Django Templates

In Module 4, we created a very simple view to display a message in the browser. This is obviously a long way from a fully functioning, modern website. For one we're missing a site template.

Site templates, at their most basic, are HTML files that are displayed by your browser. All websites—from simple, static pages to interactive web applications that work on multiple devices—are built on HTML.

Modern interactive websites are more complex. For example, a modern website will add Cascading Style Sheets (CSS), semantic markup and JavaScript in the front end to create the user experience with a back end like Django supplying the data to show in the template. However, the fundamentals stay the same.

Django's approach to web design is simple—keep Django logic and code separate from design. This means that it's possible for a designer to create a complete front end that includes HTML, CSS, imagery and user interaction without ever having to write a single line of Python or Django code.

In this module you'll learn how to build a simple HTML site template and then add Django template tags to the HTML files to create a template capable of displaying data from the database.

Before we do this, we need to dive back into your project settings and structure so you can understand where Django looks for templates and how it decides which template to display.

Template Settings

For Django to show your site template, it first must know where to look for the template file(s). This is achieved by the TEMPLATES list in settings.py. By default the TEMPLATES list contains a dictionary of settings for your templates. Let's have a closer look at this file:

On line 58 the DIRS key contains a list of paths to folders containing templates. Paths can be absolute or relative. The default is an empty list.

And in line 59, APP_DIRS is set to True by default. When APP_DIRS is set to True Django will look for a folder named templates in each of your apps.

Not all template files will be tied to a particular app. The DIRS setting is useful for linking to templates that exist elsewhere in your project structure. In our project, we'll have a site template that is not a part of the pages app, so we need to add a path to DIRS. Pause the video and enter this text into your editor or copy it from your transcript (changes in bold):

```
'BACKEND': django.template.backends.django.DjangoTemplates',
'DIRS': [os.path.join(BASE_DIR, 'mfdw_site/templates')],
'APP_DIRS': True,

# ...
```

This looks complicated, but is easy to understand—os.path.join is a Python command to create a file path by joining strings together (concatenating). In this example, we're appending mfdw_site/templates to our project path to create the full path to our templates.

Before we move on, we need to create a templates folder in our site folder. Once you have created the new folder, your project structure should look like this:

```
\mfdw_project
   \mfdw_root
         \mfdw_site
               \templates
         # more files ...
```

Static Files

Django treats static files—images, CSS and JavaScript—differently to templates. Django's creators wanted it to be fast and scalable, so right from the beginning Django was designed to make it easy to serve static media from a different server to the one the main Django application was running on.

Django achieves speed and scalability by keeping static media in a different directory to the rest of the application. This directory is defined in the settings.py file and is called static by default:

This line should be at or near the end of your settings.py file. We need to add another setting so that Django can find the static files for our site. Add the following below the STATIC_URL setting. Pause the video and enter the code into your editor or copy it from your transcript:

```
STATICFILES_DIRS = [
    os.path.join(BASE_DIR, 'mfdw_site/static'),
]
```

The STATICFILES_DIRS list serves the same function for static files as the DIRS list does for templates. In this case we're telling Django to look for static files in the static folder in our site folder. Now we need to create a static folder in our site folder. Once you have created the new folder, your project directory will look like this:

```
\mfdw_root
   \mfdw_site
         \static
         \templates
   # more files ...
```

Site Template and Static Files

Now we've configured Django to be able to serve our templates, it's time to create our site template and static files.

We'll be creating four files:

1. base.html. The base HTML template for our site
2. main.css. CSS styles for our template
3. logo.jpg. 30x30 pixels image for the logo
4. top_banner.png. 800x200 pixels banner for the site

As the base.html and CSS files are quite large, I don't expect you to enter them manually, however if you do want to type them out, they're listed in the transcript. The comment at the top of each listing shows you where to create the file.

I also don't expect you to create the two small media files that we're using in the template, so I have included them as well as the CSS and base.html files in the MODULE6_STATIC.zip file which you can download from the menu on the left.

The base.html file needs to be put in the \mfdw_site\templates\ folder. Let's have a closer look at this file:

This file is mostly plain HTML5 markup. Note the semantic elements—<aside>, <section> and <footer>. Semantic elements provide additional meaning to the browser on how a piece of content should be treated. If you are not familiar with HTML and want to learn more, you can check out W3schools.

The base.html template also includes your first Django template tag—{% static %}. The static tag is used to link media in your templates to the STATIC_ROOT of your project. As we are in development and haven't configured Django for production, STATIC_ROOT is the same as your STATIC_URL setting (/static/). This is how we put the static tag to use in the template:

First in Line 1, we load the static tag into the template;

Then, wherever we need to load static media, we pass the media filename (e.g., logo.jpg) to the static tag, which will automatically prepend the static media directory (e.g., /static/ logo.jpg). You can see this in action in lines 7, 12 and 13.

