Assignment 3

Issayev Abzal

Date: 10.11.2024

Course: Web Application Development

Github repo:
https://github.com/Karisbala/WebDev2024/tree/main/Assignment3

Table of Contents

Introduction

Models are the only definitive source of information about data. Models contain the basic fields and behaviors of the data to be stored. Typically, each model is mapped to a single database table [1].

A view function (or view for short) is a Python function that takes a web request and returns a web response. This response can be the HTML content of a web page, a redirect, a 404 error, an XML document, an image, or anything else. The view itself contains all the logic needed to return this response [2].

Django templates are text documents or Python strings marked up using the Django template language. Some syntax is recognized and interpreted by the template engine. The main ones are variables and tags. Templates are processed in context. The render replaces variables with values found in context and executes tags. The rest of the output is output as is [3].

Models, views and templates are the basic building blocks of Django and help structure and simplify web development. Models describe the structure and relationships of data in an application and allow data to be easily stored, retrieved and managed without having to query complex databases. They form the backbone of the application and make data processing organized and simple.

Exercise Descriptions

Exercise 1: Creating a Basic Model

Objective: Set up a basic Django model for blog posts.
Description of the implementation steps:

- Create a new Django app called blog.
- Define a Post model with fields: title, content, author, and published_date.
- Implement a __str__ method to return the title of each post.

Expected Outcome: A Django model that can store post information, with a string representation for easy identification.

Exercise 2: Model Relationships

Objective: Extend the Post model to include a Category model with a many-to-many relationship.
Description of the implementation steps:

- Define a Category model with a name field in models.py.
- Add a many-to-many relationship field in the Post model that links to Category.
- Migrate changes to create the updated database schema.

Expected Outcome: The Post model has a many-to-many relationship with Category, allowing each post to belong to multiple categories.

Exercise 3: Custom Manager

Objective: Create a custom manager for Post to filter published posts and posts by a specific author.
Description of the implementation steps:

- Define a PostManager class inheriting from models.Manager.
- Implement a published method that filters posts by published_date.
- Implement a by_author method that retrieves posts by a specific author.
- Set objects = PostManager() in the Post model [4].

Expected Outcome: Post.objects.published() and Post.objects.by_author("Author Name") will return filtered posts as per the criteria.

Exercise 4: Function-Based Views

Objective: Create function-based views for listing all posts and displaying a single post.
Description of the implementation steps:

- Create a post_list view that retrieves and displays all posts.
- Create a post_detail view that retrieves and displays a single post based on its ID.
- Define URLs for both views in urls.py.
- Create templates for displaying the posts.

Expected Outcome: The views display a list of posts and individual post details via function-based views.

Exercise 5: Class-Based Views

Objective: Refactor views into class-based views.
Description of the implementation steps:

- Replace post_list and post_detail views with ListView and DetailView [5].
- Update urls.py to use the new views.
- Ensure the templates work with the context provided by the class-based views.

Expected Outcome: PostListView and PostDetailView display the list and detail views for posts.

Exercise 6: Handling Forms

Objective: Create a form to add new posts and implement a view to handle form submissions.
Description of the implementation steps:

- Create a PostForm using ModelForm for the Post model [6].
- Implement a post_create view that handles form submission and validation.
- Create a post_form.html template for displaying the form.

Expected Outcome: A form that lets users add new posts, saving the data upon submission.

Exercise 7: Basic Template Rendering

Objective: Set up a basic template to display posts with formatting for the publication date.
Description of the implementation steps:

- Create a post_list.html template to display the list of posts, including titles, authors, and publication dates.
- Use the date template tag to format the published_date.

Expected Outcome: A cleanly formatted list of blog posts with titles, authors, and publication dates.

Exercise 8: Template Inheritance

Objective: Set up a base template with common elements for reuse.
Description of the implementation steps:

- Create a base.html template with a header, footer, and {% block content %} for page-specific content.
- Extend base.html in post_list.html and post_detail.html templates [7].

Expected Outcome: Consistent layout with a header and footer across the site.

