



DJANGO

for

BEGINNERS

Build websites with Python & Django

WILLIAM S. VINCENT

Django for Beginners

Build websites with Python & Django

William S. Vincent

© 2018 William S. Vincent

Table of Contents

[Introduction](#)

[Why Django](#)

[Why this book](#)

[Book Structure](#)

[Book layout](#)

[Conclusion](#)

[Chapter 1: Initial Setup](#)

[The Command Line](#)

[Install Python 3 on Mac OS X](#)

[Install Python 3 on Windows](#)

[Install Python 3 on Linux](#)

[Virtual Environments](#)

[Install Django](#)

[Install Git](#)

[Text Editors](#)

[Conclusion](#)

[Chapter 2: Hello World app](#)

[Initial Setup](#)

[Create an app](#)

[Views and URLConfs](#)

[Hello, world!](#)

[Git](#)

[Bitbucket](#)

[Conclusion](#)

[Chapter 3: Pages app](#)

[Initial Setup](#)

[Templates](#)

[Class-Based Views](#)

[URLs](#)

[Add an About Page](#)

[Extending Templates](#)

[Tests](#)

[Git and Bitbucket](#)
[Local vs Production](#)
[Heroku](#)
[Additional Files](#)
[Deploy](#)
[Conclusion](#)

[Chapter 4: Message Board app](#)

[Initial Setup](#)
[Create a database model](#)
[Activating models](#)
[Django Admin](#)
[Views/Templates/URLs](#)
[Adding new posts](#)
[Tests](#)
[Bitbucket](#)
[Heroku configuration](#)
[Heroku deployment](#)
[Conclusion](#)

[Chapter 5: Blog app](#)

[Initial Setup](#)
[Database Models](#)
[Admin](#)
[URLs](#)
[Views](#)
[Templates](#)
[Static files](#)
[Individual blog pages](#)
[Tests](#)
[Git](#)
[Conclusion](#)

[Chapter 6: Forms](#)

[Forms](#)
[Update Form](#)
[Delete View](#)
[Tests](#)

[Conclusion](#)

[Chapter 7: User Accounts](#)

[Login](#)

[Updated homepage](#)

[Logout link](#)

[Signup](#)

[Bitbucket](#)

[Heroku config](#)

[Heroku deployment](#)

[Conclusion](#)

[Chapter 8: Custom User Model](#)

[Setup](#)

[Custom User Model](#)

[Forms](#)

[Superuser](#)

[Conclusion](#)

[Chapter 9: User Authentication](#)

[Templates](#)

[URLs](#)

[Admin](#)

[Conclusion](#)

[Chapter 10: Bootstrap](#)

[Pages app](#)

[Tests](#)

[Bootstrap](#)

[Signup Form](#)

[Next Steps](#)

[Chapter 11: Password Change and Reset](#)

[Password Change](#)

[Customizing password change](#)

[Password reset](#)

[Custom Templates](#)

[Conclusion](#)

[Chapter 12: Email](#)

[SendGrid](#)

[Custom emails](#)

[Conclusion](#)

[Chapter 13: Newspaper app](#)

[Articles app](#)

[URLs and Views](#)

[Edit/Delete](#)

[Create page](#)

[Conclusion](#)

[Chapter 14: Permissions and Authorization](#)

[Improved CreateView](#)

[Authorizations](#)

[Mixins](#)

[Updating views](#)

[Conclusion](#)

[Chapter 15: Comments](#)

[Model](#)

[Admin](#)

[Template](#)

[Conclusion](#)

[Conclusion](#)

[Django Resources](#)

[Python Books](#)

[Blogs to Follow](#)

[Feedback](#)

Introduction

Welcome to *Django for Beginners*, a project-based approach to learning web development with the [Django](#) web framework. In this book you will build five progressively more complex web applications, starting with a simple “Hello, World” app, progressing to a blog app with forms and user accounts, and finally a newspaper app using a custom user model, email integration, foreign keys, authorization, permissions, and more.

By the end of this book you should feel confident creating your own Django projects from scratch using current best practices.

Django is a free, open source web framework written in the [Python](#) programming language and used by millions of programmers every year. Its popularity is due to its friendliness to both beginners and advanced programmers: Django is robust enough to be used by the largest websites in the world—Instagram, Pinterest, Bitbucket, Disqus—but also flexible enough to be a good choice for early-stage startups and prototyping personal projects.

This book is regularly updated and features the latest versions of both Django (2.0) and Python (3.6x). It also uses [Pipenv](#) which is now the officially recommended package manager by [Python.org](#) for managing Python packages and virtual environments. Throughout we’ll be using modern best practices from the Django, Python, and web development communities, especially the thorough use of testing.

Why Django A web framework is a collection of modular tools that abstracts away much of the difficulty—and repetition—inherent in web development. For example, most websites need the same basic functionality: the ability to connect to a database, set URL routes, display content on a page, handle security properly, and so on. Rather than recreate all of this from scratch, programmers over the years have created web frameworks in all the major programming languages: Django and [Flask](#) in Python, [Rails](#) in Ruby, and [Express](#) in JavaScript among many,

many others.

Django inherited Python’s “batteries-included” approach and includes out-of-the box support for common tasks in web development:

- user authentication
- templates, routes, and views
- admin interface
- robust security
- support for multiple database backends
- and much much more

This approach makes our job as web developers much, much easier. We can focus on what makes our web application unique rather than reinventing the wheel when it comes to standard web application functionality.

In contrast, several popular frameworks—most notably Flask in Python and Express in JavaScript—adopt a “microframework” approach. They provide only the bare minimum required for a simple web page and leave it up to the developer to install and configure third-party packages to replicate basic website functionality. This approach provides more flexibility to the developer but also yields more opportunities for mistakes.

As of 2018 Django has been under active development for over 13 years which makes it a grizzled veteran in software years. Millions of programmers have already used Django to build their websites. And this is undeniably a good thing. Web development is hard. It doesn’t make sense to repeat the same code—and mistakes—when a large community of brilliant developers has already solved these problems for us.

At the same time, Django remains [under active development](#) and has a yearly release schedule. The Django community is constantly adding new features and security improvements. If you’re building a website from scratch Django is a fantastic choice.

Why this book I wrote this book because while Django is extremely well documented there is a severe lack of beginner-friendly tutorials available. When I first learned Django years ago I struggled to even complete the [official](#)

[polls tutorial](#). Why was this so hard I remember thinking?

With more experience I recognize now that the writers of the Django docs faced a difficult choice: they could emphasize Django's **ease-of-use** or its **depth**, but not both. They choose the latter and as a professional developer I appreciate the choice, but as a beginner I found it so...**frustrating!**

My goal is that this book fills in the gaps and showcases how beginner-friendly Django really is.

You don't need previous Python or web development experience to complete this book. It is intentionally written so that even a total beginner can follow along and feel the magic of writing their own web applications from scratch. However if you are serious about a career in web development, you will eventually need to invest the time to learn Python, HTML, and CSS properly. A list of recommended resources for further study is included in the Conclusion.

Book Structure

We start by properly covering how to configure a local development environment in **Chapter 1**. We're using bleeding edge tools in this book: the most recent version of Django (2.0), Python (3.6), and [Pipenv](#) to manage our virtual environments. We also introduce the command line and discuss how to work with a modern text editor.

In **Chapter 2** we build our first project, a minimal *Hello, World* application that demonstrates how to set up new Django projects. Because establishing good software practices is important, we'll also save our work with *git* and upload a copy to a remote code repository on [Bitbucket](#).

In **Chapter 3** we make, test, and deploy a *Pages* app that introduces templates and class-based views. Templates are how Django allows for DRY (Don't Repeat Yourself) development with HTML and CSS while class-based views require a minimal amount of code to use and extend core functionality in Django. They're awesome as you'll soon see. We also add our first tests and deploy to [Heroku](#) which has a free tier we'll use throughout this book. Using platform-as-a-service providers like Heroku transforms development from a painful, time-consuming process into something that takes just a few mouse clicks.

In **Chapter 4** we build our first database-backed project, a *Message Board* app. Django provides a powerful [ORM](#) that allows us to write concise Python for our database tables. We'll explore the built-in admin app which provides a graphical way to interact with our data and can be even used as a Content Management System (CMS) similar to Wordpress. Of course we also write tests for all our code, store a remote copy on Bitbucket, and deploy to Heroku.

Finally in **Chapters 5-7** we're ready for our final project: a robust blog application that demonstrates how to perform CRUD (Create-Read-Update-Delete) functionality in Django. We'll find that Django's generic class-based views mean we have to write only a small amount of actual code for this! Then we'll add forms and integrate Django's built-in user authentication system (log in, log out, sign up).

In **Chapter 8** we start a **Newspaper** site and introduce the concept of custom

user models, a Django best practice that is rarely covered in tutorials. Simply put all new projects should use a custom user model and in this chapter you'll learn how. **Chapter 9** covers user authentication, **Chapter 10** adds Bootstrap for styling, and **Chapters 11-12** implement password reset and change via email. In **Chapters 13-15** we add articles and comments to our project, along with proper permissions and authorizations. We even learn some tricks for customizing the admin to display our growing data.

The **Conclusion** provides an overview of the major concepts introduced in the book and a list of recommended resources for further learning.

While you can pick and choose chapters to read, the book's structure is very deliberate. Each app/chapter introduces a new concept and reinforces past teachings. I **highly recommend** reading it in order, even if you're eager to skip ahead. Later chapters won't cover previous material in the same depth as earlier chapters.

By the end of this book you'll have an understanding of how Django works, the ability to build apps on your own, and the background needed to fully take advantage of additional resources for learning intermediate and advanced Django techniques.

Book layout

There are many code examples in this book, which are denoted as follows:

Code

```
# This is Python code
print(Hello, World)
```

For brevity we will use dots . . . to denote existing code that remains unchanged, for example, in a function we are updating.

Code

```
def make_my_website:
...
print("All done!")
```

We will also use the command line console frequently (starting in **Chapter 1: Initial Setup** to execute commands, which take the form of a \$ prefix in traditional Unix style.

Command Line

```
$ echo "hello, world"
```

The result of this particular command is the next line will state:

Command Line

```
"hello, world"
```

We will typically combine a command and its output. The command will always be prefaced by a \$ and the output will not. For example, the command and result above will be represented as follows:

Command Line

```
$ echo "hello, world"
hello, world
```

Complete source code for all examples can be found in the [official Github repository](#).

Conclusion Django is an excellent choice for any

developer who wants to build modern, robust web applications with a minimal amount of code. It is popular, under active development, and thoroughly battle-tested by the largest websites in the world. In the next chapter we'll learn how to configure any computer for Django development.

Chapter 1: Initial Setup

This chapter covers how to properly configure your computer to work on Django projects. We start with an overview of the command line and use it to install the latest versions of both Django (2.0) and Python (3.6x). Then we discuss virtual environments, git, and working with a text editor.

By the end of this chapter you'll be ready to create and modify new Django projects in just a few keystrokes.

The Command Line The command line is a powerful, text-only view of your computer. As developers we will use it extensively throughout this book to install and configure each Django project.

On a Mac, the command line is found in a program called Terminal located at /Applications/Utilities. To find it, open a new Finder window, open the Applications folder, scroll down to open the Utilities folder, and double-click the application called Terminal.

On Windows, there is a built-in command line program but it is difficult to use. I recommend instead using [Babun](#), a free and open-source command line program. On the Babun homepage click on the “Download now” button, double-click on the downloaded file to install Babun, and upon completion drag the installer to the Recycle Bin. To use Babun go to the Start menu, select Programs, and click on Babun.

Going forward when the book refers to the “command line” it means to open a new console on your computer using either Terminal or Babun.

While there are many possible commands we can use, in practice there are six used most frequently in Django development.

- cd (change down a directory)
- cd .. (change up a directory)
- ls (list files in your current directory)
- pwd (print working directory)

- `mkdir` (make directory)
- `touch` (create a new file)

Open your command line and try them out. The \$ dollar sign is our command line prompt: all commands in this book are intended to be typed **after** the \$ prompt.

For example, assuming you're on a Mac, let's change into our Desktop directory.

Command Line

```
$ cd ~/Desktop
```

Note that our current location ~/Desktop is automatically added before our command line prompt. To confirm we're in the proper location we can use `pwd` which will print out the path of our current directory.

Command Line

```
~/Desktop $ pwd  
/Users/wsv/Desktop
```

On my Mac computer this shows that I'm using the user wsv and on the desktop for that account.

Let's create a new directory folder with `mkdir`, `cd` into it, and add a new file `index.html`.

Command Line

```
~/Desktop $ mkdir new_folder  
~/Desktop $ cd new_folder  
~/Desktop/new_folder $ touch index.html
```

Now use `ls` to list all current files in our directory. You'll see there's just the newly created `index.html`.

Command Line

```
~/Desktop/new_folder $ ls  
index.html
```

As a final step return to the Desktop directory with `cd ..` and use `pwd` to confirm the location.

Command Line

```
~/Desktop/new_folder $ cd ..  
~/Desktop $ pwd
```

/Users/wsv/desktop

Advanced developers can use their keyboard and command line to navigate through their computer with ease; with practice this approach is much faster than using a mouse.

In this book I'll give you the exact instructions to run—you don't need to be an expert on the command line—but over time it's a good skill for any professional software developer to develop. Two good free resources for further study are the [Command Line Crash Course](#) and [Codecademy's Course on the Command Line](#).

Instructions are included below for Mac, Windows, and Linux computers.

Install Python 3 on Mac OS X

Although Python 2 is installed by default on Mac computers, Python 3 is not. You can confirm this by typing `python --version` in the command line console and hitting Enter:

Command Line

```
$ python --version  
Python 2.7.13
```

To check if Python 3 is already installed try running the same command using `python3` instead of `python`.

Command Line

```
$ python3 --version
```

If your computer outputs `3.6.x` (any version of 3.6 or higher) then you're in good shape, however most likely you'll see an error message since we need to install Python 3 directly.

Our first step is to install Apple's [Xcode](#) package, so run the following command to install it:

Command Line

```
$ xcode-select --install
```

Click through all the confirmation commands (Xcode is a large program so this might take a while to install depending on your internet connection).

Next, install the package manager [Homebrew](#) via the longish command below:
Command Line

```
$ /usr/bin/ruby -e "$(curl -fsSL  
https://raw.githubusercontent.com/Homebrew/install/\\  
master/install)"
```

To confirm Homebrew installed correctly, run this command:

Command Line

```
$ brew doctor  
Your system is ready to brew.
```

And now to install the latest version of Python, run the following command:

Command Line

```
$ brew install python3
```

Now let's confirm which version was installed:

Command Line

```
$ python3 --version  
Python 3.6.4
```

To open a Python 3 interactive shell—this lets us run Python commands directly on our computer—simply type `python3` from the command line:

Command Line

```
$ python3  
Python 3.6.4 (default, Jan 7 2018, 13:05:00)  
[GCC 4.2.1 Compatible Apple LLVM 9.0.0 (clang-900.0.39.2)] on  
darwin  
Type "help", "copyright", "credits" or "license" for more  
information.  
>>>
```

To exit the Python 3 interactive shell at any time type `Control-d` (the “Control” and “d” key at the same time).

You can still run Python shells with Python 2 by simply typing `python`:

Command Line

```
$ python
Python 2.7.13 (default, Dec 18 2016, 07:03:39)
[GCC 4.2.1 Compatible Apple LLVM 8.0.0 (clang-800.0.42.1)] on
darwin
Type "help", "copyright", "credits" or "license" for more
information.
```

Install Python 3 on Windows Python is not included by default on Windows, however we can check if any version exists on the system. Open a command line console by entering `command` on the Start Menu. Or you can hold down the SHIFT key and right-click on your desktop, then select Open Command Window Here.

Type the following command and hit RETURN:

Command Line

```
python --version
Python 3.6.4
```

If you see output like this, Python is already installed. _Most likely it will not be!

To download Python 3, go to the [downloads section](#) of the official Python website. Download the installer and make sure to click the *Add Python to PATH* option, which will let use use `python` directly from the command line. Otherwise we'd have to enter our system's full path and modify our environment variables manually.

After Python has installed, run the following command in a new command line console:

Command Line

```
python --version
Python 3.6.4
```

If it works, you're done!

Install Python 3 on Linux Adding Python 3 to a Linux distribution takes a bit more work. Here are recommended recent guides [for Centos](#) and [for Debian](#). If you need additional help adding Python to your PATH please refer to [this Stack Overflow answer](#).

Virtual Environments [Virtual environments](#) are an indispensable part of Python programming. They are an isolated container containing all the software dependencies for a given project. This is important because by default software like Python and Django is installed *in the same directory*. This causes a problem when you want to work on multiple projects on the same computer. What if *ProjectA* uses Django 2.0 but *ProjectB* from last year is still on Django 1.10? Without virtual environments this becomes very difficult; with virtual environments it's no problem at all.

There are many areas of software development that are hotly debated, but using virtual environments for Python development is not one. **You should use a dedicated virtual environment for each new Python project.**

Historically Python developers have used either `virtualenv` or `pyenv` to configure virtual environments. But in 2017 prominent Python developer Kenneth Reitz released [Pipenv](#) which is now [the officially recommended Python packaging tool](#).

Pipenv is similar to `npm` and `yarn` from the Node ecosystem: it creates a `Pipfile` containing software dependencies and a `Pipfile.lock` for ensuring deterministic builds. “Determinism” means that each and every time you download the software in a new virtual environment, you will have *exactly the same configuration*. Sebastian McKenzie, the creator of [Yarn](#) which first introduced this concept to JavaScript packaging, has a concise blog post [explaining what determinism is and why it matters](#).

The end result is that we will create a new virtual environment with Pipenv for

each new Django Project.

To install Pipenv we can use pip3 which Homebrew automatically installed for us alongside Python 3.

Command Line

```
$ pip3 install pipenv
```

Install Django To see Pipenv in action, let's create a new directory and install Django. First navigate to the Desktop, create a new directory django, and enter it with cd.

Command Line

```
$ cd ~/Desktop  
$ mkdir django  
$ cd django
```

Now use Pipenv to install Django.

Command Line

```
$ pipenv install django==2.0.6
```

If you look within our directory there are now two new files: Pipfile and Pipfile.lock. We have the information we need for a new virtual environment but we have not activated it yet. Let's do that with pipenv shell.

Command Line

```
$ pipenv shell
```

If you are on a Mac you should now see parentheses on your command line with the environment activated. It will take the format of the directory name and random characters. On my computer, I see the below but you will see something slightly different: it will start with django - but end with a random series of characters.

Note that due to an [open bug](#) Windows users will not see visual feedback of the virtual environment at this time. But if you can run django-admin startproject in the next section then you know your virtual environment has Django installed.

Command Line

```
(django-JmZ1NTQw) $
```

This means it's working! Create a new Django project called `test` with the following command. Don't forget that period `.` at the end.

Command Line

```
(django-JmZ1NTQw) $ django-admin startproject test_project .
```

It's worth pausing here to explain why you should add a period `.` to the command. If you just run `django-admin startproject test_project` then by default Django will create this directory structure:

```
└── manage.py
└── test_project
    ├── __init__.py
    ├── settings.py
    ├── urls.py
    └── wsgi.py
```

See how it creates a new directory `test_project` and then within it a `manage.py` file and a `test_project` directory? That feels redundant to me since we already created and navigated into a `django` folder on our Desktop. By running `django-admin startproject test_project .` with the period at the end—which says, install in the current directory—the result is instead this:

```
└── manage.py
    └── test_project
        ├── __init__.py
        ├── settings.py
        ├── urls.py
        └── wsgi.py
```

The takeaway is that it **doesn't really matter** if you include the period or not at the end of the command, but I prefer to include the period and so that's how we'll do it in this book.

Now let's confirm everything is working by running Django's local web server.

Command Line

```
(django-JmZ1NTQw) $ python manage.py runserver
```

If you visit <http://127.0.0.1:8000/> you should see the following image:



django

[View release notes for Django 2.0](#)



The install worked successfully! Congratulations!

You are seeing this page because `DEBUG=True` is in your settings file and you have not configured any URLs.



Django Documentation
Topics, references, & how-to's



Tutorial: A Polling App
Get started with Django



Django Community
Connect, get help, or contribute

[Django welcome page](#)

To stop our local server type `Control-c`. Then exit our virtual environment using the command `exit`.

Command Line

```
(django-JmZ1NTQw) $ exit
```

We can always reactivate the virtual environment again using `pipenv shell` at any time.

We'll get lots of practice with virtual environments in this book so don't worry if it's a little confusing right now. The basic pattern is to install new packages with `pipenv`, activate them with `pipenv shell`, and then exit when done with `exit`.

It's worth noting that only one virtual environment can be active in a command

line tab at a time. In future chapters we will be creating a brand new virtual environment for each new project. Either make sure to exit your current environment or open up a new tab for new projects.

Install Git [Git](#) is an indispensable part of modern software development. It is a [version control system](#) which can be thought of as an extremely powerful version of *track changes* in Microsoft Word or Google Docs. With git, you can collaborate with other developers, track all your work via commits, and revert to any previous version of your code even if you accidentally delete something important!

On a Mac, because Homebrew is already installed we can simply type brew install git on the command line:

Command Line

```
$ brew install git
```

On Windows you should download Git from [Git for Windows](#). Click the “Download” button and follow the prompts for installation. Hit “next” on all steps except the fifth one, “Adjusting your PATH environment.” Instead choose the bottom option: “Use Git and optional Unix tools from the Windows Command Prompt.”

Once installed, we need to do a one-time *system* setup to configure it by declaring the name and email address you want associated with all your Git commits (more on this shortly).

Within the command line console type the following two lines. Make sure to update them your name and email address.

Command Line

```
$ git config --global user.name "Your Name"  
$ git config --global user.email "yourname@email.com"
```

You can always change these configs later if you desire by retyping the same commands with a new name or email address.

Text Editors The final step is our text editor. While the

command line is where we execute commands for our programs, a text editor is where the actual code is written. The computer doesn't care what text editor you use—the end result is just code—but a good text editor can provide helpful hints and catch typos for you.

Experienced developers often prefer using either [Vim](#) or [Emacs](#), both decades-old, text-only editors with loyal followings. However each has a steep learning curve and requires memorizing many different keystroke combinations. I don't recommend them for newcomers.

Modern text editors combine the same powerful features with an appealing visual interface. My current favorite is [Visual Studio Code](#) which is free, easy to install, and enjoys widespread popularity. If you're not already using a text editor, download and install [Visual Studio Code](#) now.

Conclusion Phew! Nobody really likes configuring a local development environment but fortunately it's a one-time pain. We have now learned how to work with virtual environments and installed the latest version of Python and git. Everything is ready for our first Django app.

Chapter 2: Hello World app

In this chapter we'll build a Django project that simply says "Hello, World" on the homepage. This is [the traditional way](#) to start a new programming language or framework. We'll also work with *git* for the first time and deploy our code to Bitbucket.

Initial Setup To start navigate to a new directory on your computer. For example, we can create a `helloworld` folder on the Desktop with the following commands.

Command Line

```
$ cd ~/Desktop  
$ mkdir helloworld  
$ cd helloworld
```

Make sure you're not already in an existing virtual environment at this point. If you see text in parentheses () before the dollar sign \$ then you are. To exit it, type `exit` and hit Return. The parentheses should disappear which means that virtual environment is no longer active.

We'll use `pipenv` to create a new virtual environment, install Django and then activate it.

Command Line

```
$ pipenv install django==2.0.6  
$ pipenv shell
```

If you are on a Mac you should see parentheses now at the beginning of your command line prompt in the form (`helloworld-XXX`) where XXX represents random characters. On my computer I see (`helloworld-415ivvZC`). I'll display (`helloworld`) here in the text but you will see something slightly different on your computer. If you are on Windows you will not see a visual prompt at this time.

Create a new Django project called `helloworld_project` making sure to include the period . at the end of the command so that it is installed in our current directory.

Command Line

```
(helloworld) $ django-admin startproject helloworld_project .
```

If you use the `tree` command you can see what our Django project structure now looks like. (**Note:** If `tree` doesn't work for you, install it with Homebrew: `brew install tree`.)

Command Line

```
(helloworld) $ tree
```

```
.
└── helloworld_project
    ├── __init__.py
    ├── settings.py
    ├── urls.py
    └── wsgi.py
```

```
1 directory, 5 files
```

The `settings.py` file controls our project's settings, `urls.py` tells Django which pages to build in response to a browser or url request, and `wsgi.py`, which stands for [web server gateway interface](#), helps Django serve our eventual web pages. The last file `manage.py` is used to execute various Django commands such as running the local web server or creating a new app.

Django comes with a built-in web server for local development purposes. We can start it with the `runserver` command.

Command Line

```
(helloworld) $ python manage.py runserver
```

If you visit <http://127.0.0.1:8000/> you should see the following image:



django

[View release notes for Django 2.0](#)



The install worked successfully! Congratulations!

You are seeing this page because `DEBUG=True` is in your settings file and you have not configured any URLs.



[Django Documentation](#)
Topics, references, & how-to's



[Tutorial: A Polling App](#)
Get started with Django



[Django Community](#)
Connect, get help, or contribute

[Django welcome page](#)

Create an app Django uses the concept of projects and apps to keep code clean and readable. A single Django project contains one or more apps within it that all work together to power a web application. This is why the command for a new Django project is `startproject`! For example, a real-world Django e-commerce site might have one app for user authentication, another app for payments, and a third app to power item listing details. Each focuses on an isolated piece of functionality.

We need to create our first app which we'll call pages. From the command line, quit the server with `Control+c`. Then use the `startapp` command.

Command Line

```
(helloworld) $ python manage.py startapp pages
```

If you look again inside the directory with the `tree` command you'll see Django has created a `pages` directory with the following files:

Command Line

```
pages
├── __init__.py
├── admin.py
├── apps.py
└── migrations
    └── __init__.py
├── models.py
└── tests.py
└── views.py
```

Let's review what each new `pages` app file does:

- `admin.py` is a configuration file for the built-in Django Admin app
- `apps.py` is a configuration file for the app itself
- `migrations/` keeps track of any changes to our `models.py` file so our database and `models.py` stay in sync
- `models.py` is where we define our database models, which Django automatically translates into database tables
- `tests.py` is for our app-specific tests
- `views.py` is where we handle the request/response logic for our web app

Even though our new app exists within the Django project, Django doesn't "know" about it until we explicitly add it. In your text editor open the `settings.py` file and scroll down to `INSTALLED_APPS` where you'll see six built-in Django apps already there. Add our new `pages` app at the bottom:

Code

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

Views and URLConfs In Django, Views determine what content is displayed on a given page while URLConfs determine where that content is going.

When a user requests a specific page, like the homepage, the URLConf uses a [regular expression](#) to map that request to the appropriate view function which then returns the correct data.

In other words, our *view* will output the text “Hello, World” while our *url* will ensure that when the user visits the homepage they are redirected to the correct view.

Let’s start by updating the `views.py` file in our `pages` app to look as follows:

Code

```
# pages/views.py
from django.http import HttpResponse

def homePageView(request):
    return HttpResponse('Hello, World!')
```

Basically we’re saying whenever the view function `homePageView` is called, return the text “Hello, World!” More specifically, we’ve imported the built-in [HttpResponse](#) method so we can return a response object to the user. We’ve created a function called `homePageView` that accepts the `request` object and returns a response with the string `Hello, World!`.

Now we need to configure our urls. Within the `pages` app, create a new `urls.py` file.

Command Line

```
(helloworld) $ touch pages/urls.py
```

Then update it with the following code:

Code

```
# pages/urls.py
from django.urls import path
```

```
from . import views

urlpatterns = [
    path('', views.homePageView, name='home')
]
```

On the top line we import path from Django to power our urlpattern and on the next line we import our views. The period used here `from . import views` means reference the current directory, which is our pages app containing both `views.py` and `urls.py`. Our urlpattern has three parts:

- a Python regular expression for the empty string ''
- specify the view which is called `homePageView`
- add an optional url name of 'home'

In other words, if the user requests the homepage, represented by the empty string '' then use the view called `homePageView`.

We're *almost* done. The last step is to configure our project-level `urls.py` file too. Remember that it's common to have multiple apps within a single Django project, so they each need their own route.