Listing 1—base.html

```
# \mfdw_site\templates\base.html

1  {% load static %}
2  <!doctype html>
3  <html>
4  <head>
5  <meta charset="utf-8">
6  <title>Untitled Document</title>
7  <link href="{% static 'main.css' %}" rel="stylesheet"
   ype="text/css">
8  </head>
9  <body>
10 <div id="wrapper">
11   <header id="header">
12 <div id="logo"><img src="{% static 'logo.jpg' %}"
   alt=""/></div>
13 <div id="topbanner"><img src="{% static 'top_
   banner.png' %}" width="800" height="200" alt=""/></div>
14   </header>
15   <aside id="leftsidebar">
16     <nav id="nav">
17 <ul><li>Menu 1</li><li>Menu 2</li><li>Menu 3</li></ ul>
18     </nav>
19   </aside>
20   <section id="main">
21     <h1>Welcome!</h1>
22     <p>This is the site template</p>
23   </section>
24 <footer id="footer">Copyright &copy; 2017 Meandco Web
   Design</footer>
25 </div>
26 </body>
27 </html>
```

Listing 2—main.css

Main.css needs to be put into the \mfdw_site\static\ folder.

```
# \mfdw_site\static\main.css

@charset "utf-8";
#header {
  border-style: none;
  width: 800px;
  height: auto;
}
#wrapper {
  margin-top: 0px;
  margin-left: auto;
  margin-right: auto;
  background-color: #FFFFFF;
  width: 800px;
}
body {
  background-color: #E0E0E0;
font-family: Gotham, "Helvetica Neue", Helvetica, Arial,
  sans-serif;
font-size: 0.9em;
text-align: justify;
  color: #474747;
}
#footer {
  text-align: center;
  font-size: 0.8em;
  margin-top: 5px;
  padding-top: 10px;
  padding-bottom: 10px;
  background-color: #FFFFFF;
  border-top: thin solid #BBBBBB;
  clear: both;
  color: #969696;
}
#nav li {
  padding-top: 10px;
  padding-bottom: 10px;
  font-size: 1em;
  list-style-type: none;
  border-bottom: thin solid #5F5F5F;
  color: #4C4C4C;
  left: 0px;
  list-style-position: inside;
  margin-left: -10px;
}
#nav li a {
  text-decoration: none;
}
```

```
#leftsidebar {
 width: 180px;
 height: 350px;
 float: left;
}
#main {
 width: 560px;
 float: left;
 margin-left: 20px;
 margin-right: 10px;
 padding-right: 10px;
}
```

This file is standard CSS. If you are not familiar with CSS, you can either learn more about style sheets as you go, or if you want to learn more now, you can check out W3schools. I've added links to W3schools tutorials in the Course Resources section in the menu on the left.

The logo.jpg and top_banner.png files can either be extracted from the Module 6 zip file, or you can create your own. Either way, they both need to be put into the `\mfdw_site\static\` folder.

Updating Your View

Now we have the template and static files in place, we need to update our views.py file. Pause the video and enter the code into your editor or copy it from your transcript (change in bold):

```
    # pages\views.py

from django.shortcuts import render
# from django.http import HttpResponse

def index(request):
    # return HttpResponse("<h1>The Meandco Homepage</ h1>")
    return render(request, 'base.html' )
```

For our new view, we have replaced the call to HttpResponse() with a call to render(). I have commented out the original lines so that you can more easily see the changes. You don't have to remove the import from django.http, but it's good practice not to import modules that you are no longer using.

render() is a special Django helper function that creates a shortcut for communicating with a web browser. If you remember from Module 4, when Django receives a request from a browser, it finds the right view and the view returns a response to the browser.

In the example from Module 4, we simply returned some HTML text. However, when we wish to use a template, Django first must load the template, create a context—which is basically a dictionary of variables and associated data that is passed back to the browser—and then return a response.

You can code each of these steps separately in Django, but in the vast majority of cases it's more common (and easier) to use Django's render() function which provides a shortcut that provides all three steps in a single function.

When you supply the original request, the template and the context directly to render(), it returns the appropriately formatted response without you having to code the intermediate steps.

In our modified views.py, we are simply returning the original request object from the browser and the name of our site template. We'll be getting to the context a bit later in the module.

Once you have modified your views.py file, save it and fire up the development server. If you navigate to the site root, you should see your shiny new site template.

It Broke!—Django's Error Page

Creating and coding templates for the first time is almost certain to fail as it's difficult to get everything right first go, so in this video I'm going to digress a bit and take a closer look at Django's error page.

For the sake of this exercise, I'm going to deliberately create a template error by renaming our base.html file.

If you had an error in your template structure or settings, you will get a page that looks like this.

Take some time to explore the error page and get to know the various bits of information it gives you. Here are some things to note:

- At the top of the page, you get the key information about the exception: the type of exception, any parameters to the exception (the TemplateDoesNotExist message in this case), the file in which the exception was raised and the offending line number.
- As this is a template error, Django will display a Template-loader postmortem to show you where things went wrong.
- Under the key exception information, the page displays the full Python traceback for this exception. This is like the standard traceback you get in Python's command-line interpreter, except it's more interactive.
- For each level (frame) in the traceback, Django displays the name of the file, the function/method name, the line number and the source code of that line. Click the line of source code (in dark gray), and you'll see several lines from before and

after the erroneous line, to give you some context on the code that led to the error.

