DJANGO *for* BEGINNERS

Build websites with Python & Django

WILLIAM S. VINCENT

Django for Beginners

Build websites with Python & Django

William S. Vincent

Also By William S. Vincent

Django for APIs

Django for Professionals

Contents

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 *Pages* app, a *Message Board* app, a *Blog* app with forms and user accounts, and finally a *Newspaper* app that uses a custom user model, email integration, foreign keys, authorization, permissions, and more. By the end of this book you will 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. A "web framework" is software that abstracts away many of the common challenges related to building a website, such as connecting to a database, handling security, user accounts, and so on. These days most developers rely on web frameworks rather than trying to build a website truly from scratch. Django in particular was first released in 2005 and has been in continuous development since then. Today, it is one of the most popular web frameworks available, 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 and Python. It also uses Pipenv for managing Python packages and virtual environments, though using Pip works fine as well. Throughout we'll be using modern best practices from the Django, Python, and web development communities including 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, including:

- user authentication
- testing
- database models, forms, URL routes, and templates
- admin interface
- security and performance upgrades
- support for multiple database backends

This approach allows web developers to focus on what makes a web application unique rather than reinventing the wheel every time for standard, secure 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 2019 Django has been under active development for over 14 years which makes it a grizzled veteran in software years. Millions of programmers have already used Django to build their websites, which 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. And best of all it's written in the wonderfully readable yet still powerful Python programming language. In short, 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 now recognize 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 with this book is to fill in the gaps and showcase how beginner-friendly Django really can be.

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 properly learn Python, HTML, and CSS. 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 (3.1), Python (3.8), 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* app 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 GitHub.

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 are quite powerful yet require a minimal amount of code. 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 GitHub, and deploy to Heroku.

In **Chapters 5-7** we're ready for our next project: a robust *Blog* app that implements CRUD (Create-Read-Update-Delete) functionality. By using Django's generic class-based views we only 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 for sign up, log in, and log out functionality.

The remainder of the book, **Chapters 8-16**, is dedicated to building a robust **Newspaper** site, starting with the introduction to custom user models in **Chapter 8**, a Django best practice that is rarely addressed in tutorials. **Chapter 9** covers user authentication, **Chapter 10** adds Bootstrap for styling, and **Chapters 11-12** implement password reset and change via email. With **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. And in **Chapter 16**, environment variables are introduced alongside proper deployment techniques.

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

While you could pick and choose chapters to read, the book's structure is deliberate. Each app/chapter introduces a new concept and reinforces past teachings. I highly recommend reading the book 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 a solid 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 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
```

Official Source Code

Complete source code for all chapters can be found in the official GitHub repository. While it's best to type all the code by hand yourself, if you do find yourself stuck with a coding example or seeing a strange error, make sure to check your code against the official repo. And if you're still stuck, try copy and pasting the official source code. A common error is subtle white spacing differences that are almost impossible to detect to the naked eye.

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 Set Up

This chapter covers how to properly configure your computer to work on Django projects. We start with an overview of the command line and how to install the latest version of Django and Python. 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*. To find it, open a new Finder window, open the *Applications* directory, scroll down to open the *Utilities* directory, and double-click the application called Terminal.

On Windows machines there are actually two built-in command shells: the *Command* shell and *PowerShell*. You should use PowerShell, which is the more powerful of the two.

Going forward when the book refers to the "command line" it means to open a new console on your computer, using either Terminal or PowerShell.

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 on Mac)
- `dir` (list files in your current directory on Windows)
- `pwd` (print working directory)
- `mkdir` (make directory)

- `touch` (create a new file on Mac)

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.

Now let's create a new directory with `mkdir`, `cd` into it, and add a new file `index.html` with the `touch` command. Note that Windows machines unfortunately do not support a native `touch` command. In future chapters when instructed to create a new file, do so within your text editor of choice.

Command Line

```
~/Desktop $ mkdir new_dir && cd new_dir
~/Desktop/new_dir $ 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_dir $ 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_dir $ 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. A good free resource for further study is the Command Line Crash Course.

Install Python 3

It takes some configuration to properly install Python 3 on a Mac, Windows, Linux, or Chromebook computer and there are multiple approaches. Many developers–especially beginners–follow the advice on the official Python website to download distinct versions of Python directly onto their computer and then adjust the PATH variable accordingly.

The problem with this approach is that updating the PATH variable correctly is tricky, by downloading Python directly updates are harder to maintain, and there are now much easier ways to install and start using Python quickly.

I host a dedicated website, InstallPython3.com, with up-to-date guides for installing Python 3 on Mac, Windows, or Linux computers. Please refer there to install Python correctly on your local machine.

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 3.1 but *ProjectB* from last year is still on Django 2.2? 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**.

In this book we will use Pipenv to manage virtual environments. Pipenv is similar to `npm` and `yarn` from the JavaScript/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. Note the use of ~= which will ensure security updates for Django, such as 3.1.1, 3.1.2, and so on.

Command Line

```
$ pipenv install django~=3.1.0
```

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 around the name of your current directory on your command line which indicates the virtual environment is activated. Since we're in a `django` directory that means we should see `(django)` at the beginning of the command line prompt. Windows users will not see the shell prompt. If you can run `django-admin startproject` in the next section then you know your virtual environment has Django installed properly.

Command Line

```
(django) $
```

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

Command Line

```
(django) $ django-admin startproject config .
```

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

Layout

```
└── config
    ├── config
    │   ├── __init__.py
    │   ├── asgi.py
    │   ├── settings.py
    │   ├── urls.py
    │   └── wsgi.py
    └── manage.py
```

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

Layout

```
├── config
│   ├── __init__.py
│   ├── asgi.py
│   ├── settings.py
│   ├── urls.py
│   └── wsgi.py
└── manage.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.

> As you progress in your journey learning Django, you'll start to bump up more and more into similar situations where there are different opinions within the Django community on the correct best practice. Django is eminently customizable, which is a great strength, however the tradeoff is that this flexibility comes at the cost of seeming complexity. Generally speaking,

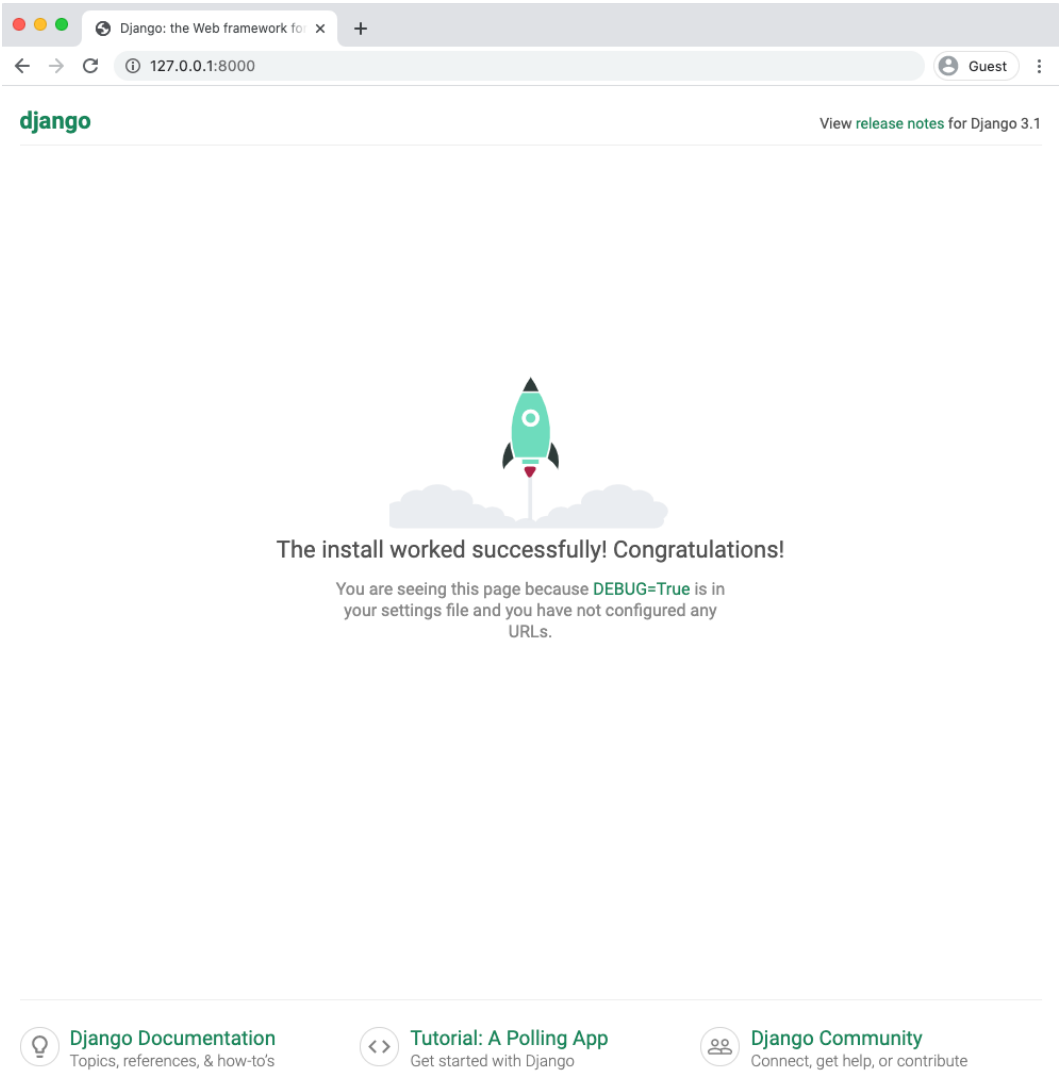
> it's a good idea to research any such issues that arise, make a decision, and then stick with it!

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

Command Line

```
(django) $ python manage.py runserver
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).