Update the `helloworld_project/urls.py` file as follows:

Code

```
# helloworld_project/urls.py
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 urlpattern 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.

It's often confusing to beginners that we don't need to import the `pages` app here, yet we refer to it in our urlpattern as `pages.urls`. The reason we do it this way is that the method `django.urls.include()` expects us to pass in a module, or app, as the first argument. So without using `include` we *would* need

to import our pages app, but since we do use `include` we don't have to at the project level!

Hello, world!

We have all the code we need now! To confirm everything works as expected, restart our Django server:

Command Line

```
(helloworld) $ python manage.py runserver
```

If you refresh the browser for <http://127.0.0.1:8000/> it now displays the text "Hello, world!"



Hello world homepage

Git In the previous chapter we also installed `git` which is a version control system. Let's use it here. The first step is to initialize (or add) `git` to our repository.

Command Line

```
(helloworld) $ git init
```

If you then type `git status` you'll see a list of changes since the last git commit. Since this is our first commit, this list is all of our changes so far.

Command Line

```
(helloworld) $ git status
On branch master
```

No commits yet

```
Untracked files:
(use "git add <file>..." to include in what will be committed)
```

```
db.sqlite3
helloworld_project/
manage.py
pages/
Pipfile
```

Pipfile.lock

```
nothing added to commit but untracked files present (use "git add" to track)
```

We next want to add *all* changes by using the command `add -A` and then `commit` the changes along with a message describing what has changed.

Command Line

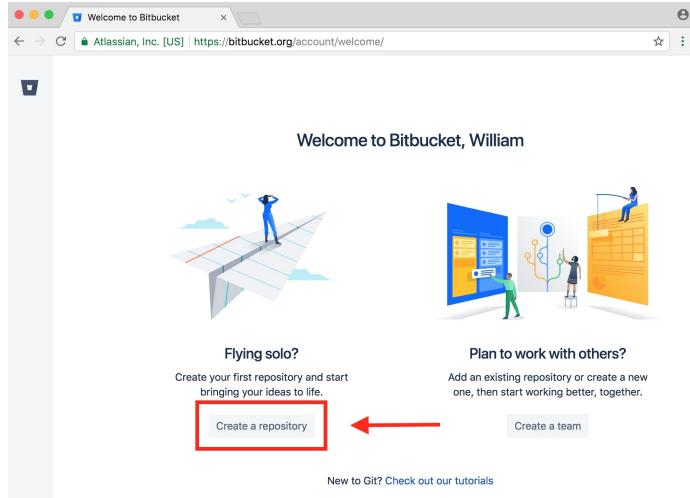
```
(helloworld) $ git add -A  
(helloworld) $ git commit -m 'initial commit'
```

Please note Windows users may receive an error [git commit error: pathspec 'commit' did not match any file\(s\) known to git](#) which appears to be related to using single quotes '' as opposed to double quotes "". If you see this error, using double quotes for all commit messages going forward.

Bitbucket It's a good habit to create a remote repository of your code for each project. This way you have a backup in case anything happens to your computer and more importantly, it allows for collaboration with other software developers. The two most popular choices are [Bitbucket](#) and [Github](#).

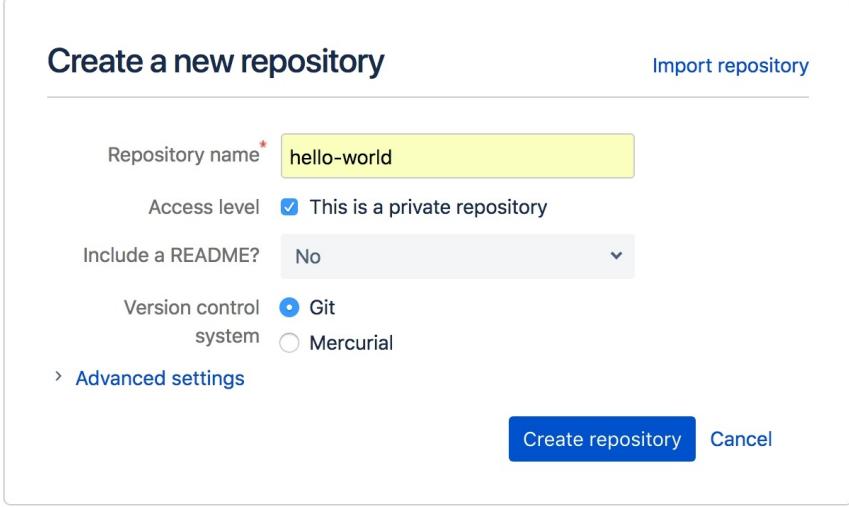
In this book we will use Bitbucket because it allows private repositories **for free**. Github charges a fee. Public repositories are available for anyone on the internet to use; private repositories are not. When you're learning web development, it's best to stick to private repositories so you don't inadvertently post critical information such as passwords online.

To get started on Bitbucket, [sign up for a free account](#). After confirming your account via email you'll be redirected to the Bitbucket welcome page. Click on the link for "Create a repository".



Bitbucket welcome page

Then on the “Create Repo” page enter in the name of your repository: “hello-world”. Then click the blue “Create repository button”:



A screenshot of the “Create a new repository” form. The repository name is set to “hello-world”. The “Access level” is “This is a private repository”. The “Include a README?” dropdown is set to “No”. The “Version control system” is “Git”. At the bottom are “Create repository” and “Cancel” buttons.

Bitbucket create repo

On the next page, click on the link on the bottom for “I have an existing project” which will open a dropdown:

Bitbucket existing project

We're already in the directory for our repo so skip Step 1. In Step 2, we'll use two commands to add our project to Bitbucket. Note that your command will differ from mine since you have a different username. The general format is the below where <USER> is your Bitbucket username.

Command Line

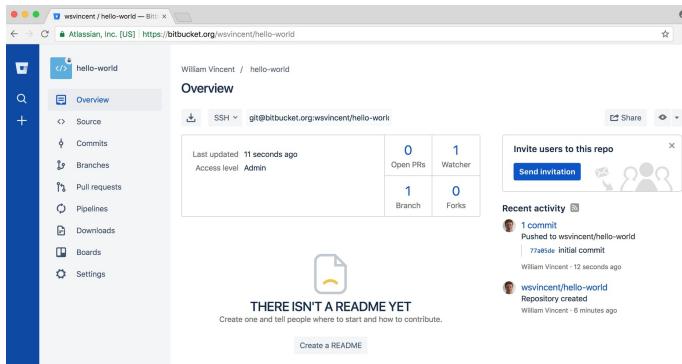
```
(helloworld) $ git remote add origin https://<USER>@bitbucket.org/<USER>/hello-world\\.git
```

After running this command to configure git with this Bitbucket repository, we must “push” our code into it.

Command Line

```
(helloworld) $ git push -u origin master
```

Now if you go back to your Bitbucket page and refresh it, you'll see the code is now online! Click on the “Source” tab on the left to see it all.



Bitbucket overview

Since we're done, go ahead and exit our virtual environment with the `exit` command.

Command Line

```
(helloworld) $ exit
```

You should now see no parentheses on your command line, indicating the virtual environment is no longer active.

Conclusion Congratulations! We've covered a lot of fundamental concepts in this chapter. We built our first Django application and learned about Django's project/app structure. We started to learn about views, urls, and the internal web server. And we worked with git to track our changes and pushed our code into a private repo on Bitbucket.

Continue on to **Chapter 3: Simple app** where we'll build and deploy a more complex Django application using templates and class-based views.

Chapter 3: Pages app

In this chapter we'll build, test, and deploy a *Pages* app that has 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.

Initial Setup As in Chapter 2: Hello World app, our initial setup involves 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
- update `settings.py`

On the command line make sure you're not working in an existing virtual environment. You can tell if there's anything in parentheses before your command line prompt. If you are simply type `exit` to leave it.

We will again create a new directory `pages` for our project on the Desktop but you can put your code anywhere you like on your computer. It just needs to be in its own directory.

Within a new command line console start by typing the following:

Command Line

```
$ cd ~/Desktop
$ mkdir pages
$ cd pages
$ pipenv install django==2.0.6
$ pipenv shell
(pages) $ django-admin startproject pages_project .
(pages) $ python manage.py startapp pages
```

I'm using `(pages)` here to represent the virtual environment but in reality mine has the form of `(pages-un0YeQ9e)`. Your virtual environment name will be unique, too, something like `(pages-XXX)`.

Open your text editor and navigate to the file `settings.py`. Add the `pages` app to our project under `INSTALLED_APPS`:

Code

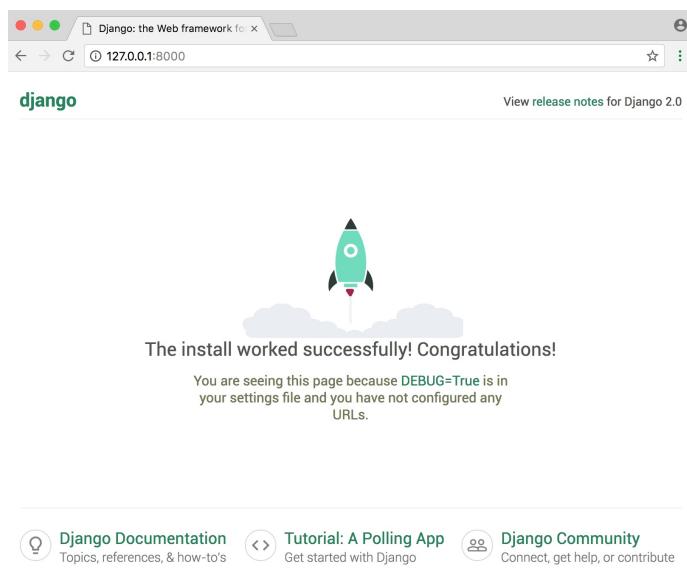
```
# pages_project/settings.py
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'pages', # new
]
```

Start the local web server with `runserver`.

Command Line

```
(pages) $ python manage.py runserver
```

And then navigate to <http://127.0.0.1:8000/>.



Django welcome page

Templates Every web framework needs a convenient way to generate HTML files. In Django, the approach is to use templates so that individual HTML files can be served by a view to a web page specified by the URL.

It's worth repeating this pattern since you'll see it over and over again in Django development: Templates, Views, and URLs. The order in which you create them doesn't much matter since all three are required and work closely together. The URLs control the initial route, the entry point into a page, such as /about, the views contain the logic or the "what", and 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.

So: Templates, Views, URLs. This pattern will hold true for **every Django web page you make**. However it will take some repetition before you internalize it.

Ok, moving on. The question of where to place the templates directory can be confusing for beginners. By default, Django looks within each app for templates. In our pages app it will expect a `home.html` template to be located in the following location:

```
└── templates
    └── pages
        └── home.html
```

This means we would need to create a new `templates` directory, a new directory with the name of the app, `pages`, and finally our template itself which is `home.html`.

A common question is: Why this repetitive structure? The short answer is that the Django template loader wants to be really sure it finds the correct template and this is how it's programmed to look for them.

Fortunately there's another often-used approach to structuring the templates in a Django project. And that is to instead create a single, project-level `templates` directory that is available to *all apps*. This is the approach we'll use. By making a small tweak to our `settings.py` file we can tell Django to *also* look in this project-level folder for templates.

First quit our server with `control-c`. Then create a project-level folder called `templates` and an HTML file called `home.html`.

Command Line

```
(pages) $ mkdir templates
(pages) $ touch templates/home.html
```

Next we need to update `settings.py` to tell Django to look at the project-level for templates. This is a one-line change to the setting '`DIRS`' under `TEMPLATES`.

Code

```
# pages_project/settings.py
TEMPLATES = [
{
...
'DIRS': ['templates'],
...
},
```

Then we can add a simple headline to our `home.html` file.

Code

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

Ok, our template is complete! The next step is to configure our url and view.

Class-Based Views Early versions of Django only shipped with function-based views, but developers soon found themselves repeating the same patterns over and over again. Write a view that lists all objects in a model. Write a view that displays only one detailed item from a model. And so on.

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.

Classes are a fundamental part of Python but a thorough discussion of them is beyond the scope of this book. If you need an introduction or refresher, I suggest reviewing the [official Python docs](#) which have an excellent tutorial on classes and their usage.

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

Code

```
# 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.

URLs 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.

Code

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

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

The code here should be review at this point. We add `include` on the second line to point the existing URL to the `pages` app. Next create an app-level `urls.py` file.

Command Line

```
(pages) $ touch pages/urls.py
```

And add the following code.

Code

```
# pages/urls.py
from django.urls import path

from . import views

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

]

This pattern is almost identical to what we did in Chapter 2 with one major difference. When using Class-Based Views, you always add `as_view()` at the end of the view name.

And we're done! If you start up the web server with `python manage.py runserver` and navigate to <http://127.0.0.1:8000/> you can see our new homepage.



Homepage

Add an About Page The process for adding an about page is very 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`.

Command Line

```
(pages) $ touch templates/about.html
```

Then populate it with a short HTML headline.

Code

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

Create a 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):
    template_name = 'about.html'
```

And then connect it to a url at about/.

Code

```
# pages/urls.py
from django.urls import path

from . import views

urlpatterns = [
    path('', views.HomePageView.as_view(), name='home'),
    path('about/', views.AboutPageView.as_view(), name='about'),
]
```

Start up the web server with `python manage.py runserver`.

Navigate to <http://127.0.0.1:8000/about> and you can see our new “About page”.



About page

Extending Templates The real power of templates is their ability to be extended. If you think about most web sites, there is content that is repeated on every page (header, footer, etc). Wouldn't it be nice if we, as developers, could have *one canonical place* for our header code that would be inherited by all other templates?

Well we can! Let's create a `base.html` file containing a header with links to our two pages. First `Control-c` and then type the following.

Command Line

```
(pages) $ touch templates/base.html
```

Django has a minimal templating language for adding links and basic logic in our templates. You can see the full list of built-in template tags [here in the official docs](#). Template tags take the form of `{% something %}` where the “something” is the template tag itself. You can even create your own custom template tags, though we won't do that in this book.

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? This is why. The `url` tag uses 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' %}`.

Code

```
<!-- 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.

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.

Code

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

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

Code

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

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

Now if you start up the server with `python manage.py runserver` and open up our webpages again at <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.

Nice, right?



Homepage.

Homepage with header



About page.

About page with header

There's *a lot* more we can do with templates and in practice you'll typically create a base .html file and then add additional templates on top of it in a robust Django project. We'll do this later on in the book.

Tests Finally we come to tests. Even in an application this basic, it's important to add tests and get in the habit of always adding them to our Django projects. In the words of Jacob Kaplan-Moss, one of Django's original creators, “Code without tests is broken as designed.”

Writing tests is important because it automates the process of confirming that the code works as expected. In an app like this one, we can manually look and see that the home page and about page exist and contain the intended content. But as a Django project grows in size there can be hundreds if not thousands of individual web pages and the idea of manually going through each page is not possible. Further, whenever we make changes to the code—adding new features, updating existing ones, deleting unused areas of the site—we want to be sure that we have not inadvertently broken some other piece of the site. Automated tests let us write one time how we expect a specific piece of our project to behave and then let the computer do the checking for us.

Fortunately Django comes with robust, built-in [testing tools](#) for writing and running tests.

If you look within our pages app, Django already provided a `tests.py` file we can use. Open it and add the following code:

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](#). That's a fancy way of saying it ensures that a given webpage actually exists, but says nothing about the content of said page.

To run the tests quit the server Control-c and type `python manage.py test` on the command line:

Command Line

```
(pages) $ python manage.py test
Creating test database for alias 'default'...
.
-----
.
Ran 2 tests in 0.028s

OK
Destroying test database for alias 'default'...
```

Success! We'll do much more with testing in the future, especially once we start working with databases. For now, it's important to see how easy it is to add tests each and every time we add new functionality to our Django project.

Git and Bitbucket It's time to track our changes with *git* and push them up to Bitbucket. We'll start by initializing our directory.

Command Line

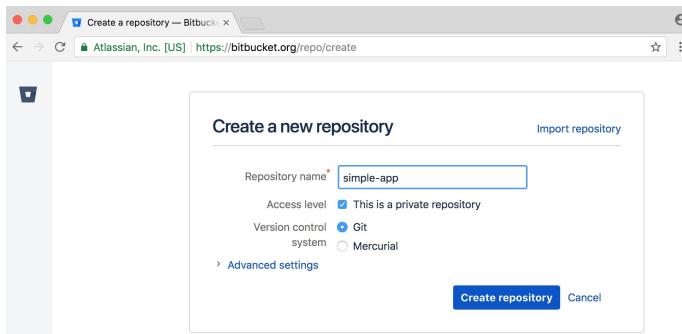
```
(pages) $ git init
```

Use `git status` to see all our code changes then `git add -A` to add them all. Finally we'll add our first commit message.

Command Line

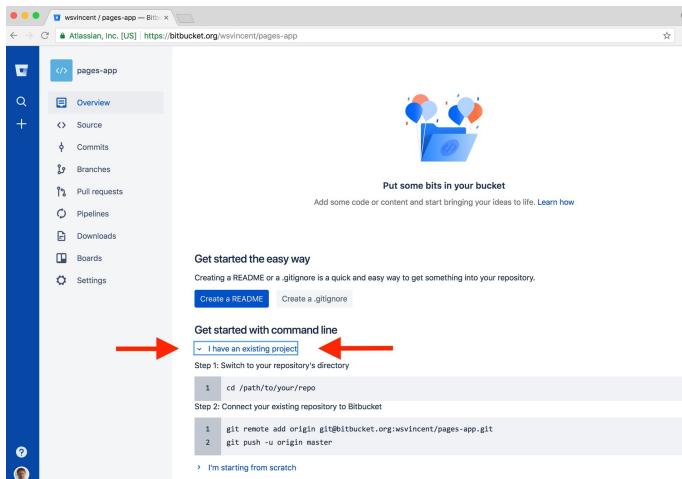
```
(pages) $ git status  
(pages) $ git add -A  
(pages) $ git commit -m 'initial commit'
```

Over on Bitbucket [create a new repo](#) which we'll call pages-app.



Bitbucket Create Page

On the next page click on the bottom link for “I have an existing project”. Copy the two commands to connect and then push the repository to Bitbucket.



Bitbucket Existing Project

It should look like this, replacing `wsvincent` with your Bitbucket username:
Command Line

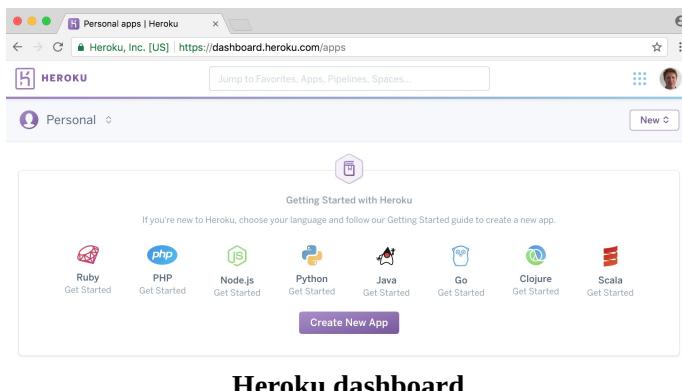
```
(pages) $ git remote add origin git@bitbucket.org:wsvincent/pages-
```

```
app.git  
(pages) $ git push -u origin master
```

Local vs Production Up to this point we've been using Django's own internal web server to power our *Pages* application locally on our computer. But you can't share a localhost address with someone else. To make our site available on the Internet where everyone can see it, we need to deploy our code to an external server that anyone can use to see our site. This is called putting our code into *production*. Local code lives only on our computer; production code lives on an external server.

There are many server providers available but we will use [Heroku](#) because it is free for small projects, widely-used, and has a relatively straightforward deployment process.

Heroku You can sign up for a free [Heroku](#) account on their website. After you confirm your email Heroku will redirect you to the dashboard section of the site.



Heroku dashboard

Now we need to install Heroku's *Command Line Interface (CLI)* so we can deploy from the command line. We want to install Heroku globally so it is available across our entire computer, so open up a new command line tab: Command + t on a Mac, Control + t on Windows. If we installed Heroku within our virtual environment, it would only be available there.

Within this new tab, on a Mac use Homebrew to install Heroku:
Command Line

```
$ brew install heroku
```

On Windows, see the [Heroku CLI page](#) to correctly install either the 32-bit or 64-bit version.

Once installation is complete you can close our new command line tab and return to the initial tab with the pages virtual environment active.

Type the command `heroku login` and use the email and password for Heroku you just set.

Command Line

```
(pages) $ heroku login  
Enter your Heroku credentials:  
Email: will@wsvincent.com  
Password: *****  
Logged in as will@wsvincent.com
```

Additional Files We need to make the following four changes to our Pages project so it's ready to deploy online with Heroku:

- update `Pipfile.lock`
- make a new `Procfile` file
- install `gunicorn` as our web server
- make a one-line change to `settings.py` file

Within your existing `Pipfile` specify the version of Python we're using, which is 3.6. Add these two lines at the bottom of the file.

Code

```
# Pipfile  
[requires]  
python_version = "3.6"
```

Then run `pipenv lock` to generate the appropriate `Pipfile.lock`.

Command Line

```
(pages) $ pipenv lock
```

Heroku actually looks in our `Pipfile.lock` for information on our virtual environment, which is why we add the language setting here.

Next create a `Procfile` which is specific to Heroku.

Command Line

```
(pages) $ touch Procfile
```

Open the `Procfile` with your text editor and add the following:

Code

```
web: gunicorn pages_project.wsgi --log-file -
```

This says to use our existing `pages_project.wsgi` file but with [gunicorn](#), which is a web server suitable for production, instead of Django's own server which is only usable for local development.

Command Line

```
(pages) $ pipenv install gunicorn==19.8.1
```

The final step is a one-line change to `settings.py`. Scroll down to the section called `ALLOWED_HOSTS` and add a '*' so it looks as follows:

Code

```
# pages_project/settings.py
ALLOWED_HOSTS = ['*']
```

The [`ALLOWED_HOSTS`](#) setting represents which host/domain names our Django site can serve. This is a security measure to prevent HTTP Host header attacks, which are possible even under many seemingly-safe web server configurations. However we've used the wildcard Asterisk * which means all domains are acceptable to keep things simple. In a production-level Django site you would explicitly list which domains were allowed.

Use `git status` to check our changes, add the new files, and then commit them:

Command Line

```
(pages) $ git status
(pages) $ git add -A
(pages) $ git commit -m "New updates for Heroku deployment"
```

Finally push to Bitbucket so we have an online backup of our code changes.

Command Line

```
(pages) $ git push -u origin master
```

Deploy The last step is to actually deploy with Heroku. If you've ever configured a server yourself in the past, you'll be amazed at how much simpler the process is with a platform-as-a-service provider like Heroku.

Our process will be as follows:

- create a new app on Heroku and push our code to it
- add a git remote “hook” for Heroku
- configure the app to ignore static files, which we’ll cover in later chapters
- start the Heroku server so the app is live
- visit the app on Heroku’s provided URL

We can do the first step, creating a new Heroku app, from the command line with `heroku create`. Heroku will create a random name for our app, in my case `cryptic-oasis-40349`. Your name will be different.

Command Line

```
(pages) $ heroku create
Creating app... done, ⬤ cryptic-oasis-40349
https://cryptic-oasis-40349.herokuapp.com/ | https://git.heroku.com/cryptic-oasis-40\
349.git
```

Now we need to add a “hook” for Heroku within git. This means that git will store both our settings for pushing code to Bitbucket and to Heroku. My Heroku app is called `cryptic-oasis-40349` so my command is as follows.

Command Line

```
(pages) $ heroku git:remote -a cryptic-oasis-40349
```

You should replace `cryptic-oasis-40349` with the app-name Heroku provides.

We only need to do one set of Heroku configurations at this point, which is to tell Heroku to ignore static files like CSS and JavaScript which Django by default tries to optimize for us. We’ll cover this in later chapters so for now just run the following command.

Command Line

```
(pages) $ heroku config:set DISABLE_COLLECTSTATIC=1
```

Now we can push our code to Heroku. Because we set our “hook” previously, it will go to Heroku.

Command Line

```
(pages) $ git push heroku master
```

If we just typed `git push origin master` then the code is pushed to Bitbucket, not Heroku. Adding `heroku` to the command sends the code to Heroku. This is a little confusing the first few times.

Finally we need to make our Heroku app live. As websites grow in traffic they need additional Heroku services but for our basic example we can use the lowest level, `web=1`, which also happens to be free!

Type the following command.

Command Line

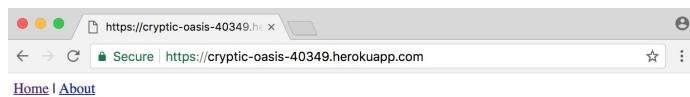
```
(pages) $ heroku ps:scale web=1
```

We’re done! The last step is to confirm our app is live and online. If you use the command `heroku open` your web browser will open a new tab with the URL of your app:

Command Line

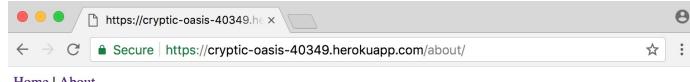
```
(pages) $ heroku open
```

Mine is at <https://cryptic-oasis-40349.herokuapp.com/about/>. You can see both the homepage and about page are up now.



Homepage.

Homepage on Heroku



[Home](#) | [About](#)

About page.

[About page on Heroku](#)

Conclusion Congratulations on building and deploying your second Django project! This time we used templates, class-based views, explored URLConfs more fully, added basic tests, and used Heroku. Next up we'll move on to our first database-backed project and see where Django really shines.

Chapter 4: Message Board app

In this chapter we will use a database for the first time to build a basic *Message Board* application where users can post and read short messages. We'll explore Django's powerful built-in admin interface which provides a visual way to make changes to our data. And after adding tests we will push our code to Bitbucket and deploy the app on Heroku.

Django provides built-in support for several types of database backends. With just a few lines in our `settings.py` file it can support PostgreSQL, MySQL, Oracle, or SQLite. But the simplest—**by far**—to use is [SQLite](#) because it runs off a single file and requires no complex installation. By contrast, the other options require a process to be running in the background and can be quite complex to properly configure. Django uses SQLite by default for this reason and it's a perfect choice for small projects.

Initial Setup Since we've already set up several Django projects at this point in the book, we can quickly run through our commands to begin a new one. We need to do the following:

- create a new directory for our code on the Desktop called `mb`
- install Django in a new virtual environment
- create a new project called `mb_project`
- create a new app call `posts`
- update `settings.py`

In a new command line console, enter the following commands. Note that I'm using (`mb`) here to represent the virtual environment name even though it's actually (`mb-XXX`) where `XXX` represents random characters.

Command Line

```
$ cd ~/Desktop
$ mkdir mb
$ cd mb
$ pipenv install django==2.0.6
$ pipenv shell
(mb) $ django-admin startproject mb_project .
(mb) $ python manage.py startapp posts
```

Tell Django about the new app posts by adding it to the bottom of the INSTALLED_APPS section of our settings.py file. Open it with your text editor of choice.

Code

```
# mb_project/settings.py
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'posts', # new
]
```

Then execute the `migrate` command to create an initial database based on Django's default settings.

Command Line

```
(mb) $ python manage.py migrate
```

If you look inside our directory with the `ls` command, you'll see there's now a `db.sqlite3` file representing our [SQLite](#) database.

Command Line

```
(mb) $ ls
db.sqlite3 mb_project manage.py
```

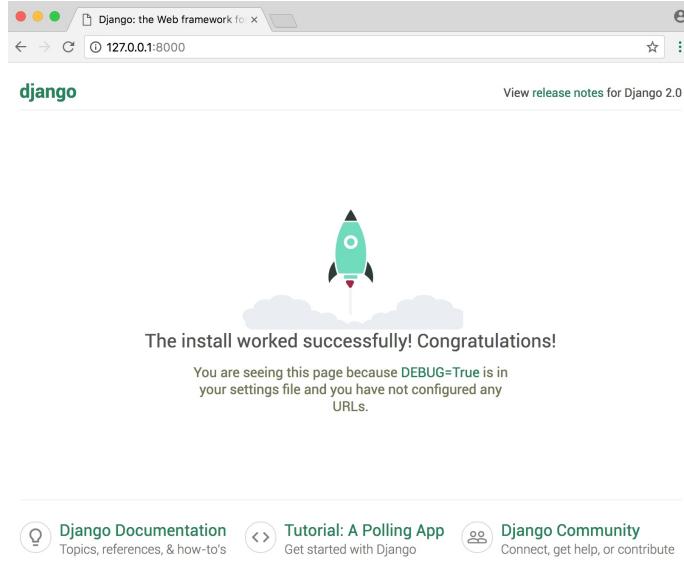
Aside: Technically a `db.sqlite3` file is created the first time you run *either* `migrate` or `runserver`. Using `runserver` configures a database using Django's default settings, however `migrate` will sync the database with the current state of any database models contained in the project and listed in `INSTALLED_APPS`. In other words, to make sure the database reflects the current state of your project you'll need to run `migrate` (and also `makemigrations`) each time you update a model. More on this shortly.

To confirm everything works correctly, spin up our local server.

Command Line

```
(mb) $ python manage.py runserver
```

And navigate to <http://127.0.0.1:8000/> to see the familiar Django installed correctly page.



Django welcome page

Create a database model Our first task is to create a database model where we can store and display posts from our users. Django will turn this model into a database table for us. In real-world Django projects, it's often the case that there will be many complex, interconnected database models but in our simple message board app we only need one.

I won't cover database design in this book but I have written a short guide which [you can find here](#) if this is all new to you.

Open the `posts/models.py` file and look at the default code which Django provides:

Code

```
# posts/models.py
from django.db import models

# Create your models here
```

Django imports a module `models` to help us build new database models, which will "model" the characteristics of the data in our database. We want to create a model to store the textual content of a message board post, which we can do so as follows:

Code

```
# posts/models.py
from django.db import models

class Post(models.Model):
    text = models.TextField()
```

Note that we've created a new database model called `Post` which has the database field `text`. We've also specified the *type of content* it will hold, `TextField()`. Django provides many [model fields](#) supporting common types of content such as characters, dates, integers, emails, and so on.

Activating models Now that our new model is created we need to activate it. Going forward, whenever we create or modify an existing model we'll need to update Django in a two-step process.

1. First we create a migration file with the `makemigrations` command which generate the SQL commands for preinstalled apps in our `INSTALLED_APPS` setting. Migration files **do not execute those commands** on our database file, rather they are a reference of all new changes to our models. This approach means that we have a record of the changes to our models over time.
2. Second we build the actual database with `migrate` which **does execute** the instructions in our migrations file.

Command Line

```
(mb) $ python manage.py makemigrations posts
(mb) $ python manage.py migrate posts
```

Note that you don't *have* to include a name after either `makemigrations` or `migrate`. If you simply run the commands then they will apply to all available changes. But it's a good habit to be specific. If we had two separate apps in our project, and updated the models in both, and then ran `makemigrations` it would generate a migrations file containing data on both changes. This makes debugging harder in the future. You want each migration file to be as small and isolated as possible. That way if you need to look at past migrations, there is only one change per migration rather than one that applies to multiple apps.

Django Admin Django provides us with a robust admin interface for interacting with our database. This is a truly killer feature that few web frameworks offer. It has its routes in [Django's origin as a project at a newspaper](#). The developers wanted a CMS (Content Management System) so that journalists could write and edit their stories without needing to touch “code.” Over time the built-in admin app has evolved into a fantastic, out-of-the-box tool for managing all aspects of a Django project.

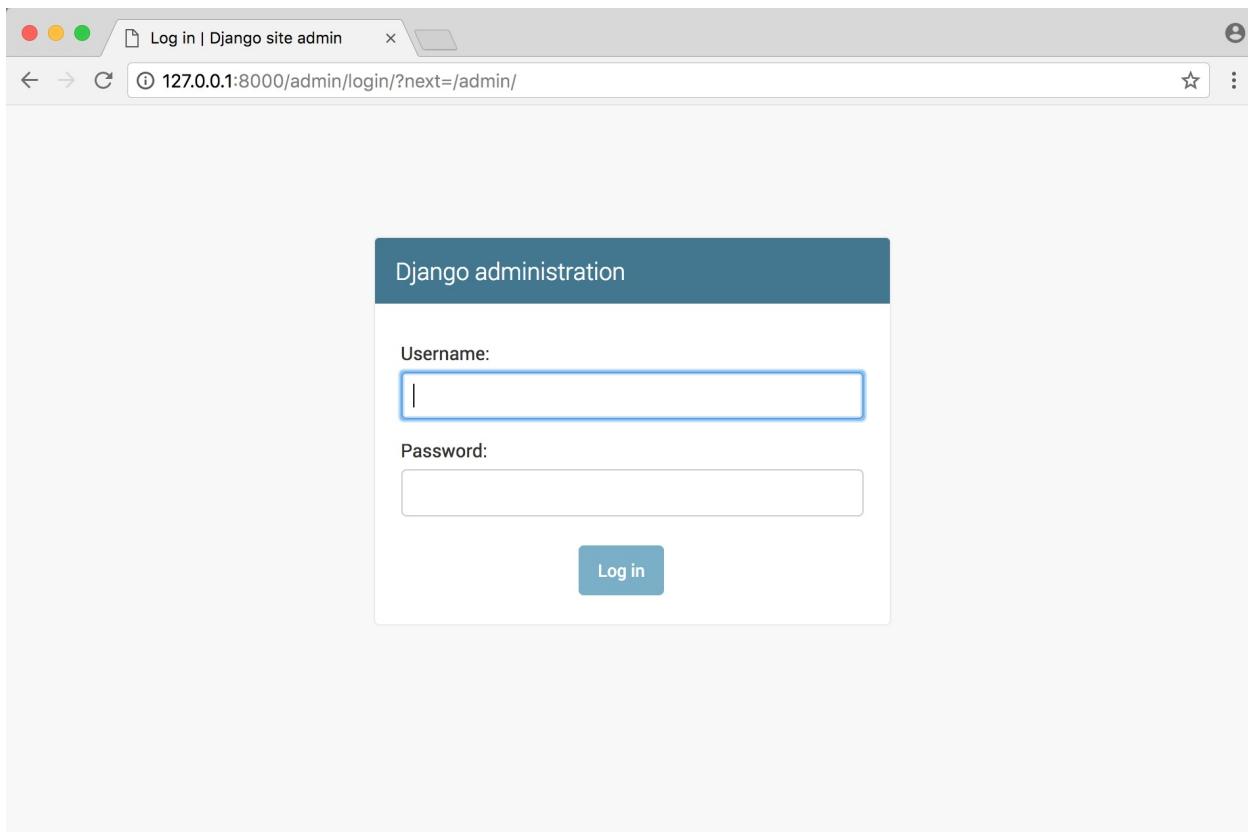
To use the Django admin, we first need to create a superuser who can login. In your command line console, type `python manage.py createsuperuser` and respond to the prompts for a username, email, and password:

Command Line

