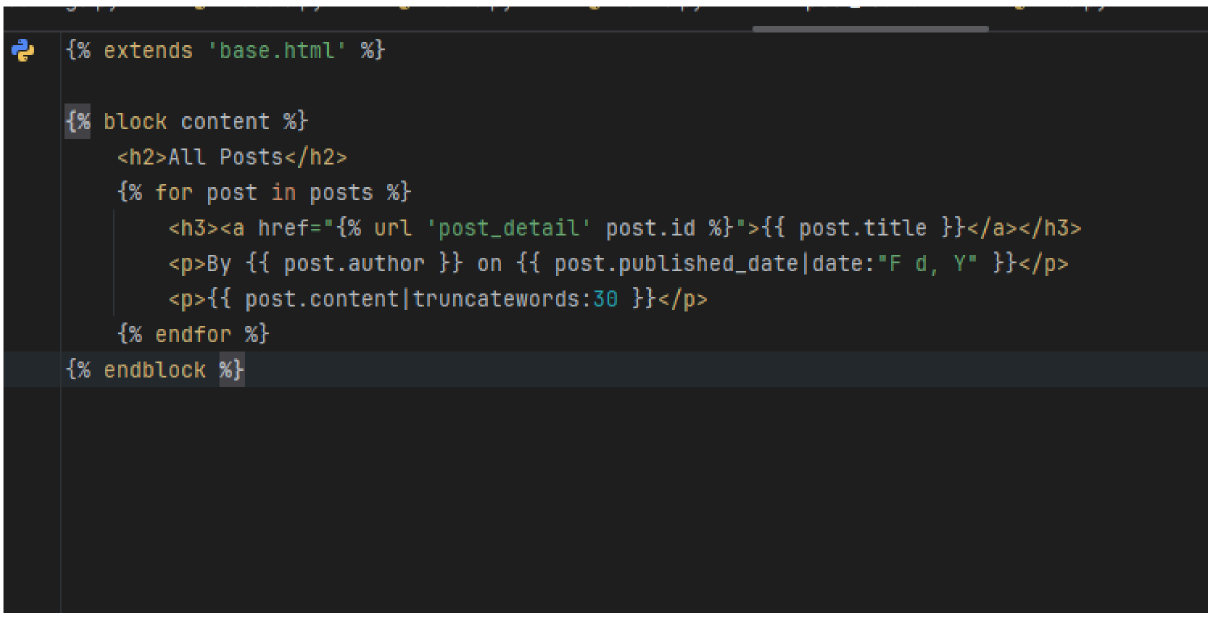
Expected Outcome: Users can now add new posts through a form on the website, and each new post will be automatically saved to the database.

Basic Template Rendering

Objective**:** Create a template to display the list of posts with key information about each post.

Description: We created a template called `post_list.html`, which displays the title, author, and publication date of each post. Django allows us to pass data from views to templates and use it to render HTML.

The template uses a loop to display each post in the list, with a link to view the details of each post. We also formatted the publication date to make the page look more organized.