You have 18 unapplied migration(s). Your project may not work properly until
you apply the migrations for app(s): admin, auth, contenttypes, sessions.
Run 'python manage.py migrate' to apply them.

August 3, 2020 - 14:52:27
Django version 3.1, using settings 'config.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

Don't worry about the text in red about "18 unapplied migrations." We'll get to that shortly but the important part, for now, is to visit `http://127.0.0.1:8000/` and make sure the following image is visible:



Django welcome page

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

Command Line

```
(django) $ 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.

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 so 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.

Once installed, we need to do a one-time *system* set up to configure it by declaring the name and email address you want associated with all your Git commits. 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 GitHub.

If you become stuck at any point, complete source code for this and all future chapters is available online on the official GitHub repo.

Initial Set Up

To begin, navigate to a new directory on your computer. For example, we can create a `helloworld` directory 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~=3.1.0
$ 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)`. If you are on Windows you will not see a visual prompt at this time.

Create a new Django project called `config` 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 config .
```

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
.
├── Pipfile
├── Pipfile.lock
├── config
│   ├── __init__.py
│   ├── asgi.py
│   ├── settings.py
│   ├── urls.py
│   └── wsgi.py
└── manage.py

1 directory, 8 files
```

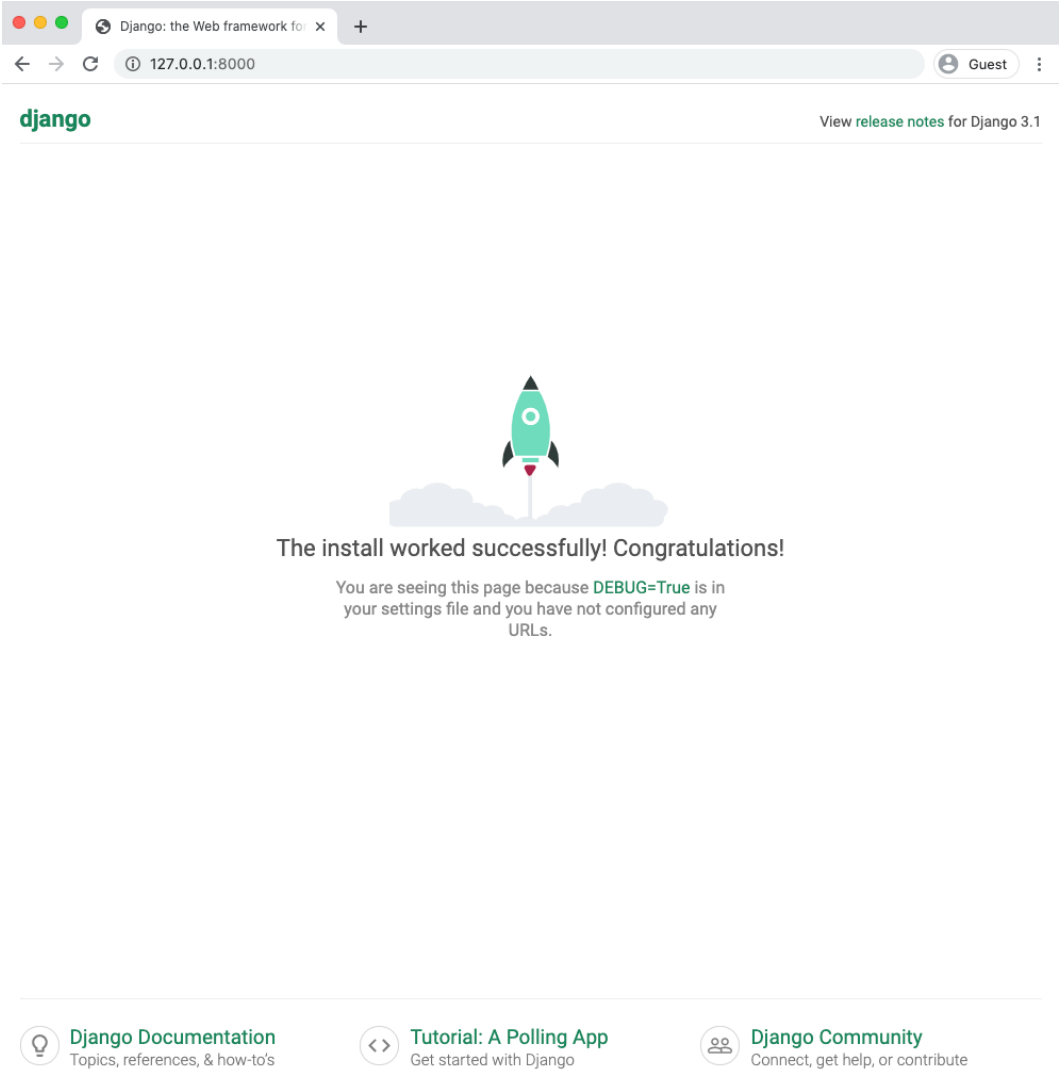
The `config/settings.py` file controls our project's settings, `urls.py` tells Django which pages to build in response to a browser or URL request, and `wsgi.py`, which stands for Web Server Gateway Interface, helps Django serve our eventual web pages. The `manage.py` file is used to execute various Django commands such as running the local web server or creating a new app. Last, but not least, is the `asgi.py` file, new to Django as of version 3.0 which allows for an optional Asynchronous Server Gateway Interface to be run.

Django comes with a built-in web server for local development purposes which we can now start 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 welcome page

Note that the full command line output will contain additional information including a warning about `18 unapplied migrations`.

Command Line

```
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).

You have 18 unapplied migration(s). Your project may not work properly until
you apply the migrations for app(s): admin, auth, contenttypes, sessions.
Run 'python manage.py migrate' to apply them.

August 3, 2020 - 14:57:42
Django version 3.1, using settings 'config.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

Technically this warning doesn't matter at this point. Django is complaining that we have not yet "migrated," or configured, our initial database. Since we won't actually use a database in this chapter, the warning won't affect the end result.

However, since warnings are still annoying to see, we can remove it by first stopping the local server with the `Control+c` command and then running `python manage.py migrate`.

Command Line

```
$ python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, sessions
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... OK
  Applying admin.0003_logentry_add_action_flag_choices... OK
  Applying contenttypes.0002_remove_content_type_name... OK
  Applying auth.0002_alter_permission_name_max_length... OK
  Applying auth.0003_alter_user_email_max_length... OK
  Applying auth.0004_alter_user_username_opts... OK
  Applying auth.0005_alter_user_last_login_null... OK
  Applying auth.0006_require_contenttypes_0002... OK
```

```
  Applying auth.0007_alter_validators_add_error_messages... OK
  Applying auth.0008_alter_user_username_max_length... OK
  Applying auth.0009_alter_user_last_name_max_length... OK
  Applying auth.0010_alter_group_name_max_length... OK
  Applying auth.0011_update_proxy_permissions... OK
  Applying auth.0012_alter_user_first_name_max_length... OK
  Applying sessions.0001_initial... OK
```

What Django has done here is migrate the built-in apps provided for us which we'll cover properly later in the book. But now, if you execute `python manage.py runserver` again, you should see the following clean output on the command line:

Command Line

```
$ python manage.py runserver
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).
August 3, 2020 - 15:23:14
Django version 3.1, using settings 'config.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

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. That's three distinct apps that all live within one top-level project.

How and when you split functionality into apps is somewhat subjective, but in general, each app should have a clear function.

Now it's time to create our first app. From the command line, quit the server with `Control+c`. Then use the `startapp` command followed by the name of our app, which will be `pages`.

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

```
(helloworld) $ tree
├── 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. Open the `config/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

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

Don't worry if you are confused at this point: it takes practice to internalize how Django projects and apps are structured. Over the course of this book we will build many projects and apps and the patterns will soon become familiar.

URLs, Views, Models, Templates

In Django, at least three (often four) separate files are required to power one single page. Within an app these are the `urls.py` file, the `views.py` file, the `models.py` file, and finally an HTML template such as `index.html`.

This interaction is fundamental to Django yet **very confusing** to newcomers so let's map out the order of a given HTTP request/response cycle. When you type in a URL, such as `https://djangoforbeginners.com`, the first thing that happens within our Django project is a *URLpattern* is found that matches the homepage. The URLpattern specifies a *view* which then determines the content for the page (usually from a database *model*) and then ultimately a *template* for styling and basic logic. The end result is sent back to the user as an HTTP response.

The complete flow looks like this:

Django request/response cycle

```
URL -> View -> Model (typically) -> Template
```

Remember how I said it can take three or four files for a given page? That's because a model is not always needed, in which case three files are enough. But generally speaking four will be used as we'll see later in this book.

The main takeaway here is that in Django *views* determine *what* content is displayed on a given page while *URLConfs* determine *where* that content is going. The *model* contains the *content* from the database and the *template* provides *styling* for it.

When a user requests a specific page, like the homepage, the *urls.py* file 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.

To see this in action, 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 which on a Mac can be done with the `touch` command; Windows users must create the file within a text editor.

Command Line

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

Then update it with the following code:

Code

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

urlpatterns = [
    path('', 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. By referring to the `views.py` file as `.views` we are telling Django to look within the current directory for a `views.py` file and import the view `homePageView` from there.