```
(mb) $ python manage.py createsuperuser
Username (leave blank to use 'wsv'): wsv
Email:
Password:
Password (again):
Superuser created successfully.
```

Note: When you type your password, it will not appear visible in the command line console for security reasons.

Restart the Django server with `python manage.py runserver` and in your browser go to <http://127.0.0.1:8000/admin/>. You should see the admin’s login screen:



Admin login page

Login by entering the username and password you just created. You will see the Django admin homepage next:

The screenshot shows the Django administration homepage. At the top, there's a header bar with the title "Django administration" and a welcome message "WELCOME, wsv. VIEW SITE / CHANGE PASSWORD / LOG OUT". Below the header, the main content area has a section titled "AUTHENTICATION AND AUTHORIZATION". It contains two entries: "Groups" and "Users", each with "Add" and "Change" buttons. To the right of the main content, there are two sidebar sections: "Recent actions" and "My actions", both of which currently show "None available".

Admin homepage

But where's our posts app? It's not displayed on the main admin page!

We need to explicitly tell Django what to display in the admin. Fortunately we can change fix this easily by opening the `posts/admin.py` file and editing it to look like this:

Code

```
# posts/admin.py
from django.contrib import admin

from .models import Post

admin.site.register(Post)
```

Django now knows that it should display our posts app and its database model Post on the admin page. If you refresh your browser you'll see that it now appears:

The screenshot shows the Django administration interface at 127.0.0.1:8000/admin/. The top navigation bar includes links for Site administration | Django site, 127.0.0.1:8000/admin/, and a user account. The main header says "Django administration" and "WELCOME, wsv. VIEW SITE / CHANGE PASSWORD / LOG OUT". On the left, there's a sidebar titled "AUTHENTICATION AND AUTHORIZATION" with "Groups" and "Users" sections, each with "Add" and "Change" buttons. Below that is a section titled "POSTS" with a "Posts" list, also with "Add" and "Change" buttons. To the right, there are two boxes: "Recent actions" (empty) and "My actions" (also empty, with a note "None available").

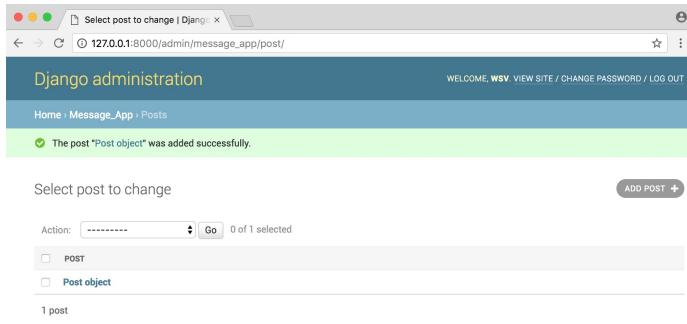
Admin homepage updated

Now let's create our first message board post for our database. Click on the + Add button opposite Posts. Enter your own text in the Text form field.

The screenshot shows the "Add post" page at 127.0.0.1:8000/admin/message_app/post/add/. The top navigation bar and header are identical to the main dashboard. The main content area has a title "Add post". It contains a "Text:" label followed by a large text input field containing the text "Hello world!". At the bottom, there are three buttons: "Save and add another", "Save and continue editing", and a prominent blue "SAVE" button.

Admin new entry

Then click the “Save” button, which will redirect you to the main Post page. However if you look closely, there’s a problem: our new entry is called “Post object”, which isn’t very helpful.



Admin new entry

Let's change that. Within the `posts/models.py` file, add a new function `__str__` as follows:

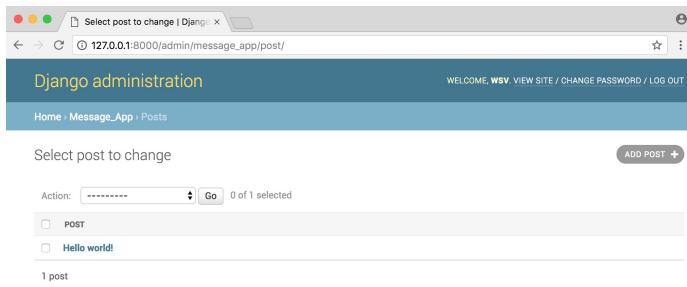
Code

```
# posts/models.py
from django.db import models

class Post(models.Model):
    text = models.TextField()

    def __str__(self):
        """A string representation of the model."""
        return self.text[:50]
```

If you refresh your Admin page in the browser, you'll see it's changed to a much more descriptive and helpful representation of our database entry.



Admin new entry

Much better! It's a best practice to add `str()` methods to all of your models to improve their readability.

Views/Templates/URLs In order to display our database content on our homepage, we have to wire up our views, templates, and URLConfs. This pattern should start to feel familiar now.

Let's begin with the view. Earlier in the book we used the built-in generic [TemplateView](#) to display a template file on our homepage. Now we want to list the contents of our database model. Fortunately this is also a common task in web development and Django comes equipped with the generic class-based [ListView](#).

In the `posts/views.py` file enter the Python code below:

Code

```
# posts/views.py
from django.views.generic import ListView
from .models import Post

class HomePageView(ListView):
    model = Post
    template_name = 'home.html'
```

In the first line we're importing `ListView` and in the second line we need to explicitly define which model we're using. In the view, we subclass `ListView`, specify our model name and specify our template reference. Internally `ListView` returns an object called `object_list` that we want to display in our template.

Our view is complete which means we still need to configure our URLs and make our template. Let's start with the template. Create a project-level directory called `templates` and a `home.html` template file.

Command Line

```
(mb) $ mkdir templates
(mb) $ touch templates/home.html
```

Then update the `DIRS` field in our `settings.py` file so that Django knows to look in this `templates` folder.

Code

```
# settings.py
TEMPLATES = [
{
```

```
...
'DIRS': ['templates'],
...
],
]
```

In our templates file `home.html` we can use the [Django Templating Language's](#) for loop to list all the objects in `object_list`. Remember that `object_list` is what `ListView` returns to us.

Code

```
<!-- templates/home.html -->
<h1>Message board homepage</h1>
<ul>
{% for post in object_list %}
<li>{{ post }}</li>
{% endfor %}
</ul>
```

The last step is to set up our URLConfs. Let's start with the project-level `urls.py` file where we simply include our posts and add `include` on the second line.

Code

```
# mb_project/urls.py
from django.contrib import admin
from django.urls import path, include

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

Then create an app-level `urls.py` file.

Command Line

```
(mb) $ touch posts/urls.py
```

And update it like so:

Code

```
# posts/urls.py
from django.urls import path

from . import views

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

Restart the server with `python manage.py runserver` and navigate to our homepage <http://127.0.0.1:8000/> which now lists out our message board posts.



Message board homepage

- Hello world!

Homepage with posts

We're basically done at this point, but let's create a few more message board posts in the Django admin to confirm that they will display correctly on the homepage.

Adding new posts To add new posts to our message board, go back into the Admin: <http://127.0.0.1:8000/admin/>

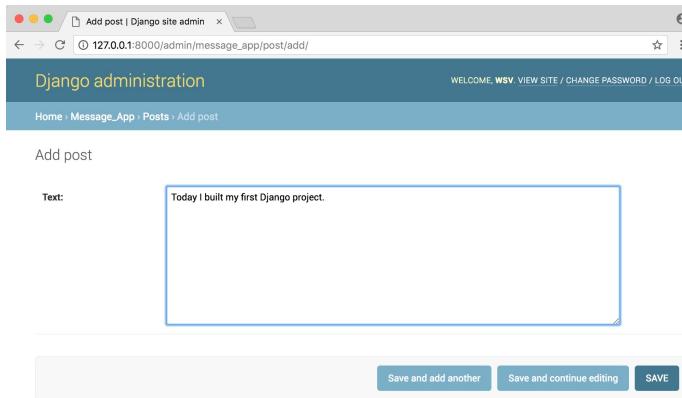
And create two more posts. Here's what mine look like:

Add post

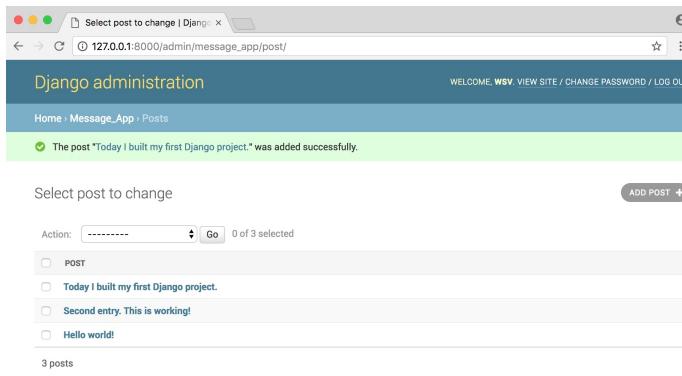
Text: Second entry. This is working!

Save and add another Save and continue editing SAVE

Admin entry



Admin entry



Updated admin entries section

If you return to the homepage you'll see it automatically displays our formatted posts. Woohoo!



Homepage with three entries

Everything works so it's a good time to initialize our directory, add the new code, and include our first git commit.

Command Line

```
(mb) $ git init  
(mb) $ git add -A  
(mb) $ git commit -m 'initial commit'
```

Tests Previously we were only testing static pages so we used [SimpleTestCase](#). But now that our homepage works with a database, we need to use [TestCase](#) which will let us create a “test” database we can check against. In other words, we don’t need to run tests on our *actual* database but instead can make a separate test database, fill it with sample data, and then test against it.

Let’s start by adding a sample post to the text database field and then check that it is stored correctly in the database. It’s important that all our test methods start with `test_` so Django knows to test them! The code will look like this:

Code

```
# posts/tests.py
from django.test import TestCase
from .models import Post

class PostModelTest(TestCase):

    def setUp(self):
        Post.objects.create(text='just a test')

    def test_text_content(self):
        post=Post.objects.get(id=1)
        expected_object_name = f'{post.text}'
        self.assertEqual(expected_object_name, 'just a test')
```

At the top we import the `TestCase` module which lets us create a sample database, then import our `Post` model. We create a new class `PostModelTest` and add a method `setUp` to create a new database that has just one entry: a post with a text field containing the string ‘just a test’.

Then we run our first test, `test_text_content`, to check that the database field actually contains just a test. We create a variable called `post` that represents the first id on our `Post` model. Remember that Django automatically sets this id for us. If we created another entry it would have an id of 2, the next one would be 3, and so on.

The following line uses [f strings](#) which are a very cool addition to Python 3.6. They let us put variables directly in our strings as long as the variables are

surrounded by brackets {}. Here we're setting expected_object_name to be the string of the value in post.text, which should be just a test.

On the final line we use assertEquals to check that our newly created entry does in fact match what we input at the top. Go ahead and run the test on the command line with `python manage.py test`.

Command Line

```
(mb) $ python manage.py test
Creating test database for alias 'default'...
System check identified no issues (0 silenced).

.
-----
Ran 1 test in 0.001s

OK
Destroying test database for alias 'default'...
```

It passed!

Don't worry if the previous explanation felt like information overload. That's natural the first time you start writing tests, but you'll soon find that most tests that you write are actually quite repetitive.

Time for our second test. The first test was on the model but now we want test our one and only page: the homepage. Specifically, we want to test that it exists (throws an HTTP 200 response), uses the home view, and uses the home.html template.

We'll need to add one more import at the top for reverse and a brand new class HomePageViewTest for our test.

Code

```
from django.test import TestCase
from django.urls import reverse
from .models import Post

class PostModelTest(TestCase):

    def setUp(self):
        Post.objects.create(text='just a test')

    def test_text_content(self):
        post=Post.objects.get(id=1)
        expected_object_name = f'{post.text}'
        self.assertEqual(expected_object_name, 'just a test')

class HomePageViewTest(TestCase):

    def setUp(self):
```

```
Post.objects.create(text='this is another test')

def test_view_url_exists_at_proper_location(self):
    resp = self.client.get('/')
    self.assertEqual(resp.status_code, 200)

def test_view_url_by_name(self):
    resp = self.client.get(reverse('home'))
    self.assertEqual(resp.status_code, 200)

def test_view_uses_correct_template(self):
    resp = self.client.get(reverse('home'))
    self.assertEqual(resp.status_code, 200)
    self.assertTemplateUsed(resp, 'home.html')
```

If you run our tests again you should see that they pass.

Command Line

```
(mb) $ python manage.py test
Creating test database for alias 'default'...
System check identified no issues (0 silenced).

-----
Ran 4 tests in 0.036s

OK
Destroying test database for alias 'default'...
```

Why does it say four tests? Remember that our `setUp` methods are not actually tests, they merely let us run subsequent tests. Our four actual tests are `test_text_content`, `test_view_url_exists_at_proper_location`, `test_view_url_by_name`, and `test_view_uses_correct_template`.

Any function that has the word `test*` at the beginning and exists in a `tests.py` file will be run when we execute the command `python manage.py test`.

We're done adding code for our testing so it's time to commit the changes to git.

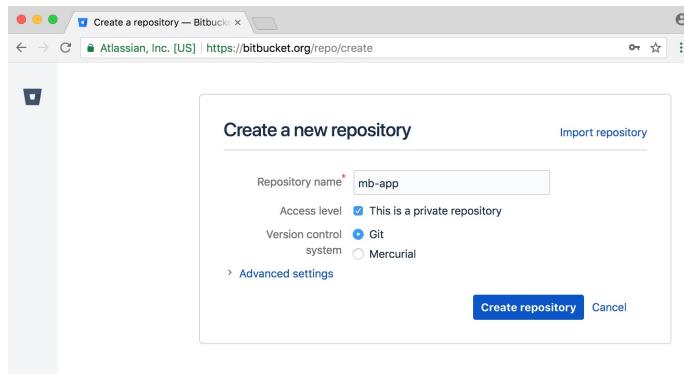
Command Line

```
(mb) $ git add -A
(mb) $ git commit -m 'added tests'
```

Bitbucket We also need to store our code on Bitbucket. This is a good habit to get into in case anything happens to your local computer and it also allows you to share and collaborate with other developers.

You should already have a Bitbucket account from **Chapter 3** so go ahead and

[create a new repo](#) which we'll call mb-app.



Bitbucket create app

On the next page click on the bottom link for “I have an existing project”. Copy the two commands to connect and then push the repository to Bitbucket.

It should look like this, replacing wsvincent (my username) with your Bitbucket username:

Command Line

```
(mb) $ git remote add origin git@bitbucket.org:wsvincent/mb-app.git  
(mb) $ git push -u origin master
```

Heroku configuration You should also already have a Heroku account setup and installed from Chapter 3. We need to make the following changes to our *Message Board* project to deploy it online:

- update Pipfile.lock
- new Procfile
- install gunicorn
- update settings.py

Within your Pipfile specify the version of Python we're using, which is 3.6. Add these two lines at the bottom of the file.

Code

```
# Pipfile  
[requires]  
python_version = "3.6"
```

Run `pipenv lock` to generate the appropriate `Pipfile.lock`.

Command Line

```
(mb) $ pipenv lock
```

Then create a `Procfile` which tells Heroku *how* to run the remote server where our code will live.

Command Line

```
(mb) $ touch Procfile
```

For now we're telling Heroku to use gunicorn as our production server and look in our `mb_project.wsgi` file for further instructions.

Command Line

```
web: gunicorn mb_project.wsgi --log-file -
```

Next install [gunicorn](#) which we'll use in production while still using Django's internal server for local development use.

Command Line

```
(mb) $ pipenv install gunicorn==19.8.1
```

Finally update `ALLOWED_HOSTS` in our `settings.py` file.

Code

```
# mb_project/settings.py
ALLOWED_HOSTS = ['*']
```

We're all done! Add and commit our new changes to git and then push them up to Bitbucket.

Command Line

```
(mb) $ git status
(mb) $ git add -A
(mb) $ git commit -m 'New updates for Heroku deployment'
(mb) $ git push -u origin master
```

Heroku deployment Make sure you're logged into your correct Heroku account.

Command Line

```
(mb) $ heroku login
```

Then run the `create` command and Heroku will randomly generate an app name for you. You can customize this later if desired.

Command Line

```
(mb) $ heroku create
Creating app... done, ⬤ agile-inlet-25811
https://agile-inlet-25811.herokuapp.com/ | https://git.heroku.com/agile-inlet-25811.git
```

Set git to use the name of your new app when you push code to Heroku. My Heroku-generated name is `agile-inlet-25811` so the command looks like this.

Command Line

```
(mb) $ heroku git:remote -a agile-inlet-25811
```

Tell Heroku to ignore static files which we'll cover in-depth when deploying our *Blog* app later in the book.

Command Line

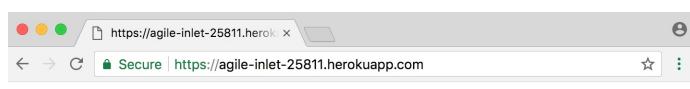
```
(mb) $ heroku config:set DISABLE_COLLECTSTATIC=1
```

Push the code to Heroku and add free scaling so it's actually running online, otherwise the code is just sitting there.

Command Line

```
(mb) $ git push heroku master
(mb) $ heroku ps:scale web=1
```

If you open the new project with `heroku open` it will automatically launch a new browser window with the URL of your app. Mine is live at: <https://agile-inlet-25811.herokuapp.com/>.



Conclusion We've now built, tested, and deployed our first database-driven app. While it's deliberately quite basic, now we know how to create a database model, update it

with the admin panel, and then display the contents on a web page. But something is missing, no?

In the real-world, users need forms to interact with our site. After all, not everyone should have access to the admin panel. In the next chapter we'll build a blog application that uses forms so that users can create, edit, and delete posts. We'll also add styling via CSS.

Chapter 5: Blog app

In this chapter we'll build a *Blog* application that allows users to create, edit, and delete posts. The homepage will list all blog posts and there will be a dedicated detail page for each individual post. We'll also introduce CSS for styling and learn how Django works with static files.

Initial Setup As covered in previous chapters, our steps for setting up a new Django project are as follows:

- create a new directory for our code on the Desktop called `blog`
- install Django in a new virtual environment
- create a new Django project called `blog_project`
- create a new app `blog`
- perform a migration to set up the database
- update `settings.py`

Execute the following commands in a new command line console. Note that the actual name of the virtual environment will be `(blog-xxx)` where `xxx` represents random characters. I'm using `(blog)` here to keep things simpler since my name will differ from yours.

And don't forget to include the period `.` at the end of the command for creating our new `blog_project`.

Command Line

```
$ cd ~/Desktop
$ mkdir blog
$ cd blog
$ pipenv install django==2.0.6
$ pipenv shell
(blog) $ django-admin startproject blog_project .
(blog) $ python manage.py startapp blog
(blog) $ python manage.py migrate
(blog) $ python manage.py runserver
```

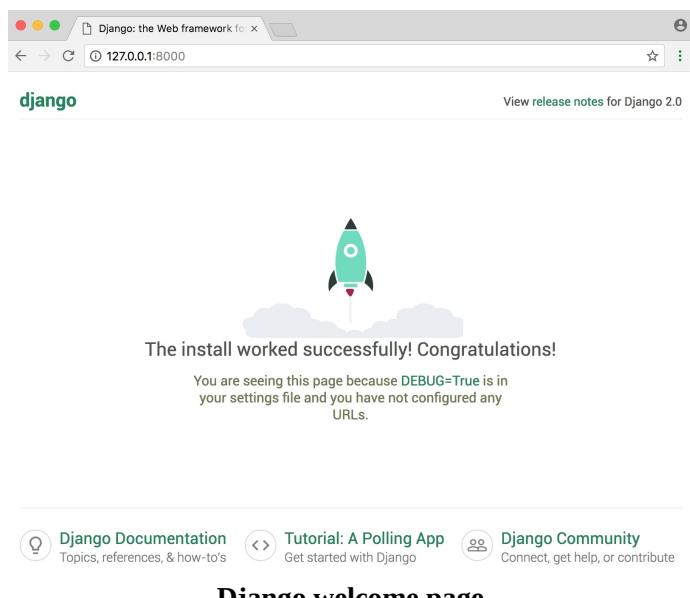
To ensure Django knows about our new app, open your text editor and add the new app to `INSTALLED_APPS` in our `settings.py` file:

Code

```
# blog_project/settings.py
INSTALLED_APPS = [
```

```
'django.contrib.admin',
'django.contrib.auth',
'django.contrib.contenttypes',
'django.contrib.sessions',
'django.contrib.messages',
'django.contrib.staticfiles',
'blog', # new
]
```

If you navigate to <http://127.0.0.1:8000/> in your browser you should see the following page.



Django welcome page

Ok, initial installation complete! Next we'll create our database model for blog posts.

Database Models What are the characteristics of a typical blog application? In our case let's keep things simple and assume each post has a title, author, and body. We can turn this into a database model by opening the `blog/models.py` file and entering the code below:

Code

```
# blog/models.py
from django.db import models
```

```
class Post(models.Model):
    title = models.CharField(max_length=200)
    author = models.ForeignKey(
        'auth.User',
        on_delete=models.CASCADE,
    )
    body = models.TextField()

    def __str__(self):
        return self.title
```

At the top we're importing the class `models` and then creating a subclass of `models.Model` called `Post`. Using this subclass functionality we automatically have access to everything within [django.db.models.Models](#) and can add additional fields and methods as desired.

For `title` we're limiting the length to 200 characters and for `body` we're using a `TextField` which will automatically expand as needed to fit the user's text. There are many field types available in Django; you can see the [full list here](#).

For the `author` field we're using a [ForeignKey](#) which allows for a *many-to-one* relationship. This means that a given user can be the author of many different blog posts but not the other way around. The reference is to the built-in `User` model that Django provides for authentication. For all many-to-one relationships such as a `ForeignKey` we must also specify an [on_delete](#) option.

Now that our new database model is created we need to create a new migration record for it and migrate the change into our database. This two-step process can be completed with the commands below:

Command Line

```
(blog) $ python manage.py makemigrations blog
(blog) $ python manage.py migrate blog
```

Our database is configured! What's next?

Admin We need a way to access our data. Enter the Django admin! First create a superuser account by typing the

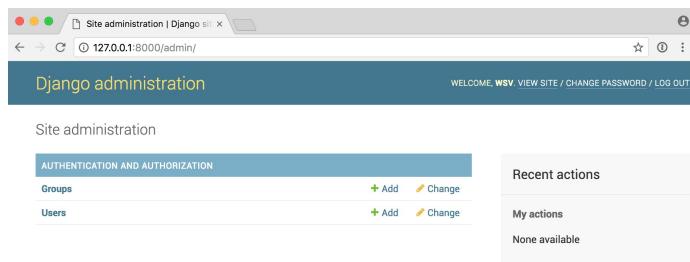
command below and following the prompts to set up an email and password. Note that when typing your password, it will not appear on the screen for security reasons.

Command Line

```
(blog) $ python manage.py createsuperuser
Username (leave blank to use 'wsv'): wsv
Email:
Password:
Password (again):
Superuser created successfully.
```

Now start running the Django server again with the command `python manage.py runserver` and open up the Django admin at <http://127.0.0.1:8000/admin/>. Login with your new superuser account.

Oops! Where's our new Post model?



Admin homepage

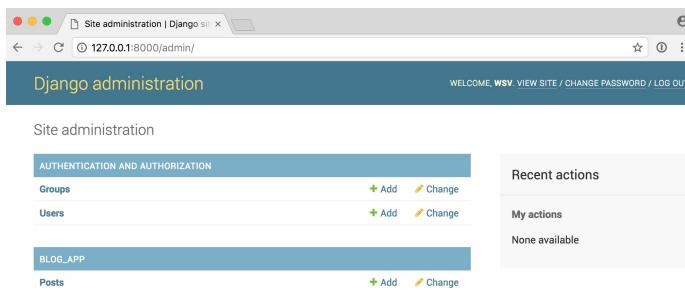
We forgot to update `blog/admin.py` so let's do that now.

Code

```
# blog/admin.py
from django.contrib import admin
from .models import Post

admin.site.register(Post)
```

If you refresh the page you'll see the update.



Admin homepage

Let's add two blog posts so we have some sample data to work with. Click on the + Add button next to Posts to create a new entry. Make sure to add an "author" to each post too since by default all model fields are required. If you try to enter a post without an author you will see an error. If we wanted to change this, we could add [field options](#) to our model to make a given field optional or fill it with a default value.

Add post

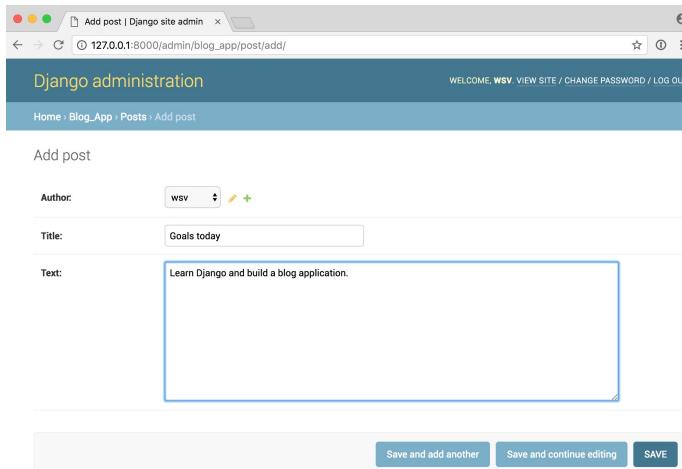
Author: wsv

Title: Hello world!

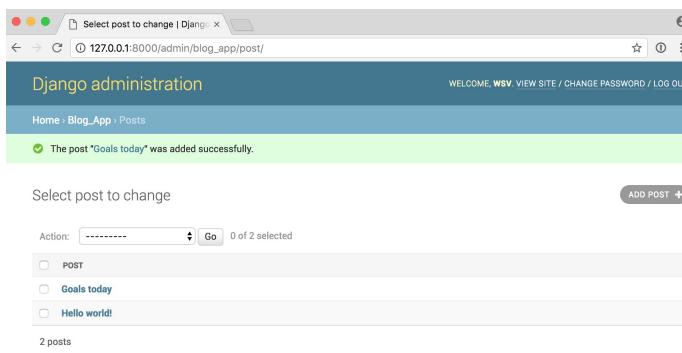
Text: My first blog post. Woohoo!

Save and add another Save and continue editing **SAVE**

Admin first post



Admin second post



Admin homepage with two posts

Now that our database model is complete we need to create the necessary views, URLs, and templates so we can display the information on our web application.

URLs We want to display our blog posts on the homepage so, as in previous chapters, we'll first configure our project-level URLConfs and then our app-level URLConfs to achieve this. Note that “project-level” means in the same parent folder as the `blog_project` and `blog` app folders.

On the command line quit the existing server with `control-c` and create a new `urls.py` file within our `blog`:

Command Line

```
(blog) $ touch blog/urls.py
```

Now update it with the code below.

Code

```
# blog/urls.py
from django.urls import path

from . import views

urlpatterns = [
    path('', views.BlogListView.as_view(), name='home'),
]
```

We're importing our soon-to-be-created views at the top. The empty string '' tells Python to match all values and we make it a named URL, `home`, which we can refer to in our views later on. While it's optional to add a [named URL](#) it's a best practice you should adopt as it helps keep things organized as your number of URLs grows.

We also should update our project-level `urls.py` file so that it knows to forward all requests directly to the `blog` app.

Code

```
# blog_project/urls.py
from django.contrib import admin
from django.urls import path, include

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

We've added `include` on the second line and a urlpattern using an empty string regular expression '' indicating that URL requests should be redirected as is to `blog`'s URLs for further instructions.

Views We're going to use class-based views but if want to see a function-based way to build a blog application, I highly recommend the [Django Girls Tutorial](#). It is excellent.

In our views file, add the code below to display the contents of our `Post` model using `ListView`.

Code

```
# blog/views.py
from django.views.generic import ListView

from .models import Post
```

```
class BlogListView(ListView):
model = Post
template_name = 'home.html'
```

On the top two lines we import [ListView](#) and our database model Post. Then we subclass ListView and add links to our model and template. This saves us a lot of code versus implementing it all from scratch.

Templates With our URLConfs and views now complete, we're only missing the third piece of the puzzle: templates. As we already saw in Chapter 4, we can inherit from other templates to keep our code clean. Thus we'll start off with a base.html file and a home.html file that inherits from it. Then later when we add templates for creating and editing blog posts, they too can inherit from base.html.

Start by creating our project-level templates directory with the two template files.

Command Line

```
(blog) $ mkdir templates
(blog) $ touch templates/base.html
(blog) $ touch templates/home.html
```

Then update settings.py so Django knows to look there for our templates.

Code

```
# blog_project/settings.py
TEMPLATES = [
{
    ...
'DIRS': ['templates'],
...
},
```

Then update the base.html template as follows.

Code

```
<!-- templates/base.html -->
<html>
<head>
<title>Django blog</title>
</head>
```

```
<body>
<header>
<h1><a href="/">Django blog</a></h1>
</header>
<div class="container">
{%
    block content %
}
{%
    endblock content %
}
</div>
</body>
</html>
```

Note that code between `{% block content %}` and `{% endblock content %}` can be filled by other templates. Speaking of which, here is the code for `home.html`.

Code

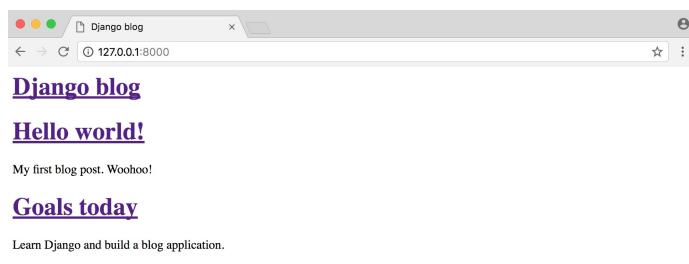
```
<!-- templates/home.html -->
{% extends 'base.html' %}

{% block content %}
{% for post in object_list %}
<div class="post-entry">
<h2><a href="">{{ post.title }}</a></h2>
<p>{{ post.body }}</p>
</div>
{% endfor %}
{% endblock content %}
```

At the top we note that this template extends `base.html` and then wraps our desired code with content blocks. We use the Django Templating Language to set up a simple *for loop* for each blog post. Note that `object_list` comes from `ListView` and contains all the objects in our view.

If you start the Django server again: `python manage.py runserver`.

And refresh <http://127.0.0.1:8000/> we can see it's working.



Blog homepage with two posts

But it looks terrible. Let's fix that!

Static files We need to add some CSS which is referred to as a static file because, unlike our dynamic database content, it doesn't change. Fortunately it's straightforward to add static files like CSS, JavaScript, and images to our Django project.