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

{% block content %}
    <h2>All Posts</h2>
    {% for post in posts %}
        <h3><a href="{% url 'post_detail' post.id %}">{{ post.title }}</a></h3>
        <p>By {{ post.author }} on {{ post.published_date|date:"F d, Y" }}</p>
        <p>{{ post.content|truncatewords:30 }}</p>
    {% endfor %}
{% endblock %}
```

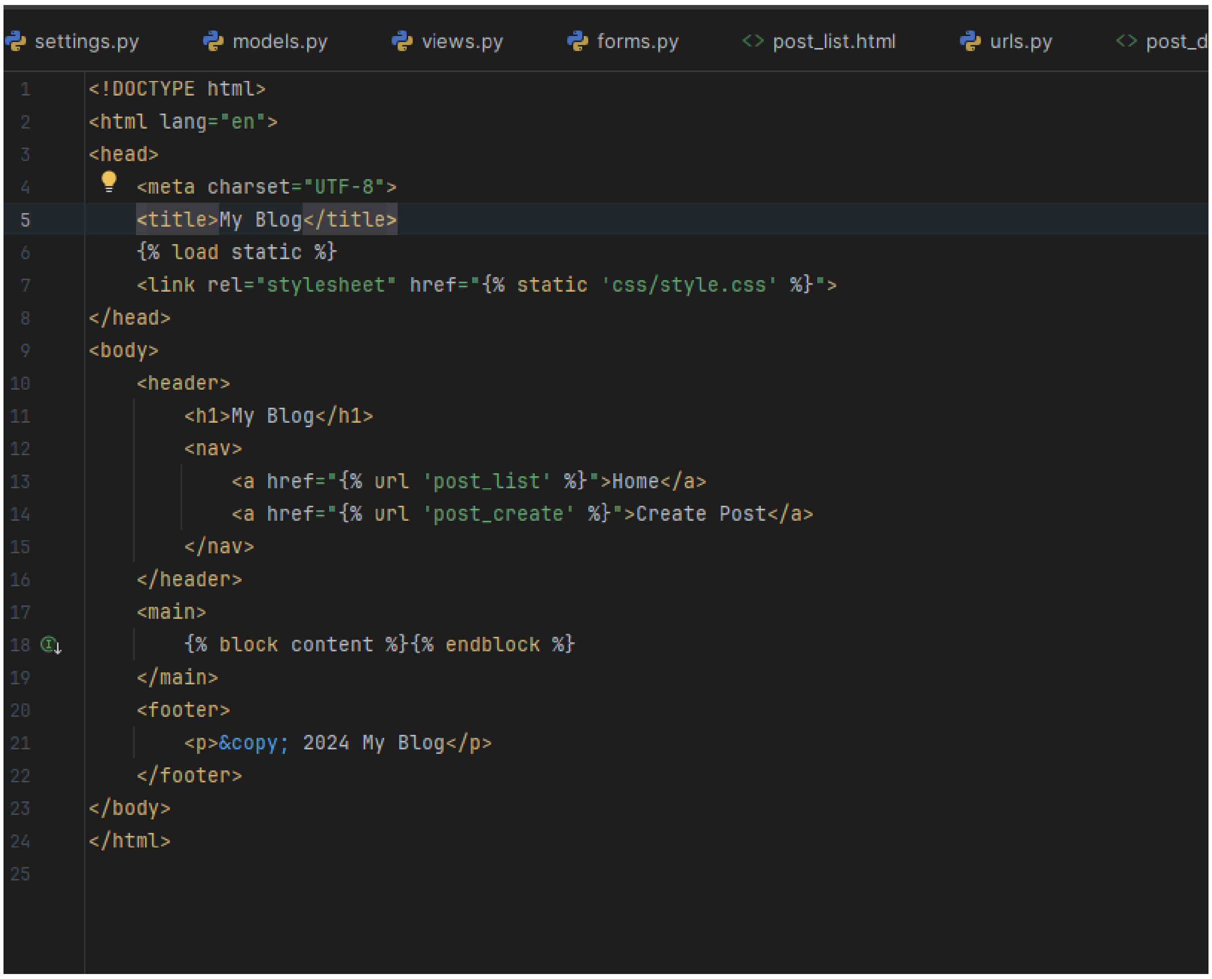
Expected Outcome: The page now displays a list of all posts with key information, allowing users to easily find interesting posts.

Template Inheritance

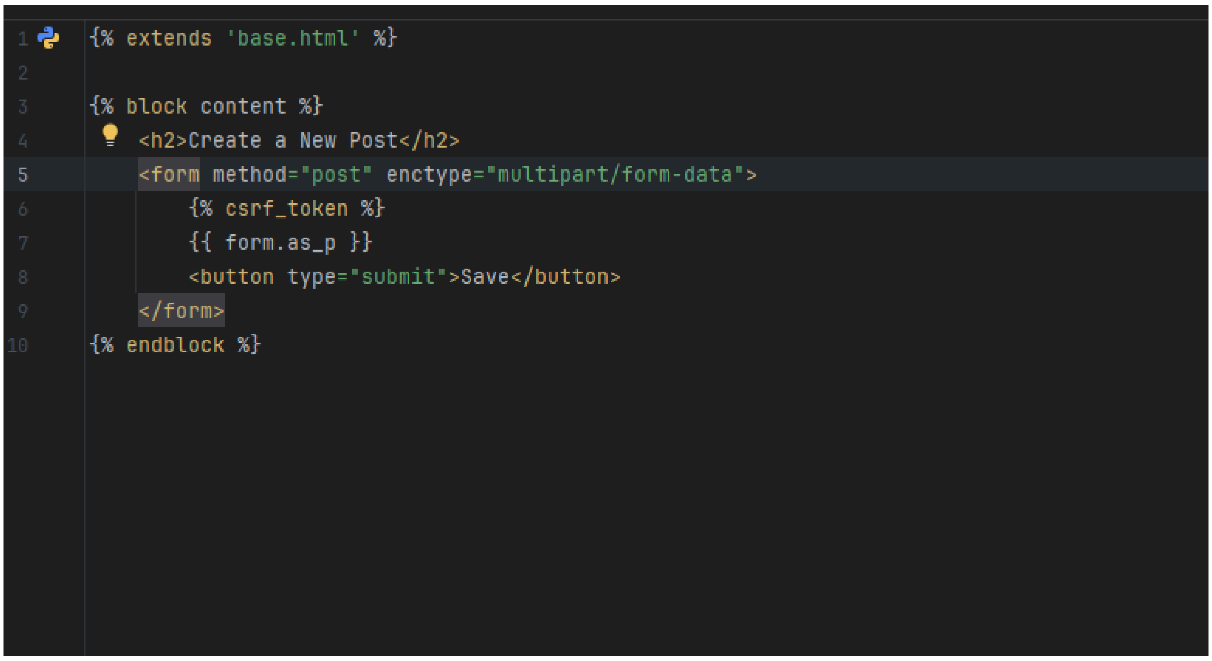
Objective: Create a base template with common elements to simplify the development and maintenance of pages.

Description: We created a base template `base.html`, which includes common HTML elements, such as the header, navigation menu, and footer. Then we extended this base template in `post_list.html` and `post_detail.html` to inherit the page structure.

By using template inheritance, we can make changes to common elements (like navigation) in one place, and these changes will automatically apply to all pages.



