

The use of Templates

Chapter 3

COMP222-Chapter 3

1

Objectives

- In this chapter we'll update our Pages app to have a homepage and an about page.
- We'll also learn about Django's class-based views and templates which are the building blocks for the more complex web applications built later on in the book.

COMP222-Chapter 3

2

Initial Setup

- In the previous chapters, we have completed the following steps:
 - create a new directory for our code
 - install Django in a new virtual environment
 - create a new Django project
 - create a new pages app with corresponding views and URLConfs
 - update settings.py

COMP222-Chapter 3

3

MTV framework (revisited)

- In Chapter 2, we have talked about Django's MTV framework, where
 - **M** stands for "Model,"
 - **T** stands for "Template,"
 - **V** stands for "View,"
- In Chapter 2, we have covered the views function. This chapter will talk about templates.

4

Templates

- Django uses templates so that individual HTML files can be served by a view to a web page specified by the URL.
- Templates, Views, and URLs work together closely.
- The URLs control the initial route, the entry point into a page, such as /about.
- The views contain the logic or the “what”.
- The template has the HTML.
- For web pages that rely on a database model, it is the view that does much of the work to decide what data is available to the template.

COMP222-Chapter 3

5

Template system basics

- A Django **template** is a text document, or a normal Python string, that is marked up using the Django template language, with the intention to separate the presentation of a document from its data.
- A template can contain template tags and variables.
- A template defines placeholders and various bits of basic logic (template tags) that regulate how the document should be displayed.
- Usually, templates are used for producing HTML.
- The template system is meant to express presentation, not program logic. The objective is to separate data and presentation.

6

Template: An example

- This example template describes an HTML page that thanks a person for placing an order with a company. Think of it as a form letter.
- Any text surrounded by a pair of braces, e.g. `{{ person_name }}` is a *variable*.
- Any text surrounded by curly braces and percent signs e.g., `{% if ordered_warranty %}` is a *template tag*.
- A *template tag* tells the template system to “do something”.
- A *filter* is used to alter the format of a variable and is attached with a *pipe character* (`|`), e.g. `{{ ship_date|date:"F j, Y" }}` where we are passing the `ship_date` variable to the `date` filter with the arguments “F j, Y”.

```
<html>
<head><title>Ordering notice</title></head>
<body>

<h1>Ordering notice</h1>
<p>Dear {{ person_name }},</p>

<p>Thanks for placing an order from {{ company }}. It's scheduled to ship
on {{ ship_date|date:"F j, Y" }}.</p>
<p>Here are the items you've ordered:</p>
<ul>
{% for item in item_list %}<li>{{ item }}</li>{% endfor %}
</ul>

{% if ordered_warranty %}
  <p>Your warranty information will be included in the packaging.</p>
{% else %}
  <p>You didn't order a warranty, so you're on your own when
the products inevitably stop working.</p>
{% endif %}

<p>Sincerely,<br />{{ company }}</p>

</body>
</html>
```

A variable

A filter

A template tag

7

Basic template-tags: if/else

The if tag may take one or several `{% elif %}` clauses.

```
{% if athlete_list %}
    Number of athletes: {{ athlete_list|length }}
{% elif athlete_in_locker_room_list %}
    <p>Athletes should be out of the locker room soon! </p>
{% elif ...
    ...
{% else %}
    <p>No athletes. </p>
{% endif %}
```

8

Basic template-tags: for

- The `{% for %}` tag allows you to loop over each item in a sequence. As in Python's for statement, the syntax is for X in Y, where Y is the sequence to loop over and X is the name of the variable to use for a particular cycle of the loop

```
<ul>
{% for athlete in athlete_list reversed %}
  <li>{{ athlete.name }}</li>
{% endfor %}
</ul>
```

- If you need to access the items in a dictionary,

```
{% for key, value in data.items %}
  {{ key }}: {{ value }}
{% endfor %}
```

