Exercise 2.4: Django Views and Templates

Learning Goals

- Summarize the process of creating views, templates, and URLs
- Explain how the “V” and “T” parts of MVT architecture work
- Create a frontend page for your web application

Reflection Questions

- Do some research on Django views. In your own words, use an example to explain how Django views work.

In Django, views handle the logic of an application and dictate what data is sent to the templates (HTML) for rendering. Here’s an example:

In a blog, that displays a list of posts;

1- **Model:** define a **Post** model/class in models.py:

```
from django.db import models
class Post(models.Model):
     title = models.CharField(max_length=200)
     content = models.TextField()
     date_posted = models.DateTimeField(auto_now_add=True)
```

2- **View:** In views.py, you create a view to fetch all posts and pass them to a template: views.py

```
from django.shortcuts import render
from .models import Post

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

3- **URL Configuration**: In urls.py, you map the view to a URL:

```
from django.urls import path
from .views import post_list
```

```
urlpatterns = [
     path('', post_list, name='post_list'),
]
```

4- **Template:** In post_list.html, you can loop through the posts:

```
. . .
<h1>Blog Posts</h1>
     <ul>
     {% for post in posts %}
          <li>{{ post.title }} - {{ post.date_posted }}</li>
     {% endfor %}
     </ul>
. . .
```

In conclusion: The view (post_list) fetches all Post objects from the database. The view passes these posts to a template (post_list.html) using a context dictionary.
The template then renders the posts as HTML for the user to see.

- Imagine you're working on a Django web development project, and you anticipate that you'll have to reuse lots of code in various parts of the project. In this scenario, will you use Django function-based views or class-based views, and why?

In a Django project where code reuse is crucial, Class-Based Views (CBVs) are preferred over Function-Based Views (FBVs) because they promote reusability and extensibility. CBVs allow you to use inheritance and built-in methods to handle common tasks with less boilerplate code. For example, if you need to display lists of objects in various parts of your app, using Django's ListView (a CBV) makes it easy to share and extend functionality across different views, making your code more maintainable and scalable.

- Read Django's documentation on the Django template language and make some notes on its basics.

Models:
Model Definition: Inherit from django.db.models.Model to define models.
Fields: Use field types like CharField, IntegerField, DateTimeField for attributes.

Relationships: Establish relationships with ForeignKey, ManyToManyField, and OneToOneField.
Meta Class: Customize model behavior with the Meta class (e.g., table name, ordering).
Views:
Function-Based Views: Write functions to handle HTTP requests and return responses.
Class-Based Views: Utilize Django's CBVs for common tasks (e.g., ListView, DetailView).
Request Object: Access request data, headers, and cookies.
Rendering Templates: Use render() to display templates and pass data.
Templates:
Template Syntax: Use Django's template language for dynamic HTML.
Tags & **Filters**: Use {% if %}, {% for %}, and filters like {{ value|default:"N/A" }}.
Context: Pass data to templates via the context argument in render().
URLs:
URL Patterns: Map URLs to views with defined patterns.
Path Converters: Use converters like <int:pk> for specific URL matches.
Namespaces: Organize URL patterns with namespaces.
Forms:
Form Classes: Create classes to handle form submissions and validation.
Field Types & **Validation**: Define fields like CharField, IntegerField and validate with clean().
Rendering Forms: Display forms in templates using {{ form }}.
Admin Interface:
Register Models: Register models using admin.site.register().
Customize Admin: Tailor the admin interface with actions, filters, and display options.
Additional Topics:
Migrations: Manage schema changes with migrations.
Static Files: Serve static assets like CSS and JavaScript.
User Authentication: Implement authentication with Django's built-in system.
Signals: Respond to events with signals.
Caching & **Testing**: Improve performance with caching and ensure reliability with tests.