```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>My Blog</title>
    {% load static %}
    <link rel="stylesheet" href="{% static 'css/style.css' %}">
</head>
<body>
    <header>
        <h1>My Blog</h1>
        <nav>
            <a href="{% url 'post_list' %}">Home</a>
            <a href="{% url 'post_create' %}">Create Post</a>
        </nav>
    </header>
    <main>
        {% block content %}{% endblock %}
    </main>
    <footer>
        <p>&copy; 2024 My Blog</p>
    </footer>
</body>
</html>
```



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

{% block content %}
    <h2>Create a New Post</h2>
    <form method="post" enctype="multipart/form-data">
        {% csrf_token %}
        {{ form.as_p }}
        <button type="submit">Save</button>
    </form>
{% endblock %}
```

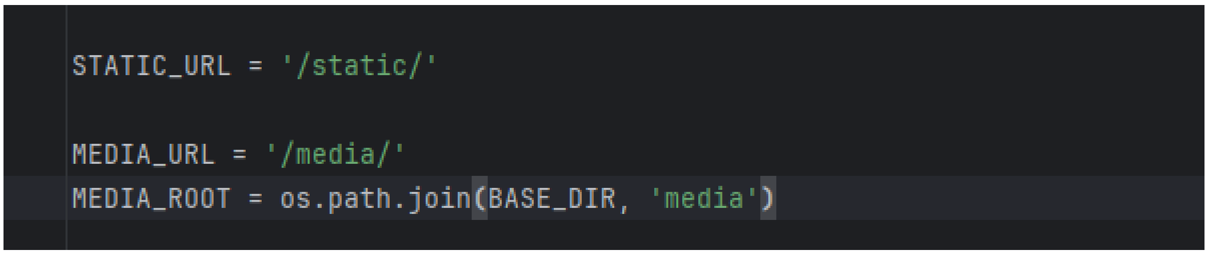
Expected Outcome: All pages on the site now have a consistent style, and it's easier to change the overall structure.

Static Files and Media

Objective: Set up static files (CSS) and media files (uploaded images) to improve the interface.

Description: We added a `static` folder to store CSS files and a `media` folder for uploaded images. In `settings.py`, we configured `MEDIA_ROOT` and `MEDIA_URL` to enable Django to handle uploaded files.

