**Chapter 05: Pages App**

Let’s build a homepage for our new project. For now this will be a static page meaning it will not interact with the database in any way. Later on it will be a dynamic page displaying books for sale but… one thing at a time.

It’s common to have multiple static pages in even a mature project such as an About page so let’s create a dedicated pages app for them. On the command line use the startapp command again to make a pages app.

docker-compose exec web python manage.py startapp pages

Then add it to our INSTALLED\_APPS setting. We’ll also update TEMPLATES so that Django will look for a project-level templates folder. By default Django looks within each app for a templates folder, but organizing all templates in one space is easier to manage.

# django\_project/settings.py

INSTALLED\_APPS = [

"django.contrib.admin",

"django.contrib.auth",

"django.contrib.contenttypes",

"django.contrib.sessions",

"django.contrib.messages",

"django.contrib.staticfiles",

# Local

"accounts.apps.AccountsConfig",

"pages.apps.PagesConfig", # new

]

TEMPLATES = [

{

...

"DIRS": [BASE\_DIR / "templates"], # new

...

}

]

Note that updating the DIRS setting means that Django will also look in this new folder; it will still look for any templates folders within an app.

**Templates**

Moving on it is time to create that new templates directory on the command line.

mkdir templates

Then with your text editor create two new files within it: templates/\_base.html and templates/home.html. The first base level file will be inherited by all other files; home.html will be our homepage.

Why call the base template \_base.html with the underscore instead of base.html? This is an optional practice but some developers prefer to add an underscore, \_, to files solely meant to be inherited by other files.

In the base file we’ll include the bare minimum needed and add block tags for both title and content. Block tags give higher-level templates the option to override just the content within the tags. For example, the homepage will have a title of “Home” but we want that to appear between html <title></title> tags. Using block tags make it easier to update this content, as needed, in inherited templates.

Why use the name content for the main content of our project? This name could be anything–main or some other generic indicator–but using content is a common naming convention in the Django world. Can you use something else? Absolutely. Is content the most common one you’ll see? Yes.

<!-- templates/\_base.html -->

<!DOCTYPE html>

<html>

<head>

<meta cahrset="utf-8">

<title>{% block title %}Bookstore{% endblock title %}</title>

</head>

<body>

<div class="container">

{% block content %}

:% endblock content %}

</div>

</body>

</html>

Now for the homepage which will simply say “This is our home page.” for now.

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

{% extends "\_base.html" %}

{% block title %}Home{% endblock title %}

{% block content %}

<h1>This is our home page.</h1>

{% endblock content %}

**URLs and Views**

Every webpage in our Django project needs a urls.py and views.py file to go along with the template. For beginners the fact that order doesn’t really matter here–we need all 3 files and really often a 4th, models.py, for the database–is confusing. Generally I prefer to start with the urls and work from there but there is no “right way” to build out this connected web of Django files.

Let’s start with our project-level urls.py to set the proper path for webpages within the pages app. Since we want to create a homepage we add no additional prefix to the URL route which is designated by the empty string "". We also import include on the second line to concisely add the pages app to our main urls.py file.

django\_project/urls.py

from django.contrib import admin

from django.urls import path, include # new

urlpatterns = [

path(“admin/”, admin.site.urls),

path(“”, include(pages.urls)), # new

]

Next we create a pages/urls.py file with our text editor. This file will import the HomePageView and set the path, again, to the empty string "". Note that we provide an optional, but recommended, named URL of "home" at the end. This will come in handy shortly.

pages/urls.py

from django.urls import path

from .views import HomePageView

urlpatterns = [

path(“”, HomePageView.as\_view(), name=”home”).

]

Finally we need a views.py file. We can leverage Django’s built-in TemplateView so that the only tweak needed is to specify our desired template, home.html.