9

Basic template-tags: for – an optional `{% empty %}` clause

- The for tag supports an optional `{% empty %}` clause that lets you define what to output if the list is empty

```
{% for athlete in athlete_list %}
  <p>{{ athlete.name }}</p>
{% empty %}
  <p>There are no athletes. Only computer programmers.</p>
{% endfor %}
```

which is equivalent to

```
{% if athlete_list %}

  {% for athlete in athlete_list %}
    <p>{{ athlete.name }}</p>
  {% endfor %}

{% else %}
  <p>There are no athletes. Only computer programmers.</p>
{% endif %}
```

10

Basic template-tags: for (cont'd)

- There is no support for breaking out of a loop before the loop is finished.
- If you want to accomplish this, change the variable you're looping over so that it includes only the values you want to loop over.
- Similarly, there is no support for a continue statement that would instruct the loop processor to return immediately to the front of the loop.

11

Basic template-tags: for – a template variable called forloop

- Within each `{% for %}` loop, you get access to a template variable called `forloop`.
- This variable has a few attributes that give you information about the progress of the loop.
- The `forloop` variable is only available within loops. After the template parser has reached `{% endfor %}`, `forloop` disappears.
 - `forloop.counter` is always set to an integer representing the number of times the loop has been entered. This is one-indexed, so the first time through the loop, `forloop.counter` will be set to 1. Here's an example:

```
{% for item in todo_list %}
  <p>{{ forloop.counter }}: {{ item }}</p>
{% endfor %}
```

- `forloop.counter0` is like `forloop.counter`, except it's zero-indexed. Its value will be set to 0 the first time through the loop.

12

Basic template-tags: for – a template variable called forloop

- `forloop.revcounter` is always set to an integer representing the number of remaining items in the loop. The first time through the loop, `forloop.revcounter` will be set to the total number of items in the sequence you're traversing. The last time through the loop, `forloop.revcounter` will be set to 1.
- `forloop.revcounter0` is like `forloop.revcounter`, except it's zero-indexed. The first time through the loop, `forloop.revcounter0` will be set to the number of elements in the sequence minus 1. The last time through the loop, it will be set to 0.
- `forloop.first` is a Boolean value set to True if this is the first time through the loop. This is convenient for special-casing:

```
{% for object in objects %}
  {% if forloop.first %}<li class="first">
    {% else %}<li>{% endif %}
    {{ object }}
  </li>
{% endfor %}
```

13

Basic template-tags: for – a template variable called forloop

- `forloop.last` is a Boolean value set to True if this is the last time through the loop. A common use for this is to put pipe characters between a list of links. Another common use for this is to put a comma between words in a list.

```
{% for link in links %}
  {{ link }}{% if not forloop.last %} | {% endif %}
{% endfor %}
```

- `forloop.parentloop` is a reference to the `forloop` object for the parent loop, in case of nested loops. Here's an example:

```
{% for country in countries %}
  <table>
    {% for city in country.city_list %}
      <tr>
        <td>Country #{{ forloop.parentloop.counter }}</td>
        <td>City #{{ forloop.counter }}</td>
        <td>{{ city }}</td>
      </tr>
    {% endfor %}
  </table>
{% endfor %}
```

14

Basic template-tags: comment

- The Django template language allows for comments. To designate a comment, use `{# #}`: `{# This is a comment #}`
- Comments using this syntax cannot span multiple lines.
- If you want to use multi-line comments, use the `{% comment %}` template tag.

```
{% comment %}
This is a
multi-line comment.
{% endcomment %}
```

15

Template Filters

- Template filters are simple ways of altering the value of variables before they're displayed. Filters use a pipe character, like this:

```
{{ name|lower }}
```

-- which converts text to lowercase

- Filters can be chained, e.g., that takes the first element in a list and converts it to uppercase: `{{ my_list|first|upper }}`
- Some filters take arguments. A filter argument comes after a colon and is always in double quotes. E.g.: `{{ bio|truncatewords:"30" }}`

16

Location for templates

- By default, Django looks within each app for templates. So a home.html template in pages app would be located at [myTestSite/pages/templates/home.html](#). However, we can edit our [settings.py](#) file to tell Django to look in a project-level folder for templates.

