Web development
Assignment 3, web app dev
Bitanov Assanali (23MD0392)
10.11.2024

Table of Contents

Introduction *3*
Django Models *4*
Django Views *4*
Django Templates *5*
Results *7*
Conclusion *9*
References *10*

Introduction

In Assignment 3, I applied my skills to create a simple blog application with task management functionality. The app was built using Django, a popular Python web framework, which helps in developing robust and scalable applications [1]. This project involved the creation of a web app where we can view, create, edit, and delete posts.

First, I focused on understanding and setting up models in Django. Models allowed me to define the structure of the blog's data, including posts, categories, and comments. I defined the Post model to store information such as the title, content, author, published date, and optional images.

Second, I implemented views in Django, which play a key role in handling user requests. The project included creating both function-based and class-based views to display the list of posts and detailed views of individual posts. I also set up views to handle form submissions for creating new posts, allowing users to submit and save content.

The next part of the project involved designing templates to display the content. I created templates for different parts of the blog, such as the post list and detail pages. Template inheritance was used to create a consistent structure across all pages, including a common header, footer, and navigation menu. Also, I included static files like CSS to style the pages and make the user interface attractive and easy to navigate.

I also learned how to manage media files in Django, especially for user-uploaded images that are displayed alongside posts. By configuring Django to handle static and media files properly, I ensured that images were stored correctly and could be retrieved whenever needed.

By the end of the project, I gained experience in using Django for full-stack web development. I learned how to handle data, manage templates, and handle media files.

Django Models

In the first three exercises, I created a basic Django app called blog, where I defined the models needed for a blogging platform. The main model I created was the Post model, which represents a single blog post. The Post class is defined in the models.py file. The file is shown in Figure 1.

```
class Post(models.Model):
    title = models.CharField(max_length=200)
    content = models.TextField()
    author = models.CharField(max_length=100)
    published_date = models.DateTimeField(auto_now_add=True)
    categories = models.ManyToManyField(Category, related_name='posts')
    image = models.ImageField(upload_to='post_images/', blank=True, null=True)

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

    def __str__(self):
        return self.title
```

Figure 1. The Post class in the models.py file

Here I defined these fields: title, content, author, published date, categories, and image. The title and author are character fields. The content field holds the body text of the post, while the published date is automatically set when the post is created, helping track when each post was made. The categories field allows multiple categories per post and links to the Category model, creating a flexible many-to-many relationship. Lastly, the image field is optional, letting users attach an image. Also, object managed stands for basic querying and CRUD operations.

Django Views

To show all posts in a page I created the PostListView class. This class gets all the posts and displays them in a list. It also uses a template to make sure the posts are shown in a user-friendly way. The class is shown in Figure 2.

```
class PostListView(ListView):
    model = Post
    template_name = 'blog/post_list.html'
    context_object_name = 'posts'

    def get_queryset(self):
        return Post.published.all()

Асанали Битанов, 2 days ago | 1 author (Асанали Битанов)
class PostDetailView(DetailView):
    model = Post
    template_name = 'blog/post_detail.html'
    context_object_name = 'post'
```

Figure 2. The PostListView and PostDetailView classes in the views.py file

The PostListView uses the ListView to automatically manage retrieving and organizing the posts. By setting the context_object_name to 'posts', this view makes it easy to access the list of posts in the post_list.html template.

To show details about a certain post I created the PostDetailView. It inherits from DetailView and is configured to retrieve one specific Post object based on its ID. The context_object_name is set to 'post', so the template post_detail.html can easily display all the details for a single post.

Next, I created a PostForm class in the forms.py file to handle the form for adding or editing posts. The PostForm class is shown in Figure 3.

```
Асанали Битанов, 2 days ago | 1 author (Асанали Битанов)
class PostForm(forms.ModelForm):
    Асанали Битанов, 2 days ago | 1 author (Асанали Битанов)
    class Meta:
        model = Post
        fields = ['title', 'content', 'author', 'image']
```

Figure 3. The PostForm class in the forms.py file

The PostForm class uses Django's ModelForm to automatically generate a form based on the Post model. By specifying the Meta class inside, we link the form to the Post model and choose specific fields to include: 'title', 'content', 'author', and 'image'. This setup makes it easy to create or update posts since the form will automatically have fields for all the necessary information.