Our URLpattern has three parts:

- a Python regular expression for the empty string `''`
- a reference to the view called `homePageView`
- an optional named URL pattern called `'home'`

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

We're *almost* done at this point. The last step is to update our `config/urls.py` file. It's common to have multiple apps within a single Django project, like `pages` here, and they each need their own dedicated URL path.

Code

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

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

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, they will first be routed to the `pages` app and then to the `homePageView` view set in the `pages/urls.py` file.

This need for two separate `urls.py` files is often confusing to beginners. Think of the top-level `config/urls.py` as the gateway to various url patterns distinct to each app.

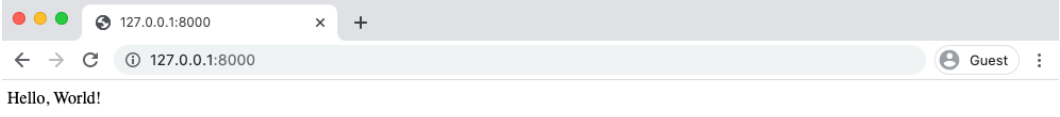
Hello, World!

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. Make sure you've stopped the local server with `Control+c`, then run the command `git init`.

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)

        Pipfile
        Pipfile.lock
        config/
        db.sqlite3
        manage.py
        pages/

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, (`-m`), describing what has changed.

Command Line

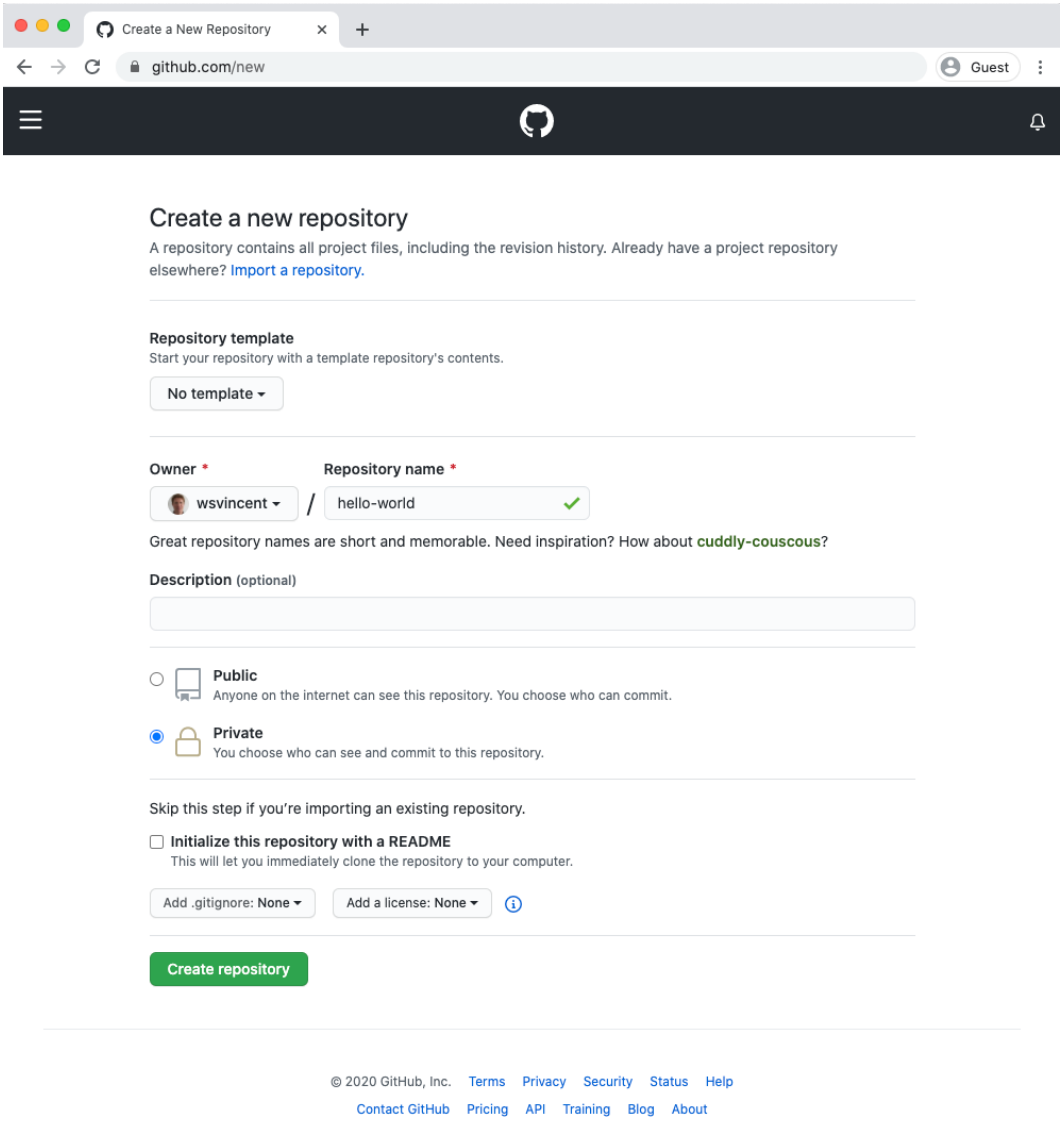
```
(helloworld) $ git add -A
(helloworld) $ git commit -m "initial commit"
```

GitHub

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. Popular choices include GitHub, Bitbucket, and GitLab. When you're learning web development, it's best to stick to private rather than public repositories so you don't inadvertently post critical information such as passwords online.

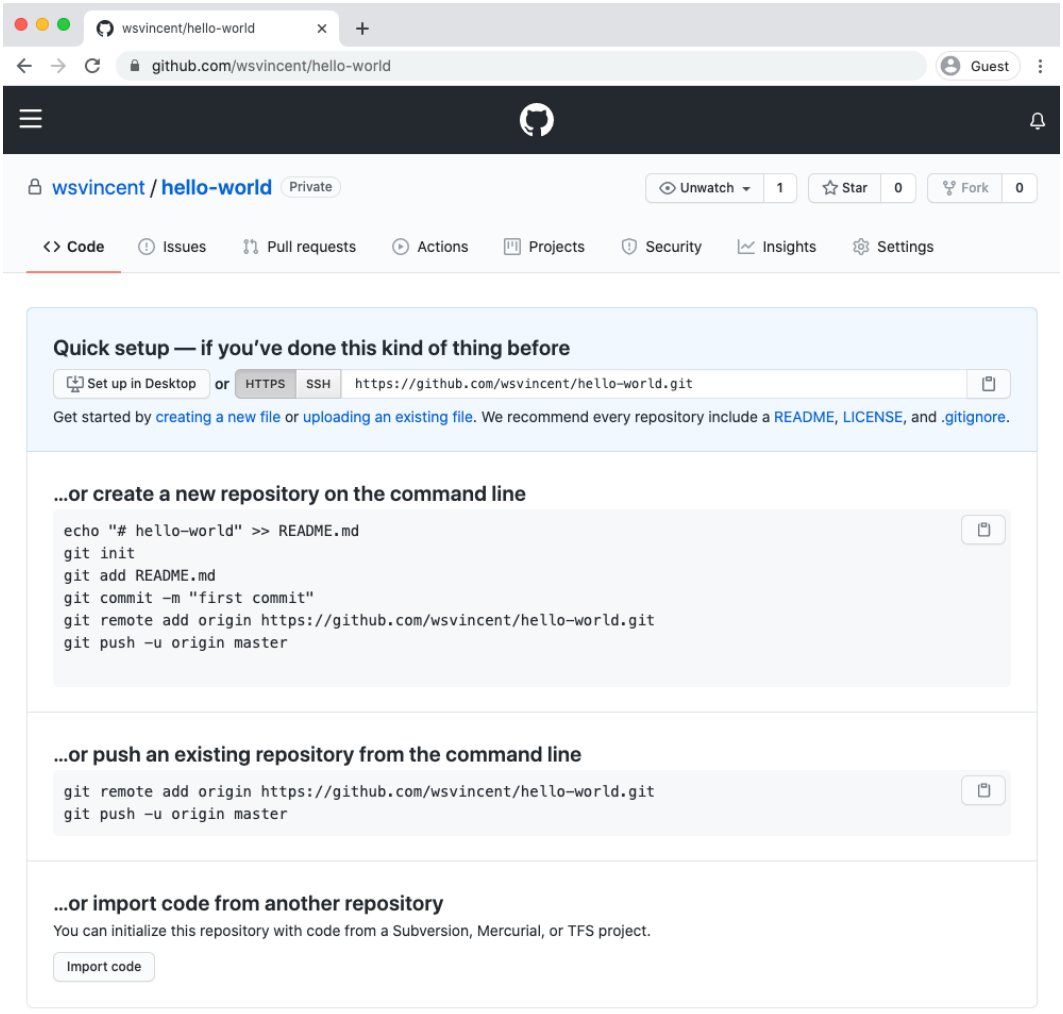
We will use GitHub in this book but all three services offer similar functionality for newcomers. Sign up for a free account on GitHub's homepage and verify your email address. Then navigate to the "Create a new repository" page located at https://github.com/new.

Enter the repository name `hello-world` and click on the radio button next to "Private" rather than "Public." Then click on the button at the bottom for "Create Repository."



GitHub New Repository

Your first repository is now created! However there is no code in it yet. Scroll down on the page to where it says "...or push an existing repository from the command line." That's what we want.



GitHub Hello, World repository

Copy the text immediately under this headline and paste it into your command line. Note that my username is `wsvincent` here; yours will be different so if you copy my snippet below it won't work! This syncs the local directory on our computer with the remote repository on the GitHub website.