Exercise 9: Static Files and Media

Objective: Add CSS styling and configure media uploads for images.
Description of the implementation steps:

- Create a styles.css file in the static directory and link it in the base.html template.
- Set up MEDIA_URL and MEDIA_ROOT in settings.py for media file handling [8].
- Update Post model to include an ImageField for images.
- Display uploaded images in the post list and detail templates.

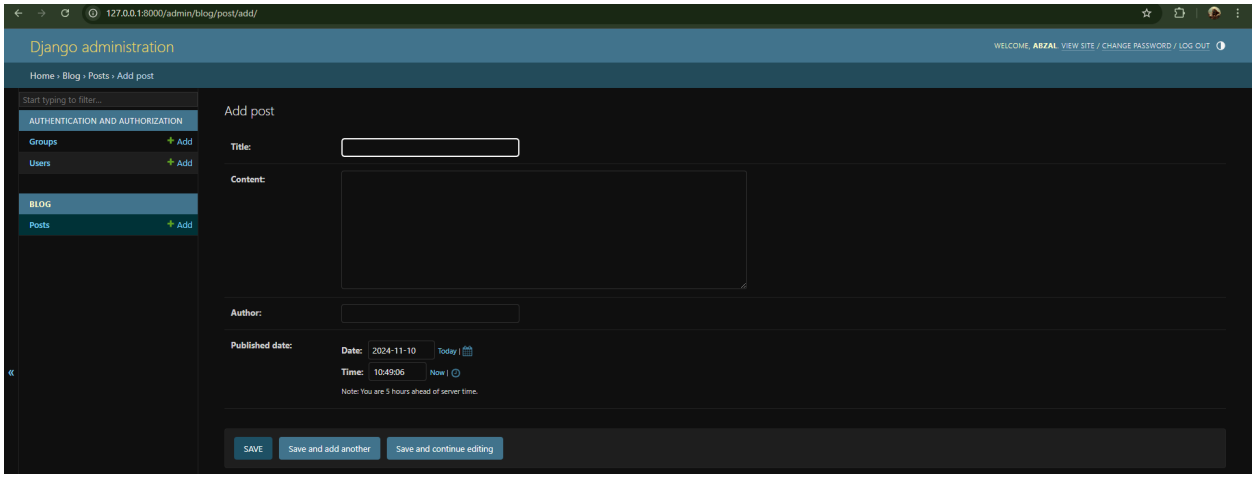
Expected Outcome: The site has basic CSS styling, and uploaded images display on the respective post pages.

Results and Code Snippets

Exercise 1: Creating a Basic Model

The Post model was created with fields for title, content, author, and published_date. A __str__ method was defined to represent each post by its title, allowing blog posts to be stored and easily identified within the app.
Post model code snippet:

```
# Create your models here.
class Post(models.Model):
    title = models.CharField(max_length=200)
    content = models.TextField()
    author = models.CharField(max_length=100)
    published_date = models.DateTimeField(default=timezone.now)

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



Screenshot 1. Admin panel UI with Post model.

Exercise 2: Model Relationships

A Category model was added, and a many-to-many relationship was set up with Post. This allowed each post to be associated with multiple categories.
Category and Post model code snippets with necessary changes:

```
class Category(models.Model):
    name = models.CharField(max_length=100, unique=True)

    def __str__(self):
        return self.name

class Post(models.Model):
    title = models.CharField(max_length=200)
    content = models.TextField()
    author = models.CharField(max_length=100)
    published_date = models.DateTimeField(default=timezone.now)
    categories = models.ManyToManyField(Category, related_name='posts')

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

Comment model code snippet:

```
class Comment(models.Model):
            post    =    models.ForeignKey(Post,    on_delete=models.CASCADE,
related_name='comments')
    author = models.CharField(max_length=100)
    content = models.TextField()
    created_date = models.DateTimeField(default=timezone.now)
```

```
    def __str__(self):
        return f"Comment by {self.author} on {self.post.title}"
```

Exercise 3: Custom Manager

A custom manager was created for Post to filter published posts and retrieve posts by specific authors.