In a production-ready Django project you would typically store this on a Content Delivery Network (CDN) for better performance, but for our purposes storing the files locally is fine.

First quit our local server with Control-c. Then create a project-level folder called static.

Command Line

```
(blog) $ mkdir static
```

Just as we did with our templates folder we need to update settings.py to tell Django where to look for these static files. We can update settings.py with a one-line change for STATICFILES_DIRS. Add it at the bottom of the file below the entry for STATIC_URL.

Code

```
# blog_project/settings.py
STATICFILES_DIRS = [os.path.join(BASE_DIR, 'static')]
```

Now create a css folder within static and add a new base.css file in it.

Command Line

```
(blog) $ mkdir static/css
(blog) $ touch static/css/base.css
```

What should we put in our file? How about changing the title to be red?

Code

```
/* static/css/base.css */
header h1 a {
color: red;
}
```

Last step now. We need to add the static files to our templates by adding {%- load staticfiles %} to the top of base.html. Because our other templates inherit from base.html we only have to add this once. Include a new line at the bottom

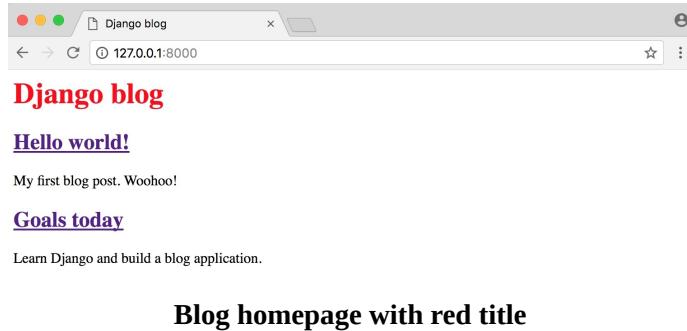
of the `<head></head>` code that explicitly references our new `base.css` file.

Code

```
<!-- templates/base.html -->
{% load static %}
<html>
<head>
<title>Django blog</title>
<link rel="stylesheet" href="{% static 'css/base.css' %}">
</head>
...
```

Phew! That was a bit of a pain but it's a one-time pain. Now we can add static files to our `static` folder and they'll automatically appear in all our templates.

Start up the server again with `python manage.py runserver` and look at our updated homepage at <http://127.0.0.1:8000/>.



Blog homepage with red title

We can do a little better though. How about if we add a custom font and some more CSS? Since this book is not a tutorial on CSS simply insert the following between `<head></head>` tags to add [Source Sans Pro](#), a free font from Google.

Code

```
<!-- templates/base.html -->
{% load static %}
<html>
<head>
<title>Django blog</title>
<link href="https://fonts.googleapis.com/css?family=Source+Sans+Pro:400" rel="stylesheet">
<link rel="stylesheet" href="{% static 'css/base.css' %}">
</head>
...
```

Then update our css file by copy and pasting the following code:

Code

```
/* static/css/base.css */
body {
```

```
font-family: 'Source Sans Pro', sans-serif;
font-size: 18px;
}

header {
border-bottom: 1px solid #999;
margin-bottom: 2rem;
display: flex;
}

header h1 a {
color: red;
text-decoration: none;
}

.nav-left {
margin-right: auto;
}

.nav-right {
display: flex;
padding-top: 2rem;
}

.post-entry {
margin-bottom: 2rem;
}

.post-entry h2 {
margin: 0.5rem 0;
}

.post-entry h2 a,
.post-entry h2 a:visited {
color: blue;
text-decoration: none;
}

.post-entry p {
margin: 0;
font-weight: 400;
}

.post-entry h2 a:hover {
color: red;
}
```

Refresh the homepage at <http://127.0.0.1:8000/> and you should see the

following.



Django blog

Hello world!

My first blog post. Woohoo!

Goals today

Learn Django and build a blog application.

Blog homepage with CSS

Individual blog pages Now we can add the functionality for individual blog pages. How do we do that? We need to create a new view, url, and template. I hope you're noticing a pattern in development with Django now!

Start with the view. We can use the generic class-based [DetailView](#) to simplify things. At the top of the file add `DetailView` to the list of imports and then create our new view called `BlogDetailView`.

Code

```
# blog/views.py
from django.views.generic import ListView, DetailView

from .models import Post

class BlogListView(ListView):
    model = Post
    template_name = 'home.html'

class BlogDetailView(DetailView):
    model = Post
    template_name = 'post_detail.html'
```

In this new view we define the model we're using, `Post`, and the template we want it associated with, `post_detail.html`. By default `DetailView` will provide a context object we can use in our template called either `object` or the lowercased name of our model, `post`. Also, `DetailView` expects either a primary key or a slug passed to it as the identifier. More on this shortly.

Now exit the local server `Control-c` and create our new template for a post

detail as follows:

Command Line

```
(blog) $ touch templates/post_detail.html
```

Then type in the following code:

Code

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

{% block content %}
<div class="post-entry">
<h2>{{ post.title }}</h2>
<p>{{ post.body }}</p>
</div>

{% endblock content %}
```

At the top we specify that this template inherits from `base.html`. Then display the title and body from our context object, which `DetailView` makes accessible as `post`.

Personally I found the naming of context objects in generic views extremely confusing when first learning Django. Because our context object from `DetailView` is either our model name `post` or `object` we could also update our template as follows and it would work exactly the same.

Code

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

{% block content %}
<div class="post-entry">
<h2>{{ object.title }}</h2>
<p>{{ object.body }}</p>
</div>

{% endblock content %}
```

If you find using `post` or `object` confusing, we can also explicitly set the name of the context object in our view. So if we wanted to call it `anything_you_want` and then use that in the template, the code would look as follows and it would work the same.

Code

```
# blog/views.py
...
class BlogDetailView(DetailView):
model = Post
template_name = 'post_detail.html'
context_object_name = 'anything_you_want'
```

Code

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

{% block content %}
<div class="post-entry">
<h2>{{ anything_you_want.title }}</h2>
<p>{{ anything_you_want.body }}</p>
</div>

{% endblock content %}
```

The “magic” naming of the context object is a price you pay for the ease and simplicity of using generic views. They’re great if you know what they’re doing but can be hard to customize if you want different behavior.

Ok, what’s next? How about adding a new URLConf for our view, which we can do as follows.

Code

```
# blog/urls.py
from django.urls import path

from . import views

urlpatterns = [
path('', views.BlogListView.as_view(), name='home'),
path('post/<int:pk>', views.BlogDetailView.as_view(), name='post_detail'),
]
```

All blog post entries will start with `post/`. Next is the primary key for our post entry which will be represented as an integer `<int:pk>`. What’s the primary key you’re probably asking? Django automatically adds an [auto-incrementing primary key](#) to our database models. So while we only declared the fields `title`, `author`, and `body` on our `Post` model, under-the-hood Django also added another field called `id`, which is our primary key. We can access it as either `id` or `pk`.

The `pk` for our first “Hello, World” post is 1. For the second post, it is 2. And so on. Therefore when we go to the individual entry page for our first post, we can expect that its urlpattern will be `post/1`.

Note: Understanding how primary keys work with DetailView is a **very common** place of confusion for beginners. It's worth re-reading the previous two paragraphs a few times if it doesn't click. With practice it will become second nature.

If you now start up the server with `python manage.py runserver` and go directly to <http://127.0.0.1:8000/post/1/> you'll see a dedicated page for our first blog post.



Woohoo! You can also go to <http://127.0.0.1:8000/post/2/> to see the second entry.

To make our life easier, we should update the link on the homepage so we can directly access individual blog posts from there. Currently in `home.html` our link is empty: ``. Update it to ``.

Code

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

{% block content %}
{% for post in object_list %}
<div class="post-entry">
<h2><a href="{% url 'post_detail' post.pk %}">{{ post.title }}</a></h2>
<p>{{ post.body }}</p>
</div>
{% endfor %}
{% endblock content %}
```

We start off by telling our Django template we want to reference a URLConf by using the code `{% url ... %}`. Which URL? The one named `post_detail`, which is the name we gave `BlogDetailView` in our URLConf just a moment ago. If we look at `post_detail` in our URLConf, we see that it expects to be passed an argument `pk` representing the primary key for the blog post. Fortunately, Django has already created and included this `pk` field on our `post` object. We

pass it into the URLConf by adding it in the template as post.pk.

To confirm everything works, refresh the main page at <http://127.0.0.1:8000/> and click on the title of each blog post to confirm the new links work.

Tests We need to test our model and views now. We want to ensure that the Post model works as expected, including its str representation. And we want to test both ListView andDetailView.

Here's what sample tests look like in the blog/tests.py file.

Code

```
# blog/tests.py
from django.contrib.auth import get_user_model
from django.test import Client, TestCase
from django.urls import reverse

from .models import Post

class BlogTests(TestCase):

    def setUp(self):
        self.user = get_user_model().objects.create_user(
            username='testuser',
            email='test@email.com',
            password='secret'
        )

        self.post = Post.objects.create(
            title='A good title',
            body='Nice body content',
            author=self.user,
        )

    def test_string_representation(self):
        post = Post(title='A sample title')
        self.assertEqual(str(post), post.title)

    def test_post_content(self):
        self.assertEqual(f'{self.post.title}', 'A good title')
        self.assertEqual(f'{self.post.author}', 'testuser')
        self.assertEqual(f'{self.post.body}', 'Nice body content')

    def test_post_list_view(self):
        response = self.client.get(reverse('home'))
        self.assertEqual(response.status_code, 200)
        self.assertContains(response, 'Nice body content')
        self.assertTemplateUsed(response, 'home.html')

    def test_post_detail_view(self):
        response = self.client.get('/post/1/')
        no_response = self.client.get('/post/1000000/')
        self.assertEqual(response.status_code, 200)
        self.assertEqual(no_response.status_code, 404)
```

```
self.assertContains(response, 'A good title')
self.assertTemplateUsed(response, 'post_detail.html')
```

There's a lot that's new in these tests so we'll walk through them slowly. At the top we import [get_user_model](#) to reference our active User. We import [TestCase](#) which we've seen before and also [Client\(\)](#) which is new and used as a dummy Web browser for simulating GET and POST requests on a URL. In other words, whenever you're testing views you should use `Client()`.

In our `setUp` method we add a sample blog post to test and then confirm that both its string representation and content are correct. Then we use `test_post_list_view` to confirm that our homepage returns a 200 HTTP status code, contains our body text, and uses the correct `home.html` template. Finally `test_post_detail_view` tests that our detail page works as expected and that an incorrect page returns a 404. It's always good to both test that something **does** exist and that something incorrect **doesn't** exist in your tests.

Go ahead and run these tests now. They should all pass.

Command Line

```
(testy) $ python manage.py test
```

Git Now is also a good time for our first *git commit*. First initialize our directory.

Command Line

```
(testy) $ git init
```

Then review all the content we've added by checking the status. Add all new files. And make our first commit.

Command Line

```
(testy) $ git status
(testy) $ git add -A
(testy) $ git commit -m 'initial commit'
```

Conclusion We've now built a basic blog application from scratch! Using the Django admin we can create, edit, or delete the content. And we used `DetailView` for the first time to create a detailed individual view of each blog post

entry.

In the next section **Chapter 6: Blog app with forms**, we'll add forms so we don't have to use the Django admin at all for these changes.

Chapter 6: Forms

In this chapter we'll continue working on our blog application from [Chapter 5](#) by adding forms so a user can create, edit, or delete any of their blog entries.

Forms Forms are very common and very complicated to implement correctly. Any time you are accepting user input there are security concerns ([XSS Attacks](#)), proper error handling is required, and there are UI considerations around how to alert the user to problems with the form. Not to mention the need for redirects on success.

Fortunately for us [Django's built-in Forms](#) abstract away much of the difficulty and provide a rich set of tools to handle common use cases working with forms.

To start, update our base template to display a link to a page for entering new blog posts. It will take the form `` where `post_new` is the name for our URL.

Your updated file should look as follows:

Code

```
<!-- templates/base.html -->
{% load staticfiles %}
<html>
<head>
<title>Django blog</title>
<link href="https://fonts.googleapis.com/css?family=Source+Sans+Pro:400" rel="stylesheet">
<link rel="stylesheet" href="{% static 'css/base.css' %}">
</head>
<body>
<div class="container">
<header>
<div class="nav-left">
<h1><a href="/">Django blog</a></h1>
</div>
<div class="nav-right">
<a href="{% url 'post_new' %}">+ New Blog Post</a>
</div>
```

```
</header>
{% block content %}
{% endblock content %}
</div>
</body>
</html>
```

Let's add a new URLConf for post_new now:

Code

```
# blog/urls.py
from django.urls import path

from . import views

urlpatterns = [
    path('', views.BlogListView.as_view(), name='home'),
    path('post/<int:pk>/', views.BlogDetailView.as_view(),
         name='post_detail'),
    path('post/new/', views.BlogCreateView.as_view(), name='post_new'),
]
```

Our url will start with post/new/, the view is called BlogCreateView, and the url will be named post_new. Simple, right?

Now let's create our view by importing a new generic class called CreateView and then subclass it to create a new view called BlogCreateView.

Code

```
# blog/views.py
from django.views.generic import ListView, DetailView
from django.views.generic.edit import CreateView
from . models import Post

class BlogListView(ListView):
    model = Post
    template_name = 'home.html'

class BlogDetailView(DetailView):
    model = Post
    template_name = 'post_detail.html'

class BlogCreateView(CreateView):
    model = Post
    template_name = 'post_new.html'
    fields = '__all__'
```

Within `BlogCreateView` we specify our database model `Post`, the name of our template `post_new.html`, and all fields with '`__all__`' since we only have two: `title` and `author`.

The last step is to create our template, which we will call `post_new.html`.

Command Line

```
(blog) $ touch templates/post_new.html
```

And then add the following code:

Code

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

{% block content %}
<h1>New post</h1>
<form action="" method="post">{% csrf_token %}
{{ form.as_p }}
<input type="submit" value="Save" />
</form>
{% endblock %}
```

Let's breakdown what we've done:

- On the top line we inherit from our base template.
- Use HTML `<form>` tags with the POST method since we're *sending* data. If we were receiving data from a form, for example in a search box, we would use GET.
- Add a `{% csrf_token %}` which Django provides to protect our form from cross-site scripting attacks. **You should use it for all your Django forms.**
- Then to output our form data we use `{{ form.as_p }}` which renders it within paragraph `<p>` tags.
- Finally specify an input type of submit and assign it the value "Save".

To view our work, start the server with `python manage.py runserver` and go to the homepage at <http://127.0.0.1:8000/>.



Django blog

[+ New Blog Post](#)

Hello world!

My first blog post. Woohoo!

Goals today

Learn Django and build a blog application.

Homepage with New button

Click on our link for “+ New Blog Post” which will redirect you to:
<http://127.0.0.1:8000/post/new/>.



New post

Author:

Title:

Text:

Blog new page

Go ahead and try to create a new blog post and submit it.



New post

Author:

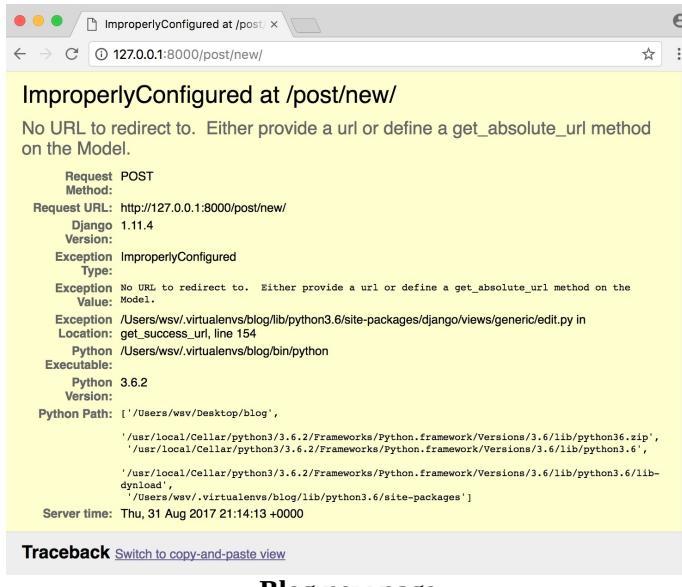
Title:

Text:

I wonder if this will work?

Blog new page

Oops! What happened?



[Blog new page](#)

Django's error message is quite helpful. It's complaining that we did not specify where to send the user after successfully submitting the form. Let's send a user to the detail page after success; that way they can see their completed post.

We can follow Django's suggestion and add a [get_absolute_url](#) to our model. This is a best practice that you should always do. It sets a canonical URL for an object so even if the structure of your URLs changes in the future, the reference to the specific object is the same. In short, you should add a `get_absolute_url()` and `__str__()` method to each model you write.

Open the `models.py` file. Add an import on the second line for [reverse](#) and a new `get_absolute_url` method.

Command Line

```
# blog/models.py
from django.db import models
from django.urls import reverse

class Post(models.Model):
    title = models.CharField(max_length=200)
    author = models.ForeignKey(
        'auth.User',
        on_delete=models.CASCADE,
    )
    body = models.TextField()

    def __str__(self):
        return self.title
```

```
def get_absolute_url(self):
    return reverse('post_detail', args=[str(self.id)])
```

[Reverse](#) is a very handy utility function Django provides us to reference an object by its URL template name, in this case `post_detail`. If you recall our URL pattern it is the following:

Code

```
path('post/<int:pk>/', views.BlogDetailView.as_view(),
      name='post_detail'),
```

That means in order for this route to work we must *also* pass in an argument with the `pk` or primary key of the object. Confusingly `pk` and `id` are interchangeable in Django though the Django docs recommend using `self.id` with `get_absolute_url`. So we're telling Django that the ultimate location of a Post entry is its `post_detail` view which is `posts/<int:pk>/` so the route for the first entry we've made will be at `posts/1`.

Try to create a new blog post again at <http://127.0.0.1:8000/post/new/> and you'll find upon success you are redirected to the detailed view page where the post appears.

The screenshot shows a web browser window titled "Django blog". The address bar displays the URL "127.0.0.1:8000/post/new/". The main content area has a red header "Django blog" and a blue link "+ New Blog Post". Below this, the title "New post" is displayed. A form is present with the following fields:

- Author: wsv
- Title: Is this form working?
- Text: Yes it is!

A "Save" button is located at the bottom left of the form.

Blog new page with input

You'll also notice that our earlier blog post is also there. It was successfully sent to the database, but Django didn't know how to redirect us after that.



Django blog

[+ New Blog Post](#)

Hello world!

My first blog post. Woohoo!

Goals today

Learn Django and build a blog application.

3rd post

I wonder if this will work?

Is this form working?

Yes it is!

Blog homepage with four posts

While we could go into the Django admin to delete unwanted posts, it's better if we add forms so a user can update and delete existing posts directly from the site.

Update Form The process for creating an update form so users can edit blog posts should feel familiar. We'll again use a built-in Django class-based generic view, [UpdateView](#), and create the requisite template, url, and view.

To start, let's add a new link to `post_detail.html` so that the option to edit a blog post appears on an individual blog page.

Code

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

{% block content %}
<div class="post-entry">
<h2>{{ object.title }}</h2>
<p>{{ object.body }}</p>
</div>

<a href="{% url 'post_edit' post.pk %}">+ Edit Blog Post</a>
{% endblock content %}
```

We've added a link using `<a href>...` and the Django template engine's `{% url ... %}` tag. Within it we've specified the target name of our url, which will be called `post_edit` and also passed the parameter needed, which is the primary

key of the post post.pk.

Next we create the template for our edit page called post_edit.html.

Command Line

```
(blog) $ touch templates/post_edit.html
```

And add the following code:

Code

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

{% block content %}
<h1>Edit post</h1>
<form action="" method="post">{% csrf_token %}
{{ form.as_p }}
<input type="submit" value="Update" />
</form>
{% endblock %}
```

We again use HTML <form></form> tags, Django's csrf_token for security, form.as_p to display our form fields with paragraph tags, and finally give it the value "Update" on the submit button.

Now to our view. We need to import UpdateView on the second-from-the-top line and then subclass it in our new view BlogUpdateView.

Code

```
# blog/views.py
from django.views.generic import ListView, DetailView
from django.views.generic.edit import CreateView, UpdateView
from .models import Post

class BlogListView(ListView):
model = Post
template_name = 'home.html'

class BlogDetailView(DetailView):
model = Post
template_name = 'post_detail.html'

class BlogCreateView(CreateView):
model = Post
template_name = 'post_new.html'
fields = '__all__'
```

```
class BlogUpdateView(UpdateView):
model = Post
fields = ['title', 'body']
template_name = 'post_edit.html'
```

Notice that in `BlogUpdateView` we are explicitly listing the fields we want to use `['title', 'body']` rather than using `'__all__'`. This is because we assume that the author of the post is not changing; we only want the title and text to be editable.

The final step is to update our `urls.py` file as follows:

Code

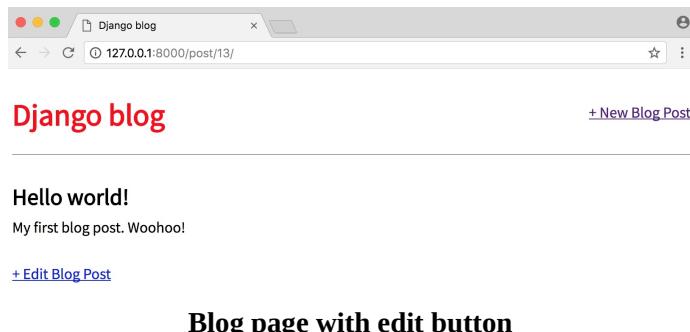
```
# blog/urls.py
from django.urls import path

from . import views

urlpatterns = [
path('', views.BlogListView.as_view(), name='home'),
path('post/<int:pk>/', views.BlogDetailView.as_view(),
name='post_detail'),
path('post/new/', views.BlogCreateView.as_view(), name='post_new'),
path('post/<int:pk>/edit/',
views.BlogUpdateView.as_view(), name='post_edit'),
]
```

At the top we add our view `BlogUpdateView` to the list of imported views, then created a new url pattern for `/post/pk/edit` and given it the name `post_edit`.

Now if you click on a blog entry you'll see our new Edit button.



If you click on “+ Edit Blog Post” you'll be redirected to <http://127.0.0.1:8000/post/1/edit/> if it's your first blog post.



Django blog

[+ New Blog Post](#)

Edit post

Title:

Text:

My first blog post. Woohoo!

Blog edit page

Note that the form is pre-filled with our database's existing data for the post. Let's make a change...



Django blog

[+ New Blog Post](#)

Edit post

Title:

Text:

My first blog post. Woohoo!
I can add new content now.

Blog edit page

And after clicking the "Update" button we are redirected to the detail view of the post where you can see the change. This is because of our `get_absolute_url` setting. Navigate to the homepage and you can see the change next to all the other entries.



Django blog

[+ New Blog Post](#)

Hello world! (EDITED)

My first blog post. Woohoo! I can add new content now.

Goals today

Learn Django and build a blog application.

3rd post

I wonder if this will work?

Is this form working?

Yes it is!

Blog homepage with edited post

Delete View The process for creating a form to delete blog posts is very similar to that for updating a post. We'll use yet another generic class-based view, [DeleteView](#), to help and need to create a view, url, and template for the functionality.

Let's start by adding a link to delete blog posts on our individual blog page, `post_detail.html`.

Code

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

{% block content %}
<div class="post-entry">
<h2>{{ object.title }}</h2>
<p>{{ object.body }}</p>
</div>

<p><a href="{% url 'post_edit' post.pk %}">+ Edit Blog Post</a></p>
<p><a href="{% url 'post_delete' post.pk %}">+ Delete Blog Post</a></p>
{% endblock content %}
```

Then create a new file for our delete page template. First quit the local server Control-c and then type the following command:

Command Line

```
(blog) $ touch templates/post_delete.html
```

And fill it with this code:

Code

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

{% block content %}
<h1>Delete post</h1>
<form action="" method="post">{% csrf_token %}
<p>Are you sure you want to delete "{{ post.title }}"?</p>
<input type="submit" value="Confirm" />
</form>
{% endblock %}
```

Note we are using `post.title` here to display the title of our blog post. We could also just use `object.title` as it too is provided by `DetailView`.

Now update our `views.py` file, by importing `DeleteView` and `reverse_lazy` at the top, then create a new view that subclasses `DeleteView`.

Code

```
# blog/views.py
from django.views.generic import ListView, DetailView
from django.views.generic.edit import CreateView, UpdateView, DeleteView
from django.urls import reverse_lazy

from .models import Post

class BlogListView(ListView):
model = Post
template_name = 'home.html'

class BlogDetailView(DetailView):
model = Post
template_name = 'post_detail.html'

class BlogCreateView(CreateView):
model = Post
template_name = 'post_new.html'
fields = '__all__'

class BlogUpdateView(UpdateView):
model = Post
fields = ['title', 'body']
template_name = 'post_edit.html'

class BlogDeleteView(DeleteView):
model = Post
template_name = 'post_delete.html'
success_url = reverse_lazy('home')
```

We use `reverse_lazy` as opposed to just `reverse` so that it won't execute the URL redirect until our view has finished deleting the blog post.

Finally add a url by importing our view `BlogDeleteView` and adding a new pattern:

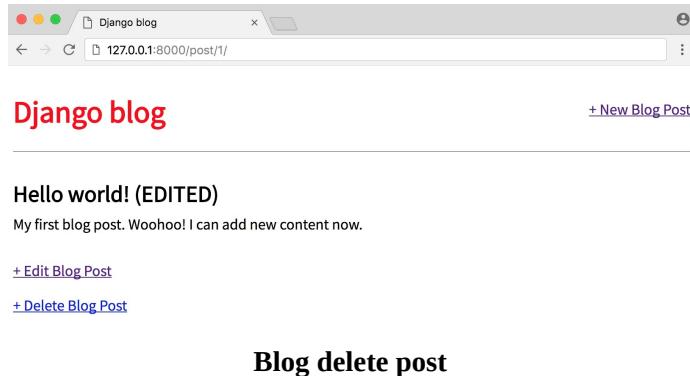
Code

```
# blog/urls.py
from django.urls import path

from . import views

urlpatterns = [
    path('', views.BlogListView.as_view(), name='home'),
    path('post/<int:pk>/', views.BlogDetailView.as_view(),
         name='post_detail'),
    path('post/new/', views.BlogCreateView.as_view(), name='post_new'),
    path('post/<int:pk>/edit/',
         views.BlogUpdateView.as_view(), name='post_edit'),
    path('post/<int:pk>/delete/',
         views.BlogDeleteView.as_view(), name='post_delete'),
]
```

If you start the server again `python manage.py runserver` and refresh the individual post page you'll see our "Delete Blog Post" link.



Clicking on the link takes us to the delete page for the blog post, which displays the name of the blog post.



Django blog

[+ New Blog Post](#)

Delete post

Are you sure you want to delete "Hello world! (EDITED)"?

Blog delete post page

If you click on the “Confirm” button, it redirects you to the homepage where the blog post has been deleted!



Django blog

[+ New Blog Post](#)

Goals today

Learn Django and build a blog application.

3rd post

I wonder if this will work?

Is this form working?

Yes it is!

Homepage with post deleted

So it works!

Tests Time for tests to make sure everything works now—and in the future—as expected. We’ve added a `get_absolute_url` method to our model and new views for create, update, and edit posts. That means we need four new tests:

- `def test_get_absolute_url`
- `def test_post_create_view`
- `def test_post_update_view`
- `def test_post_delete_view`

Update your existing `tests.py` file as follows.

Code

```
# blog/tests.py
from django.contrib.auth import get_user_model
from django.test import Client, TestCase
from django.urls import reverse

from .models import Post


class BlogTests(TestCase):

    def setUp(self):
        self.user = get_user_model().objects.create_user(
            username='testuser',
            email='test@email.com',
            password='secret'
        )

        self.post = Post.objects.create(
            title='A good title',
            body='Nice body content',
            author=self.user,
        )

    def test_string_representation(self):
        post = Post(title='A sample title')
        self.assertEqual(str(post), post.title)

    def test_get_absolute_url(self):
        self.assertEqual(self.post.get_absolute_url(), '/post/1/')

    def test_post_content(self):
        self.assertEqual(f'{self.post.title}', 'A good title')
        self.assertEqual(f'{self.post.author}', 'testuser')
        self.assertEqual(f'{self.post.body}', 'Nice body content')

    def test_post_list_view(self):
        response = self.client.get(reverse('home'))
        self.assertEqual(response.status_code, 200)
        self.assertContains(response, 'Nice body content')
        self.assertTemplateUsed(response, 'home.html')

    def test_post_detail_view(self):
        response = self.client.get('/post/1/')
        no_response = self.client.get('/post/100000/')
        self.assertEqual(response.status_code, 200)
        self.assertEqual(no_response.status_code, 404)
        self.assertContains(response, 'A good title')
        self.assertTemplateUsed(response, 'post_detail.html')

    def test_post_create_view(self):
        response = self.client.post(reverse('post_new'), {
            'title': 'New title',
            'body': 'New text',
            'author': self.user,
        })
        self.assertEqual(response.status_code, 200)
        self.assertContains(response, 'New title')
        self.assertContains(response, 'New text')

    def test_post_update_view(self):
        response = self.client.post(reverse('post_edit', args='1'), {
            'title': 'Updated title',
            'body': 'Updated text',
        })
```

```
self.assertEqual(response.status_code, 302)

def test_post_delete_view(self):
    response = self.client.get(
        reverse('post_delete', args='1'))
    self.assertEqual(response.status_code, 200)
```

We expect the url of our test to be at post/1/ since there's only one post and the 1 is its primary key Django adds automatically for us. To test create view we make a new response and then ensure that the response goes through (status code 200) and contains our new title and body text. For update view we access the first post—which has a pk of 1 which is passed in as the only argument—and we confirm that it results in a 302 redirect. Finally we test our delete view by confirming that if we delete a post the status code is 200 for success.

There's always more tests that can be added but this at least has coverage on all our new functionality.

Conclusion In a small amount of code we've built a blog application that allows for creating, reading, updating, and deleting blog posts. This core functionality is known by the acronym [CRUD: Create-Read-Update-Delete](#). While there are multiple ways to achieve this same functionality—we could have used function-based views or written our own class-based views—we've demonstrated how little code it takes in Django to make this happen.

In the next chapter we'll add user accounts and login, logout, and signup functionality.

Chapter 7: User Accounts

So far we've built a working blog application that uses forms, but we're missing a major piece of most web applications: user authentication.

Implementing proper user authentication is famously hard; there are many security gotchas along the way so you really don't want to implement this yourself. Fortunately Django comes with a powerful, built-in [user authentication system](#) that we can use.

Whenever you create a new project, by default Django installs the auth app, which provides us with a [User object](#) containing:

- username
- password
- email
- first_name
- last_name

We will use this User object to implement login, logout, and signup in our blog application.

Login Django provides us with a default view for a login page via [LoginView](#). All we need to add are a project-level urlpattern for the auth system, a login template, and a small update to our settings.py file.

First update the project-level urls.py file. We'll place our login and logout pages at the accounts/ URL. This is a one-line addition on the next-to-last line.

Code