Command Line

```
(helloworld) $ git remote add origin https://github.com/wsvincent/hello-world.git
```

The last step is to "push" our code to GitHub.

Command Line

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

Hopefully this command works and you can go back to your GitHub page and refresh it to see your local code now hosted online.

SSH Keys

Unfortunately, there is a good chance that the last command yielded an error if you are a new developer and do not have SSH keys already configured.

Command Line

```
ERROR: Repository not found.
fatal: Could not read from remote repository.

Please make sure you have the correct access rights
and the repository exists.
```

This cryptic message means we need to configure SSH keys. This is a one-time thing but a bit of a hassle to be honest.

SSH is a protocol used to ensure private connections with a remote server. Think of it as an additional layer of privacy on top of username/password. The process involves generating unique SSH keys and storing them on your computer so only GitHub can access them.

First, check whether you have existing SSH keys. Github has a guide to this that works for Mac, Windows, and Linux. If you *don't* have existing public and private keys, you'll need to generate them. GitHub, again, has a guide on doing this.

Once complete you should be able to execute the `git push -u origin master` command successfully!

It's normal to feel overwhelmed and frustrated if you become stuck with SSH keys. GitHub has a lot of resources to walk you through it but the reality is its very intimidating the first time. If you're truly stuck, continue with the book and come back to SSH Keys and GitHub with a full nights sleep. I can't count the number of times a clear head has helped me process a difficult programming issue.

Assuming success with GitHub, go ahead and exit our virtual environment with the `exit` command.

Command Line

```
(helloworld) $ exit
```

You should no longer see 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 Django web server. And we worked with Git to track our changes and pushed our code into a private repo on GitHub.

Continue on to **Chapter 3: Pages 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 will build, test, and deploy a *Pages* app with a homepage and about page. We'll 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 Set Up

As in **Chapter 2: Hello World App**, our initial set up involves the following steps:

- create a directory for our code
- install Django in a new virtual environment
- create a new Django project
- create a new `pages` app
- update `config/settings.py`

On the command line make sure you're not working in an existing virtual environment. If there is text before the dollar sign ($) in parentheses, then you are! Make sure to type `exit` to leave it.

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

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

Command Line

```
$ cd ~/Desktop
$ mkdir pages && cd pages
$ pipenv install django~=3.1.0
$ pipenv shell
(pages) $ django-admin startproject config .
(pages) $ python manage.py startapp pages
```

Open your text editor and navigate to the file `config/settings.py`. Add the `pages` app at the bottom of the `INSTALLED_APPS` setting:

Code

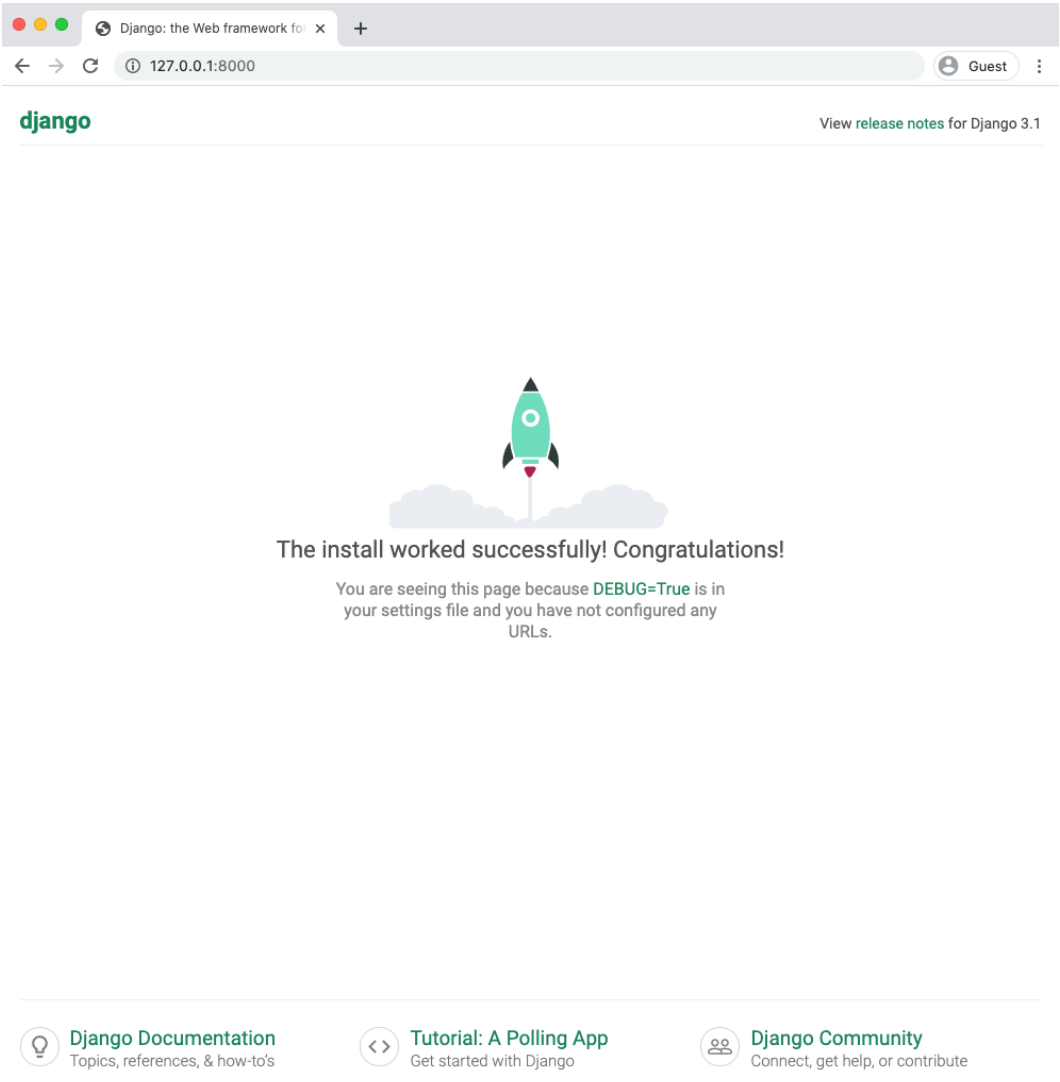
```
# config/settings.py
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'pages', # new
]
```

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 and in Django the approach is to use *templates*: individual HTML files that can be linked together and also include

basic logic.

Recall that in the previous chapter our "Hello, World" site had the phrase hardcoded into a `views.py` file as a string. That technically works but doesn't scale well! A better approach is to link a view to a template, thereby separating the information contained in each.

In this chapter we'll learn how to use templates to more easily create our desired homepage and about page. And in future chapters, the use of templates will support building websites that can support hundreds, thousands, or even millions of webpages with a minimal amount of code.

The first consideration is where to place templates within the structure of a Django project. There are two options. By default, Django's template loader will look within each app for related templates. However the structure is somewhat confusing: each app needs a new `templates` directory, another directory with the same name as the app, and then the template file.

Therefore, in our `pages` app, Django would expect the following layout:

Layout

```
└── pages
    ├── 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`.

Why this seemingly repetitive approach? The short answer is that the Django template loader wants to be really sure it finds the correct template! What happens if there are `home.html` files within two separate apps? This structure makes sure there are no such conflicts.

There is, however, another approach which is to instead create a single project-level `templates` directory and place *all* templates within there. By making a small tweak to our `config/settings.py` file we can tell Django to *also* look in this directory for templates. That is the approach we'll use.

First, quit the running server with the `Control+c` command. Then create a directory called `templates` and an HTML file called `home.html`.

Command Line

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

Next we need to update `config/settings.py` to tell Django the location of our new `templates` directory. This is a one-line change to the setting `'DIRS'` under `TEMPLATES`.

Code

```
# config/settings.py
TEMPLATES = [
    {
        ...
        'DIRS': [str(BASE_DIR.joinpath('templates'))], # new
        ...
    },
]
```

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 files.

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, `HomePageView`, 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 `config/urls.py` file to point at our `pages` app and then within `pages` we match views to URL routes.

Let's start with the `config/urls.py` file.

Code

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

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

The code here should look familiar 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
```

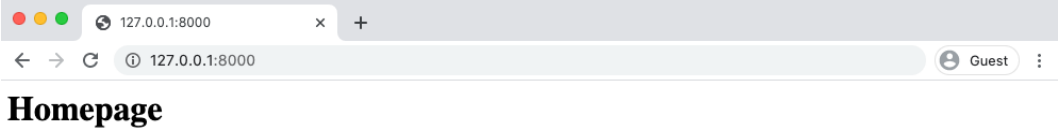
And add the following code.

Code

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

urlpatterns = [
    path('', 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! Start up the local web server with `python manage.py runserver` and navigate to `http://127.0.0.1:8000/` to see our new homepage.



Homepage

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

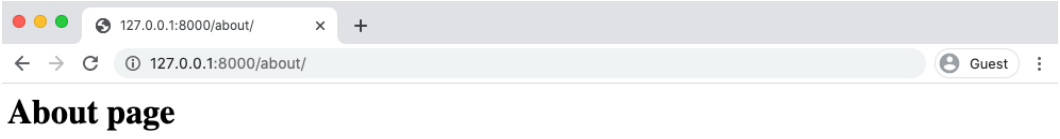
```
# pages/views.py
from django.views.generic import TemplateView


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


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

And then import the view name and connect it to a URL at `about/`.

Code

```
# pages/urls.py
from django.urls import path
from .views import HomePageView, AboutPageView # new

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

Start up the web server with `python manage.py runserver`. Navigate to `http://127.0.0.1:8000/about` and the new About page is visible.