By adding a CSS file (`style.css`), we improved the appearance of the pages, and we added functionality to upload and display images for each post, making the application more visually appealing.



```
STATIC_URL = '/static/'

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

Expected Outcome: The website now looks more polished due to CSS styling, and users can upload images for posts, enhancing the application's functionality and appeal.

Code Snippets

1. Defining the Post Model

This is the code for the `Post` model, which stores each blog post's information, including its title, content, author, and publication date and image.

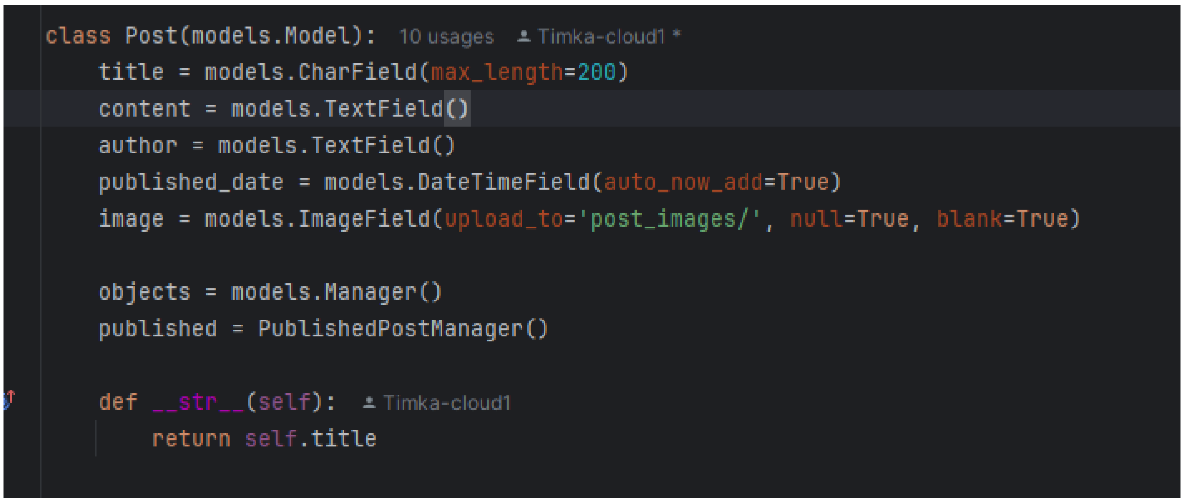
title = models.CharField(max_length=200): Creates a character field with a maximum length of 200 characters for the post title.

content = models.TextField(): Creates a text field for the main content of the post.

author = models.TextField(): Creates an author for post.

published_date = models.DateTimeField(null=True, blank=True): Records the date and time the post was published. null=True and blank=True allow this field to be empty.

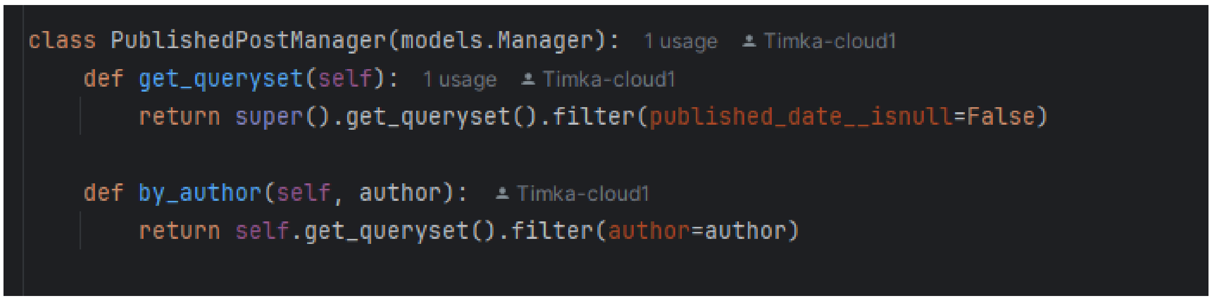
def __str__(self): This method returns the title of the post when it's represented as a string.



```
class Post(models.Model):  10 usages  Timka-cloud1 *
    title = models.CharField(max_length=200)
    content = models.TextField()
    author = models.TextField()
    published_date = models.DateTimeField(auto_now_add=True)
    image = models.ImageField(upload_to='post_images/', null=True, blank=True)

    objects = models.Manager()
    published = PublishedPostManager()

    def __str__(self):  Timka-cloud1
        return self.title
```

Custom Manager for `Post` Model

This custom manager allows us to retrieve only published posts and filter posts by a specific author.



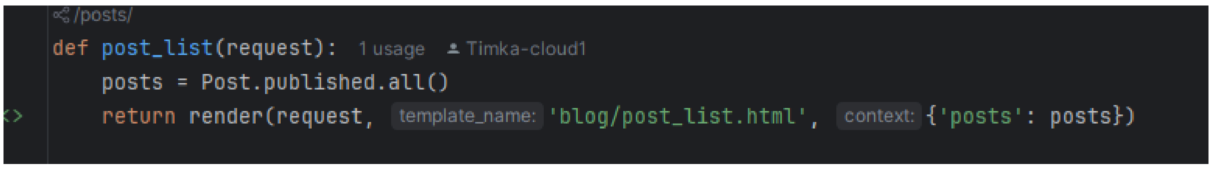
```
class PublishedPostManager(models.Manager):  1 usage  Timka-cloud1
    def get_queryset(self):  1 usage  Timka-cloud1
        return super().get_queryset().filter(published_date__isnull=False)