```
# blog_project/urls.py
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('accounts/', include('django.contrib.auth.urls')),
    path('', include('blog.urls')),
]
```

As the [LoginView](#) documentation notes, by default Django will look within a templates folder called registration for a file called login.html for a login form. So we need to create a new directory called registration and the requisite file within it. From the command line type Control-c to quit our local server. Then enter the following:

Command Line

```
(blog) $ mkdir templates/registration  
(blog) $ touch templates/registration/login.html
```

Now type the following template code for our newly-created file.

Code

```
<!-- templates/registration/login.html -->  
{% extends 'base.html' %}  
  
{% block content %}  
<h2>Login</h2>  
<form method="post">  
  {% csrf_token %}  
  {{ form.as_p }}  
  <button type="submit">Login</button>  
</form>  
{% endblock content %}
```

We're using HTML <form></form> tags and specifying the POST method since we're sending data to the server (we'd use GET if we were requesting data, such as in a search engine form). We add {% csrf_token %} for security concerns, namely to prevent a XSS Attack. The form's contents are outputted between paragraph tags thanks to {{ form.as_p }} and then we add a "submit" button.

The final step is we need to specify *where* to redirect the user upon a successful login. We can set this with the LOGIN_REDIRECT_URL setting. At the bottom of the settings.py file add the following:

Code

```
# settings.py  
LOGIN_REDIRECT_URL = 'home'
```

Now the user will be redirected to the 'home' template which is our homepage.

We're actually done at this point! If you now start up the Django server again with python manage.py runserver and navigate to our login page:

<http://127.0.0.1:8000/accounts/login/>

You'll see the following:



Django blog

[+ New Blog Post](#)

Login

Username:

Password:

[Login page](#)

Upon entering the login info for our superuser account, we are redirected to the homepage. Notice that we didn't add any *view* logic or create a database model because the Django auth system provided both for us automatically. Thanks Django!

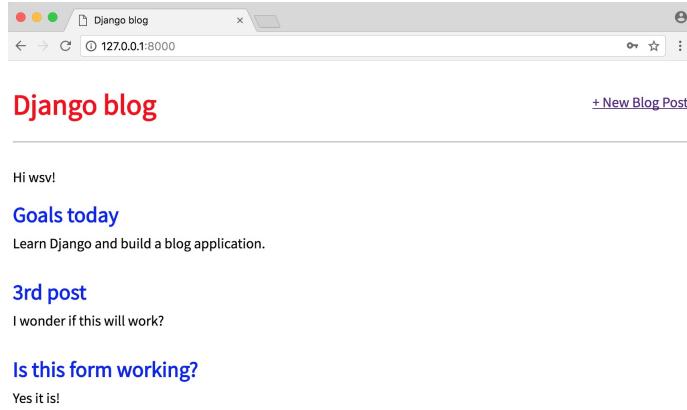
Updated homepage Let's update our `base.html` template so we display a message to users whether they are logged in or not. We can use the `is_authenticated` attribute for this.

For now, we can simply place this code in a prominent position. Later on we can style it more appropriately. Update the `base.html` file with new code starting beneath the closing `</header>` tag.

Code

```
<!-- templates/base.html -->
...
</header>
{% if user.is_authenticated %}
<p>Hi {{ user.username }}!</p>
{% else %}
<p>You are not logged in.</p>
<a href="{% url 'login' %}">login</a>
{% endif %}
{% block content %}
{% endblock content %}
```

If the user is logged in we say hello to them by name, if not we provide a link to our newly created login page.



Hi wsv!

Goals today
Learn Django and build a blog application.

3rd post
I wonder if this will work?

Is this form working?
Yes it is!

+ New Blog Post

Homepage logged in

It worked! My superuser name is wsv so that's what I see on the page.

Logout link We added template page logic for logged out users but...how do we log out now? We could go into the Admin panel and do it manually, but there's a better way. Let's add a logout link instead that redirects to the homepage. Thanks to the Django auth system, this is dead-simple to achieve.

In our `base.html` file add a one-line `{% url 'logout' %}` link for logging out.

Command Line

```
<!-- templates/base.html-->
...
{% if user.is_authenticated %}
<p>Hi {{ user.username }}!</p>
<p><a href="{% url 'logout' %}">logout</a></p>
{% else %}
...
```

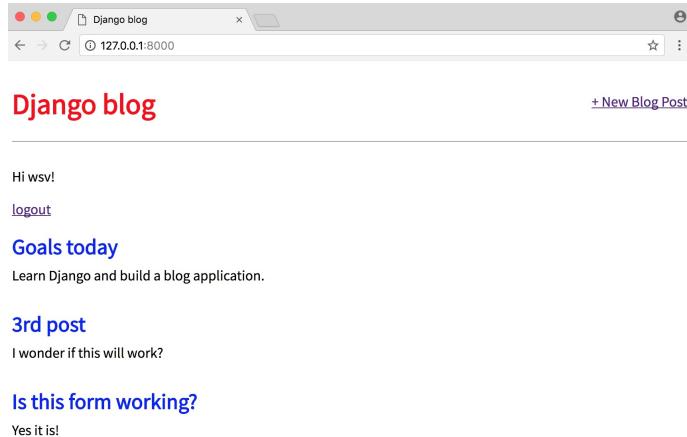
That's all we need to do as the necessary *view* is provided to us by the Django auth app. We do need to specify where to redirect a user upon logout though.

Update `settings.py` to provide a redirect link which is called, appropriately, `LOGOUT_REDIRECT_URL`. We can add it right next to our login redirect so the bottom of the file should look as follows:

Code

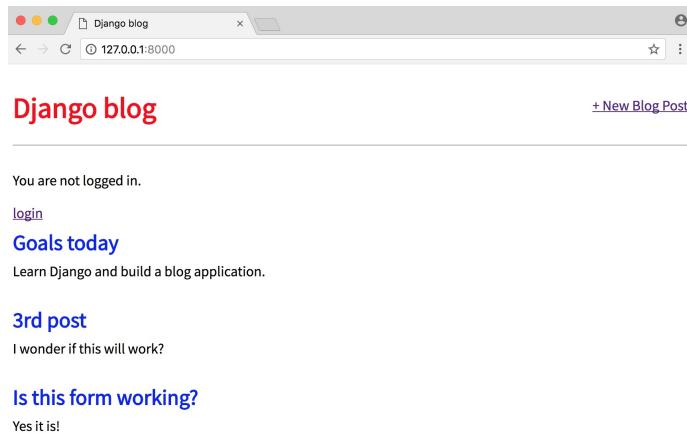
```
# blog_project/settings.py
LOGIN_REDIRECT_URL = 'home'
LOGOUT_REDIRECT_URL = 'home'
```

If you refresh the homepage you'll see it now has a "logout" link for logged in users.



Homepage logout link

And clicking it takes you back to the homepage with a "login" link.



Homepage logged out

Go ahead and try logging in and out several times with your user account.

Signup We need to write our own view for a signup page to register new users, but Django provides us with a form

class, [UserCreationForm](#), to make things easier. By default it comes with three fields: username, password1, and password2.

There are many ways to organize your code and url structure for a robust user authentication system. Here we will create a dedicated new app, accounts, for our signup page.

Command Line

```
(blog) $ python manage.py startapp accounts
```

Add the new app to the INSTALLED_APPS setting in our settings.py file.

Code

```
# blog_project/settings.py
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'blog',
    'accounts',
]
```

Next add a project-level url pointing to this new app directly **below** where we include the built-in auth app.

Code

```
# blog_project/urls.py
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('accounts/', include('django.contrib.auth.urls')),
    path('accounts/', include('accounts.urls')), # new
    path('', include('blog.urls')),
]
```

The order of our urls matters here because Django reads this file top-to-bottom. Therefore when we request them /accounts/signup url, Django will first look in auth, not find it, and **then** proceed to the accounts app.

Let's go ahead and create our accounts/urls.py file.

Command Line

```
(blog) $ touch accounts/urls.py
```

And add the following code:

Code

```
# accounts/urls.py
from django.urls import path

from . import views

urlpatterns = [
    path('signup/', views.SignUpView.as_view(), name='signup'),
]
```

We're using a not-yet-created view called `SignUpView` which we already know is class-based since it is capitalized and has the `as_view()` suffix. Its path is just `signup/` so the overall path will be `accounts/signup/`.

Now for the view which uses the built-in `UserCreationForm` and generic `CreateView`.

Code

```
# accounts/views.py
from django.contrib.auth.forms import UserCreationForm
from django.urls import reverse_lazy
from django.views import generic

class SignUpView(generic.CreateView):
    form_class = UserCreationForm
    success_url = reverse_lazy('login')
    template_name = 'signup.html'
```

We're subclassing the generic class-based view `CreateView` in our `SignUpView` class. We specify the use of the built-in `UserCreationForm` and the not-yet-created template at `signup.html`. And we use `reverse_lazy` to redirect the user to the login page upon successful registration.

Why use `reverse_lazy` here instead of `reverse`? The reason is that for all generic class-based views the urls are not loaded when the file is imported, so we have to use the lazy form of `reverse` to load them later when they're available.

Now let's add `signup.html` to our project-level `templates` folder:

Command Line

```
(blog) $ touch templates/signup.html
```

Add then populate it with the code below.

Code

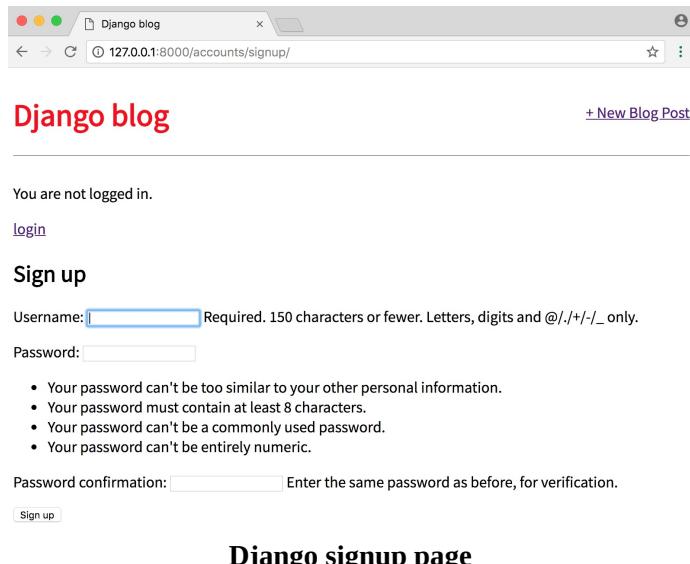
```
<!-- templates/signup.html -->
{% extends 'base.html' %}

{% block content %}
<h2>Sign up</h2>
<form method="post">
{% csrf_token %}
{{ form.as_p }}
<button type="submit">Sign up</button>
</form>
{% endblock %}
```

This format is very similar to what we've done before. We extend our base template at the top, place our logic between `<form></form>` tags, use the `csrf_token` for security, display the form's content in paragraph tags with `form.as_p`, and include a submit button.

We're now done! To test it out, navigate to our newly created page:

<http://127.0.0.1:8000/accounts/signup/>

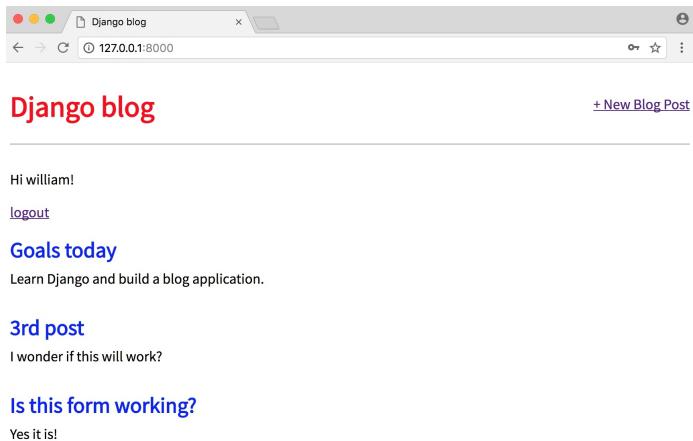


Django signup page

Notice there is a lot of extra text that Django includes by default. We can customize this using something like the built-in [messages framework](#) but for now try out the form.

I've created a new user called "william" and upon submission was redirected to the login page. Then after logging in successfully with my new user and password, I was redirected to the homepage with our personalized "Hi

username” greeting.



A screenshot of a web browser window titled "Django blog". The address bar shows "127.0.0.1:8000". The page content includes a greeting "Hi william!", a "logout" link, a section titled "Goals today" with the text "Learn Django and build a blog application.", a link titled "3rd post" with the text "I wonder if this will work?", and a link titled "Is this form working?" with the text "Yes it is!". There is also a link "+ New Blog Post" at the top right.

Homepage for user wsvincent

Our ultimate flow is therefore: Signup -> Login -> Homepage. And of course we can tweak this however we want. The SignupView redirects to login because we set `success_url = reverse_lazy('login')`. The Login page redirects to the homepage because in our `settings.py` file we set `LOGIN_REDIRECT_URL = 'home'`.

It can seem overwhelming at first to keep track of all the various parts of a Django project. That's normal. But I promise with time they'll start to make more sense.

Bitbucket It's been a while since we made a git commit. Let's do that and then push a copy of our code onto Bitbucket.

First check all the new work that we've done with `git status`.

Command Line

```
(blog) $ git status
```

Then add the new content.

Command Line

```
(blog) $ git commit -m 'forms and user accounts'
```

[Create a new repo](#) on Bitbucket which you can call anything you like. I'll choose

the name blog-app. Therefore *after creating the new repo on the Bitbucket site* I can type the following two commands. Make sure to replace my username wsvincent with yours from Bitbucket.

Command Line

```
(blog) $ git remote add origin git@bitbucket.org:wsvincent/blog-app.git  
(blog) $ git push -u origin master
```

All done! Now we can deploy our new app on Heroku.

Heroku config This is our third time deploying an app. As with our *Message Board* app, there are four changes we need to make so it can be deployed on Heroku.

- update Pipfile.lock
- new Procfile
- install gunicorn
- update settings.py

We'll specify a Python version in our Pipfile and then run pipenv lock to apply it to the Pipfile.lock. We'll add a Procfile which is a Heroku-specific configuration file, install gunicorn to run as our production web server in place of Django's local server, and finally update the ALLOWED_HOSTS so anyone can view our app.

Open the Pipfile with your text editor and at the bottom add the following two lines.

Code

```
# Pipfile  
[requires]  
python_version = "3.6"
```

We're using 3.6 here rather than the more specific 3.6.4 so that our app is automatically updated to the most recent version of Python 3.6x on Heroku.

Now run pipenv lock to update our Pipfile.lock since Heroku will use it to generate a new environment on Heroku servers for our app.

Command Line

```
(blog) $ pipenv lock
```

Create a new Procfile file.

Command Line

```
(blog) $ touch Procfile
```

Within your text editor add the following line to Procfile. This tells tells Heroku to use gunicorn rather than the local server which is not suitable for production.

Code

```
web: gunicorn blog_project.wsgi --log-file -
```

Now install [gunicorn](#).

Command Line

```
(blog) $ pipenv install gunicorn==19.8.1
```

Finally update ALLOWED_HOSTS to accept all domains, which is represented by the asterisk *.

Code

```
# blog_project/settings.py
ALLOWED_HOSTS = ['*']
```

We can commit our new changes and push them up to Bitbucket.

Command Line

```
(blog) $ git status
(blog) $ git add -A
(blog) $ git commit -m 'Heroku config files and updates'
(blog) $ git push -u origin master
```

Heroku deployment To deploy on Heroku first confirm that you're logged in to your existing Heroku account.

Command Line

```
(blog) $ heroku login
```

Then run the create command which tells Heroku to make a new container for our app to live in. If you just run heroku create then Heroku will assign you a random name, however you can specify a custom name but it must be *unique on Heroku*. In other words, since I'm picking the name dfb-blog you can't. You need some other combination of letters and numbers.

Command Line

```
(blog) $ heroku create dfb-blog
```

Now configure git so that when you push to Heroku, it goes to your new app name (replacing dfb-blog with your custom name).

Command Line

```
(blog) $ heroku git:remote -a dfb-blog
```

There's one more step we need to take now that we have static files, which in our case is CSS. Django does not support serving static files in production however the [WhiteNoise](#) project does. So let's install it.

Command Line

```
(blog) $ pipenv install whitenoise==3.3.1
```

Then we need to update our static settings so it will be used in production. In your text editor open `settings.py`. Add whitenoise to the `INSTALLED_APPS` **above** the built-in `staticfiles` app and also to `MIDDLEWARE` on the third line. Order matters for both `INSTALLED_APPS` and `MIDDLEWARE`.

At the bottom of the file add new lines for both `STATIC_ROOT` and `STATICFILES_STORAGE`. It should look like the following.

Code

```
# blog_project/settings.py
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'whitenoise.runserver_nostatic', # new!
    'django.contrib.staticfiles',
    'blog',
    'accounts',
]

MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'whitenoise.middleware.WhiteNoiseMiddleware', # new!
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
]

...
```

```
STATIC_ROOT = os.path.join(BASE_DIR, 'staticfiles') # new!
STATIC_URL = '/static/'
STATICFILES_DIRS = [os.path.join(BASE_DIR, 'static')]
STATICFILES_STORAGE = 'whitenoise.storage.CompressedManifestStaticFilesStorage' # ne\
w!
```

Make sure to add and commit your new changes. Then push it to Bitbucket.

Command Line

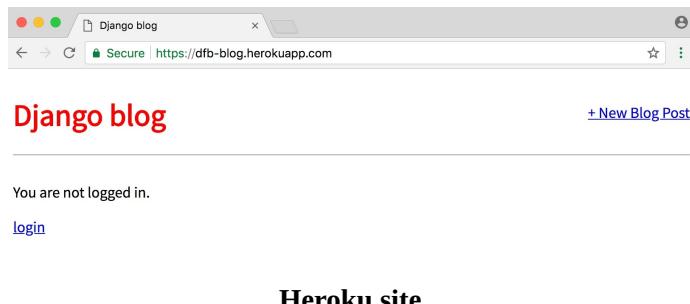
```
(blog) $ git add -A
(blog) $ git commit -m 'Heroku config'
(blog) $ git push origin master
```

Finally we can push our code to Heroku and add a web process so the dyno is running.

Command Line

```
(blog) $ git push heroku master
(blog) $ heroku ps:scale web=1
```

The URL of your new app will be in the command line output or you can run `heroku open` to find it. Mine is located at <https://dfb-blog.herokuapp.com/>.



Heroku site

Conclusion With a minimal amount of code, the Django framework has allowed us to create a login, logout, and signup user authentication flow. Under-the-hood it has taken care of the many security gotchas that can crop up if you try to create your own user authentication flow from scratch.

Chapter 8: Custom User Model

Django's built-in [User model](#) allows us to start working with users right away, as we just did with our *Blog app* in the previous chapters. However the [official Django documentation](#) *highly recommends* using a custom user model for new projects. The reason is that if you want to make any changes to the User model down the road—for example adding an age field—using a custom user model from the beginning makes this quite easy. But if you do not create a custom user model, updating the default User model in an existing Django project is very, very challenging.

So **always use a custom user model** for all new Django projects. However the official documentation [example](#) is not actually what many Django experts recommend. It uses the quite complex `AbstractBaseUser` when if we just use `AbstractUser` things are far simpler and still customizable. This is the approach we will take in this chapter where we start a new *Newspaper* app properly with a custom user model. The choice of a newspaper app pays homage to Django's roots as a web framework built for editors and journalists at the Lawrence Journal-World.

Setup The first step is to create a new Django project from the command line. We need to do several things:

- create and navigate into a new directory for our code
- create a new virtual environment `news`
- install Django
- make a new Django project `newspaper_project`
- make a new app `users`

We're calling our app for managing users `users` here but you'll also see it frequently called `accounts` in open source code. The actual name doesn't matter as long as you are consistent when referring to it throughout the project.

Here are the commands to run:

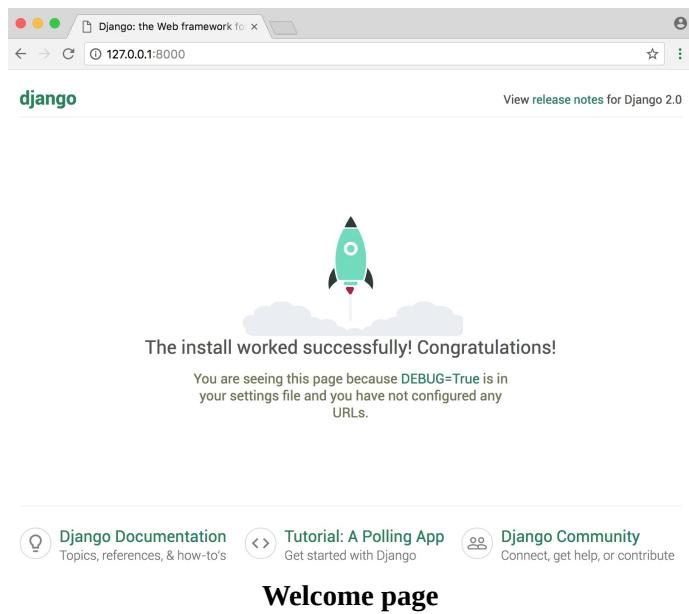
Command Line

```
$ cd ~/Desktop  
$ mkdir news
```

```
$ cd news
$ pipenv install django==2.0.6
$ pipenv shell
(news) $ django-admin startproject newspaper_project .
(news) $ python manage.py startapp users
(news) $ python manage.py runserver
```

Note that we **did not** run `migrate` to configure our database. It's important to wait until **after** we've created our new custom user model before doing so given how tightly connected the user model is to the rest of Django.

If you navigate to <http://127.0.0.1:8000> you'll see the familiar Django welcome screen.



Custom User Model Creating our custom user model requires four steps:

- update `settings.py`
- create a new `CustomUser` model
- create new forms for `UserCreation` and `UserChangeForm`
- update the admin

In `settings.py` we'll add the `users` app to our `INSTALLED_APPS`. Then at the bottom of the file use the `AUTH_USER_MODEL` config to tell Django to use our new custom user model in place of the built-in `User` model. We'll call our custom

user model `CustomUser` so, since it exists within our `users` app we refer to it as `users.CustomUser`.

Code

```
# newspaper_project/settings.py
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'users', # new
]
...
AUTH_USER_MODEL = 'users.CustomUser'
```

Now update `users/models.py` with a new User model which we'll call `CustomUser`. We'll also add our first extra field for the “age” of our users. We can use Django's [PositiveIntegerField](#) which means the integer must be either positive or zero.

Code

```
# users/models.py
from django.contrib.auth.models import AbstractUser
from django.db import models

class CustomUser(AbstractUser):
    age = models.PositiveIntegerField(default=0)
```

That's really all the code we need! Since we're extending `AbstractUser` our `CustomUser` is basically a copy of the default `User` model. The only update is our new `age` field.

Forms If we step back for a moment, what are the two ways in which we would interact with our new `CustomUser` model? One case is when a user signs up for a new account on our website. The other is within the `admin` app which allows us, as superusers, to modify existing users. So we'll need to update the two built-in forms for this functionality: [UserCreationForm](#) and [UserChangeForm](#).

Stop the local server with `Control+c` and create a new file in the `users` app called `forms.py`.

Command Line

```
(news) $ touch users/forms.py
```

We'll update it with the following code to extend the existing `UserCreationForm` and `UserChangeForm` forms.

Code

```
# users/forms.py
from django import forms
from django.contrib.auth.forms import UserCreationForm, UserChangeForm
from .models import CustomUser

class CustomUserCreationForm(UserCreationForm):

    class Meta(UserCreationForm.Meta):
        model = CustomUser
        fields = UserCreationForm.Meta.fields

class CustomUserChangeForm(UserChangeForm):

    class Meta:
        model = CustomUser
        fields = UserChangeForm.Meta.fields
```

For both new forms we are setting the `model` to our `CustomUser` and using the default fields by using `Meta.fields`.

The concept of fields on a form can be confusing at first so let's take a moment to explore it further. Our `CustomUser` model contains all the fields of the default `User` model **and** our additional `age` field which we set. But what are these default fields? It turns out there [are many](#) including `username`, `first_name`, `last_name`, `email`, `password`, `groups`, and more. Yet when a user signs up for a new account on Django the default form only asks for a `username`, `email`, and `password`. This tells us that the default setting for fields on `UserCreationForm` is just `username`, `email`, and `password` even though there are many more fields available.

This is might not click for you since understanding forms and models properly takes some time. In the next chapter we will create our own `signup`, `login`, and `logout` pages which will tie together our `CustomUser` model and forms more clearly. So hang tight!

The final step is to update our `admin.py` file since Admin is tightly coupled to the default `User` model. We will extend the existing [UserAdmin](#) class to use our new `CustomUser` model and our two new forms.

Code

```
# users/admin.py
from django.contrib import admin
from django.contrib.auth.admin import UserAdmin

from .forms import CustomUserCreationForm, CustomUserChangeForm
from .models import CustomUser

class CustomUserAdmin(UserAdmin):
    add_form = CustomUserCreationForm
    form = CustomUserChangeForm
    list_display = ['email', 'username', 'age']
    model = CustomUser

admin.site.register(CustomUser, CustomUserAdmin)
```

Note that our `CustomUserAdmin` also has a `list_display` setting so that it only displays the `email`, `username`, and `age` fields even though there are in fact many more on the `CustomUser` model at this point.

Ok we're done! Go ahead and run `makemigrations` and `migrate` for the first time to create a new database that uses the custom user model.

Command Line

```
(news) $ python manage.py makemigrations
(news) $ python manage.py migrate
```

Superuser Let's create a superuser account to confirm that everything is working as expected.

On the command line type the following command and go through the prompts.

Command Line

```
(news) $ python manage.py createsuperuser
```

The fact that this works is the first proof our custom user model works as expected. Let's view things in the admin too to be extra sure.

Start up the web server.

Command Line

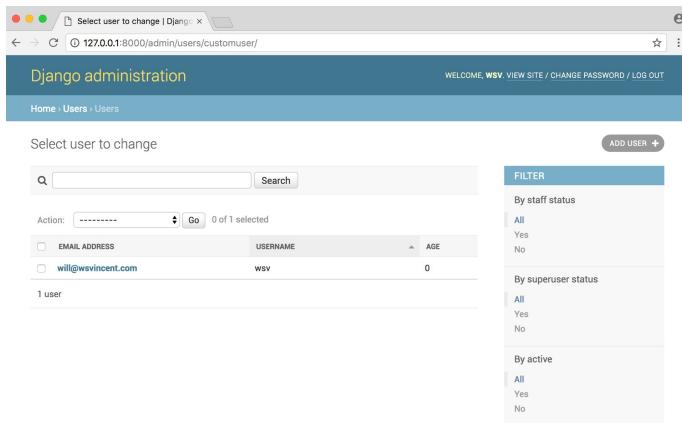
```
(news) $ python manage.py runserver
```

Then navigate to the admin at <http://127.0.0.1:8000/admin> and log in.



Admin page

If you click on the link for “Users” you should see your superuser account. Note that the three fields we see are email, username, and age since we set that as the `list_display` in our `CustomUserAdmin`.

A screenshot of the "Select user to change" page in the Django admin. The URL is "127.0.0.1:8000/admin/users/customuser/". The main area shows a table with one row for "wsv". The columns are "EMAIL ADDRESS" (will@wsvincent.com), "USERNAME" (wsv), and "AGE" (0). Above the table is a search bar and an "ADD USER" button. To the right is a "FILTER" sidebar with three sections: "By staff status" (All, Yes, No), "By superuser status" (All, Yes, No), and "By active" (All, Yes, No).

Admin one user

Conclusion With our custom user model complete, we can now focus on building out the rest of our *Newspaper* app. In the next chapter we will configure and customize signup, login, and logout pages.

Chapter 9: User Authentication

Now that we have a working custom user model we can add the functionality every website needs: the ability to signup, login, and logout users. Django provides everything we need for login and logout but we will need to create our own form to sign up new users. We'll also build a basic homepage with links to all three features so we don't have to type in the URLs by hand every time.

Templates By default the Django template loader looks for templates in a nested structure within each app. So a `home.html` template in `users` would need to be located at `users/templates/users/home.html`. But a project-level templates folder approach is cleaner and scales better so that's what we'll use.

Let's create a new `templates` directory and within it a `registration` folder as that's where Django will look for the login template.

Command Line

```
(news) $ mkdir templates  
(news) $ mkdir templates/registration
```

Now we need to tell Django about this new directory by updating the configuration for '`DIRS`' in `settings.py`. This is a one-line change.

Code

```
# newspaper_project/settings.py  
TEMPLATES = [  
{  
    ...  
    'DIRS': ['templates'],  
    ...  
}]
```

If you think about what happens when you login or logout of a site, you are immediately redirected to a subsequent page. We need to tell Django where to send users in each case. The `LOGIN_REDIRECT_URL` and `LOGOUT_REDIRECT_URL` settings do that. We'll configure both to redirect to our homepage which will have the named URL of '`home`'.

Remember that when we create our URL routes we have the option to add a name to each one. So when we make the homepage URL we'll make sure call it 'home'.

Add these two lines at the bottom of the `settings.py` file.

Code

```
# newspaper_project/settings.py
LOGIN_REDIRECT_URL = 'home'
LOGOUT_REDIRECT_URL = 'home'
```

Now we can create four new templates:

Command Line

```
(news) $ touch templates/registration/login.html
(news) $ touch templates/base.html
(news) $ touch templates/home.html
(news) $ touch templates/signup.html
```

Here's the HTML code for each file to use. The `base.html` will be inherited by every other template in our project. By using a block like `{% block content %}` we can later override the content *just in this place* in other templates.

Code

```
<!-- templates/base.html -->
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Newspaper App</title>
</head>
<body>
<main>
  {% block content %}
  {% endblock %}
</main>
</body>
</html>
```

Code

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

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

{% block content %}
{% if user.is_authenticated %}
Hi {{ user.username }}!
<p><a href="{% url 'logout' %}">logout</a></p>
{% else %}
<p>You are not logged in</p>
<a href="{% url 'login' %}">login</a> |
```

```
<a href="#">% url 'signup' %

---


```

Code

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

{%
  block title %}Login{%
  endblock %

}

{%
  block content %
}
<h2>Login</h2>
<form method="post">
{%
  csrf_token %
}
{{ form.as_p }}
<button type="submit">Login</button>
</form>
{%
  endblock %
}
```

Code

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

{%
  block title %}Sign Up{%
  endblock %

}

{%
  block content %
}
<h2>Sign up</h2>
<form method="post">
{%
  csrf_token %
}
{{ form.as_p }}
<button type="submit">Sign up</button>
</form>
{%
  endblock %
}
```

Our templates are now all set. Still to go are our urls and views.

URLs Let's start with the url routes. In our project-level urls.py file we want to have our home.html template appear as the homepage. But we don't want to build a dedicated pages app just yet, so we can use the shortcut of importing TemplateView and setting the template_name right in our url pattern.

Next we want to “include” both the users app and the built-in auth app. The reason is that the built-in auth app already provides views and urls for login and logout. But for signup we will need to create our own view and url. To ensure that our URL routes are consistent we place them *both* at users/ so the eventual URLs will be /users/login, /users/logout, and /users/signup.

