



Build and Deploy Django project - pages

2020-12-25

*Source: "Django for Beginners_Build websites with Python and Django 3.0",
chapter 3, William S. Vincent (2020)*



Key Steps



1. Initial Setup
2. Use Templates
3. Use Class-based views
4. Explore URLConfs
5. Add Basic Tests
6. Use Heroku

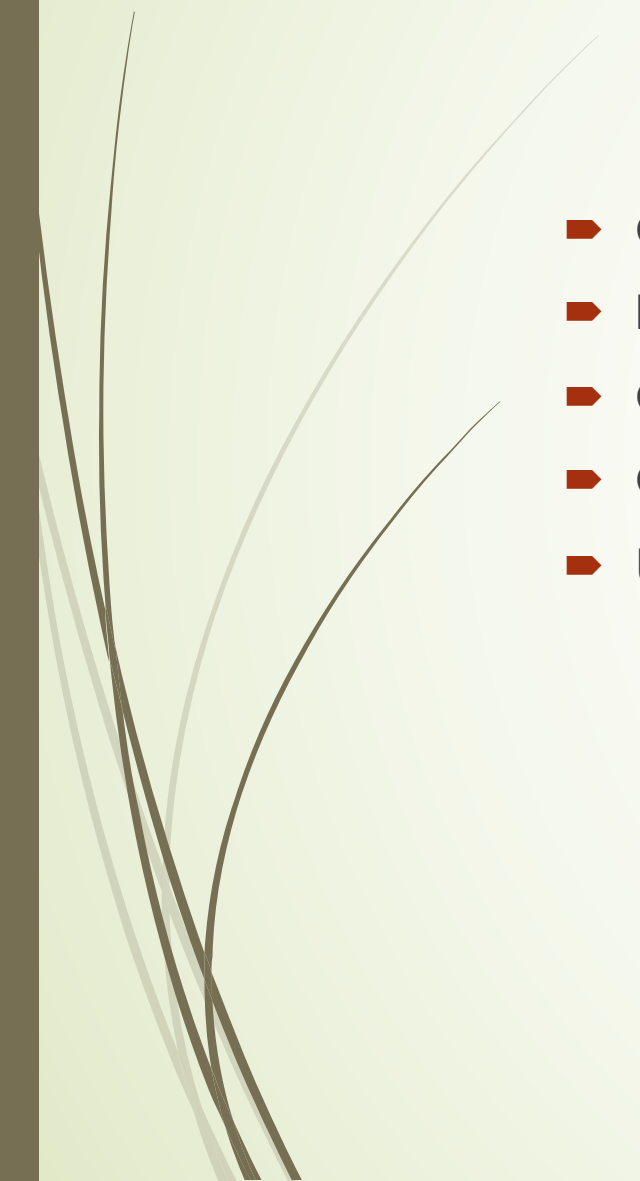
Project “Pages”: has a homepage and an about page

A decorative graphic on the left side of the slide. It features a solid red arrow pointing to the right, positioned horizontally. Behind the arrow and extending upwards and to the right are several thin, dark grey curved lines that sweep across the page.

Initial Setup



Initial Setup

- Create a directory for our code
 - Install Django in a new virtual environment
 - Create a new Django project
 - Create a new *pages* app
 - Update *settings.py*
- 

Within Command Line Console

- `$ cd ~/Desktop`
- `$ mkdir pages && cd pages`
- `$ pipenv install Django==3.0.1`
- `$ pipenv shell`
- `(pages) $ django-admin startproject pages_project .`
- `(pages) $ python manage.py startapp pages`

If not have pipenv on MacOS, run “brew install pipenv” first.