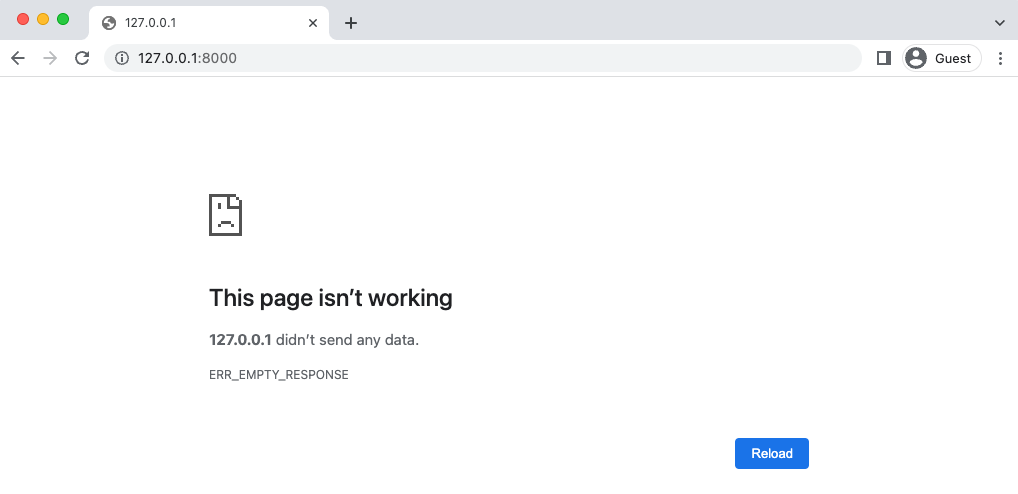
pages/views.py

from django.views.generic import TemplateView

class HomePageView(TemplateView):

template\_name = “home.html

We’re almost done. If you navigate to the homepage now at http://127.0.0.1:8000/ you’ll actually see an error. But what’s causing it? Since we’re running the container in background detached mode–that -d flag–we must explicitly check the logs to see console output.



Django welcome page

In the shell type docker-compose logs which will turn up an error “ModuleNotFoundError: No module named ‘pages.urls’”. What’s happening is that Django does not automatically update the django\_project/settings.py file for us based on a change. In a non-Docker world stopping and restarting the server does the trick since the settings variables are loaded upfront. We must do the same here which means typing docker-compose down and then docker-compose up -d to load the new books app in properly.

docker-compose down

docker-compose up -d

Refresh the homepage now and it will work.



**Tests**

Time for tests. For our homepage we can use Django’s SimpleTestCase which is a special subset of Django’s TestCase that is designed for webpages that do not have a model included.

Testing can feel overwhelming at first, but it quickly becomes a bit boring. You’ll use the

same structure and techniques over and over again. In your text editor, update the existing pages/tests.py file. We’ll start by testing the template.

# pages/tests.py

from django.test import SimpleTestCase

from django.urls import reverse

# Create your tests here.

class HomepageTests(SimpleTestCase):

def test\_url\_exists\_at\_correct\_location(self):

response = self.client.get("/")

self.assertEqual(response.status\_code, 200)

def test\_homepage\_url\_name(self):

response = self.client.get(reverse("home"))

self.assertEqual(response.status\_code, 200)

At the top we import SimpleTestCase as well as reverse which is useful for testing our URLs. Then we create a class called HomepageTests that extends SimpleTestCase and within it add a method for each unit test.

Note that we’re adding self as the first argument of each unit test. This is a Python convention that is worth repeating.

It is best to be overly descriptive with your unit test names but be aware that each method must start with test to be run by the Django test suite.

The two tests here both check that the HTTP status code for the homepage equals 200 which means that it exists. It does not yet tell us anything specific about the contents of the page. For test\_url\_exists\_at\_correct\_location we’re creating a variable called response that accesses the homepage (/) and then uses Python’s assertEqual to check that the status code matches 200. A similar pattern exists for test\_homepage\_url\_name except that we are calling the URL name of home via the reverse method. Recall that we added this to the pages/urls.py file as a best practice. Even if we change the actual route of this page in the future, we can still refer to it by

the same home URL name.