Django Templates

Then I created the templates. The base.html file is the main template used by all pages in the app. It has a header, footer, and navigation menu. The base.html is shown in Figure 4.

```
Асанали Битанов, 23 hours ago | 1 author (Асанали Битанов)
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>{% block title %}Blog{% endblock %}</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 'create_post' %}">New Post</a>
        </nav>
    </header>

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

    <footer>
        <p>&copy; 2024 Assanali Bitanov Blog</p>
    </footer>
</body>
</html>
```

Figure 4. The base.html file

The header has links to the "Home" page, which shows all blog posts, and the "New Post" page, where users can add a new post. We also included a CSS file for styling, which is saved in the static folder. The {% block content %} tag lets other pages add their own content while keeping the main layout the same.

Then I created more templates, such as post_list, post_detail, create_post and delete_post. All of them are build on base.html. The post_list.html is shown in Figure 5.

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

{% block title %}Blog Posts{% endblock %}

{% block content %}
    <h2>All Posts</h2>
    <ul>
        {% for post in posts %}
            <li>
                <h3>{{ post.title }}</h3>
                <p>By {{ post.author }}</p>
                <p>Published on {{ post.published_date|date:"F j, Y" }}</p>
                <p>{{ post.content|truncatewords:30 }}</p>
                <a href="{% url 'post_detail' post.id %}">Read more</a>
            </li>
        {% empty %}
            <li>No posts are available.</li>
        {% endfor %}
    </ul>
{% endblock %}
```

Figure 5. The post_list.html file

Each post displays its title, author, publication date, and a short preview of the content. The date is formatted, and the content preview is limited to 30 words. Users can click "Read more" to see the full post. If there are no posts, a message says "No posts are available." This simple design helps users browse posts easily while keeping the page neat and organized.

Then I created style.css file to enhance the interface of the app. The style.css file is shown in Figure 6.

```
body {
    font-family: Arial, sans-serif;
    margin: 0;
    padding: 0;
    background-color: #f8f9fa;
}

header, footer {
    background-color: #343a40;
    color: white;
    padding: 10px;
    text-align: center;
}