Code

```
# newspaper_project/urls.py
from django.contrib import admin
from django.urls import path, include
from django.views.generic.base import TemplateView

urlpatterns = [
    path('', TemplateView.as_view(template_name='home.html'), name='home'),
    path('admin/', admin.site.urls),
    path('users/', include('users.urls')),
    path('users/', include('django.contrib.auth.urls')),
]
```

Now create a `urls.py` file in the `users` app.

Command Line

```
(news) $ touch users/urls.py
```

Update `users/urls.py` with the following code:

Code

```
# users/urls.py
from django.urls import path
from . import views

urlpatterns = [
    path('signup/', views.SignUp.as_view(), name='signup'),
]
```

The last step is our `views.py` file which will contain the logic for our signup form. We're using Django's generic `CreateView` here and telling it to use our `CustomUserCreationForm`, to redirect to login once a user signs up successfully, and that our template is named `signup.html`.

Code

```
# users/views.py
from django.urls import reverse_lazy
from django.views import generic

from .forms import CustomUserCreationForm

class SignUp(generic.CreateView):
    form_class = CustomUserCreationForm
    success_url = reverse_lazy('login')
    template_name = 'signup.html'
```

Ok, phew! We're done. Let's test things out.

Start up the server with `python manage.py runserver` and go to the homepage

at <http://127.0.0.1:8000/>.



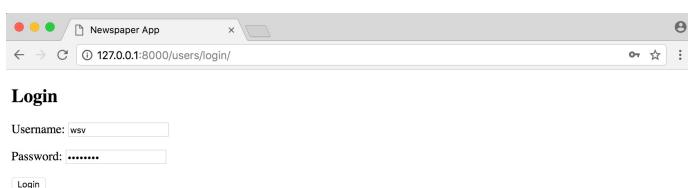
Homepage logged in

We logged in to the admin in the previous chapter so you should see a personalized greeting here. Click on the *logout* link.



Homepage logged out

Now we're on the logged out homepage. Go ahead and click on *login* link and use your **superuser** credentials.



Login

Upon successfully logging in you'll be redirected back to the homepage and see the same personalized greeting. It works!



Homepage logged in

Now use the *logout* link to return to the homepage and this time click on the *signup* link.



Homepage logged out

You'll be redirected to our signup page.

Sign up

Username: Required. 150 characters or fewer. Letters, digits and @/./+/-/_ only.

Password:

- Your password can't be too similar to your other personal information.
- Your password must contain at least 8 characters.
- Your password can't be a commonly used password.
- Your password can't be entirely numeric.

Password confirmation: Enter the same password as before, for verification.

[Sign up](#)

Signup page

Create a new user. Mine is called testuser. After successfully submitting the form you'll be redirected to the login page. Login with your new user and you'll again be redirected to the homepage with a personalized greeting for the new user.

Hi testuser!

[logout](#)

Homepage for testuser

Everything works as expected.

Admin Let's also log in to the admin to view our two user accounts. Navigate to: <http://127.0.0.1:8000/admin> and ...

Django administration

You are authenticated as testuser, but are not authorized to access this page. Would you like to login to a different account?

Username:

Password:

[Log in](#)

Admin login wrong

What's this! Why can't we log in?

Well we're logged in with our new testuser account not our superuser account. Only a superuser account has the permissions to log in to the admin! So use your superuser account to log in instead.

After you've done that you should see the normal admin homepage. Click on Users and you can see our two users: the one we just created and your previous superuser name (mine is wsv).

The screenshot shows the Django Admin 'Users' list page. At the top, there's a search bar and a 'FILTER' sidebar with options for staff status (All, Yes, No), superuser status (All, Yes, No), and active status (All, Yes, No). Below the sidebar, a table lists two users: 'testuser' and 'wsv'. Each user row has a checkbox next to it. The 'testuser' row also includes columns for 'EMAIL ADDRESS' (will@wsvincent.com) and 'AGE' (0). The 'wsv' row also includes 'EMAIL ADDRESS' (will@wsvincent.com) and 'AGE' (0). The table has a header row with columns for 'EMAIL ADDRESS', 'USERNAME', and 'AGE'.

Users in the Admin

Everything is working but you may notice that there is no email field for our testuser. Why is that? Well, look back at the signup page at users/signup/ and you'll see that it only asks for a username and password, not an email! This is just how the default Django setting works. However we can easily change it. Let's return to our users/forms.py file.

Currently it looks like this:

Code

```
# users/forms.py
from django import forms
from django.contrib.auth.forms import UserCreationForm,
UserChangeForm
from .models import CustomUser

class CustomUserCreationForm(UserCreationForm):

    class Meta(UserCreationForm.Meta):
        model = CustomUser
        fields = UserCreationForm.Meta.fields
```

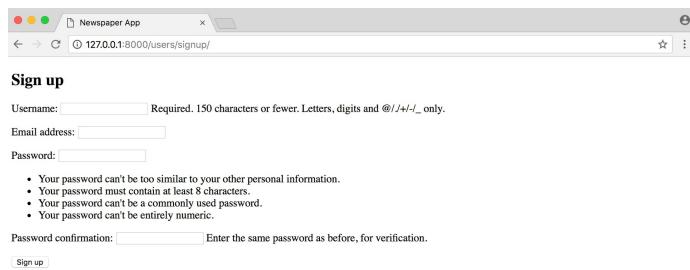
```
class CustomUserChangeForm(UserChangeForm):  
  
    class Meta:  
        model = CustomUser  
        fields = UserChangeForm.Meta.fields
```

Under `fields` we're using `Meta.fields` which just displays the default settings of username/password. But we can also explicitly set which fields we want displayed so let's update it to ask for a username/email/password by setting it to `('username', 'email',)`. We don't need to include the password field because it's required! But all the other fields can be configured however we choose.

Code

```
# users/forms.py  
...  
  
class CustomUserCreationForm(UserCreationForm):  
  
    class Meta(UserCreationForm.Meta):  
        model = CustomUser  
        fields = ('username', 'email', ) # new  
  
  
class CustomUserChangeForm(UserChangeForm):  
  
    class Meta:  
        model = CustomUser  
        fields = ('username', 'email', ) # new
```

Now if you try out the signup page again at <http://127.0.0.1:8000/users/signup/> you can see the additional “Email address” field is there.



New sign up page

Sign up with a new user account. I've named mine `testuser2` with an email address of `testuser2@email.com`. If we then switch back to the admin page—and login using our superuser account to do so—the three users are now evident.

The screenshot shows the Django Admin interface for a custom user model. The URL is 127.0.0.1:8000/admin/users/customuser/. The page title is "Select user to change | Django". The main content area shows a table with three rows of user data:

EMAIL ADDRESS	USERNAME	AGE
testuser	testuser	0
testuser2@email.com	testuser2	0
will@wsvincent.com	wsv	0

Below the table, it says "3 users". To the right of the table are three filter panels:

- FILTER**
 - By staff status
 - All
 - Yes
 - No
 - By superuser status
 - All
 - Yes
 - No
 - By active
 - All
 - Yes
 - No

Three users in the Admin

Django's user authentication flow requires a little bit of setup but you should be starting to see that it also provides us incredible flexibility to configure signup and log in *exactly* how we want.

Conclusion So far our *Newspaper* app has a custom user model and working sign up, login, and logout pages. But you may have noticed our site doesn't look very good. In the next chapter we'll add [Bootstrap](#) for styling and create a dedicated pages app.

Chapter 10: Bootstrap

Web development requires a lot of skills. Not only do you have to program the website to work correctly, users expect it to look good, too. When you're creating everything from scratch, it can be overwhelming to also add all the necessary HTML/CSS for a beautiful site.

Fortunately there's [Bootstrap](#), the most popular framework for building responsive, mobile-first projects. Rather than write all our own CSS and JavaScript for common website layout features, we can instead rely on Bootstrap to do the heavy lifting. This means with only a small amount of code on our part we can quickly have great looking websites. And if we want to make custom changes as a project progresses, it's easy to override Bootstrap where needed, too.

When you want to focus on the functionality of a project and not the design, Bootstrap is a great choice. That's why we'll use it here.

Pages app In the previous chapter we displayed our homepage by including view logic in our `urls.py` file. While this approach works, it feels somewhat hackish to me and it certainly doesn't scale as a website grows over time. It is also probably somewhat confusing to Django newcomers. Instead we can and should create a dedicated pages app for all our static pages. This will keep our code nice and organized going forward.

On the command line use the `startapp` command to create our new pages app. If the server is still running you may need to type `Control+c` first to quit it.

Command Line

```
(news) $ python manage.py startapp pages
```

Then immediately update our `settings.py` file. I often forget to do this so it is a good practice to just think of creating a new app as a two-step process: run the `startapp` command then update `INSTALLED_APPS`.

Code

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

Now we can update our project-level `urls.py` file. Go ahead and remove the import of `TemplateView`. We will also update the '' route to include the `pages` app.

Code

```
# newspaper_project/urls.py
from django.contrib import admin
from django.urls import path, include

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

It's time to add our homepage which means Django's standard urls/views/templates dance. We'll start with the `pages/urls.py` file. First create it.

Command Line

```
(news) $ touch pages/urls.py
```

Then import our not-yet-created views, set the route paths, and make sure to name each url, too.

Code

```
# pages/urls.py
from django.urls import path

from . import views

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

The `views.py` code should look familiar at this point. We're using Django's

TemplateView generic class-based view which means we only need to specify our template_name to use it.

Code

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

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

We already have an existing home.html template. Let's confirm it still works as expected with our new url and view. Navigate to the homepage at <http://127.0.0.1:8000/> to confirm it remains unchanged.



It should show the name of your logged in superuser account which we used at the end of the last chapter.

Tests We've added new code and functionality which means it's time for tests. You can never have enough tests in your projects. Even though they take some upfront time to write, they always save you time down the road and give confidence as a project grows in complexity.

There are two ideal times to add tests: either before you write any code (test-driven-development) or immediately after you've added new functionality and it's clear in your mind.

Currently our project has four pages:

- home
- signup
- login
- logout

However we only need to test the first two. Login and logout are part of Django

and rely on internal views and url routes. They therefore already have test coverage. If we made substantial changes to them in the future, we *would* want to add tests for that. But as a general rule, you do not need to add tests for core Django functionality.

Since we have urls, templates, and views for each of our two new pages we'll add tests for each. Django's [SimpleTestCase](#) will suffice for testing the homepage but the signup page uses the database so we'll need to use [TestCase](#) too.

Here's what the code should look like in your pages/tests.py file.

Code

```
# pages/tests.py
from django.contrib.auth import get_user_model
from django.test import SimpleTestCase, TestCase
from django.urls import reverse

class HomePageTests(SimpleTestCase):

    def test_home_page_status_code(self):
        response = self.client.get('/')
        self.assertEqual(response.status_code, 200)

    def test_view_url_by_name(self):
        response = self.client.get(reverse('home'))
        self.assertEqual(response.status_code, 200)

    def test_view_uses_correct_template(self):
        response = self.client.get(reverse('home'))
        self.assertEqual(response.status_code, 200)
        self.assertTemplateUsed(response, 'home.html')

class SignupPageTests(TestCase):

    username = 'newuser'
    email = 'newuser@email.com'

    def test_signup_page_status_code(self):
        response = self.client.get('/users/signup/')
        self.assertEqual(response.status_code, 200)

    def test_view_url_by_name(self):
        response = self.client.get(reverse('signup'))
        self.assertEqual(response.status_code, 200)

    def test_view_uses_correct_template(self):
        response = self.client.get(reverse('signup'))
        self.assertEqual(response.status_code, 200)
        self.assertTemplateUsed(response, 'signup.html')

    def test_signup_form(self):
        new_user = get_user_model().objects.create_user(
            self.username, self.email)
        self.assertEqual(get_user_model().objects.all().count(), 1)
```

```
self.assertEqual(get_user_model().objects.all()  
[0].username, self.username)  
self.assertEqual(get_user_model().objects.all()  
[0].email, self.email)
```

On the top line we use [get_user_model\(\)](#) to reference our custom user model. Then for both pages we test three things:

- the page exists and returns a HTTP 200 status code
- the page uses the correct url name in the view
- the proper template is being used

Our signup page also has a form so we should test that, too. In the test `test_signup_form` we're verifying that when a username and email address are POSTed (sent to the database), they match what is stored on the `CustomUser` model.

Quit the local server with `control+c` and then run our tests to confirm everything passes.

Command Line

```
(news) $ python manage.py test
```

Bootstrap If you've never used Bootstrap before you're in for a real treat. It accomplishes so much in so little code.

There are two ways to add Bootstrap to a project: you can download all the files and serve them locally or rely on a Content Delivery Network (CDN). The second approach is simpler to implement provided you have a consistent internet connection so that's what we'll use here.

[Bootstrap comes with a starter template](#) that includes the basic files needed. Notably there are four that we incorporate:

- `Bootstrap.css`
- `jQuery.js`
- `Popper.js`
- `Bootstrap.js`

Here's what the updated `base.html` file should look like. Generally you should type all code examples yourself but as this is one is quite long, it's ok to copy

and paste here.

Code

```
<!-- templates/base.html -->
<!doctype html>
<html lang="en">
<head>
<!-- Required meta tags -->
<meta charset="utf-8">
<meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit\nt=no">

<!-- Bootstrap CSS -->
<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/css\bootstrap.min.css" integrity="sha384-Gn5384xqQ1aoWXA+058RXPxPg6fy4IWvTNh0E263XmFcJ1\SAwiGgFAW/dAiS6JXm" crossorigin="anonymous">

<title>Hello, world!</title>
</head>
<body>
<h1>Hello, world!</h1>

<!-- Optional JavaScript -->
<!-- jQuery first, then Popper.js, then Bootstrap JS -->
<script src="https://code.jquery.com/jquery-3.2.1.slim.min.js" integrity="sha384-KJ3o2DKtIkYIK3UENzm7KCKr/rE9/Qpg6aAZGJwFDMVNA/GpGFF93hXpG5KKN" crossorigin="anonymous"></script>
<script src="https://cdn.jsdelivr.net/npm/popper.js@1.12.9/dist/umd/popper.min.js" integrity="sha384-ApNbgh9B+Y1QKtv3Rn7W3mgPxhU9K/ScQsAP7hUibX39j7fakFPskvXusvfa0b4Q" crossorigin="anonymous"></script>
<script src="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/js/bootstrap.min.js" integrity="sha384-JZR6Spejh4U02d8j0t6vLEHfe/JQGiRRSQxSffWpi1MquVdAyjuar5+76PVCmY1" crossorigin="anonymous"></script>
</body>
</html>
```

If you start the server again with `python manage.py runserver` and refresh the homepage at <http://127.0.0.1:8000/> you'll see that only the font size has changed at the moment.

Let's add a navigation bar at the top of the page which contains our links for the homepage, login, logout, and signup. Notably we can use the `if/else` tags in the Django templating engine to add some basic logic. We want to show a "log in" and "sign up" button to users who are logged out, but a "log out" and "change password" button to users logged in.

Here's what the code looks like. Again, it's ok to copy/paste here since the focus of this book is on learning Django not HTML, CSS, and Bootstrap.

Code

```
<!-- templates/base.html -->
<!doctype html>
<html lang="en">
<head>
```

```

<!-- Required meta tags -->
<meta charset="utf-8">
<meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">

<!-- Bootstrap CSS -->
<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/css/bootstrap.min.css" integrity="sha384-Gn5384xqQ1aoWXA+058RXPxPg6fy4IWvTNh0E263XmFcJ1SAwiGgFAW/dAiS6JXm" crossorigin="anonymous">

<title>{% block title %}Newspaper App{% endblock title %}</title>
</head>
<body>
<nav class="navbar navbar-expand-md navbar-dark bg-dark mb-4">
<a class="navbar-brand" href="{% url 'home' %}">Newspaper</a>
<button class="navbar-toggler" type="button" data-toggle="collapse" data-target="#navbarCollapse" aria-controls="navbarCollapse" aria-expanded="false" aria-label="Toggle navigation">
<span class="navbar-toggler-icon"></span>
</button>
<div class="collapse navbar-collapse" id="navbarCollapse">
{%
  if user.is_authenticated %
}
<ul class="navbar-nav ml-auto">
<li class="nav-item">
<a class="nav-link dropdown-toggle" href="#" id="userMenu" data-toggle="dropdown" aria-haspopup="true" aria-expanded="false">
{{ user.username }}
</a>
<div class="dropdown-menu dropdown-menu-right" aria-labelledby="userMe">
<a class="dropdown-item" href="{% url 'password_change' %}">Change password</a>
<div class="dropdown-divider"></div>
<a class="dropdown-item" href="{% url 'logout' %}">Log out</a>
</div>
</li>
</ul>
{%
  else %
}
<form class="form-inline ml-auto">
<a href="{% url 'login' %}" class="btn btn-outline-secondary">Log in</a>
<a href="{% url 'signup' %}" class="btn btn-primary ml-2">Sign up</a>
</form>
{%
  endif %
}
</div>
</nav>
<div class="container">
{%
  block content %
}
{%
  endblock %
}
</div>

<!-- Optional JavaScript -->
<!-- jQuery first, then Popper.js, then Bootstrap JS -->
<script src="https://code.jquery.com/jquery-3.2.1.slim.min.js" integrity="sha384-KJ3o2DKtIkYYIK3UENm7KCkRr/rE9/Qpg6aAZGJwFDMVNA/GpGFF93hXpG5Kkn" crossorigin="anonymous"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.12.9/umd/popper.min.js" integrity="sha384-ApNbgh9B+Y1QKtv3Rn7W3mgPxhU9K/ScQsAP7hUibX39j7fakFPskvXusvfa0b4Q" crossorigin="anonymous"></script>
<script src="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/js/bootstrap.min.js" integrity="sha384-JZR6Spejh4U02d8j0t6vLEHfe/JQGiRRSQxSffWpi1MquVdAyjUar5+76PVCmY1" crossorigin="anonymous"></script>
</body>
</html>

```

If you refresh the homepage at <http://127.0.0.1:8000/> our new nav has magically appeared!



Homepage with Bootstrap nav logged in

Click on the username in the upper right hand corner—wsv in my case—to see the nice dropdown menu Bootstrap provides.



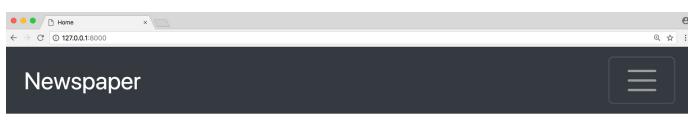
Homepage with Bootstrap nav logged in and dropdown

If you click on the “logout” link then our nav bar changes offering links to either “log in” or “sign up.”



Homepage with Bootstrap nav logged out

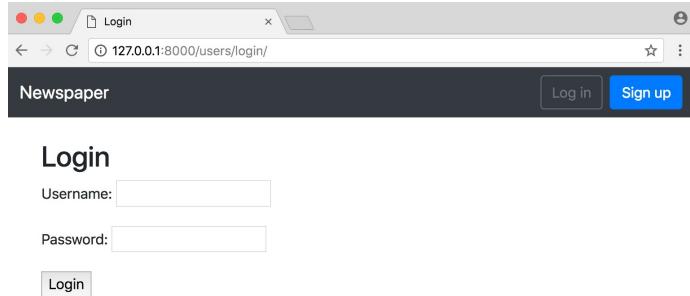
Better yet if you shrink the size of your browser window Bootstrap automatically resizes and makes adjustments so it looks good on a mobile device, too.



Homepage mobile with hamburger icon

Refresh the homepage and you’ll see it in action. You can even change the width of the web browser to see how the side margins change as the screen size increases and decreases.

If you click on the “logout” button and then “log in” from the top nav you can also see that our login page <http://127.0.0.1:8000/users/login> looks better too.



The screenshot shows a web browser window with a title bar "Login". The address bar shows the URL "127.0.0.1:8000/users/login/". The main content area has a dark header with the word "Newspaper" and two buttons: "Log in" and "Sign up". Below the header is a "Login" form with fields for "Username" and "Password", and a "Login" button at the bottom.

Bootstrap login

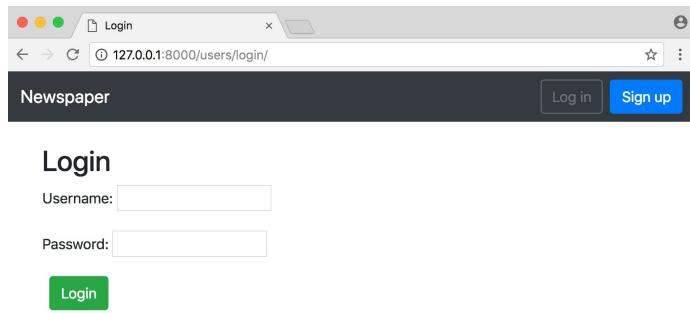
The only thing that looks off is our “Login” button. We can use Bootstrap to add some nice styling such as making it green and inviting.

Change the “button” line in `templates/registration/login.html` as follows.

Code

```
<!-- templates/registration/login.html -->
...
<button class="btn btn-success ml-2" type="submit">Login</button>
...
```

Now refresh the page to see our new button.



The screenshot shows the same web browser window and login form as the previous one, but the "Login" button has been styled with a green background and white text, matching the "Sign up" button above it.

Bootstrap login with new button

Signup Form Our signup page at <http://127.0.0.1:8000/users/signup/> has Bootstrap stylings but also distracting helper text. For example after “Username” it says “Required. 150 characters or fewer. Letters, digits and @/./+/-/_ only.”

!
Updated
navbar
logged
out

Where did that text come from, right? Whenever something feels like “magic” in Django rest assured that it is decidedly not. Likely the code came from an internal piece of Django.

The fastest method I’ve found to figure out what’s happening under-the-hood in Django is to simply go to the [Django source code on Github](#), use the search bar and try to find the specific piece of text.

For example, if you do a search for “150 characters or fewer” you’ll find yourself on the `django/contrib/auth/models.py` page [located here](#) on line 301. The text comes as part of the auth app, on the `username` field for `AbstractUser`.

We have three options now:

- override the existing `help_text`
- hide the `help_text`
- restyle the `help_text`

We’ll choose the third option since it’s a good way to introduce the excellent 3rd party package [django-crispy-forms](#). Working with forms is a challenge and `django-crispy-forms` makes it easier to write DRY code.

First stop the local server with `Control+c`. Then use Pipenv to install the package in our project.

Command Line

```
(news) $ pipenv install django-crispy-forms==1.7.2
```

Add the new app to our `INSTALLED_APPS` list in the `settings.py` file. As the number of apps starts to grow, I find it helpful to distinguish between 3rd party apps and local apps I’ve added myself. Here’s what the code looks like now.

Code

```
# newspaper_project/settings.py
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
```

```
'django.contrib.sessions',
'django.contrib.messages',
'django.contrib.staticfiles',

# 3rd Party
'crispy_forms',

# Local
'users',
'pages',
]
```

Since we're using Bootstrap4 we should also add that config to our `settings.py` file. This goes on the bottom of the file.

Code

```
# newspaper_project/settings.py
CRISPY_TEMPLATE_PACK = 'bootstrap4'
```

Now in our `signup.html` template we can quickly use crispy forms. First we load `crispy_forms_tags` at the top and then swap out `{{ form.as_p }}` for `{{ form|crispy }}`.

Code

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

{% load crispy_forms_tags %}

{% block title %}Sign Up{% endblock %}

{% block content %}
<h2>Sign up</h2>
<form method="post">
{% csrf_token %}
{{ form|crispy }}
<button type="submit">Sign up</button>
</form>
{% endblock %}
```

If you start up the server again with `python manage.py runserver` and refresh the signup page we can see the new changes.

Sign up

Username*

Email address

Password*

- Your password can't be too similar to your other personal information.
- Your password must contain at least 8 characters.
- Your password can't be a commonly used password.
- Your password can't be entirely numeric.

Password confirmation*

Enter the same password as before, for verification.

Sign up

Crispy signup page

Much better. Although how about if our “Sign up” button was a little more inviting? Maybe make it green? Bootstrap has [all sorts of button styling options](#) we can choose from. Let’s use the “success” one which has a green background and white text.

Update the `signup.html` file on the line for the sign up button.

Code

```
<!-- templates/signup.html -->
...
<button class="btn btn-success" type="submit">Sign up</button>
...
```

Refresh the page and you can see our updated work.

Sign up

Username*

Email address

Password*

- Your password can't be too similar to your other personal information.
- Your password must contain at least 8 characters.
- Your password can't be a commonly used password.
- Your password can't be entirely numeric.

Password confirmation*

Enter the same password as before, for verification.

Sign up

Crispy signup page green button

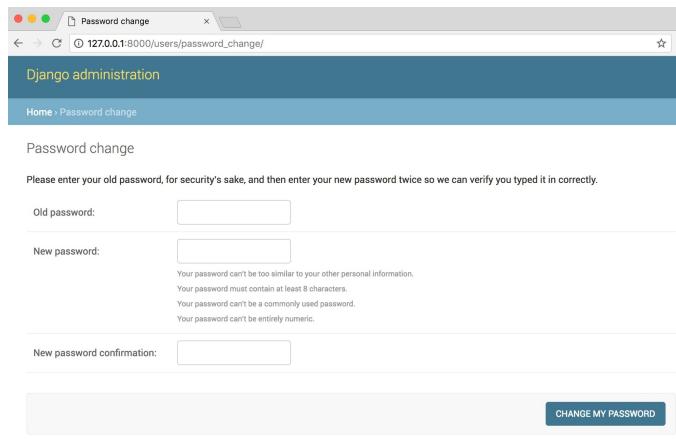
Next Steps Our *Newspaper* app is starting to look pretty good. The last step of our user auth flow is to configure password change and reset. Here again Django has taken care of the heavy lifting for us so it requires a minimal amount of code on our part.

Chapter 11: Password Change and Reset

In this chapter we will complete the authorization flow of our *Newspaper* app by adding password change and reset functionality. Users will be able to change their current password or, if they've forgotten it, to reset it via email.

Just as Django comes with built-in views and urls for login and logout, so too it also comes with views/urls for both password change and reset. We'll go through the default versions first and then learn how to customize them with our own Bootstrap-powered templates and email service.

Password Change Letting users change their passwords is a common feature on many websites. Django provides a default implementation that already works at this stage. To try it out first click on the “log in” button to make sure you're logged in. Then navigate to the “Password change” page at http://127.0.0.1:8000/users/password_change/.



The screenshot shows a web browser window with the title 'Password change'. The address bar displays the URL '127.0.0.1:8000/users/password_change/'. The page header reads 'Django administration' with a 'Home > Password change' link. The main content area is titled 'Password change' and contains instructions: 'Please enter your old password, for security's sake, and then enter your new password twice so we can verify you typed it in correctly.' Below this, there are four input fields: 'Old password:' (disabled), 'New password:' (disabled), and 'New password confirmation:' (disabled). Each field has a corresponding error message below it: 'Your password can't be too similar to your other personal information.', 'Your password must contain at least 8 characters.', 'Your password can't be a commonly used password.', and 'Your password can't be entirely numeric.'. At the bottom right is a large blue 'CHANGE MY PASSWORD' button.

Password change

Enter in both your old password and then a new one. Then click the “Change My Password” button.

You'll be redirected to the “Password change successful” page located at: http://127.0.0.1:8000/users/password_change/done/.



Password change done

Customizing password change Let's customize these two password change pages so that they match the look and feel of our *Newspaper* site. Because Django already has created the views and URLs for us, we only need to add new templates.

On the command line create two new template files in the registration folder.

Command Line

```
(news) $ touch templates/registration/password_change_form.html  
(news) $ touch templates/registration/password_change_done.html
```

Update `password_change_form.html` with the following code.

Code

```
<!-- templates/registration/password_change_form.html -->  
{% extends 'base.html' %}  
  
{% block title %}Password Change{% endblock %}  
  
{% block content %}  
<h1>Password change</h1>  
<p>Please enter your old password, for security's sake, and then enter your new pa\\  
ssword twice so we can verify you typed it in correctly.</p>  
  
<form method="POST">  
{% csrf_token %}  
{% form.as_p %}  
<input class="btn btn-success" type="submit" value="Change my password">  
</form>  
{% endblock %}
```

At the top we extend `base.html` and set our page title. Because we used “block” titles in our `base.html` file we can override them here. The form uses POST since we’re sending data and a `csrf_token` for security reasons. By using `form.as_p` we’re simply displaying in paragraphs the content of the default password reset form. And finally we include a submit button that uses Bootstrap’s `btn btn-success` styling to make it green.

Go ahead and refresh the page at http://127.0.0.1:8000/users/password_change/ to see our changes.

Old password:

New password:

- Your password can't be too similar to your other personal information.
- Your password must contain at least 8 characters.
- Your password can't be a commonly used password.
- Your password can't be entirely numeric.

New password confirmation:

Change my password

New password change form

Next up is the `password_change_done` template.

Code

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

{% block title %}Password Change Successful{% endblock %}

{% block content %}
<h1>Password change successful</h1>
<p>Your password was changed.</p>
{% endblock content %}
```

It also extends `base.html` and includes a new title. However there's no form on the page, just new text.

The new page is at http://127.0.0.1:8000/users/password_change/done/.

Password change successful

Your password was changed.

New password change done

That wasn't too bad, right? Certainly it was a lot less work than creating everything from scratch, especially all the code around securely updating a user's password.

Next up is our password reset functionality.

Password reset Password reset handles the common case of users forgetting their passwords. The steps are very similar to configuring password change, as we just did. Django already provides a default implementation that we will use and then customize the templates so it matches the rest of our site.

The only configuration required is telling Django **how** to send emails. After all, a user can only reset a password if they have access to the email linked to the account. In production we'll use the email service [SendGrid](#) to actually send the emails but for testing purposes we can rely on Django's [console backend](#) setting which outputs the email text to our command line console instead.

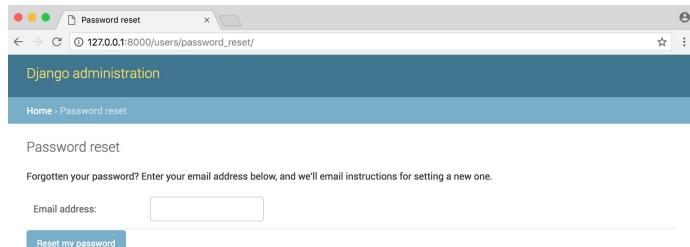
At the bottom of the `settings.py` file make the following one-line change.

Code

```
# newspaper_project/settings.py
EMAIL_BACKEND = 'django.core.mail.backends.console.EmailBackend'
```

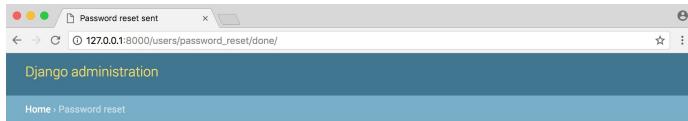
And we're all set! Django will take care of all the rest for us. Let's try it out.

Navigate to http://127.0.0.1:8000/users/password_reset/ to view the default password reset page.



Default password reset page

Make sure the email address you enter matches one of your user accounts. Upon submission you'll then be redirected to the password reset done page at: http://127.0.0.1:8000/users/password_reset/done/.



Default password reset done page

Which says to check our email. Since we've told Django to send emails to the command line console, the email text will now be there. This is what I see in my console.