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. We could name this file anything but using `base.html` is a common convention. Type `Control+c` and then create the new file.

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 two routes in `pages/urls.py`? This is why. The `url` tag uses these names to automatically create links for us.

The URL route for our homepage is called `home`. To configure a link to it we use the following syntax: `{% url 'home' %}`.

Code

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

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

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

Let's update our `home.html` and `about.html` files 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

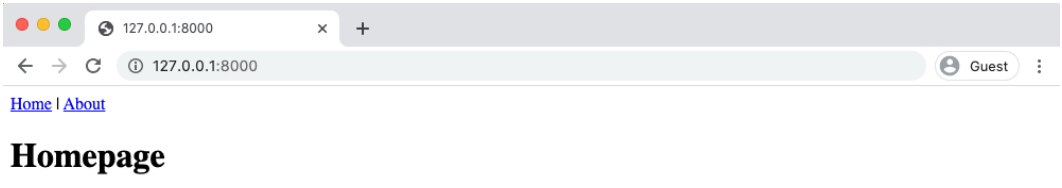
```
<!-- templates/home.html -->
{% extends 'base.html' %}

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

Code

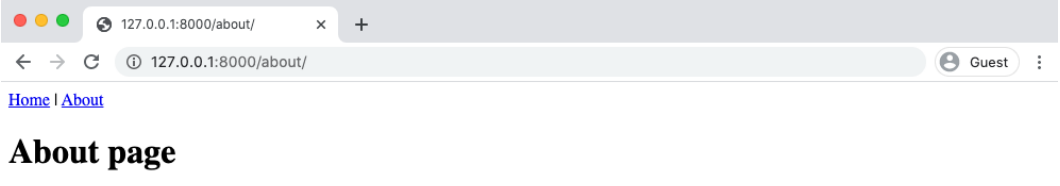
```
<!-- templates/about.html -->
{% extends 'base.html' %}

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

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 with header



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 Django co-creator Jacob Kaplan-Moss, "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.

And fortunately, Django comes with 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
System check identified no issues (0 silenced).
..
----------------------------------------------------------------------
Ran 2 tests in 0.014s

OK
```

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 GitHub

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

Command Line

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

Use `git status` to see all our code changes and 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 GitHub create a new repo called `pages-app` and make sure to select the "Private" radio button. Then click on the "Create repository" button.

On the next page, scroll down to where it says "...or push an existing repository from the command line." Copy and paste the two commands there into your terminal.

It should look like the below albeit instead of `wsvincent` as the username it will be your GitHub username.

Command Line

```
(pages) $ git remote add origin https://github.com/wsvincent/pages-app.git
(pages) $ git push -u origin master
```

Local vs Production

To make our site available on the Internet where everyone can see it, we need to deploy our code to an external server and database. This is called putting our code into *production*. Local code lives only on our computer; production code lives on an external server available to everyone.

The `startproject` command creates a new project configured for local development via the file `config/settings.py`. This ease-of-use means when it does come time to push the project into production, a number of settings have to be changed.

One of these is the web server. Django comes with its own basic server, suitable for local usage, but it *is not* suitable for production. There are two options available: Gunicorn and uWSGI. Gunicorn is the simpler to configure and more than adequate for our projects so that will be what we use.

For our hosting provider 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.

Now we need to install Heroku's *Command Line Interface* (*CLI*) so we can deploy from the command line. Currently, we are operating within a virtual environment for our *Pages* project but we want Heroku available globally, that is everywhere on our machine. An easy way to do so is open up a new command line tab–`Command+t` on a Mac, `Control+t` on Windows–which is not operating in a virtual environment. Anything installed here will be global.

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

Command Line

```
$ brew install heroku/brew/heroku
```

On Windows, see the Heroku CLI page to correctly install either the 32-bit or 64-bit version. If you are using Linux there are specific install instructions available on the Heroku website.

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@learndjango.com
Password: ********************************
Logged in as will@learndjango.com
```

Deployment typically requires a number of discrete steps. It is common to have a "checklist" for these since there quickly become too many to remember. At this stage, we are intentionally keeping things basic so there are only two additional steps required, however this list will grow in future projects as we add additional security and performance features.

Here is the deployment checklist:

- install `Gunicorn`
- add a `Procfile` file
- update `ALLOWED_HOSTS`

`Gunicorn` can be installed using Pipenv.

Command Line

```
(pages) $ pipenv install gunicorn==19.9.0
```

The `Procfile` is a Heroku-specific file that can include multiple lines but, in our case, will have just one specifying the use of Gunicorn as the production web server. Create the `Procfile` now.

Command Line

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

Open the `Procfile` with your text editor and add the following line.

Procfile

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

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. For now, we'll use the wildcard asterisk, `*`, which means all domains are acceptable. Later in the book we'll see how to explicitly list the domains that should be allowed.

Within the `config/settings.py`, scroll down to `ALLOWED_HOSTS` and add a `'*'` so it looks as follows:

Code

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

That's it. Keep in mind we've taken a number of security shortcuts here but the goal is to push our project into production in as few steps as possible.

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 GitHub so we have an online backup of our code changes.

Command Line

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

Deployment

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
- disable the collection of static files (we'll cover this in a later chapter)
- push the code up to Heroku
- 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 `fathomless-hamlet-26076`. Your name will be different.

Command Line

```
(pages) $ heroku create
Creating app... done, ⬢ fathomless-hamlet-26076
https://fathomless-hamlet-26076.herokuapp.com/ |
https://git.heroku.com/fathomless-hamlet-26076.git
```

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.

Command Line

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

If we had just typed `git push origin master`, then the code would have been pushed to GitHub, 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

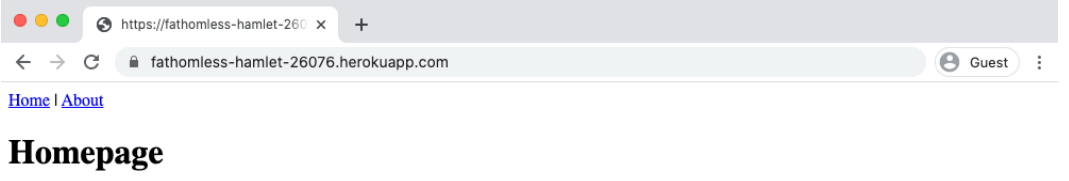
```
(pages) $ heroku ps:scale web=1
```

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://fathomless-hamlet-26076.herokuapp.com/`.



Homepage on Heroku

You do not have to log out or exit from your Heroku app. It will continue running at this free tier on its own, though you should type `exit` to leave the local virtual environment and be ready for the next chapter.

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, a *Message Board* website, 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 GitHub and deploy the app on Heroku.

Thanks to the powerful Django ORM (Object-Relational Mapper), there is built-in support for multiple database backends: PostgreSQL, MySQL, MariaDB, Oracle, or SQLite. This means that we, as developers, can write the same Python code in a `models.py` file and it will automatically be translated into each database correctly. The only configuration required is to update the DATABASES section of our `config/settings.py` file. This is truly an impressive feature!

For local development, Django defaults to using SQLite because it is file-based and therefore far simpler to use than the other database options, which require a dedicated server to be running separate from Django itself.

Initial Set Up

Since we've already set up several Django projects at this point in the book, we can quickly run through the standard 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 `config`
- create a new app call `posts`
- update `config/settings.py`

In a new command line console, enter the following commands:

Command Line

```
$ cd ~/Desktop
$ mkdir mb && cd mb
$ pipenv install django~=3.1.0
$ pipenv shell
(mb) $ django-admin startproject config .
(mb) $ python manage.py startapp posts
```

Next we must alert Django to the new app, `posts`, by adding it to the top of the `INSTALLED_APPS` section of our `config/settings.py` file.

Code

```
# config/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
Pipfile    Pipfile.lock    config    db.sqlite3    manage.py    posts
```