main {
    padding: 20px;
}

h1, h2 {
    color: #343a40;
}

a {
    color: #007bff;
    text-decoration: none;
}
```

Figure 6. The style.css file

It sets a basic font style for the whole page using Arial, making the text easy to read. The body has a light gray background color (#f8f9fa), while the header and footer are styled with a dark color (#343a40) and white text to make them clear. Both header and footer have padding and centered text for a neat appearance. The main content area (main) has extra padding to give more space around the content, making it easier to read. Headings (h1 and h2) are given the same.

Results

As a result we have a fully functioning blog app. The main page of the app is the list of posts page. The page is shown in Figure 8.

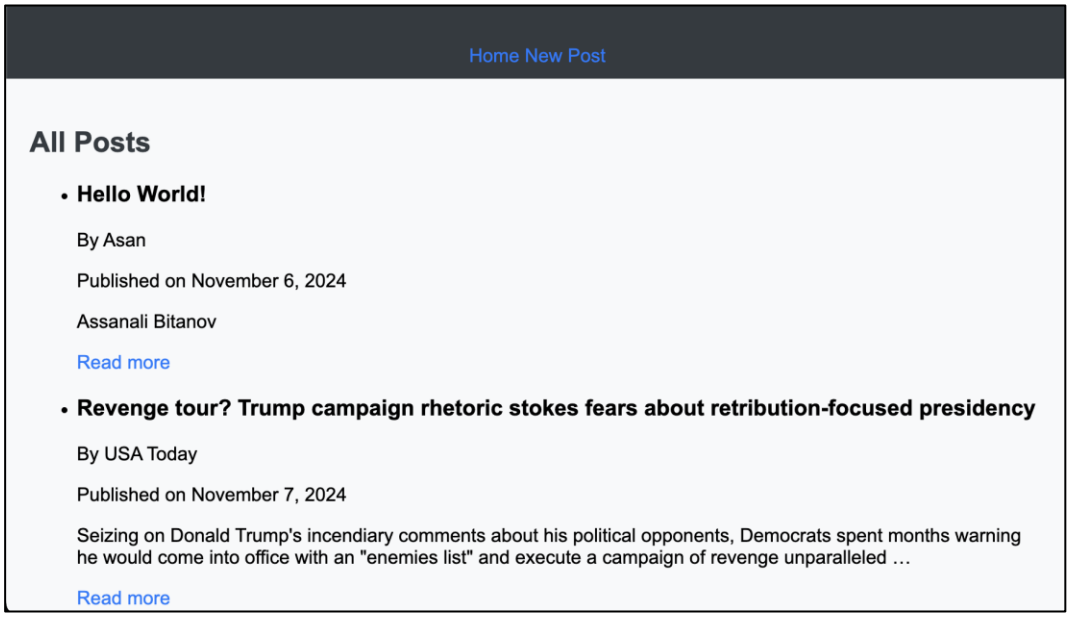

Figure 8. The main page

The main page of the blog app shows a list of all posts in a simple and clear way. Each post includes the title, author's name, publication date, and a short preview of the content. For each post, there's a "Read more" link that lets users view the full content if they're interested. If no posts are available, a message tells users that there are no posts yet. The page is easy to read and use, with simple styling and a clean layout that makes it easy to look through the posts and find something to read.

By pressing to Read more button we move to the post detail page. The page is shown in Figure 9.

Figure 9. The detail post page.

The post detail page shows a full view of a single blog post. Here, users can see the complete content of the post, along with the title, author, and publication date. If an image was uploaded with the post, it is displayed as well, sized appropriately to keep it from taking up too much space on the page.

To create a new post we press the New Post button on the main page. The Create New Post page is shown in Figure 10.

Figure 10. The create a new post form

The create new post page allows users to add a new blog post to the site. On this page, there is a form where users can enter the post's title, content, author name, and optionally upload an image. Once the user fills in the details and submits the form, the post is saved and will appear on the main list of posts.

Conclusion

In this project, I focused on building a blog application using Django to learn more about web development and working with databases. My main goal was to understand how to create a functional and user-friendly blog site with features like adding, viewing, and managing posts. I worked through several exercises that helped me learn Django's core components, including models, views, forms, and templates, as well as the role of media and static files.

I began by setting up the structure of the Django project and creating the main components in the app. Using Django models, I created classes like Post and Category to represent my blog data. This included defining fields for post title, content, author, categories, and images, allowing me to build a flexible data structure. I also used relationships, like many-to-many and foreign key relationships, to connect posts with categories and comments, enhancing the blog's functionality. Creating and applying migrations helped me see how to manage database changes effectively.

In the views section, I learned how to display data using both function-based and class-based views. I started by implementing function-based views to show a list of posts and individual post details. Then, I refactored these into class-based views using Django's ListView and DetailView to streamline the code. This gave me knowlegde into different approaches for organizing views in Django and when to use each one. Also, I created forms for adding new posts, where users can input details for each post. This helped me understand how to handle form data and validate user input.

Templates were a key part of the project, as they allowed me to design the blog interface. I created a base.html template with a consistent header, footer, and navigation links, which other templates could extend. For the post list and detail pages, I used templates to display content in a clear and organized way, adding custom styles with CSS. Adding a media folder for post images also helped me understand how Django handles static and media files, allowing me to display uploaded images effectively.

This project provided hands-on experience with Django's features, from data modeling to views and templates. I learned how to structure a web application, connect it to a database, and make it interactive for users. Working with media files, adding CSS styles, and managing URLs gave me a well-rounded understanding of building a blog from scratch. These skills will be valuable for future web development projects, as I now know how to create a complete and dynamic web application using Django.

References

1. Django makes it easier to build better web apps more quickly and with less code. [Django] https://www.djangoproject.com/