- Click Local vars under any frame in the stack to view a table of local variables and their values in that frame at the exact point in the code at which the exception was raised. This debugging information can be a great help.
- Note the Switch to copy-and-paste view text under the Traceback header. Click the link and the traceback will switch to an alternate version that can be easily copied and pasted. Use this when you want to share your exception traceback with others to get technical support.
- Underneath, the Share this traceback on a public Web site button will do this work for you in just one click. Click it to post the traceback to dpaste where you'll get a distinct URL that you can share with other people.
- Next, the Request information section includes a wealth of information about the incoming Web request that spawned the error: GET and POST information, cookie values and meta information.
- Below the Request information section, the Settings section lists all the settings for the current Django installation.

The Django error page can display a range of different information depending on the type of error. You should consider it your number one troubleshooting tool when your Django app is not working.

It's obvious that much of this information is sensitive. As it exposes the innards of your Python code and Django configuration, a malicious person could use it to attempt to reverse-engineer your web application.

For that reason, the Django error page is only displayed when a Django project is in debug mode. When we created the project with startproject, Django automatically put the site in debug mode. This is OK for now; just know that you must never run a production site in debug mode. We'll be setting DEBUG to False when we deploy the site to the Internet in Module 12.

The Pages Template

Now we've got the site template up and running, we need to create a template for our pages app. If you remember the DRY principle from Module 1, we don't want to repeat the same information in all our templates, so we want our pages template to inherit from the site template.

Implementing inheritance is easy in Django—you define replaceable blocks in each template so that child templates can replace sections of the parent template with content unique to the child. This is easier to understand with an example, so let's modify our base.html file. Pause the video and enter the code into your editor or copy it from your transcript (changes in bold):

```
# \mfdw_site\templates\base.html

# ...
<section id="main">
  {% block content %}
    <h1>Welcome!</h1>
    <p>This is the site template</p>
  {% endblock content %}
</section>
# ...
```

The two lines of code we have added are a set of Django block tags.

Block tags have the following form:

{% block <name> %}{% endblock <name> %}

The second <name> declaration isn't required although it's highly recommended, especially if you are using multiple block tags. You can name your block tags anything you like—in our example, we're naming the block tag content.

A block tag defines a block of template code that can be replaced by any child templates that inherit from the template.

Next, we need to create a template for our pages app that inherits from the site template. If you remember from earlier in the module, the APP_DIRS setting defaults to True. This means that Django will search all your apps for a folder named templates.

Go ahead and create a templates folder inside your pages app now. We then need to create another folder inside that. This second folder is named pages after the app. Your pages app folder structure should look like this when you are done:

```
\pages
   \templates
            \pages
```

So, why a second pages folder?

What if you have two apps in your project that each have a template named index.html? Django uses short circuit logic when searching for templates, so when it goes searching for templates/index.html, it will use the first instance it finds and that may not be in the app you wanted!

Adding the inner pages folder is an example of namespacing your templates. By adding this folder you can make sure Django retrieves the right template.

Let's go ahead and create our page template—page.html. Pause the video and enter this code into your editor or copy it from your transcript:

```
# \pages\templates\pages\page.html

1  {% extends "base.html" %}
2
3  {% block content %}
4  <h1>Welcome!</h1>
5    <p>This is the page template</p>
6  {% endblock content %}
```

Let's have a closer look at this:

Line 1 is where the magic of inheritance comes in. We're telling Django that the page template extends, or adds to, the base (site) template.

And in lines 3 to 6, we're declaring a set of block tags named content. This block will replace the block of the same name in the parent template.

Notice that we have not repeated a single line of code from base.html— we're loading the file with the extends tag and replacing the content block with a new content block.

Now that we've created the page template, we need to modify views.py to show the template. Pause the video and enter the code into your editor or copy it from your transcript (changes in bold):

```
1  from django.shortcuts import render
2  from django.http import HttpResponse
3
4  def index(request):
5      return render(request, 'pages/page.html' )
```

Only a small change this time—instead of using the site template, we're now using the page template. If you run the development server, your home page should look like this.

Notice that it's saying "This is the page template", not "This is the site template". This demonstrates that, with only a few lines of code, Django's templates allow you focus on the things that are different on the page and ignore the parts that are the same.

Which I'm sure you'll agree is really cool.

Module Summary

We've covered a bit of ground in this module, so it's good to step back and have a look at what we've achieved.

First, we learned how Django discovers templates, how it separates static files from your applications to make it easier to scale a Django website and how to add and modify template settings.

Along the way we explored the Django error page and how it can be used to troubleshoot errors in our apps. Finally, we learned how easy it is to turn a basic HTML template into a Django template capable of inheriting common content.

Of course, the templates are still static—the content is hard-coded into the template. In the next module we'll learn how to display dynamic content in a Django page as well as writing some more complex views.