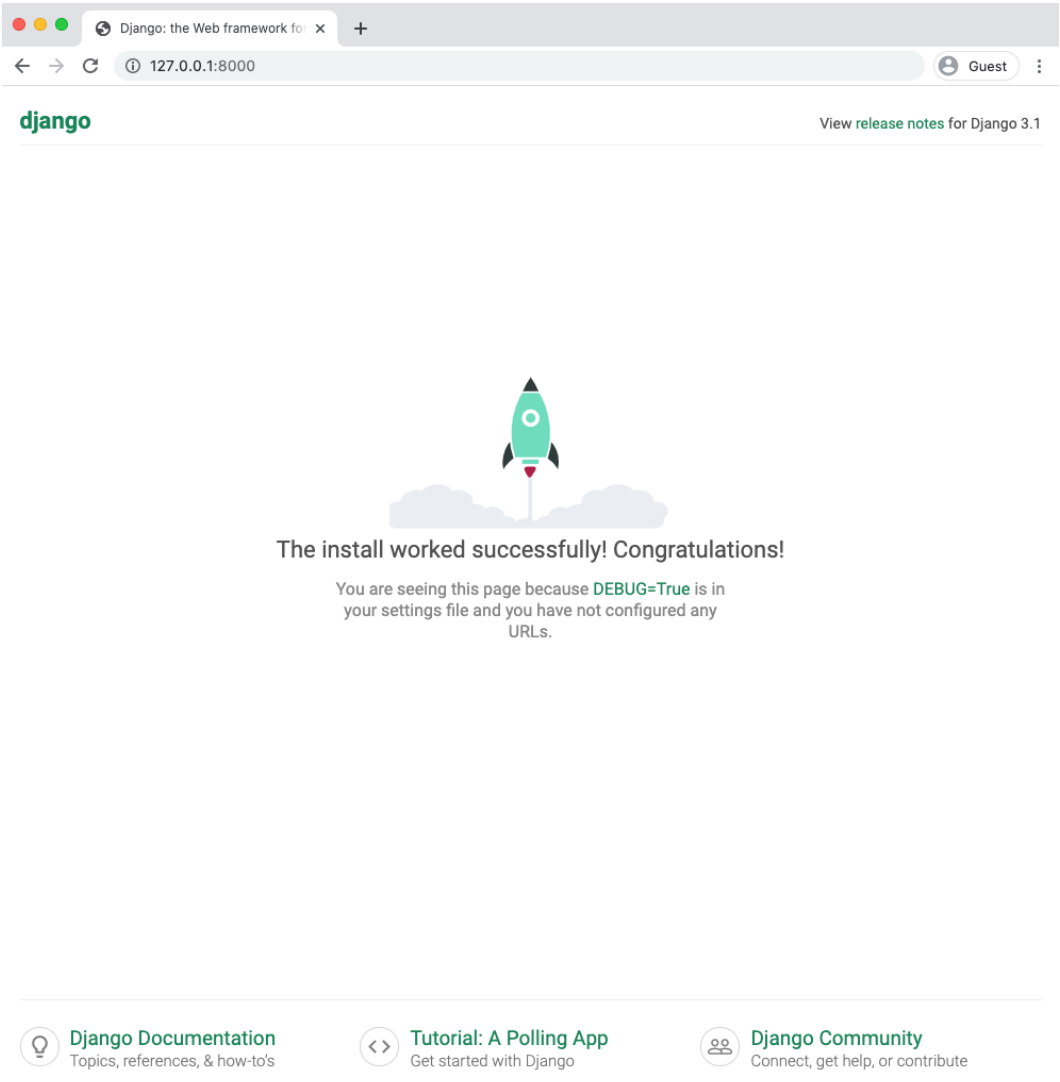
> Technically, a `db.sqlite3` file is created the first time you run *either* `migrate` or `runserver`, however `migrate` will sync the database with the current state of any database models contained in the project and listed in `INSTALLED_APPS`. In other words, to make sure the database reflects the 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
```

In your web browser, navigate to `http://127.0.0.1:8000/` to see the familiar Django welcome 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's ORM will automatically turn this model into a database table for us. In a real-world

Django project, there are often 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 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 migrations file with the `makemigrations` command. Migration files create a reference of any changes to the database models which means we can track changes–and debug errors as necessary–over time.
2. Second, we build the actual database with the `migrate` command which executes the instructions in our migrations file.

Make sure the local server is stopped by typing `Control+c` on the command line and then run the commands `python manage.py makemigrations posts` and `python manage.py migrate`.

Command Line

```
(mb) $ python manage.py makemigrations posts
Migrations for 'posts':
  posts/migrations/0001_initial.py
    - Create model Post
(mb) $ python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, posts, sessions
Running migrations:
  Applying posts.0001_initial... OK
```

Note that you don't *have* to include a name after `makemigrations`. If you simply run `python manage.py makemigrations`, a migrations file will be created for *all* available changes throughout the Django project. That is fine in a small project such as ours with only a single app, but most Django projects have more than one app! Therefore ,if you made model changes in multiple apps the resulting migrations file would include *all* those changes! This is not ideal. Migrations file should be as small and concise as possible as this makes it easier to debug in the future or even roll back changes as needed. Therefore, as a best practice, adopt the habit of always including the name of an app when executing the `makemigrations` command!

Django Admin

One of Django's killer features is its robust built-in admin interface that provides a visual way to interact with data. It came about because Django was originally built as a newspaper CMS (Content Management System). The idea was that journalists could write and edit their stories

in the admin 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.

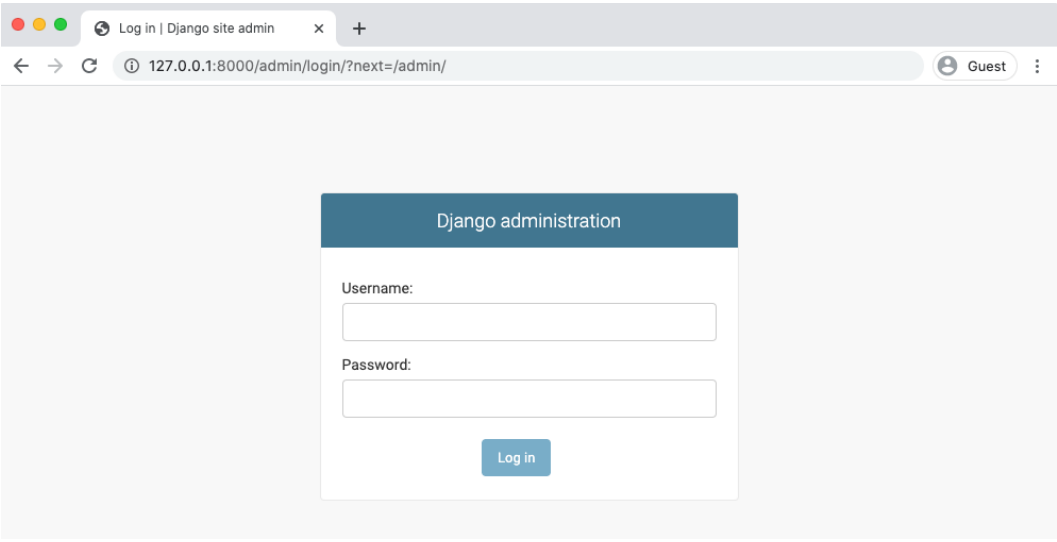
To use the Django admin, we first need to create a `superuser` who can log in. In your command line console, type `python manage.py createsuperuser` and respond to the prompts for a username, email, and password:

Command Line

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

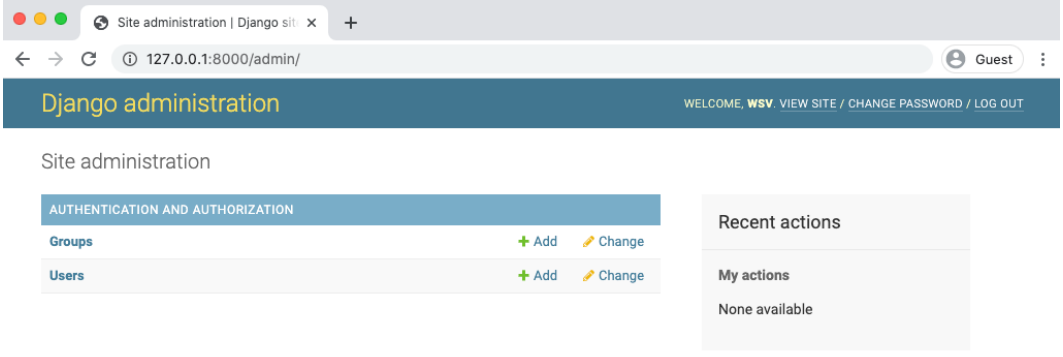
> When you type your password, it will not appear visible in the command line console for security reasons.

Restart the Django server with `python manage.py runserver` and in your web browser go to `http://127.0.0.1:8000/admin/`. You should see the log in screen for the admin:



Admin login page

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



Admin homepage

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

Just as we must explicitly add new apps to the `INSTALLED_APPS` config, so, too, must we update an app's `admin.py` file for it to appear in the admin.

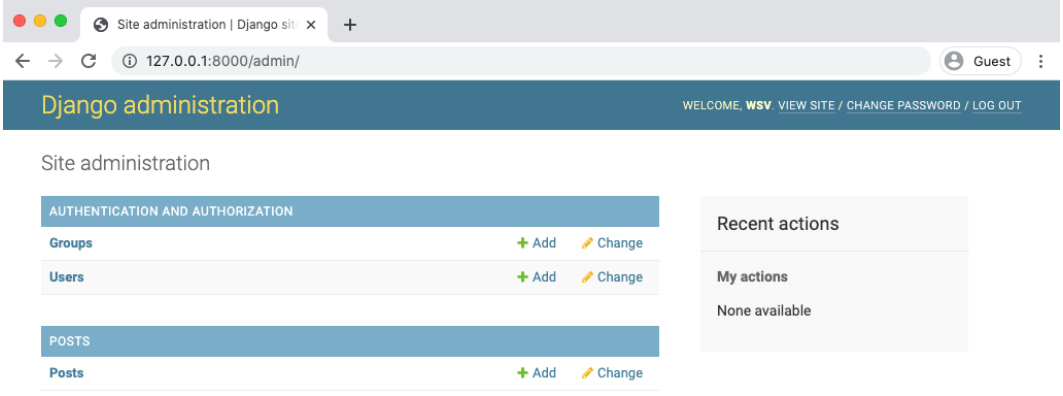
In your text editor open up `posts/admin.py` and add the following code so that the Post model

is displayed.

Code

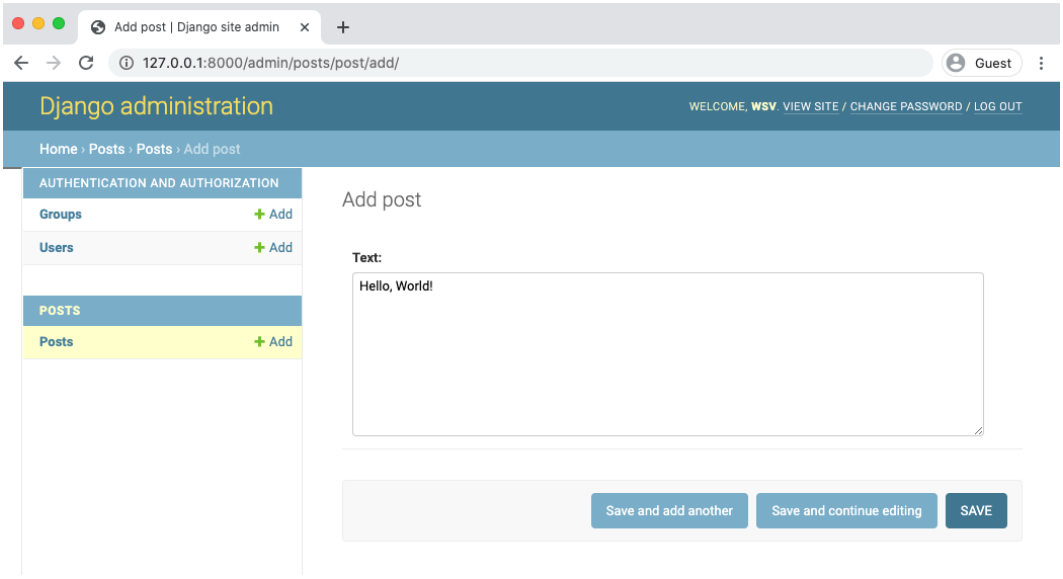
```
# posts/admin.py
from django.contrib import admin