Then after above, use <http://localhost:8000> to view the page

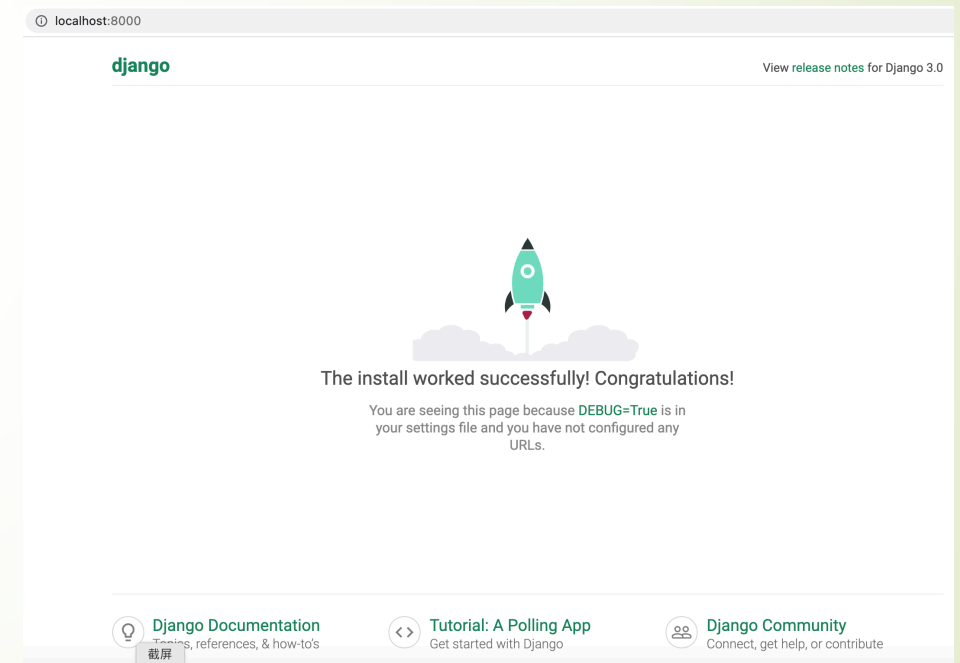
pages_project/setting.py

- Add the *pages* app at the bottom of project under *INSTALLED_APPS*:

- Code

```
# pages_project/settings.py
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'pages.app.PagesConfig', # new
]
```

- Then, start the local web server via
 - (pages) \$ python manage.py runserver



A decorative graphic on the left side of the slide. It features a solid red arrow pointing to the right, positioned horizontally. Behind the arrow and extending upwards and to the right are several thin, dark grey curved lines that sweep across the page.

Templates



About Templates



- Every web framework needs a convenient way to generate HTML files and in Django the approach is to use *templates*: individual HTML file that can be linked together and also include basic logic.
- Approach 1: have the phrase hardcoded into a *views.py* file as a string.
- Approach 2: use templates to more easily create desired homepage and about page. Where to place templates? Two options
 - Option 1: Layout:
 - Pages → templates → pages → home.html
 - Option 2: instead of creating a single project-level *templates* directory and place *all* templates within there, by making a small tweak to our *settings.py* file we can tell Django to also look in this directory for templates.



Create folder and HTML file

- Quit the running server with the *Control+c* command.
- Then create a directory called *templates* and an HTML file called *home.html*
 - (page) \$ `mkdir templates`
 - (page) \$ `touch templates/home.html`

Update *settings.py*

- Update *settings.py* to tell Django the location of our new *templates* directory. This is one-line change to the setting '*DIRS*' under *TEMPLATES*

```
# pages_project/settings.py
TEMPLATES = [
    {
        ...
        'DIRS': [os.path.join(BASE_DIR, 'templates')], # new
        ...
    },
]
```



Add a simple headline to *home.html*

➤ Code:

```
<!-- templates/home.html -->  
<h1>Homepage</h1>
```



Class-Based Views



Background

- Early versions of Django only shipped with function-based views, but developers soon found themselves repeating the same patterns over and over again.
- Function-based generic views were introduced to abstract these patterns and streamline development of common patterns. However, there was **no easy way to extend or customize these views**. As a result, Django introduced class-based generic views that make it easy to use and also extend views covering common use cases.
- In this view, we will use the **built-in TemplateView** to display our template.

Update *pages/views.py* file

- Code:

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

- Note: we've capitalized our view, *HomePageView*, since it's now a Python class.
 - Class, 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, here is *home.html*.



URLs



How to do?

- This “Update our URLConfs” is the last step for building the website, we need to make updates in two locations.
 - First, we update the **pages_project/urls.py** file to point at our **pages** app
 - Then, within **pages** we match views to URL routes.
- 

Update *pages/urls.py* file

```
# pages_project/urls.py
from django.contrib import admin
from django.urls import path, include      # new add "include"

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('pages.urls')),      # new
]
```

We add **include** on the 2nd line to point the existing URL to the **pages** app.

