Yufan Xu

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.

➢ A Django view is a Python function that takes a web request and returns a response, which can be as simple as an HTML file or an image, or as complicated as retrieved data from a database based on the user’s input in the request. Taking the GitHub website as an example, if the URL entered in the browser is https://github.com/ (which is the same as https://github.com/home), the website returns the homepage that displays what the website offers and prompts you to sign up for an account. If the URL entered is https://github.com/login, the website takes you to the login page because of the “/login” portion of the URL. Upon successful login, your username is appended to the URL and redirects you to your custom homepage/dashboard (looks like https://github.com/yuxu1). When a repository name is appended to the end of that URL, the website then searches for a repository that matches that name in the logged in user’s repositories (https://github.com/yuxu1/recipe-app).

❖ 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?

➢ I would use class-based views (CBVs), because CBVs, as the name implies, are based on Python classes. Its class-based nature would make it easier to reuse or extend pieces of code in various parts of the project, helping to avoid code duplication using built-in views.

❖ Read Django’s documentation on the Django template language and make some notes on its basics.

➢ **Template**: a text file; can generate any text-based format (HTML, XML, CSV, etc)

➢ **Variables**: look like this **{{ variable }}**
  - gets evaluated and replaced with the result when encountered in a template; variable names consist of any combination of alphanumeric characters and underscores but cannot start with an underscore or a number
  - Attributes of a variable accessed using a dot (**.**)

➢ **Filters**: look like this: **{{ name | lower }}**, displaying the value of the **{{ name }}** variable after being filtered through the **lower** filter
  - Filters are applied using a pipe (|)

- **Tags**: look like this: **{% tag %}**, comes with about two dozen built-in from Django
  - More complex than variables: some create text in the output; some control flow by performing loops or logic; some load external information into the template to be used by later variables
  - Some require beginning and ending tags

- **Comments**: syntax **{# #}**, for single-line comments only (multiline comments need the comment tag)

- **Template Inheritance**: most powerful part of Django's template engine; allows you to build a base "skeleton" template that contains all the common elements of your site and defines ***blocks*** that child templates can override (and fill in with content)

- **Automatic HTML escaping**: feature of Django; by default every template automatically escapes the output of every variable tag, specifically "**<**", "**>**", single quotes, double quotes, and "**&**" (converted to &lt, &gt, &#x27, &quot, and &amp respectively)
  - Prevents issues when generating HTML from templates that contain characters that can affect the resulting HTML