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 appears:



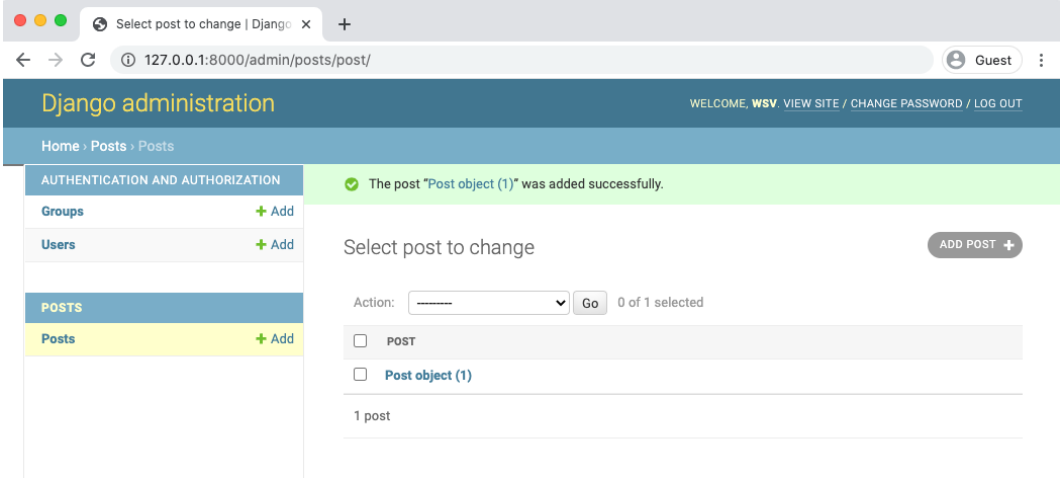
Admin homepage updated

Let's create our first message board post for our database. Click on the `+ Add` button opposite `Posts` and enter your own content in the `Text` form field.



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 descriptive!



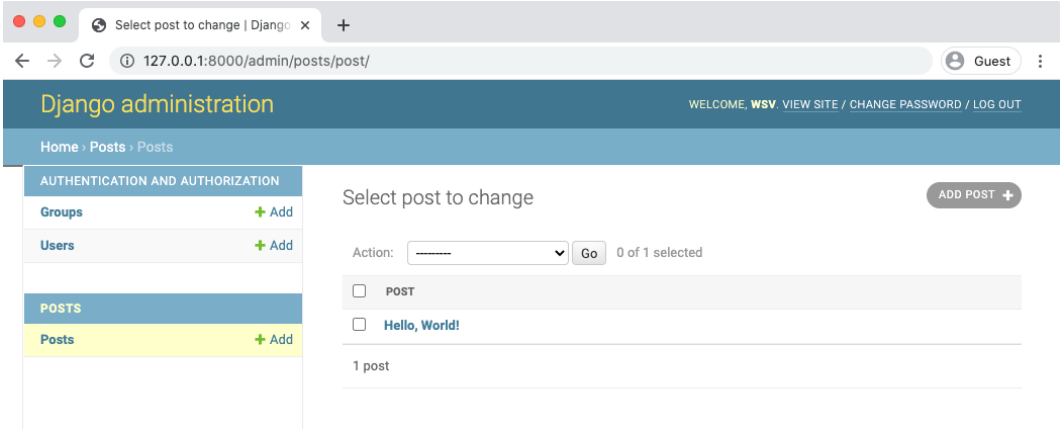
Admin new entry

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):
        return self.text[:50]
```

This will display the first 50 characters of the `text` field. 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'
```

On the first line we're importing `ListView` and in the second line we import the `Post` model. In the view, `HomePageView`, we subclass `ListView` and specify the correct model and 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 new directory called `templates` and within it a `home.html` template file.

Command Line

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

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

Code

```
# config/settings.py
TEMPLATES = [
    {
        ...
        'DIRS': [str(BASE_DIR.joinpath('templates'))], # new
        ...
    },
]
```

ListView automatically returns to us a context variable called `object_list` that we can loop over via the built-in for template tag. We'll create our own variable called `post` and can then access the desired field we want displayed, `text`, as `post.text`.

Code

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

The name `object_list` isn't very friendly, is it? Instead we can provide an explicit name via context_object_name attribute. Django is, as ever, eminently customizable.

Back in our `posts/views.py` file add the following:

Code

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


class HomePageView(ListView):
    model = Post
    template_name = 'home.html'
    context_object_name = 'all_posts_list' # new
```

Adding an explicit name in this way makes it easier for other members of a team, for example a designer, to understand and reason about what is available in the template context.

Don't forget to update our template, too, so that it references `all_posts_list` rather than `object_list`.

Code

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

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

Code

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

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

Then create a `urls.py` file within the `posts` app.

Command Line

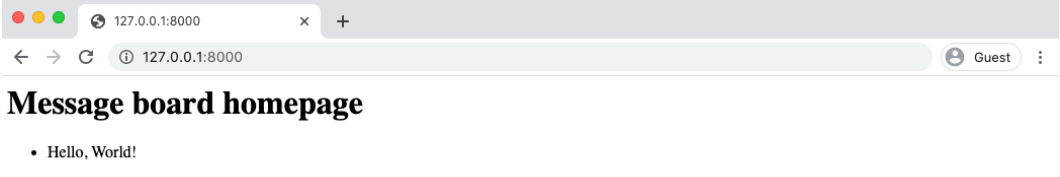
```
(mb) $ touch posts/urls.py
```

And update it like so:

Code

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

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

Restart the server with `python manage.py runserver` and navigate to our homepage, which now lists out our message board post.

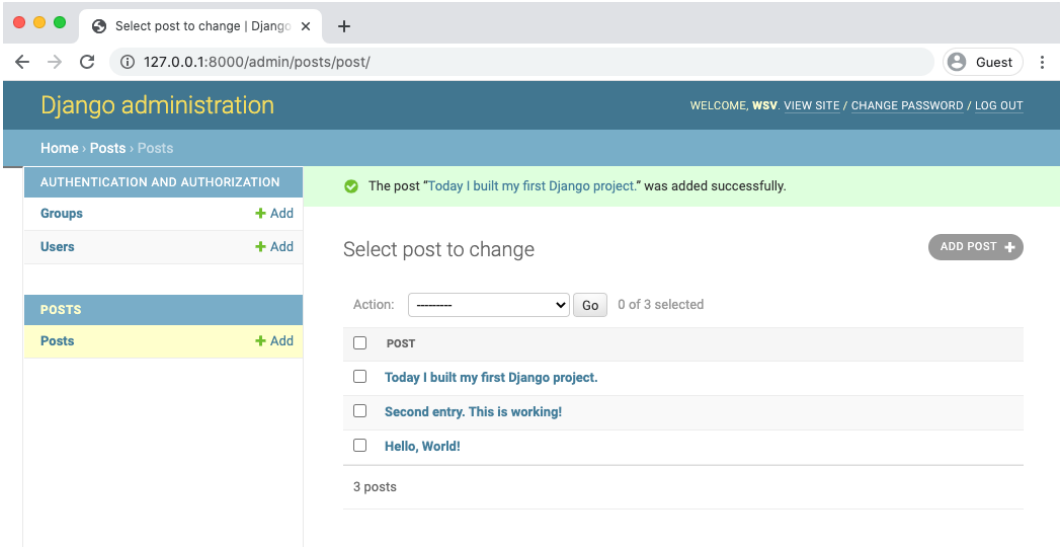


Homepage with posts

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.

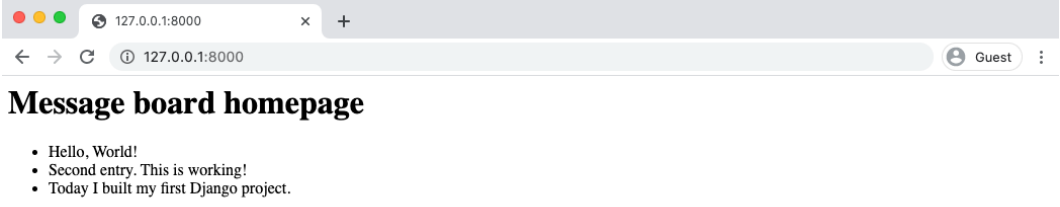
Adding New Posts

To add new posts to our message board, go back into the Admin and create two more posts. Here's what mine look like:



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 that which is a must safer and more performant approach.

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 the phrase `test_` so that 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 imported the `TestCase` module which lets us create a sample database and our `Post` model. We created a new class, `PostModelTest`, and added a `setUp` method to create a new database that has just one entry: a post with a text field containing the string 'just a test.'

Then we can run our first test, `test_text_content`, to check that the database field actually contains `just a test`. We created 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 next line uses f strings, a very cool addition to Python 3.6, which 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 assertEqual 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 command `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 you write are actually quite repetitive.

Time for our next group of tests. The first test looked at the model but now we want evaluate the homepage itself:

- does it actually exist and return a HTTP 200 response?
- does it use `HomePageView` as the view?

- does it use `home.html` as the template?

We can include all of these tests in a new class called `HomePageViewTest`. Note that rather than access the view name directly we will instead import reverse and refer to the named URL of `home`. Why do it this way? URL schemes can and do change over the course of a project, but the named URL likely will not so this is a way to future-proof your tests.

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

Code

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


class PostModelTest(TestCase):
    ...

class HomePageViewTest(TestCase): # new

    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 0.015s

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

Why does the output say four tests and not six? The answer is that our `setUp` methods are not actually tests, they are helper functions. Only functions that start with the name `test*` and exist in a `tests.py` file will be run as tests when we execute the `python manage.py test` command.

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"
```