Create an app-level *urls.py* file

- Command line: (page) \$ touch pages/urls.py
- Add following code:

```
# pages/urls.py
from django.urls import path
from .views import HomePageView
urlpatterns = [
    path('', HomePageView.as_view(), name='home'),
]
```

- When using Class-Based Views, also ass **as_view()** at the end of view name.
- Now using command: \$ python manage.py runserver, then from <http://127.0.0.1:8000/> you can see our new homepage,



Add an About Page

- The process for adding an about page is very similar to what we just deid.
 - We'll create a new template file, a new view, and a new url route
 - Quit the server with **Control+c**
- 

Create new template

- Create a new template called *about.html*
 - Command: *(page) \$ touch templates/about.html*
- Then populate it with a short HTML headline:

```
<!-- templates/about.html -->  
<h1>About page</h1>
```

Create new view for the page

➡ Code:

```
# pages/views.py
from django.views.generic import TemplateView

class HomePageView(TemplateView):
    template_name = 'home.html'

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

Connect view to a URL at *about/*

► Code:

```
# pages/urls.py
from django.urls import path
from .views import HomePageView, AboutPageView      # new
urlpatterns = [
    path('about/', AboutPageView.as_view(), name='about'),      # news
    path('', HomePageView.as_view(), name='home'),
]
```

- Then, start up the web server via `python manage.py runserver`, go to <http://127.0.0.1:8000/about> and you can see our new “About Page”

A decorative graphic on the left side of the slide. It features a solid red arrow pointing to the right, positioned horizontally. Behind the arrow and extending upwards and to the right are several thin, curved, light brown lines that create a sense of movement or flow.

Extending Templates



Background



- The real power of templates is their ability to be extended. It would be nice if we could have **one canonical place** for our header code that would be inherited by all other templates.
- We can do this in Django through creating a **base.html** file containing a header with links to our two pages.
- We could name this file anything, but let's using **base.html** which is a common convention.
- Type **Control+c** and then create the new file:
 - (page) \$ touch templates/base.html



Handle Links/Logic in Templates

- 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 two routes in *pages/urls.py*? This is why. The *url* use these names to automatically create links for us.
- The URL route for our homepage is called **home** therefore to configure a link to it we would use the following: **{% url 'home' %}**

Create *base.html*

► Code:

```
<!-- templates/base.html -->
<header>
    <a href="{% url 'home' %}">Home</a> | <a href="{% url 'about' %}">About</a>
</header>

{% block content %}
{% endblock content %}
```

- At the bottom we've added a **block** tag called **content**. Blocks can be overwritten by child templates via inheritance. While it's optional to name our closing **endblock** - you can just write **{% endblock %}** if you prefer – doing so helps with readability, especially in larger template files.

Update *home.html* & *about.html*

- Now update our *home.html* and *about.html* file to extend the *base.html* template. This means we can reuse the same code from one template in another template.
- The Django templating language comes with **extends** method here.

```
<!-- templates/home.html -->
{% extends 'base.html' %}
```

```
{% block content %}
<h1>Homepage</h1>
{% endblock content %}
```

```
<!-- templates/about.html -->
{% extends 'base.html' %}
```

```
{% block content %}
<h1>About page</h1>
{% endblock content %}
```

- Now if you start up server with **python manage.py runserver** and open the webpage <http://127.0.0.1:8000/> and <http://127.0.0.1:8000/about> you'll see the header is magically included in **both** locations.



Tests



About Tests

- In the word of Django co-creator **Jacob Kaplan-Moss**, “Code without tests is broken as designed.”
- Django comes with robust, built-in **testing tools** for writing and running test.
- In **pages** app, Django already provides a **tests.py** file we can use.

Update *tests.py* Code

```
# pages/tests.py
from django.test import SimpleTestCase

class SimpleTests(SimpleTestCase):
    def test_home_page_status_code(self):
        response = self.client.get('/')
        self.assertEqual(response.status_code, 200)
    def test_about_page_status_code(self):
        response = self.client.get('/about/')
        self.assertEqual(response.status_code, 200)
```

We're using **SimpleTestCase** here since we aren't using a database.

If we were using a database, we'd instead use **TestCase**.

Then we perform a check if the status code for each page is 200, which is the **standard response for a successful HTTP request**.

This is a fancy way of saying it ensures that a given webpage actually exists, but says nothing about the content of said page.

Run the Tests via Command Line

- (page) \$ python manage.py test
- Creating test database for alias 'default'...
- System check identified no issues (0 silenced).
- ..
- -----
- Ran 2 tests in 0.014s
- OK
- Destroying test database for alias 'default' ...