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debug',
                'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',
                'django.contrib.messages.context_processors.messages',
            ],
        },
    },
]
```

- Create a project-level folder called templates and an HTML file called home.html.
- Next is a one-line change to the setting 'DIRS' under TEMPLATES in [settings.py](#) file.
`'DIRS': [os.path.join(BASE_DIR, 'templates')],`
- BASE_DIR is a variable that stores the path to the directory in which your project's settings.py module is contained.
- We make use of `os.path.join()` to mash together the BASE_DIR variable and 'templates'. For example, it would yield `<workspace>/myTestSite/templates/`.

COMP222-Chapter 3

17

Class-Based Views

- In our view we'll use the built-in `TemplateView` to display our template. Update the [pages/views.py](#) file.

```
# pages/views.py

from django.views.generic import TemplateView

class HomePageView(TemplateView):

    template_name = 'home.html'
```

- Note that we've capitalized our view since it's now a Python class. Classes, unlike functions, should always be capitalized.
- The `TemplateView` already contains all the logic needed to display our template, we just need to specify the template's name.
- Then we can add a simple headline to our home.html file.
`<h1>Homepage.</h1>`

COMP222-Chapter 3

18

Class-Based Views vs function-based views

```
# pages/views.py
from django.views.generic import TemplateView
class HomePageView(TemplateView):
    template_name = 'home.html'
```

Class-based view (capitalized) of this chapter

```
# pages/views.py
from django.http import HttpResponse
def homePageView(request):
    return HttpResponse('Hello, World!')
```

function-based view of previous chapter

COMP222-Chapter 3

19

URLs – Edit project-level urls.py

- The last step is to update our URLConfs.
- Recall from Chapter 2 that we need to make updates in two locations.
 - First we update the project-level urls.py file to point at our pages app and then within pages we match the views to routes.
- Let's start with the project-level `urls.py` file.

```
from django.contrib import admin
from django.urls import path, include
urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('pages.urls')),
]
```

- We've imported `include` on the second line next to `path` and then created a new `urlpatterns` for our pages app.
- Now whenever a user visits the homepage at `/`, they will first be routed to the pages app and then to the `HomePageView` view.

COMP222-Chapter 3

20

URLs – Edit app-level urls.py

- Next edit our app-level urls.py file (within the pages app) to use the Class-Based view with the following code:

```
# pages/urls.py
```

```
from django.urls import path
```

```
from . import views
```

The period reference the current directory, which is our pages app containing both views.py and urls.py.

```
urlpatterns = [
```

```
    path('', views.HomePageView.as_view(), name='home'),
```

```
]
```

- If the user requests the homepage, represented by the empty string "", then use the view called `HomePageView`.
- When using Class-Based Views, you always add `as_view()` at the end of the view name.

- Refresh your browser to see the new homepage.

COMP222-Chapter 2

21

Add an About page

- The process for adding an about page is similar to what we just did.
 - We'll create a new template file, a new view, and a new url route.
- Quit the server with Control-c and create a new template called `about.html` in the project-level template folder with `<h1>About page.</h1>` as content.
- Update the `pages/views.py` file by adding the following:

```
class AboutPageView(TemplateView):
    template_name = 'about.html'
```

- And then connect it to a url at `about/`. This is done by adding the following line to the app-level `urls.py` file (i.e. `pages/urls.py`).

```
path('about/', views.AboutPageView.as_view(), name='about'),
```

- Refresh your browser to see the new “About page”.

COMP222-Chapter 3

22

Extending Templates

- Most web sites contain content that is repeated on every page (header, footer, etc).
- It will be nice if there is one place for our header code that would be inherited by all other templates.
- Let's create a base.html file containing a header with links to our two pages, the homepage and the about page, as written in the [pages/urls.py](#) file.
- Before we start, let's talk about template tags.

COMP222-Chapter 3

23

Template tags

- Django has a minimal templating language for adding links and basic logic in our templates.
- Template tags take the form of `{% something %}` where the “something” is the template tag itself.
- To add URL links in our project, we can use the built-in url template tag which takes the URL pattern name as an argument.
- Remember how we added optional URL names to our url routers?