To run our tests execute the command prefaced with docker-compose exec web so that it runs within Docker itself.

docker-compose exec web python manage.py test

Creating test database for alias 'default'...

System check identified no issues (0 silenced).

..

----------------------------------------------------------------------

Ran 4 tests in 0.126s

OK

Destroying test database for alias 'default'...

Why does it say 4 tests when we only created 2? Because we’re testing the entire Django projec and in the previous chapter under users/tests.py we added two tests for the custom user model. If we wanted to only run tests for the pages app we simply append that name onto the command so docker-compose exec web python manage.py test pages.

**Testing Templates**

So far we’ve tested that the homepage exists, but we should also confirm that it uses the correct template. SimpleTestCase comes with a method assertTemplateUsed just for this purpose! Let’s use it.

from django.test import SimpleTestCase

from django.urls import reverse

# Create your tests here.

class HomepageTests(SimpleTestCase):

def test\_url\_exists\_at\_correct\_location(self):

response = self.client.get("/")

self.assertEqual(response.status\_code, 200)

def test\_homepage\_url\_name(self): # new

response = self.client.get(reverse("home"))

self.assertEqual(response.status\_code, 200)

def test\_homepage\_template(self):

response = self.client.get("/")

self.assertTemplateUsed(response, "home.html")

We’ve created a response variable again and then checked that the template home.html is used. Let’s run the tests again.

docker-compose exec web python manage.py test pages

Creating test database for alias 'default'...

System check identified no issues (0 silenced).

...

----------------------------------------------------------------------

Ran 3 tests in 0.009s

OK

Destroying test database for alias 'default'...

Did you notice something different in that command? We added the name of our app pages so that only the tests within that app were run. At this early state it’s fine to run all the tests, but in larger projects if you know that you’ve only added tests within a specific app, it can save time to just run the updated/new tests and not the entire suite.

**Testing HTML**

Let’s now confirm that our homepage has the correct HTML code and also does not have incorrect text. It’s always good to test both that tests pass and that tests we expect to fail do, actually, fail!

# pages/tests.py

from django.test import SimpleTestCase

from django.urls import reverse

# Create your tests here.

class HomepageTests(SimpleTestCase):

def test\_url\_exists\_at\_correct\_location(self):

response = self.client.get("/")

self.assertEqual(response.status\_code, 200)

def test\_homepage\_url\_name(self):

response = self.client.get(reverse("home"))

self.assertEqual(response.status\_code, 200)

def test\_homepage\_template(self):

response = self.client.get("/")

self.assertTemplateUsed(response, "home.html")

def test\_homepage\_contains\_correct\_html(self): # new

response = self.client.get("/")

self.assertContains(response, "home page")

def test\_homepage\_does\_not\_contain\_incorrect\_html(self): # new

response = self.client.get("/")

self.assertNotContains(response, "Hi there! I should not be on the page.")

Run the tests again.

docker-compose exec web python manage.py test

Creating test database for alias 'default'...

System check identified no issues (0 silenced).

.....

----------------------------------------------------------------------

Ran 7 tests in 0.279s

OK

Destroying test database for alias 'default'...

**setUp()**

Have you noticed that we seem to be repeating ourself with these unit tests? For each one we are loading a response variable which seems wasteful and prone to errors. It’d be better to stick to something more DRY (Don’t Repeat Yourself) such as doing this once at the top of the tests with a function called setUp that loads the response into a response variable.

Our current test\_homepage\_url test is now redundant since setUp first runs reverse on our named template “home” so we can remove that test.