Command Line

```
Content-Type: text/plain; charset="utf-8"
MIME-Version: 1.0
Content-Transfer-Encoding: 7bit
Subject: Password reset on 127.0.0.1:8000
From: webmaster@localhost
To: will@wsvincent.com
Date: Thu, 22 Mar 2018 20:31:48 -0000
Message-ID: <152175070807.39206.18266082938043731152@1.0.0.127.in-addr.arpa>
```

You're receiving this email because you requested a password reset for your user account at 127.0.0.1:8000.

Please go to the following page and choose a new password:

<http://127.0.0.1:8000/users/reset/MQ/4up-678712c114db2ead7780/>

Your username, in case you've forgotten: wsv

Thanks for using our site!

The 127.0.0.1:8000 team

Your email text should be identical except for three lines:

- the “To” on the sixth line contains the email address of the user
- the URL link contains a secure token that Django randomly generates for us and can be used only once
- Django helpfully reminds us of our username

We will customize all of the email default text shortly but for now focus on finding the link provided. In the message above mine is:

<http://127.0.0.1:8000/users/reset/MQ/4up-678712c114db2ead7780/>

Enter this link into your web browser and you'll be redirected to the “change password page”.

Enter new password

Please enter your new password twice so we can verify you typed it in correctly.

New password:

Confirm password:

[Change my password](#)

Default change password page

Now enter in a new password and click on the “Change my password” button. The final step is you’ll be redirected to the “Password reset complete” page.

Password reset complete

Your password has been set. You may go ahead and log in now.

[Log in](#)

Default password reset complete

To confirm everything worked, click on the “Log in” link and use your new password. It should work.

Custom Templates As with “Password change” we only need to create new templates to customize the look and feel of password reset.

Create four new template files.

Command Line

```
(news) $ touch templates/registration/password_reset_form.html
(news) $ touch templates/registration/password_reset_done.html
(news) $ touch templates/registration/password_reset_confirm.html
(news) $ touch templates/registration/password_reset_complete.html
```

Start with the password reset form which is `password_reset_form.html`.

Code

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

{% block title %}Forgot Your Password?{% endblock %}

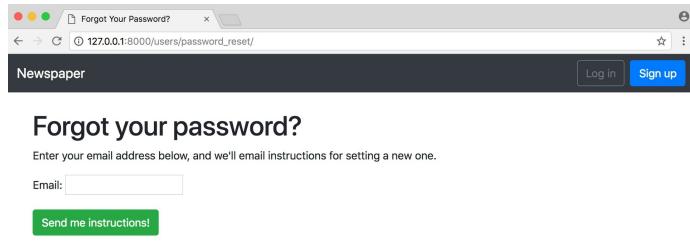
{% block content %}
<h1>Forgot your password?</h1>
```

```
<p>Enter your email address below, and we'll email instructions for setting a new on\\
e.</p>

<form method="POST">
{% csrf_token %}
{{ form.as_p }}
<input class="btn btn-success" type="submit" value="Send me instructions!">
</form>
{% endblock %}
```

At the top we extend `base.html` and set our page title. Because we used “block” titles in our `base.html` file we can override them here. The form uses POST since we’re sending data and a `csrf_token` for security reasons. By using `form.as_p` we’re simply displaying in paragraphs the content of the default password reset form. Finally we include a submit button and use Bootstrap’s `btn btn-success` styling to make it green.

If you navigate to http://127.0.0.1:8000/users/password_reset/ and refresh the page you can see our new page.



New password reset

Now we can update the other three pages. Each takes the same form of extending `base.html`, a new title, new content text, and for “password reset confirm” an updated form as well.

Code

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

{% block title %}Email Sent{% endblock %}

{% block content %}
<h1>Check your inbox.</h1>
<p>We've emailed you instructions for setting your password. You should receive th\\
e email shortly!</p>
{% endblock %}
```

Confirm the changes by going to
http://127.0.0.1:8000/users/password_reset/done/.



Check your inbox.

We've emailed you instructions for setting your password. You should receive the email shortly!

New reset done

Next the password reset confirm page.

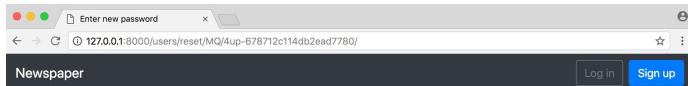
Code

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

{% block title %}Enter new password{% endblock %}

{% block content %}
<h1>Set a new password!</h1>
<form method="POST">
{% csrf_token %}
{{ form.as_p }}
<input class="btn btn-success" type="submit" value="Change my password">
</form>
{% endblock %}
```

In the command line grab the URL link from the email outputted to the console—mine was `http://127.0.0.1:8000/users/reset/MQ/4up-678712c114db2ead7780/`—and you'll see the following.



Set a new password!

[Change my password](#)

New set password

Finally here is the password reset complete code.

Code

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

{% block title %}Password reset complete{% endblock %}

{% block content %}
<h1>Password reset complete</h1>
<p>Your new password has been set. You can log in now on the <a href="{% url 'login'\%}">log in page</a>.</p>
{% endblock %}
```

You can view it at <http://127.0.0.1:8000/users/reset/done/>.



New password reset complete

Users can now reset their account password!

Conclusion In the next chapter we will connect our *Newspaper* app to the email service [SendGrid](#) to actually send our automated emails to users as opposed to outputting them in our command line console.

Chapter 12: Email

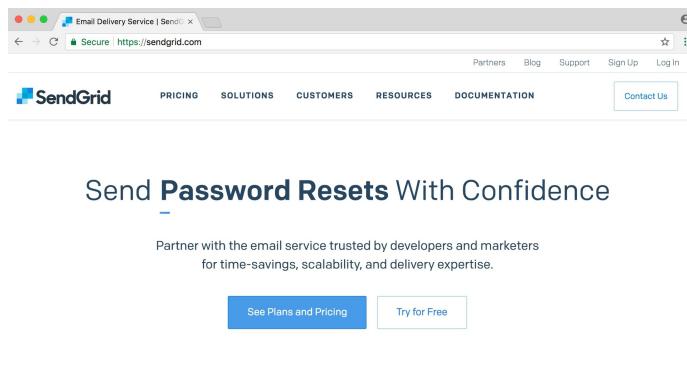
At this point you may be feeling a little overwhelmed by all the user authentication configuration we've done up to this point. That's normal. After all, we haven't even created any core *Newspaper* app features yet! Everything has been about setting up custom user accounts and the rest.

The upside to Django's approach is that it is incredibly easy to customize any piece of our website. The downside is Django requires a bit more out-of-the-box code than some competing web frameworks. As you become more and more experienced in web development, the wisdom of Django's approach will ring true.

Now we want to have our emails be actually sent to users, not just outputted to our command line console. We need to signup for an account at [SendGrid](#) and update our `settings.py` files. Django will take care of the rest. Ready?

SendGrid [SendGrid](#) is a popular service for sending transactional emails so we'll use it. Django doesn't care what service you choose though; you can just as easily use [MailGun](#) or any other service of your choice.

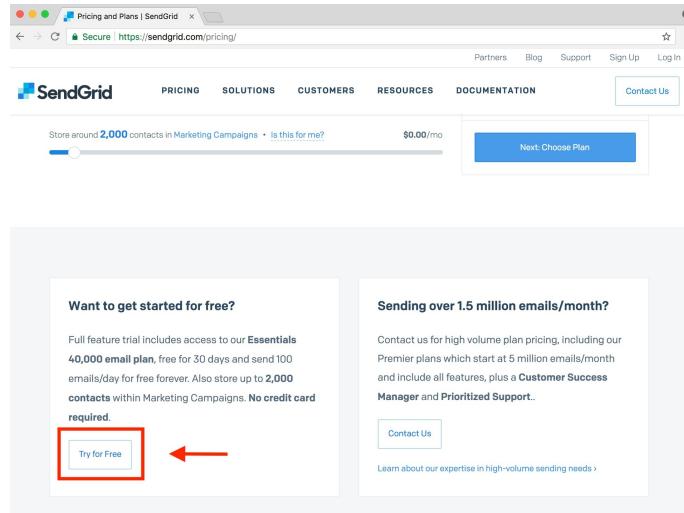
On the SendGrid homepage click on the large blue button for "See Plans and Pricing".



SendGrid homepage

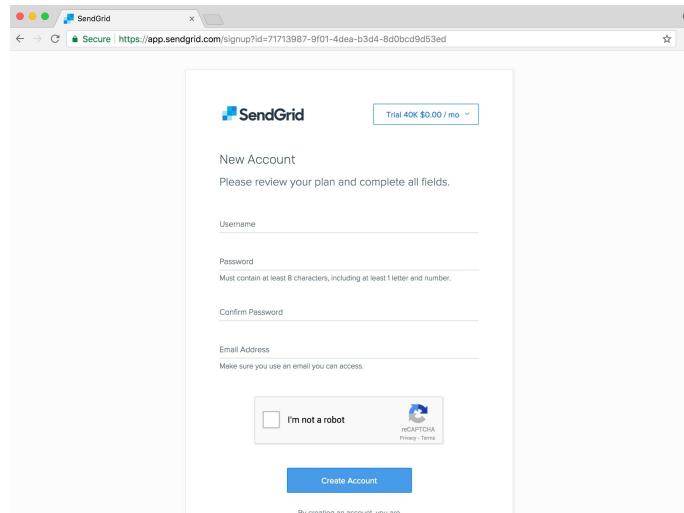
On the next page scroll down slightly and look on the left side for the "Try for Free" button. SendGrid provides a free tier we can use although they make it

somewhat difficult to find.



SendGrid pricing

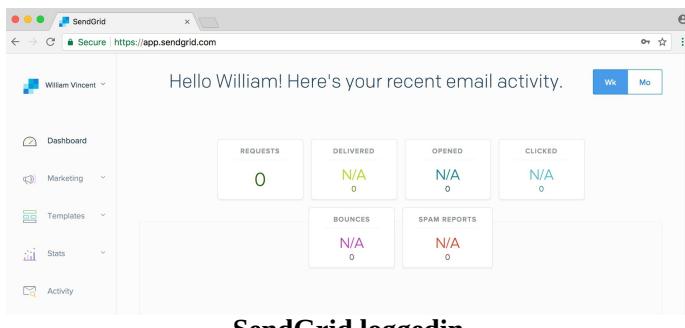
Sign up for your free account on the next page.



SendGrid new account

Make sure that the email account you use for SendGrid **is not** the same email account you have for your superuser account on the *Newspaper* project or there can be weird errors.

After confirming your new account via email—that's kinda meta, no?—you'll be asked to login and taken to your SendGrid dashboard page.



SendGrid loggedin

Now we can configure our Django code in the `settings.py` file. First we update the email backend to use SMTP.

Code

```
# newspaper_project/settings.py
EMAIL_BACKEND = 'django.core.mail.backends.smtp.EmailBackend'
```

Then right below it add the following five lines of email configuration. Note that ideally you should store secure information like your password in environment variables, but we won't here to keep things simple.

Code

```
# newspaper_project/settings.py
EMAIL_HOST = 'smtp.sendgrid.net'
EMAIL_HOST_USER = 'sendgrid_username'
EMAIL_HOST_PASSWORD = 'sendgrid_password'
EMAIL_PORT = 587
EMAIL_USE_TLS = True
```

Make sure to use enter your own SendGrid username for `EMAIL_HOST_USER` and password for `EMAIL_HOST_PASSWORD`.

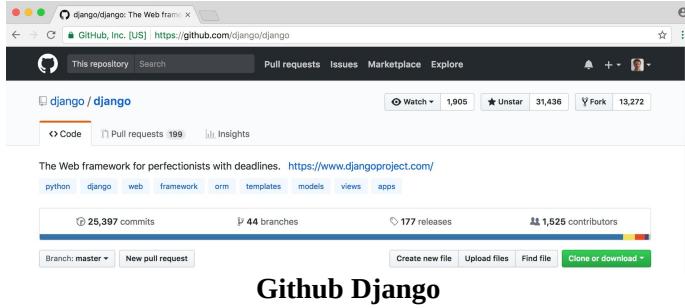
That's it. We're done! Navigate to the password reset form again at:

http://127.0.0.1:8000/users/password_reset/

You should receive an email in your inbox! The text will be exactly the same as that outputted to our command line console previously.

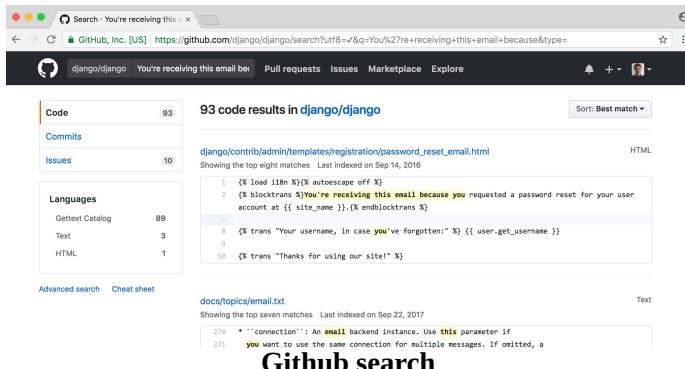
Custom emails The current email text isn't very personal, is it? Let's change things. At this point I could just show you what steps to take, but I think it's helpful if I can explain how I figured out how to do this. After all, you want to be able to customize all parts of Django as needed.

In this case, I knew what text Django was using by default but it wasn't clear where in the Django source code it was written. And since all of Django's source code is [available on Github](#) we can just search it.



Github Django

Use the Github search bar and enter a few words from the email text. If you type in "You're receiving this email because" you'll end up at this Github search page.



Github search

The first result is the one we want. It shows the code is located at `django/contrib/admin/templates/registration/password_reset_email.html`. That means in the `contrib` app the file we want is called `password_reset_email.html`.

Here is that default text from the Django source code.

Code

```
{% load i18n %}{% autoescape off %}
{% blocktrans %}You're receiving this email because you requested a password reset for your user account at {{ site_name }}.{% endblocktrans %}

{% trans "Please go to the following page and choose a new password:" %}
{% block reset_link %}
{{ protocol }}://{{ domain }}{% url 'password_reset_confirm' uidb64=uid token=token %}
{% endblock %}
{% trans "Your username, in case you've forgotten:" %} {{ user.get_username }}
```

```
{% trans "Thanks for using our site!" %}

{% blocktrans %}The {{ site_name }} team{% endblocktrans %}

{% endautoescape %}
```

Let's change it. We need to create a new `password_reset_email.html` file in our `registration` folder.

Command Line

```
(news) $ touch templates/registration/password_reset_email.html
```

Then use the following code which tweaks what Django provided by default.

Code

```
<!-- templates/registration/password_reset_email.html -->
{% load i18n %}{% autoescape off %}
{% trans "Hi" %} {{ user.get_username }},

{% trans "We've received a request to reset your password. If you didn't make this request, you can safely ignore this email. Otherwise, click the button below to reset your password." %}

{% block reset_link %}
{{ protocol }}://{{ domain }}{% url 'password_reset_confirm' uidb64=uid token=token %}
{% endblock %}
{% endautoescape %}
```

This code might look a little scary so let's break it down line-by-line. Up top we load the template tag [i18n](#) which means this text is eligible to be translated into multiple languages. Django has robust [internationalization support](#) though covering it is beyond the scope of this book.

We're greeting the user by name thanks to `user.get_username`. Then we use the built-in `reset_link` block to include the custom URL link. You can read more about Django's [password management approach](#) in the official docs.

Let's also update the email's subject title. To do this we'll create a new file `templates/registration/password_reset_subject.txt`.

Command Line

```
(news) $ touch templates/registration/password_reset_subject.txt
```

Then add the following line of code to the `password_reset_subject.txt` file.

```
Please reset your password
```

And we're all set. Go ahead and try out our new flow again by entering a new password at http://127.0.0.1:8000/users/password_reset/. Then check your email and it will have our new content and subject.

Conclusion We've now finished implementing a complete user authentication flow. Users can sign up for a new account, login, logout, change their password, and reset their password. It's time to build out our actual *Newspaper* app.

Chapter 13: Newspaper app

It's time to build out our *Newspaper* app. We'll have an articles page where journalists can post articles, set up permissions so only the author of an article can edit or delete it, and finally add the ability for other users to write comments on each article which will introduce the concept of foreign keys.

Articles app To start create an articles app and define our database models. There are no hard and fast rules around what to name your apps except that you can't use the name of a built-in app. If you look at the INSTALLED_APPS section of settings.py you can see which app names are off-limits: admin, auth, contenttypes, sessions, messages, and staticfiles. A general rule of thumb is to use the plural of an app name—posts, payments, users, etc.—unless doing so is obviously wrong as in the common case of blog where the singular makes more sense.

Start by creating our new articles app.

Command Line

```
(news) $ python manage.py startapp articles
```

Then add it to our INSTALLED_APPS and update the time zone since we'll be timestamping our articles. You can find your time zone in [this Wikipedia list](#). For example, I live in Boston, MA which is in the Eastern time zone of the United States. Therefore my entry is America/New_York.

Code

```
# newspaper_project/settings.py
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',

    # 3rd Party
    'crispy_forms',
```

```
# Local
'users',
'pages',
'articles', # new
]

TIME_ZONE = 'America/New_York'
```

Next up we define our database model which contains four fields: title, body, date, and author. Note that we're letting Django automatically set the time and date based on our TIME_ZONE setting. For the author field we want to [reference our custom user model](#) 'users.CustomUser' which we set in the settings.py file as AUTH_USER_MODEL.

We can do this via [get_user_model](#). And we also implement the best practices of defining a get_absolute_url from the beginning and a __str__ method for viewing the model in our admin interface.

Code

```
# articles/models.py
from django.conf import settings
from django.contrib.auth import get_user_model
from django.db import models
from django.urls import reverse

class Article(models.Model):
    title = models.CharField(max_length=255)
    body = models.TextField()
    date = models.DateTimeField(auto_now_add=True)
    author = models.ForeignKey(
        get_user_model(),
        on_delete=models.CASCADE,
    )

    def __str__(self):
        return self.title

    def get_absolute_url(self):
        return reverse('article_detail', args=[str(self.id)])
```

Since we have a brand new app and model, it's time to make a new migration file and then apply it to the database.

Command Line

```
(news) $ python manage.py makemigrations articles
(news) $ python manage.py migrate
```

At this point I like to jump into the admin to play around with the model before building out the urls/views/templates needed to actually display the data on the

website. But first we need to update `admin.py` so our new app is displayed.

Code

```
# articles/admin.py
from django.contrib import admin

from . import models

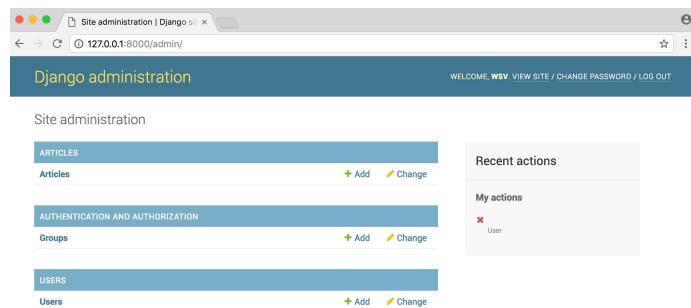
admin.site.register(models.Article)
```

Now we start the server.

Command Line

```
(news) $ python manage.py runserver
```

Navigate to <http://127.0.0.1:8000/admin/> and log in.



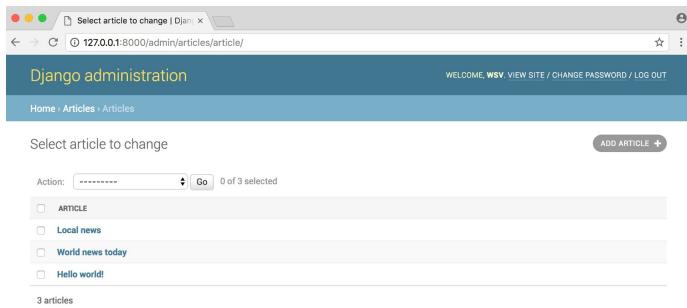
Admin page

If you click on “Articles” at the top of the page we can enter in some sample data. You’ll likely have three users available at this point: your superuser, testuser, and testuser2 accounts. Use your superuser account as the author of all three articles.

A screenshot of a web browser displaying the "Add article" form in the Django administration interface. The title bar says "Add article | Django site admin". The address bar shows the URL "127.0.0.1:8000/admin/articles/article/add/". The main content area has a dark blue header with the text "Django administration". Below the header, the breadcrumb navigation shows "Home > Articles > Articles > Add article". The form itself has fields for "Title" (containing "Hello world!") and "Body" (containing "This is my first article."). Below the body field is a "Author" dropdown menu with "wsv" selected. At the bottom of the form are three buttons: "Save and add another", "Save and continue editing", and a large blue "SAVE" button.

Admin articles add page

I've added three new articles as you can see on the updated Articles page.



Admin three articles

If you click on an individual article you will see that the title, body, and author are displayed but not the date. That's because the date was automatically added by Django for us and therefore can't be changed in the admin. We *could* make the date editable—in more complex apps it's common to have both a `created_at` and `updated_at` field—but to keep things simple we'll just have the date be set upon creation by Django for us for now. Even though date is not displayed here we will still be able to access it in our templates so it can be displayed on web pages.

URLs and Views The next step is to configure our URLs and views. Let's have our articles appear at `articles/`. Import `include` on the second line and add a URL pattern for articles in our project-level `urls.py` file.

Code

```
# newspaper_project/urls.py
from django.contrib import admin
from django.urls import path, include # new

urlpatterns = [
path('', include('pages.urls')),
path('articles/', include('articles.urls')), # new
path('admin/', admin.site.urls),
path('users/', include('users.urls')),
path('users/', include('django.contrib.auth.urls')),
]
```

Next we create an `articles/urls.py` file.

Command Line

```
(news) $ touch articles/urls.py
```

Then populate it with our routes. Let's start with the page to list all articles at `articles/` which will use the view `ArticleListView`.

Code

```
# articles/urls.py
from django.urls import path

from . import views

urlpatterns = [
    path('', views.ArticleListView.as_view(), name='article_list'),
]
```

Now create our view using the built-in generic `ListView` from Django.

Code

```
# articles/views.py
from django.views.generic import ListView

from . import models

class ArticleListView(ListView):
    model = models.Article
    template_name = 'article_list.html'
```

The only two fields we need to specify are the model `Article` and our template name which will be `article_list.html`.

The last step is to create our template. We can make an empty file from the command line.

Command Line

```
(news) $ touch templates/article_list.html
```

Bootstrap has a built-in component called [Cards](#) that we can customize for our individual articles. Recall that `ListView` returns an object called `object_list` which we can iterate over using a `for` loop.

Within each `article` we display the title, body, author, and date. We can even provide links to “edit” and “delete” functionality that we haven't built yet.

Code

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

{% block title %}Articles{% endblock %}

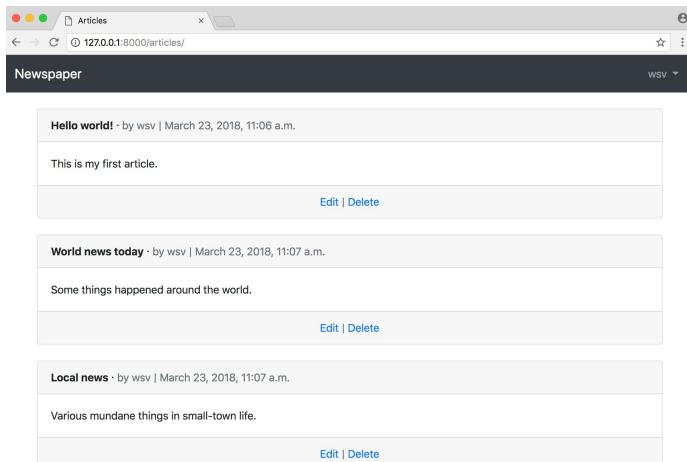
{% block content %}
    {% for article in object_list %}
```

```

<div class="card">
<div class="card-header">
<span class="font-weight-bold">{{ article.title }}</span> &middot;
<span class="text-muted">by {{ article.author }} | {{ article.date }}</span>
</div>
<div class="card-body">
{{ article.body }}
</div>
<div class="card-footer text-center text-muted">
<a href="#">Edit</a> | <a href="#">Delete</a>
</div>
<br />
{% endfor %}
{% endblock content %}

```

Spin up the server again with `python manage.py runserver` and check out our page at <http://127.0.0.1:8000/articles/>.



Articles page

Not bad eh? If we wanted to get fancy we could create a [custom template filter](#) so that the date outputted is shown in seconds, minutes, or days. This can be done with some if/else logic and Django's [date options](#) but we won't implement it here.

Edit/Delete How do we add edit and delete options? We need new urls, views, and templates. Let's start with the urls. We can take advantage of the fact that Django automatically adds a primary key to each database. Therefore our first article with a primary key of 1 will be at articles/1/edit/ and the delete route will be at

articles/1/delete/.

Code

```
# articles/urls.py
from django.urls import path

from . import views

urlpatterns = [
    path('', views.ArticleListView.as_view(), name='article_list'),
    path('<int:pk>/edit/',
        views.ArticleUpdateView.as_view(), name='article_edit'), # new
    path('<int:pk>',
        views.ArticleDetailView.as_view(), name='article_detail'), # new
    path('<int:pk>/delete/',
        views.ArticleDeleteView.as_view(), name='article_delete'), # new
]
```

Now write up our views which will use Django's generic class-based views for `DetailView`, `UpdateView` and `DeleteView`. We specify which fields can be updated—`title` and `body`—and where to redirect the user after deleting an article: `article_list`.

Code

```
# articles/views.py
from django.views.generic import ListView, DetailView
from django.views.generic.edit import UpdateView, DeleteView
from django.urls import reverse_lazy

from . import models

class ArticleListView(ListView):
    model = models.Article
    template_name = 'article_list.html'

class ArticleDetailView(DetailView):
    model = models.Article
    template_name = 'article_detail.html'

class ArticleUpdateView(UpdateView):
    model = models.Article
    fields = ['title', 'body', ]
    template_name = 'article_edit.html'

class ArticleDeleteView(DeleteView):
    model = models.Article
    template_name = 'article_delete.html'
    success_url = reverse_lazy('article_list')
```

Finally we need to add our new templates. Stop the server with `Control+c` and type the following.

Command Line

```
(news) $ touch templates/article_detail.html  
(news) $ touch templates/article_edit.html  
(news) $ touch templates/article_delete.html
```

We'll start with the details page which will display the title, date, body, and author with links to edit and delete. It will also link back to all articles. Recall that the Django templating language's `url` tag wants the URL name and then any arguments passed in. The name of our edit route is `article_edit` and we need to pass in its primary key `article.pk`. The delete route name is `article_delete` and it also needs a primary key `article.pk`. Our articles page is a `ListView` so it does not need any additional arguments passed in.

Code

```
<!-- templates/article_detail.html -->  
{% extends 'base.html' %}  
  
{% block content %}  
<div class="article-entry">  
<h2>{{ object.title }}</h2>  
<p>by {{ object.author }} | {{ object.date }}</p>  
<p>{{ object.body }}</p>  
</div>  
  
<p><a href="{% url 'article_edit' article.pk %}">Edit</a> | <a href="{% url 'article_delete' article.pk %}">Delete</a></p>  
<p>Back to <a href="{% url 'article_list' %}">All Articles</a>.</p>  
{% endblock content %}
```

For the edit and delete pages we can use Bootstrap's [button styling](#) to make the edit button light blue and the delete button red.

Code

```
<!-- templates/article_edit.html -->  
{% extends 'base.html' %}  
  
{% block content %}  
<h1>Edit</h1>  
<form action="" method="post">{% csrf_token %}  
&nbsp;  
<button class="btn btn-info ml-2" type="submit">Update</button>  
</form>  
{% endblock %}
```

Code

```
<!-- templates/article_delete.html -->  
{% extends 'base.html' %}  
  
{% block content %}  
<h1>Delete</h1>  
<form action="" method="post">{% csrf_token %}  
<p>Are you sure you want to delete "{{ article.title }}"?</p>  
<button class="btn btn-danger ml-2" type="submit">Confirm</button>
```

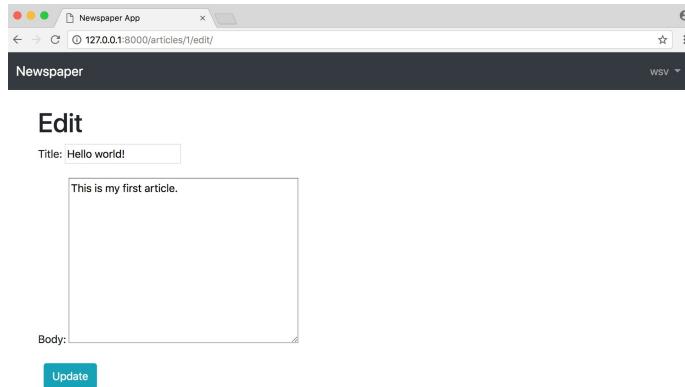
```
</form>
{% endblock %}
```

As a final step we can add the edit and delete links to our lists page at the div class for card-footer These will be the same as those added to the detail page.

Code

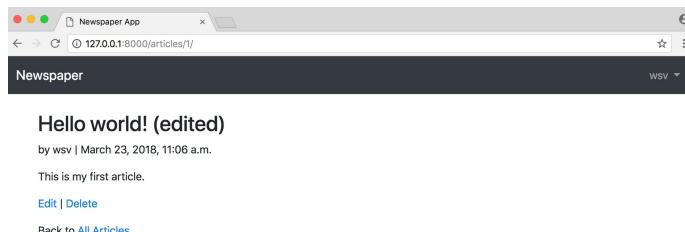
```
<!-- templates/article_list.html -->
...
<div class="card-footer text-center text-muted">
<a href="{% url 'article_edit' article.pk %}">Edit</a> | 
<a href="{% url 'article_delete' article.pk %}">Delete</a>
</div>
...
```

Ok, we're ready to view our work. Start up the server with `python manage.py runserver` and navigate to articles page at <http://127.0.0.1:8000/articles/>. Click on the link for “edit” on the first article and you'll be redirected to:
<http://127.0.0.1:8000/articles/1/edit/>



Edit page

If you update the “title” field and click update you'll be redirected to the detail page which shows the new change.



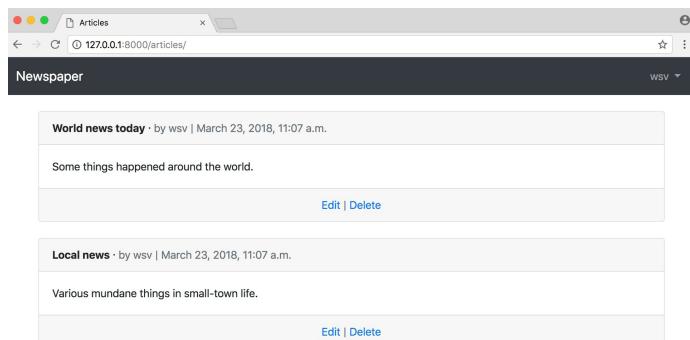
Detail page

If you click on the “Delete” link you’ll be redirected to the delete page.



Delete page

Press the scary red button for “Delete” and you’ll be redirected to the articles page which now only has two entries.



Articles page two entries

Create page The final step is a create page for new articles which we can do with Django’s createView. Our three steps are to create a view, url, and template. This flow should feel pretty familiar by now.

In our views file add `CreateView` to the imports at the top and make a new class `ArticleCreateView` that specifies our model, template, and the fields available.

Code

```
# articles/views.py
...
from django.views.generic.edit import CreateView, UpdateView, DeleteView

class ArticleCreateView(CreateView):
    model = models.Article
    template_name = 'article_new.html'
    fields = ['title', 'body', 'author', ]
...
```

Note that our `fields` has `author` since we want to associate a new article with an author, however once an article has been created we do not want a user to be

able to change the author which is why ArticleUpdateView only has the fields ['title', 'body',].

Update our urls file with the new route for the view.

Code

```
# articles/urls.py
...
urlpatterns = [
...
path('new/', views.ArticleCreateView.as_view(), name='article_new'),
...
]
```

Then quit the server Control+c to create a new template named article_new.html.

Command Line

```
(news) $ touch templates/article_new.html
```

And update it with the following HTML code.

Code

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

{% block content %}
<h1>New article</h1>
<form action="" method="post">{% csrf_token %}
{{ form.as_p }}
<button class="btn btn-success ml-2" type="submit">Save</button>
</form>
{% endblock %}
```

As a final step we should add a link to creating new articles in our nav so it is accessible everywhere on the site to logged-in users.