```
path('about/', views.AboutPageView.as_view(), name='about'),
```

 The url tag uses these names to automatically create links for us.
- The URL route for our about page is called 'about', therefore to configure a link to it we would use `{% url 'about' %}`.

COMP222-Chapter 3

24

Content of base.html

- The content of base.html is as follows:

```
<!-- templates/base.html -->

<header>

    <a href="{% url 'home' %}">Home</a> | <a href="{% url 'about' %}">About</a>

</header>

{% block content %}

{% endblock %}
```

- At the bottom we've added a block tag called content. Blocks can be **overwritten** by child templates via inheritance.

COMP222-Chapter 3

25

Extending the base.html

- Let's update our home.html and about.html to extend the base.html template.
- That means we can reuse the same code from one template in another template.
- The Django templating language comes with an **extends** method that we can use for this.

```
<!-- templates/home.html -->

{% extends 'base.html' %}

{% block content %}

<h1>Homepage.</h1>

{% endblock %}
```

```
<!-- templates/about.html -->

{% extends 'base.html' %}

{% block content %}

<h1>About page.</h1>

{% endblock %}
```

When the template engine sees the `{% extends %}` tag, noting that this template is a child template. The engine immediately loads the parent template – in this case, base.html. At that point, the template engine replaces the `{% block %}` tags in base.html with the contents of the child template.

- Refresh your browser to see the home page and the “About page”. You'll see the header is magically included in both locations.

COMP222-Chapter 3

26

Should user-submitted data be trusted blindly?

- When generating HTML from templates, there's always a risk that a variable will include characters that affect the resulting HTML. For example, consider this template fragment: `Hello, {{ name }}`.
- If the user entered his name as this: `<script>alert('hello')</script>` the template would be rendered as:
`Hello, <script>alert('hello')</script>`
which means the browser would pop-up a JavaScript alert box!
- Similarly, what if the name contained a '`<`' symbol, like `username`
- That would result in: `Hello, username`
which, in turn, would result in the remainder of the Web page being bolded!
- User-submitted data shouldn't be trusted blindly and inserted directly into your Web pages.

27

Automatic HTML escaping

- To avoid this problem, take advantage of Django's automatic HTML escaping.
- By default in Django, every template automatically escapes the output of every variable tag. Specifically, these five characters are escaped:
 - `<` is converted to `<`;
 - `>` is converted to `>`;
 - `'` (single quote) is converted to `'`
 - `"` (double quote) is converted to `"`;
 - `&` is converted to `&`;

28

How to turn off auto-escaped

- Sometimes, template variables contain data that you intend to be rendered as raw HTML, in which case you don't want their contents to be escaped.
- To disable auto-escaping for an individual variable, use the safe filter, e.g. `{{ data|safe }}`
- To control auto-escaping for a template, use the autoescape tag, e.g.


```
{% autoescape off %}
Hello {{ name }}
{% endautoescape %}
```

29

auto-escaping tag

- The auto-escaping tag passes its effect on to templates that extend the current one as well as templates included via the include tag, just like all block tags.

```
# base.html

{% autoescape off %}
<h1>{% block title %}{% endblock %}</h1>
{% block content %}
{% endblock %}
{% endautoescape %}

# child.html

{% extends "base.html" %}
{% block title %}This & that{% endblock %}
{% block content %}{{ greeting }}{% endblock %}
```

Because auto-escaping is turned off in the base template, it will also be turned off in the child template, resulting in the following rendered HTML when the greeting variable contains the string `Hello!`:

```
<h1>This & that</h1>
<b>Hello!</b>
```

30

Summary: what have you learnt?

- Django's class-based views and templates
- Edit [settings.py](#) file to tell Django to look in a project-level folder for templates
- The built-in TemplateView
- Extend templates