    def by_author(self, author):  Timka-cloud1
        return self.get_queryset().filter(author=author)
```

get_queryset(): Overrides the default queryset to return only posts with a published_date.

by_author(author): A custom method that filters posts by the provided author.

Function-Based View for Listing Posts

This function-based view (`post_list`) retrieves all published posts and sends them to a template for rendering.



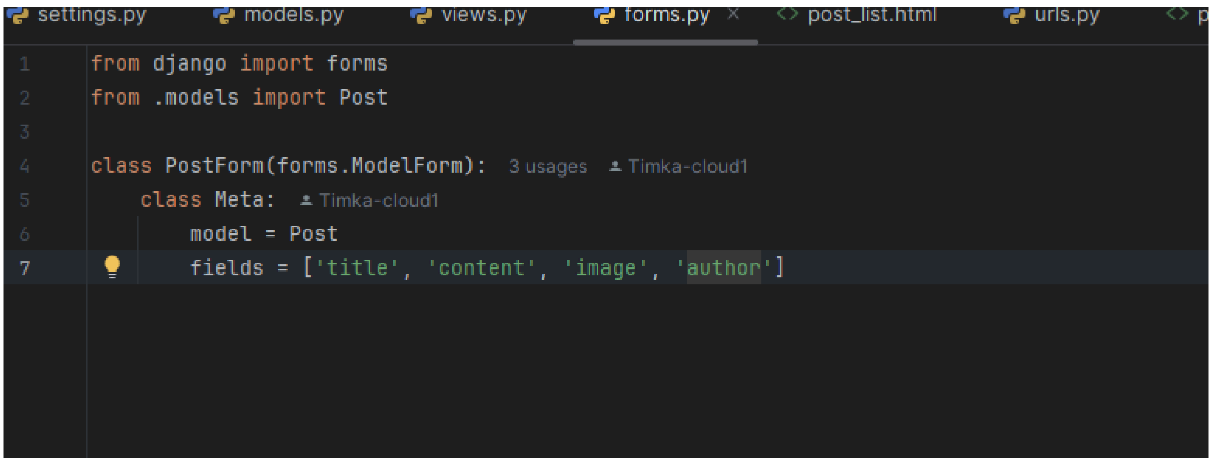
```
/posts/
def post_list(request):  1 usage  Timka-cloud1
    posts = Post.published.all()
    return render(request, template_name: 'blog/post_list.html', context: {'posts': posts})
```

posts = Post.published.all(): Uses the custom manager published to retrieve all published posts.

return render(...): Sends the posts data to the post_list.html template for display.

Form for Creating a New Post

This code defines a form (`PostForm`) for creating a new post using `ModelForm`.



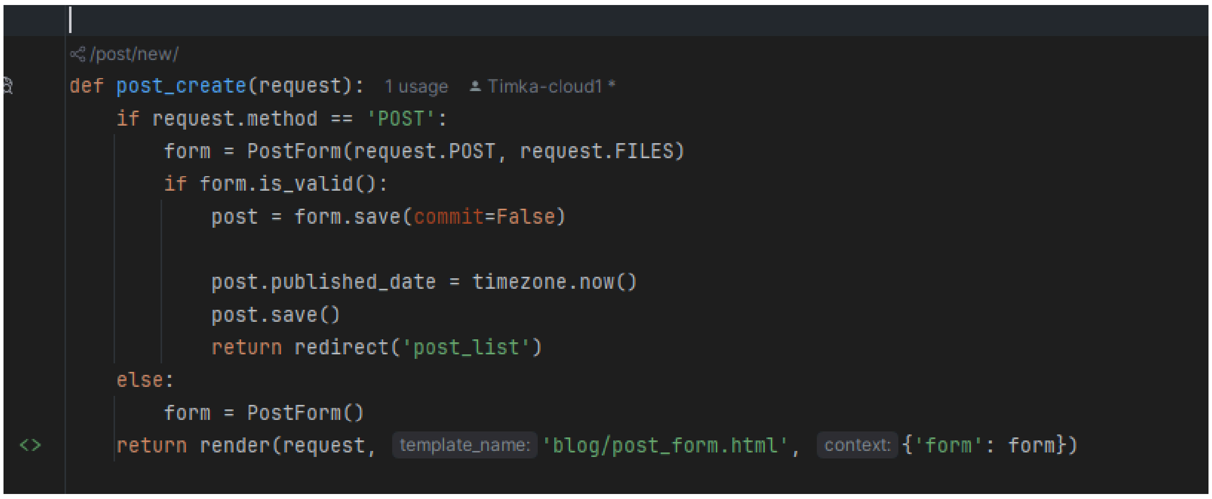
```python
from django import forms
from .models import Post