Code

```
<!-- templates/base.html -->
...
<body>
<nav class="navbar navbar-expand-md navbar-dark bg-dark mb-4">
<a class="navbar-brand" href="{% url 'home' %}">Newspaper</a>
{% if user.is_authenticated %}
<ul class="nav navbar-nav mr-auto">
<li class="nav-item"><a href="{% url 'article_new' %}">+ New</a></li>
</ul>
{% endif %}
<button class="navbar-toggler" type="button" data-toggle="collapse" data-target=\ "#navbarSupportedContent" aria-controls="navbarSupportedContent" aria-expanded="false" aria-label="Toggle navigation">
<span class="navbar-toggler-icon"></span>
</button>
```

...

And why not use Bootstrap to improve our original homepage now too? We can update `templates/home.html` as follows.

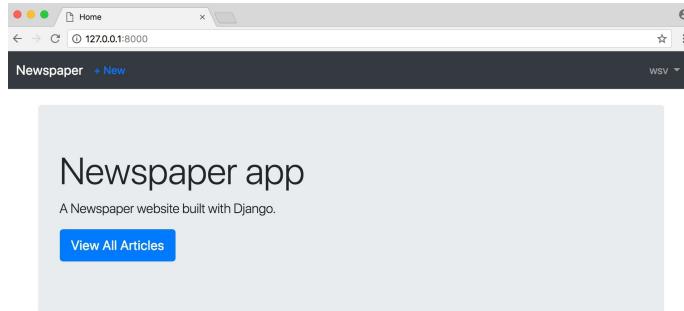
Code

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

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

{% block content %}
<div class="jumbotron">
<h1 class="display-4">Newspaper app</h1>
<p class="lead">A Newspaper website built with Django.</p>
<p class="lead"><a class="btn btn-primary btn-lg" href="{% url 'article_list' %}" role="button">\nView All Articles</a></p>
</div>
{% endblock %}
```

We're all done. Let's just confirm everything works as expected. Start up the server again `python manage.py runserver` and navigate to our homepage at: <http://127.0.0.1:8000/>.



Homepage with new link in nav

Click on the link for “+ New” in the top nav and you’ll be redirected to our create page.

New article

Title:

Body:

Author:

Save

Create page

Go ahead and create a new article. Then click on the “Save” button. You will be redirected to the detail page. Why? Because in our `models.py` file we set the `get_absolute_url` method to `article_detail`. This is a good approach because if we later change the url pattern for the detail page to, say, `articles/details/4/`, the redirect will still work. Whatever route is associated with `article_detail` will be used; there is no hardcoding of the route itself.

4th article

by wsv | March 23, 2018, 11:44 a.m.

This really works!

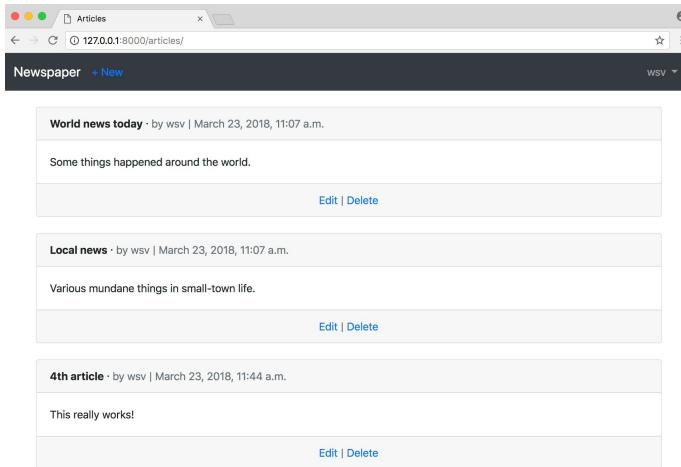
[Edit](#) | [Delete](#)

[Back to All Articles.](#)

Detail page

Note also that the primary key here is 4 in the URL. Even though we’re only displaying three articles right now, Django doesn’t reorder the primary keys just because we deleted one. In practice, most real-world sites don’t actually delete anything; instead they “hide” deleted fields since this makes it easier to maintain the integrity of a database and gives the option to “undelete” later on if needed. With our current approach once something is deleted it’s gone for good!

Click on the link for “All Articles” to see our new `/articles` page.



Updated articles page

There's our new article on the bottom as expected.

Conclusion We have created a dedicated articles app with CRUD functionality. But there are no permissions or authorizations yet, which means anyone can do anything! A logged-out user can visit all URLs and any logged-in user can make edits or deletes to an existing article, even one that's not their own! In the next chapter we will add permissions and authorizations to our project to fix this.

Chapter 14: Permissions and Authorization

There are several issues with our current *Newspaper* website. For one thing we want our newspaper to be financially sustainable. With more time we could add a payments app to charge for access, but for now we will require a user to log in to view any articles. This is known as *authorization*. It's common to set different rules around who is authorized to view areas of your site. Note that this is different than **authentication** which is the process of registering and logging-in users. Authorization restricts access; authentication enables a user signup and login flow.

As a mature web framework, Django has built-in functionality for authorization that we can quickly use. In this chapter we'll limit access to various pages only to logged-in users.

Improved CreateView At present the author on a new article can be set to any user. Instead it should be automatically set to the current user. The default CreateView provides a lot of functionality for us but in order to set the current user to author we will need to customize it. We will remove author from the fields and instead set it automatically via the `form_valid` method.

Code

```
# articles/views.py
...
class ArticleCreateView(CreateView):
    model = models.Article
    template_name = 'article_new.html'
    fields = ['title', 'body']

    def form_valid(self, form):
        form.instance.author = self.request.user
        return super().form_valid(form)
    ...
```

How did I know I could update `CreateView` like this? The answer is I looked at the source code and used Google. Generic class-based views are amazing for starting new projects but when you want to customize them, it is necessary roll up your sleeves and start to understand what's going on under the hood. The

more you use and customize built-in views, the more comfortable you will become making customizations like this. Chances are whatever you are trying to do has already been solved somewhere, either within Django itself or on a forum like [Stack Overflow](#). Don't be afraid to ask for help!

Now reload the browser and try clicking on the “+ New” link in the top nav. It will redirect to the updated create page where author is no longer a field. If you create a new article and then go into the admin you will see it is automatically set to the current logged-in user.

The screenshot shows a web browser window titled "Newspaper App" with the URL "127.0.0.1:8000/articles/new/". The page is titled "New article". It has two input fields: "Title:" and "Body:". The "Body:" field is a large text area. At the bottom is a green "Save" button.

New article link

Authorizations There are multiple issues around the lack of authorizations in our current project. Obviously we would like to restrict access to only users so we have the option of one day charging readers to our newspaper. But beyond that, any random logged-out user who knows the correct URL can access any part of the site.

Consider what would happen if a logged-out user tried to create a new article? To try it out, click on your username in the upper right corner of the nav bar, then select “Log out” from the dropdown options. The “+ New” link disappears from the nav bar but what happens if you go to it directly:

<http://127.0.0.1:8000/articles/new/>?

The page is still there.

New article

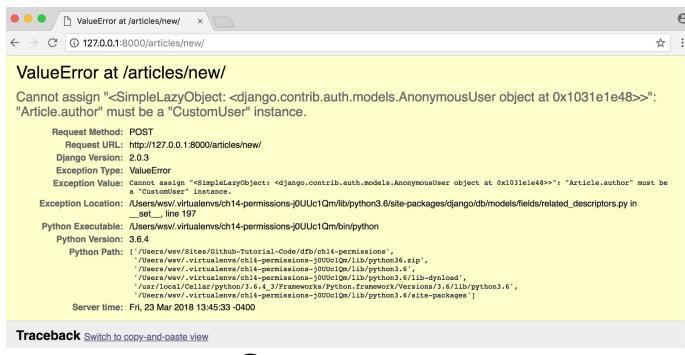
Title:

Body:

Save

Logged out new

Now try to create a new article with a title and body. Click on the “Save” button.



Create page error

An error. This is because our model **expects** an author field which is linked to the current logged-in user. But since we are not logged in, there's no author, and therefore the submission fails.

Mixins We clearly want to set some authorizations so only logged-in users can access the site. To do this we can use a *mixin*, which is a special kind of multiple inheritance that Django uses to avoid duplicate code and allow for customization. For example, the built-in generic [ListView](#) needs a way to return a template. But so does [DetailView](#) and in fact almost every other view. Rather than repeat the same code in each big generic view, Django breaks out this functionality into a “mixin” known as [TemplateResponseMixin](#). Both [ListView](#) and [DetailView](#) use this mixin to render the proper template.

If you read the Django source code, which is freely available [on Github](#), you'll see mixins used all over the place.

To restrict view access to only logged in users, Django has a [LoginRequiredMixin](#) that we can use. It's powerful and extremely concise.

Within the existing `articles/views.py` file import it at the top and then add `LoginRequiredMixin` to our `ArticleCreateView`. Make sure that the mixin is to the left of `ListView` so it will be read first. We want the `ListView` to already know we intend to restrict access.

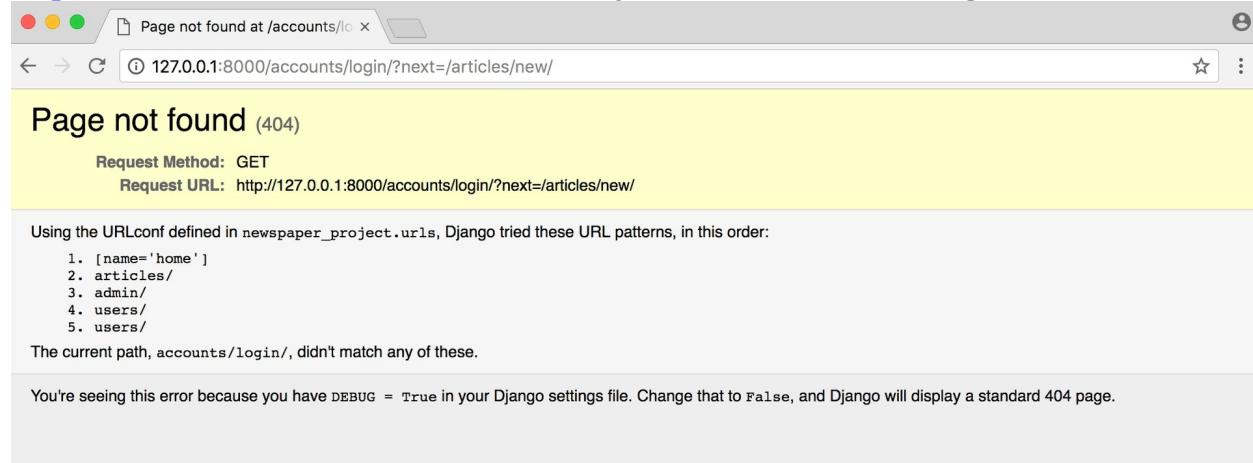
And that's it! We're done.

Code

```
# articles/views.py
from django.contrib.auth.mixins import LoginRequiredMixin # new

...
class ArticleCreateView(LoginRequiredMixin, CreateView): # new
...
```

Now return to the URL for creating new messages at <http://127.0.0.1:8000/articles/new/> and you'll see the following error:



Error page

What's happening? Django has automatically redirected us to the default location for the `login` page which is at `/accounts/login` however if you recall, in our project-level URLs we are using `users/` as our route. That's why our `login` page is at `users/login`. So how do we tell our `ArticleCreateView` about this?

If you look at the [documentation for LoginRequired mixin](#) it tells us the answer. We can add a `login_url` to override the default parameter. We're using the named URL of our login route here `login`.

Code

```
# articles/views.py
...
class ArticleCreateView(LoginRequiredMixin, CreateView):
model = models.Article
template_name = 'article_new.html'
fields = ['title', 'body',]
login_url = 'login' # new

def form_valid(self, form):
form.instance.author = self.request.user
return super().form_valid(form)
```

Try the link for creating new messages again: <http://127.0.0.1:8000/articles/new/>.

It now redirects users to the login page. Just as we desired!

Updating views Now we see that restricting view access is just a matter of adding `LoginRequiredMixin` at the beginning of all existing views and specifying the correct `login_url`. Let's update the rest of our articles views since we don't want a user to be able to create, read, update, or delete a message if they aren't logged in.

The complete `views.py` file should now look like this:

Code

```
# articles/views.py
from django.contrib.auth.mixins import LoginRequiredMixin
from django.views.generic import ListView, DetailView
from django.views.generic.edit import CreateView, UpdateView,
DeleteView
from django.urls import reverse_lazy

from . import models

class ArticleListView(LoginRequiredMixin, ListView):
model = models.Article
template_name = 'article_list.html'
login_url = 'login'
```

```
class ArticleDetailView(LoginRequiredMixin, DetailView):
model = models.Article
template_name = 'article_detail.html'
login_url = 'login'

class ArticleUpdateView(LoginRequiredMixin, UpdateView):
model = models.Article
fields = ['title', 'body', ]
template_name = 'article_edit.html'
login_url = 'login'

class ArticleDeleteView(LoginRequiredMixin, DeleteView):
model = models.Article
template_name = 'article_delete.html'
success_url = reverse_lazy('article_list')
login_url = 'login'

class ArticleCreateView(LoginRequiredMixin, CreateView):
model = models.Article
template_name = 'article_new.html'
fields = ['title', 'body', ]
login_url = 'login'

def form_valid(self, form):
form.instance.author = self.request.user
return super().form_valid(form)
```

Go ahead and play around with the site to confirm that the login redirects now work as expected. If you need help recalling what the proper URLs are, log in first and write down the URLs for each of the routes for create, edit, delete, and all articles.

Conclusion Our *Newspaper* app is almost done. We have our articles properly configured, have set permissions and authorizations, user authentication is in good shape. The last item is to add the ability for fellow logged-in users to leave comments which we'll cover in the next chapter.

Chapter 15: Comments

There are two ways we could add comments to our *Newspaper* site. The first is to create a dedicated *comments* app and link it to *articles*, however that seems like over-engineering at this point. Instead we can simply add an additional model called `Comment` to our *articles* app and link it to the `Article` model through a foreign key. By the end of this chapter users will have the ability to leave comments on any other users articles.

Model To start we can add another table to our existing database called comment. This model will have a many-to-one foreign key relationship to Article: one article can have many comments, but not the other way around. Traditionally the name of the foreign key field is simply the model it links to, so this field will be called article. The other two fields will be comment and author.

Open up the file `articles/models.py` and underneath the existing code add the following.

Code

```
# articles/models.py
...
class Comment(models.Model):
    article = models.ForeignKey(Article, on_delete=models.CASCADE)
    comment = models.CharField(max_length=140)
    author = models.ForeignKey(
        get_user_model(),
        on_delete=models.CASCADE,
    )

    def __str__(self):
        return self.comment

    def get_absolute_url(self):
        return reverse('article_list')
```

Our `Comment` model also has a `__str__` method and a `get_absolute_url` method that returns to the main `articles/` page.

Since we've updated our models it's time to make a new migration file and then

apply it. Note that by adding `articles` at the end of each command—which is optional—we are specifying we want to use just the `articles` app here. This is a good habit to use. For example, what if we made changes to models in two different apps? If we **did not** specify an app, then both apps' changes would be incorporated in the same migrations file which makes it harder, in the future, to debug errors. Keep each migration as small and contained as possible.

Command Line

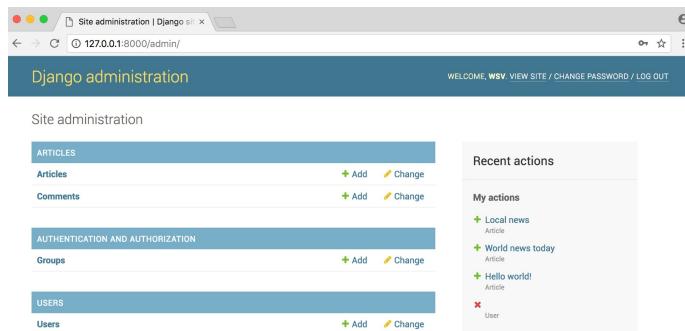
```
(news) $ python manage.py makemigrations articles  
(news) $ python manage.py migrate
```

Admin After making a new model it's good to play around with it in the admin app before displaying it on our actual website. Add comment to our `admin.py` file so it will be visible.

Code

```
# articles/admin.py  
from django.contrib import admin  
  
from . import models  
  
admin.site.register(models.Article)  
admin.site.register(models.Comment)
```

Then start up the server with `python manage.py runserver` and navigate to our main page <http://127.0.0.1:8000/admin/>



Admin page with Comments

Under our app “Articles” you’ll see our two tables: Comments and Articles. Click on the “+ Add” next to Comments. You’ll see that under Article is a dropdown of existing articles, same thing for Author, and there is a text field next to Comment.

The screenshot shows the Django admin interface for adding a comment. The URL is 127.0.0.1:8000/admin/articles/comment/add/. The page has a header 'Django administration' and a top navigation bar with links like 'Home', 'Articles', 'Comments', and 'Add comment'. Below the header, there's a form with three fields: 'Article' (set to 'Local news'), 'Comment' (containing 'Nice article!'), and 'Author' (set to 'testuser'). At the bottom of the form are three buttons: 'Save and add another', 'Save and continue editing', and a larger 'SAVE' button.

Admin Comments

Select an Article, write a comment, and then select an author that is not your superuser, perhaps `testuser` as I've done in the picture. Then click on the "Save" button.

You should next see your comment on the "Comments" page.

The screenshot shows the Django admin interface for managing comments. The URL is 127.0.0.1:8000/admin/articles/comment/. The page has a header 'Django administration' and a top navigation bar with links like 'Home', 'Articles', and 'Comments'. A green success message box at the top says 'The comment "Nice article!" was added successfully.'. Below the message is a table with a single row containing a checkbox, the word 'COMMENT', and the text 'Nice article!'. At the top left is a dropdown menu labeled 'Select comment to change' with an arrow icon. At the top right is a blue 'ADD COMMENT' button.

Admin Comment One

At this point we could add an additional admin field so we'd see the comment and the article on this page. But wouldn't it be better to just see all Comment models related to a single Post model? It turns out we can with a Django admin feature called **inlines** which displays foreign key relationships in a nice, visual way.

There are two main inline views used: [TabularInline](#) and [StackedInline](#). The only difference between the two is the template for displaying information. In a TabularInline all model fields appear on one line while in a StackedInline each field has its own line. We'll implement both so you can decide which one you prefer.

Update `articles/admin.py` as follows in your text editor.

Code

```
# articles/admin.py
```

```

from django.contrib import admin
from . import models

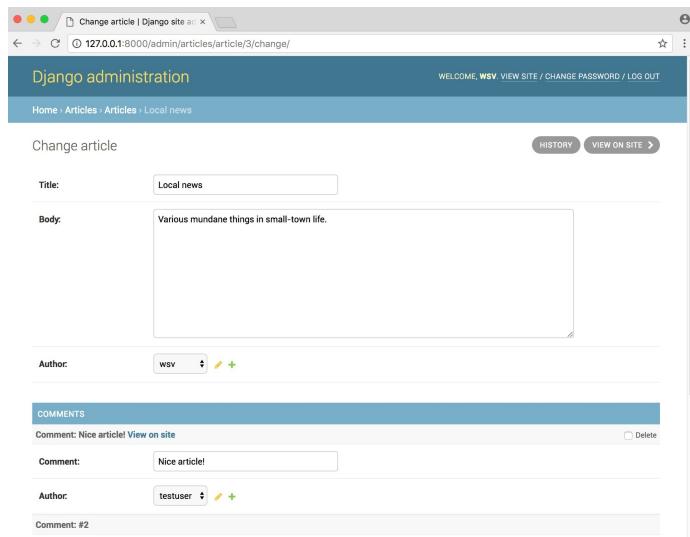
class CommentInline(admin.StackedInline):
    model = models.Comment

class ArticleAdmin(admin.ModelAdmin):
    inlines = [
        CommentInline,
    ]

admin.site.register(models.Article, ArticleAdmin)
admin.site.register(models.Comment)

```

Now go back to the main admin page at <http://127.0.0.1:8000/admin/> and click on “Articles.” Select the article which you just added a comment for which was “Local news” in my case.



Admin change page

Better right! We can see and modify all our related articles and comments in one place.

Personally though I prefer using `TabularInline` as it shows more information in less space. To switch to it we only need to change our `CommentInline` from `admin.StackedInline` to `admin.TabularInline`.

Code

```

# articles/admin.py
from django.contrib import admin
from . import models

```

```

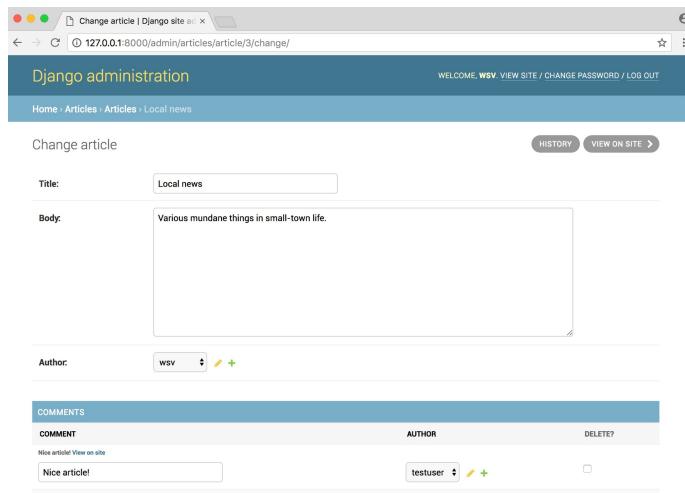
class CommentInline(admin.TabularInline):
    model = models.Comment

class ArticleAdmin(admin.ModelAdmin):
    inlines = [
        CommentInline,
    ]

admin.site.register(models.Article, ArticleAdmin)
admin.site.register(models.Comment)

```

Refresh the admin page and you'll see the new change: all fields for each model are displayed on the same line.



TabularInline page

Much better. Now we need to update our template to display comments.

Template Since comment lives within our existing articles app we only need to update the existing templates for article_list.html and article_detail.html to display our new content. We don't have to create new templates and mess around with urls and views.

What we want to do is display **all** comments related to a specific article. This is called a “query” as we’re asking the database for a specific bit of information. In our case, working with a foreign key, we want to [follow a relationship backward](#): for each Article look up related Comment models.

Django has a built-in syntax we can use known as `FOO_set` where `FOO` is the lowercased source model name. So for our `Article` model we can use `article_set` to access all instances of the model.

But personally I strongly dislike this syntax as I find it confusing and non-intuitive. A better approach is to add a `related_name` attribute to our model which lets us explicitly set the name of this reverse relationship instead. Let's do that.

To start add a `related_name` attribute to our `Comment` model. A good default is to name it the plural of the model holding the `ForeignKey`.

Code

```
# articles/models.py
...
class Comment(models.Model):
    article = models.ForeignKey(
        Article,
        on_delete=models.CASCADE,
        related_name='comments' # new
    )
```

Since we just made a change to our database model we need to create a migrations file and update the database. Stop the local server with `Control+c` and execute the following two commands. Then spin up the server again as we will be using it shortly.

Command Line

```
(news) $ python manage.py makemigrations articles
(news) $ python manage.py migrate
(news) $ python manage.py runserver
```

Understanding queries takes some time so don't be concerned if the idea of reverse relationships is confusing. I'll show you how to implement the code as desired. And once you've mastered these basic cases you can explore how to filter your querysets in great detail so they return exactly the information you want.

In our `article_list.html` file we can add our comments to the `card-footer`. Note that I've moved our edit and delete links up into `card-body`. To access each comment we're calling `article.comments.all` which means first look at the `article` model, then `comments` which is the related name of the entire `Comment` model, and select `all` included. It can take a little while to become accustomed to this syntax for referencing foreign key data in a template!

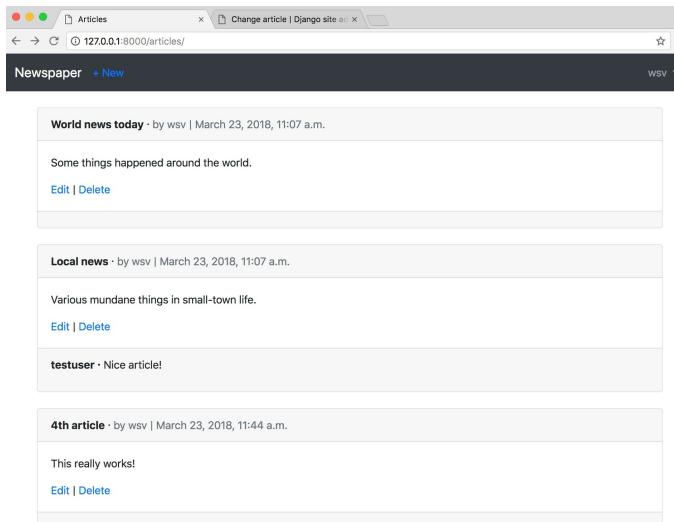
Code

```
<!-- template/article_list.html -->
{% extends 'base.html' %}

{% block title %}Articles{% endblock %}

{% block content %}
{% for article in object_list %}
<div class="card">
<div class="card-header">
<span class="font-weight-bold">{{ article.title }}</span> &middot;
<span class="text-muted">by {{ article.author }} | {{ article.date }}</span>
</div>
<div class="card-body">
<p>{{ article.body }}</p>
<a href="{% url 'article_edit' article.pk %}">Edit</a> | 
<a href="{% url 'article_delete' article.pk %}">Delete</a>
</div>
<div class="card-footer">
{% for comment in article.comments.all %}
<p>
<span class="font-weight-bold">{{ comment.author }}</span> &middot;
{{ comment }}
</p>
{% endfor %}
</div>
<br />
{% endfor %}
{% endblock content %}
```

If you refresh the articles page at <http://127.0.0.1:8000/articles/> we can see our new comment displayed on the page.



Articles page with comments

Yooahoo! It works. We can see both comments listed underneath the initial message.

Conclusion With more time we would focus on forms now so a user could write a new article directly on the articles/page as well as add comments too. But the main focus of this chapter is to demonstrate how foreign key relationships work in Django.

Our *Newspaper* app is now complete. It has a robust user authentication flow including the use of email for password resets. We are also using a custom user model so if we want to add additional fields to our `CustomUser` model it is as simple as adding an additional field. We already have an age field for all users that is currently being set to 0 by default. If we wanted to, we could add an age dropdown to the signup form and restrict user access only to users over age 13. Or we could offer discounts to users over age 65. Whatever we want to do to our `CustomUser` model is an option.

Most of web development follows the same patterns and by using a web framework like Django 99% of what we want in terms of functionality is either already included or only a small customization of an existing feature away.

Conclusion Congratulations on finishing *Django for Beginners!* After starting from absolute zero we've now built five different web applications from scratch. And we've covered all the major features of Django: templates, views, urls, users, models, security, testing, and deployment. You now have the knowledge to go off and build your own modern websites with Django.

As with any new skill, it's important to practice and apply what you've just learned. The CRUD functionality in our *Blog* and *Newspaper* sites is common in many, many other web applications. For example, can you make a Todo List web application? You already have all the tools you need.

Web development is a very deep field and there's still much more to learn about what Django has to offer. For example a more advanced Django project would likely use multiple settings files, environment variables, and PostgreSQL locally instead of SQLite. It might even use third-party packages like `django-allauth` for social authentication, `django-debug-toolbar` for debugging, and `django-extensions` for extra goodies.

The best way to learn more about Django and web development in general is to think of a project you want to build and then step-by-step learn what you need to complete it. Two additional resources that may help are [DjangoX](#), a real-world starter project, and [awesome-django](#), a curated list of awesome things related to Django.

I have also written a second book [REST APIs with Django](#) that describes how to turn any Django website into a powerful web API with a minimal amount of code. The book covers how to build three progressively more complex APIs, connect them to any front-end like React, Vue, or Angular, and dives into more advanced topics like user authentication, permissions, and documentation.

A final resource is my own personal website, wsvincent.com, which is regularly updated and features articles on some more advanced Django topics:

- [Django, PostgreSQL, and Docker](#)
- [Django Social Authentication](#)
- [Django Log In Mega-Tutorial](#)
- [Official Django REST Framework Tutorial - A Beginner's Guide](#)
- [Django Rest Framework Tutorial](#)
- [Django Rest Framework with React](#)

Django Resources To continue learning Django, I recommend working through the following free online tutorials:

- [Official Polls Tutorial](#)
- [Django Girls Tutorial](#)
- [MDN: Django Web Framework](#)
- [A Complete Beginner's Guide to Django](#)

I also strongly recommend [Two Scoops of Django 1.11: Best Practices for the Django Web Framework](#), which is the current best-practices bible for Django developers.

Python Books If you're new to Python, there are several excellent books available for beginners to advanced Pythonistas:

- [Python Crash Course](#) is a fantastic introduction to Python that also walks you through three real-world projects, including a Django application.
- [Think Python](#) introduces Python and computer science fundamentals at the same time.
- [Automate the Boring Stuff](#) is another great guide to learning and using Python in real-world settings.
- [The Hitchhiker's Guide to Python](#) covers best practices in Python programming.
- [Python Tricks](#) demonstrates how to write Pythonic code.
- [Effective Python](#) is an excellent guide not just to Python but programming in general.

- [Fluent Python](#) is amazing and provides a deep understanding of the Python language.

Blogs to Follow These sites provide regular, high-quality writings on Python and web development.

- [TestDriven](#)
- [Real Python](#)
- [Dan Bader](#)
- [Trey Hunner](#)
- [Full Stack Python](#)
- [Ned Batchelder](#)
- [Armin Ronacher](#)
- [Kenneth Reitz](#)
- [Daniel Greenfeld](#)

Feedback If you've made it through the entire book, I'd love to hear your thoughts. What did you like or dislike? What areas were especially difficult? And what new content would you like to see? I can be reached at will@wsvincent.com.