# pages/tests.py

from django.test import SimpleTestCase

from django.urls import reverse

# Create your tests here.

class HomepageTests(SimpleTestCase):

def setUp(self): # new

url = reverse("home")

self.response = self.client.get(url)

def test\_url\_exists\_at\_correct\_location(self):

self.assertEqual(self.response.status\_code, 200)

def test\_homepage\_template(self):

self.assertTemplateUsed(self.response, "home.html")

def test\_homepage\_contains\_correct\_html(self):

self.assertContains(self.response, "home page")

def test\_homepage\_does\_not\_contain\_incorrect\_html(self):

self.assertNotContains(self.response, "Hi there! I should not be on the page.")

Now run the tests again. Because setUp is a helper method and does not start with test it will not be considered a unit test in the final tally. So only 6 total tests will run.

docker-compose exec web python manage.py test

Creating test database for alias 'default'...

System check identified no issues (0 silenced).

....

----------------------------------------------------------------------

Ran 6 tests in 0.126s

OK

Destroying test database for alias 'default'...

**Resolve**

A final views check we can do is that our HomePageView “resolves” a given URL path. Django contains the utility function resolve for just this purpose. We will need to import both resolve as well as the HomePageView at the top of the file.

Our actual test, test\_homepage\_url\_resolves\_homepageview, checks that the name of the view used to resolve / matches HomePageView.

# pages/tests.py

from django.test import SimpleTestCase

from django.urls import reverse, resolve # new

from .views import HomePageView # new

# Create your tests here.

class HomepageTests(SimpleTestCase):

def setUp(self):

url = reverse("home")

self.response = self.client.get(url)

def test\_url\_exists\_at\_correct\_location(self):

self.assertEqual(self.response.status\_code, 200)

def test\_homepage\_template(self):

self.assertTemplateUsed(self.response, "home.html")

def test\_homepage\_contains\_correct\_html(self):

self.assertContains(self.response, "home page")

def test\_homepage\_does\_not\_contain\_incorrect\_html(self):

self.assertNotContains(self.response, "Hi there! I should not be on the page.")

def test\_home\_url\_resolves\_homepageview(self): # new

view = resolve("/")

self.assertEqual(view.func.\_\_name\_\_, HomePageView.as\_view().\_\_name\_\_)

Phew. That’s our last test. Let’s confirm that everything passes.

docker-compose exec web python manage.py test

Creating test database for alias 'default'...

System check identified no issues (0 silenced).

.....

----------------------------------------------------------------------

Ran 7 tests in 0.126s

OK

Destroying test database for alias 'default'...

**Git**

Time to add our new changes to source control with Git.

git status

git add .

git commit -m ’’Chapter 05. Pages App’’

**To push to GitHub if you have an account:**

1. run: cd ..

Your path should be where you root depository is(in my case Bookstore-by-Chapter):

PS C:\Users\computer’s username\Documents\your main folder\Bookstore-by-Chapter>

Yours would be different like:

PS C:\Users\computer’s username\Documents\your main folder\title of your subfolder>

git add "Chapter 05. Pages App"

git commit -m "Chapter 05. Pages App"

git push

You can compare with the official source code on Github for this chapter.

**Create a back up:**

Copy-Item -Recurse -Path "C:\Users\Jean-Marc H\Documents\Django for professionals\Bookstore-by-Chapter\*Chapter x. chapter’s title*" -Destination "C:\Users\Jean-Marc H\Documents\Django for professionals\Bookstore-by-Chapter\ *Chapter x. chapter’s title* - Backup"

**Conclusion**

We have configured our templates and added the first page to our project, a static homepage. We also added tests which should always be included with new code changes. Some developers prefer a method called Test-Driven Development where they write the tests first and then the code. Personally I prefer to write the tests immediately after which is what we’ll do here.

Both approaches work, the key thing is to be rigorous with your testing. Django projects quickly grow in size where it’s impossible to remember all the working pieces in your head. And if you are working on a team, it is a nightmare to work on an untested codebase. Who knows what will break?

In the next chapter we’ll add user registration to our project: log in, log out, and sign up.

The end