class PostForm(forms.ModelForm):
    class Meta:
        model = Post
        fields = ['title', 'content', 'image', 'author']
```

model = Post: Links the form to the Post model.

fields = ['title', 'content', 'image', ‘authour’]: Specifies which fields from the model should appear in the form.

Handling Form Submission in `post_create` View

This view function handles the submission of the `PostForm` to create a new post.



```python
def post_create(request):
    if request.method == 'POST':
        form = PostForm(request.POST, request.FILES)
        if form.is_valid():
            post = form.save(commit=False)

            post.published_date = timezone.now()
            post.save()
            return redirect('post_list')
    else:
        form = PostForm()
    return render(request, 'blog/post_form.html', {'form': form})
```

form = PostForm(request.POST, request.FILES): Populates the form with data from the POST request and uploaded files.

if form.is_valid(): Checks if the form data is valid.

post = form.save(commit=False): Creates a Post instance without saving it to the database.

post.author = request.user: Sets the current user as the author of the post.

post.published_date = timezone.now(): Sets the publication date to the current date and time.

post.save(): Saves the post to the database.

return redirect('post_list'): Redirects the user to the list of posts.

Results

By completing each exercise, we created a working Django blog application with the ability to publish and view posts, add comments, and organize posts into categories. Here are the results for each exercise:

1. Models: We successfully created models for posts, categories, and comments. The `Post` model allows us to store the title, body, author, and publication date of each post. The `Category` model allows us to organize posts into topics, and `Comment` allows users to leave comments on posts. This helps make the data structure of the application flexible and easy to manage.

2. Model Relationships: By adding relationships between models, posts can be associated with categories, and multiple comments can be added to each post. This makes our blog more functional and resembles a real web application.

3. Custom Manager: We created a custom manager to easily retrieve only published posts and filter posts by author. This simplified the database queries and made it easier to work with posts.

4. Functional and Class-Based Views: We implemented both functional and class-based views to display the list of posts and the details of a single post. This gave us the opportunity to compare the two approaches and understand how Django helps manage queries and render pages.

5. Forms: We created a form to add new posts using `ModelForm`. The form works correctly and the new post is automatically saved to the database. This is an important element of the application, as it allows users to create content.

6. Templates: We created templates to display the list of posts and the details of a single post, and used template inheritance to create a common layout with a header, navigation, and footer. This made the pages look neater and made it easier to change the overall look of the site.

7. Static files and media: We added CSS for page styling and configured the ability to upload images to posts. This improved the look and feel of the app and made it more user-friendly.

There were a few issues along the way:

1. Errors when installing Custom Manager: At first, it was difficult to configure the manager to filter published posts, but with the help of Django documentation, we were able to create a working custom manager.

2. Require authentication to create posts: It was necessary to configure the login system to restrict access to post creation only for authorized users. We solved this by connecting standard Django authentication routes.

3. Working with media files: Setting up media files required additional changes to the settings and URL configuration, but after adding them, image loading and displaying worked correctly.

Overall, each step was completed successfully and the application works as intended. This project helped me better understand how Django's components—models, views, and templates—come together to create complete web applications.

Conclusion

Working on this project gave us some good hands-on experience using Django to build web applications. We learned how models can be used to define and manage data, how views can handle user requests and pass data to templates, and how templates can help us design pages and display the information we need.

Each component of Django — models, views, and templates — plays an important role. Models allow us to store data in the database and make it convenient for use. Views handle user actions and decide what data to send to templates. Templates render the HTML pages that users see and interact with.

This project showed us how all of these pieces work together and helped us better understand how to build functional and usable web applications with Django. Working with models, views, and templates is the basis of any Django application, and now it's clear how to effectively use them in real projects.

References

Django Documentation: https://docs.djangoproject.com/