PostManager custom Manager code snippet:

```
class PostManager(models.Manager):
    def published(self):
        return self.filter(published_date__lte=timezone.now())

    def by_author(self, author_name):
        return self.filter(author=author_name)

class Post(models.Model):
    title = models.CharField(max_length=200)
    content = models.TextField()
    author = models.CharField(max_length=100)
    published_date = models.DateTimeField(default=timezone.now)
    categories = models.ManyToManyField(Category, related_name='posts')

    objects = PostManager()

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

Exercise 4: Function-Based Views

The post_list and post_detail function-based views were implemented to display all posts and individual post details.

post_list view code snippet:

```
def post_list(request):
    posts = Post.objects.published().order_by('-published_date')
    return render(request, 'blog/post_list.html', {'posts': posts})
```

post_detail view code snippet:

```
def post_detail(request, post_id):
    post = get_object_or_404(Post, id=post_id)
    return render(request, 'blog/post_detail.html', {'post': post})
```

Exercise 5: Class-Based Views

The views were refactored into class-based views (ListView and DetailView).
Refactored PostListView and PostDetailView views code snippet:

```
class PostListView(ListView):
    model = Post
    template_name = 'blog/post_list.html'
    context_object_name = 'posts'
    ordering = ['-published_date']

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

class PostDetailView(DetailView):
    model = Post
    template_name = 'blog/post_detail.html'
    context_object_name = 'post'
```

Changed urls for PostListView and PostDetailView views code snippet:

```
urlpatterns = [
    path('', PostListView.as_view(), name='post_list'),
    path('post/<int:pk>/', PostDetailView.as_view(), name='post_detail'),
]
```

Exercise 6: Handling Forms

A PostForm was created using ModelForm, and a view was implemented to handle form submission, validate data, and save posts.
Model for PostForm to handle correct fields code snippet:

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

post_create view that handles form submissions and validates data code snippet:

```
def post_create(request):
    if request.method == 'POST':
        form = PostForm(request.POST)
        if form.is_valid():
            form.save()
            return redirect('post_list')
```

```
    else:
        form = PostForm()
    return render(request, 'blog/post_form.html', {'form': form})
```

Screenshot 2. post_list correctly displaying newly created blog post.

Screenshot 3. post_detail correctly displaying details of newly created blog post.

Exercise 7: Basic Template Rendering

A basic template was set up to display posts with published_date formatted using the date template tag.

post_list html code snippet that shows date template tag used in the code:

```
<h1>All Blog Posts</h1>
<ul>
    {% for post in posts %}
        <li>
            <h2><a href="{% url 'post_detail' post.id %}">{{ post.title
}}</a></h2>
            <p>By {{ post.author }}</p>
            <p>Published on: {{ post.published_date|date:"F j, Y" }}</p>
        </li>
    {% endfor %}
</ul>
```

Screenshot 4. Published date displaying correct date format.

Exercise 8: Template Inheritance

A base.html template was created, containing a header, footer, and a {% block content %} for extending.

Extended post_list code snippet:

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

{% block content %}
<h1>All Blog Posts</h1>
<ul>
    {% for post in posts %}
        <li>
              <h2><a href="{% url 'post_detail' post.id %}">{{ post.title
}}</a></h2>
            <p>By {{ post.author }}</p>
            <p>Published on: {{ post.published_date|date:"F j, Y" }}</p>
        </li>
    {% endfor %}
</ul>
{% endblock %}
```

Extended post_detail code snippet:

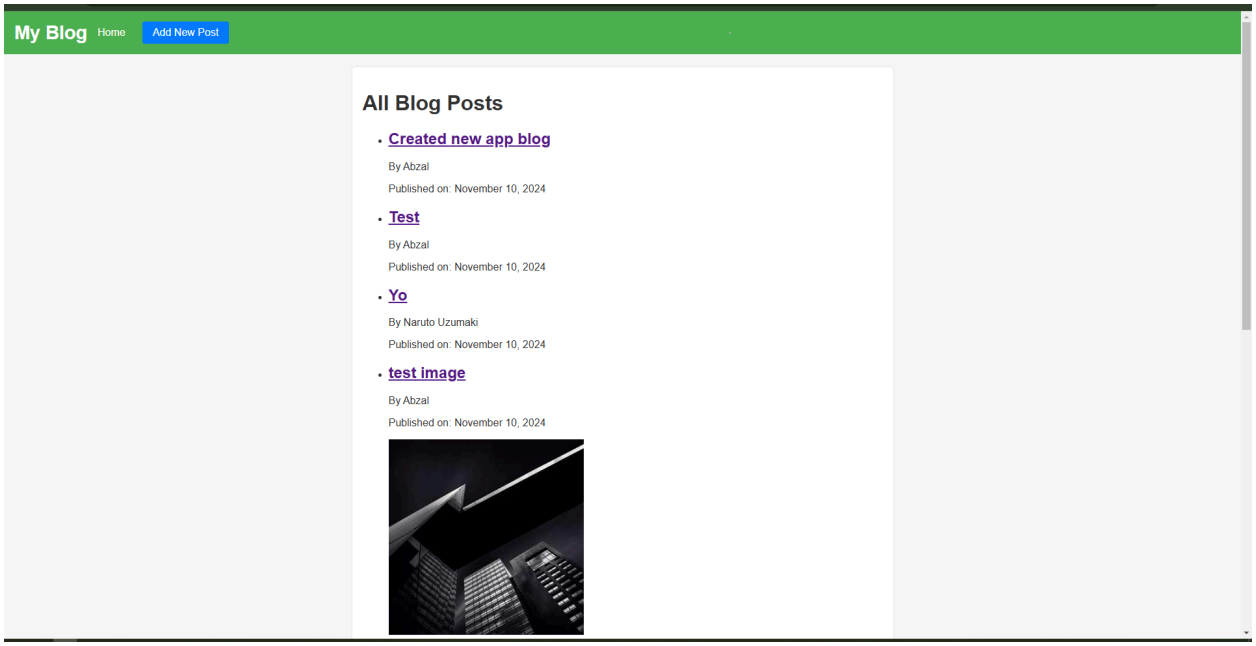
```
{% extends "blog/base.html" %}

{% block content %}
<h1>{{ post.title }}</h1>
<p>Published on: {{ post.published_date|date:"F j, Y" }}</p>
<p>By {{ post.author }}</p>
```

```
<p>{{ post.content }}</p>
{% endblock %}
```

Screenshot 5. Main screen html representation with base html and extended list and detail.

Exercise 9: Static Files and Media

CSS styling was added with static files, and media handling was configured for uploaded images.



Screenshot 6. Complete web blog app with styles.

Code snippet of Post model to include image:

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

    objects = PostManager()

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

Conclusion

Learning about Django's models, views and templates shows how these pieces work together to easily build complex web applications. Models work by organizing and storing data in a way that makes it easy to retrieve and manage. Views process user requests, handle what the user needs, and link data from the model together to display the appropriate information.

Templates manage the look and layout of web pages, allowing dynamic data to be organized and displayed in an easy-to-understand way.

Understanding these three components is important for building powerful and flexible web applications because the Django installation keeps the code organized and makes it easy to add new features and improve the application.

References

1. https://docs.djangoproject.com/en/5.1/topics/db/models/
2. https://docs.djangoproject.com/en/5.1/topics/http/views/
3. https://docs.djangoproject.com/en/5.1/topics/templates/
4. https://docs.djangoproject.com/en/5.1/topics/db/managers/
5. https://docs.djangoproject.com/en/5.1/ref/class-based-views/generic-display/
6. https://docs.djangoproject.com/en/5.1/topics/forms/modelforms/
7. https://docs.djangoproject.com/en/5.1/ref/templates/language/
8. https://docs.djangoproject.com/en/5.1/topics/files/

Appendix

Include any additional material (like screenshots of the app, if applicable).