GitHub

We also need to store our code on GitHub. You should already have a GitHub account from previous chapters so go ahead and create a new repo called `mb-app`. Select the "Private" radio button.

On the next page scroll down to where it says "â€¦or push an existing repository from the command line." Copy and paste the two commands there into your terminal, which should look like like the below after replacing `wsvincent` (my username) with your GitHub username:

Command Line

```
(mb) $ git remote add origin https://github.com/wsvincent/mb-app.git
(mb) $ git push -u origin master
```

Heroku Configuration

You should already have a Heroku account set up and installed from Chapter 3. Our deployment checklist contains the same three items:

- install `Gunicorn`
- add a `Procfile`
- update `ALLOWED_HOSTS`

`Gunicorn` can be installed via Pipenv.

Command Line

```
(mb) $ pipenv install gunicorn==19.9.0
```

Then create a `Procfile` with Heroku instructions.

Command Line

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

Add one line instructing Heroku to use `Gunicorn` as our production web server.

Procfile

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

Previously, we set `ALLOWED_HOSTS` to `*`, meaning accept all hosts, but that was a dangerous shortcut. We can, and should be, more specific. The two local hosts Django runs on are `localhost:8000` and `127.0.0.1:8000`. We also know, having deployed on Heroku previously, that any Heroku site will end with `.herokuapp.com`. We can add all three routes to our config.

Code

```
# config/settings.py
ALLOWED_HOSTS = ['.herokuapp.com', 'localhost', '127.0.0.1']
```

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

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.

Command Line

```
(mb) $ heroku create
Creating app... done, â¬¢ sleepy-brook-64719
https://sleepy-brook-64719.herokuapp.com/ |
https://git.heroku.com/sleepy-brook-64719.git
```

For now, continue to instruct Heroku to ignore static files. We'll cover them in the next section while deploying our *Blog* app.

Command Line

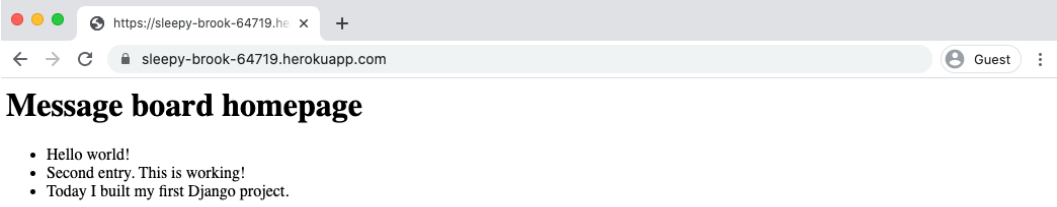
```
(mb) $ heroku config:set DISABLE_COLLECTSTATIC=1
```

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
```

You can open the URL of the new project from the command line by typing `heroku open` which will launch a new browser window. For example, mine can be seen below:



Live site

To finish up, exit our virtual environment by typing `exit` on the command line.

Conclusion

We've now built, tested, and deployed our first database-driven app. While it's deliberately quite basic, we learned how to create a database model, update it with the admin panel, and then display the contents on a web page. That's the good news. The bad news is that if you check your Heroku site in a few days, it's likely whatever posts you've added will be deleted! This is related to how Heroku handles SQLite, but really it's an indication that we should be using a production database like PostgreSQL for deployments, even if we still want to stick with SQLite locally. This is covered in Chapter 16!

In the next section, we'll learn how to deploy with PostgreSQL and build a *Blog* application so that users can create, edit, and delete posts on their own. No admin access required! We'll also add styling via CSS so the site looks better.

Chapter 5: Blog App

In this chapter we willll 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 blog post. We'll also introduce CSS for styling and learn how Django works with static files.

Initial Set Up

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 `config`
- create a new app `blog`
- perform a migration to set up the database
- update `config/settings.py`

Command Line

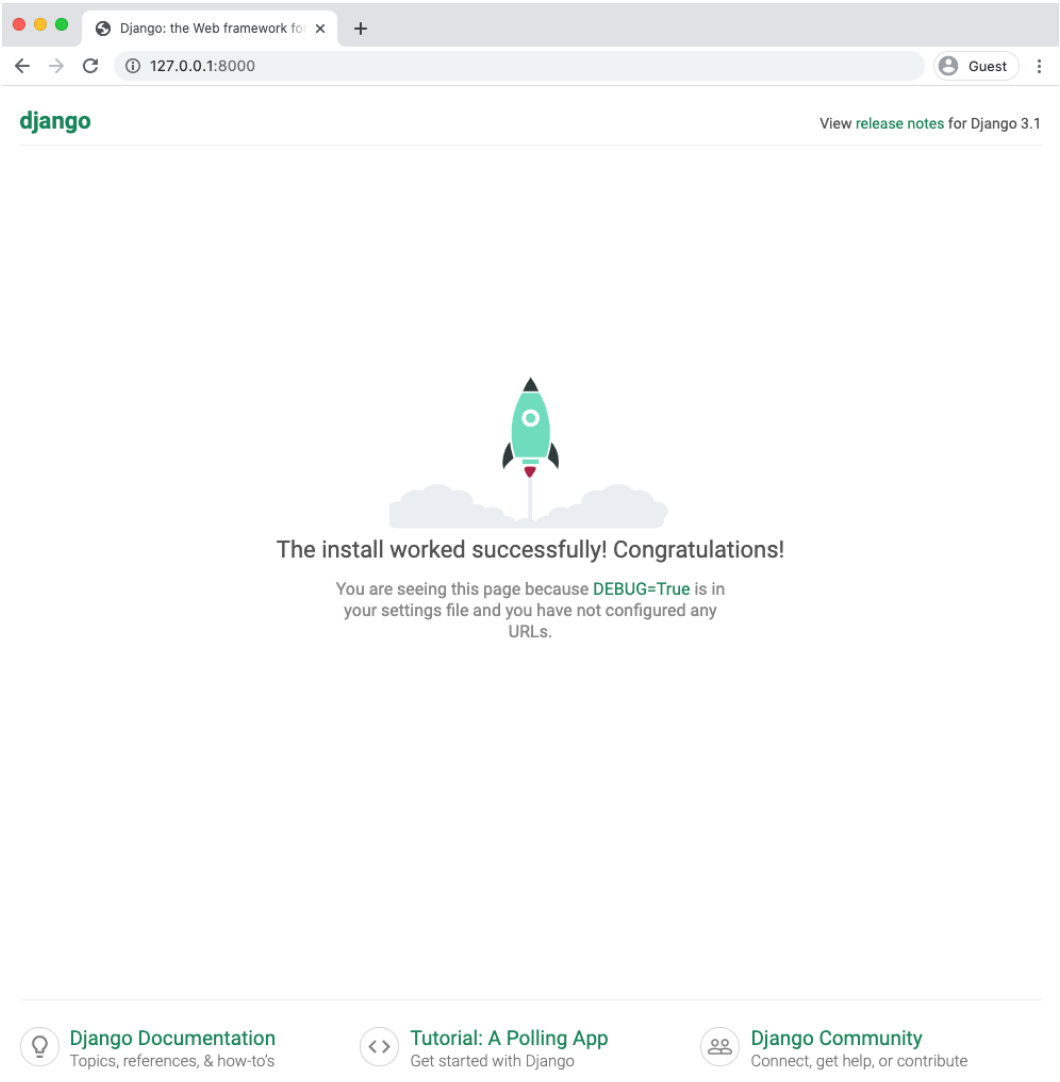
```
$ cd ~/Desktop
$ mkdir blog
$ cd blog
$ pipenv install django~=3.1.0
$ pipenv shell
(blog) $ django-admin startproject config .
(blog) $ python manage.py startapp blog
(blog) $ python manage.py migrate
(blog) $ python manage.py runserver
```

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

Code

```
# config/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 Django welcome 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. Stop the server with `Control+c`. 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

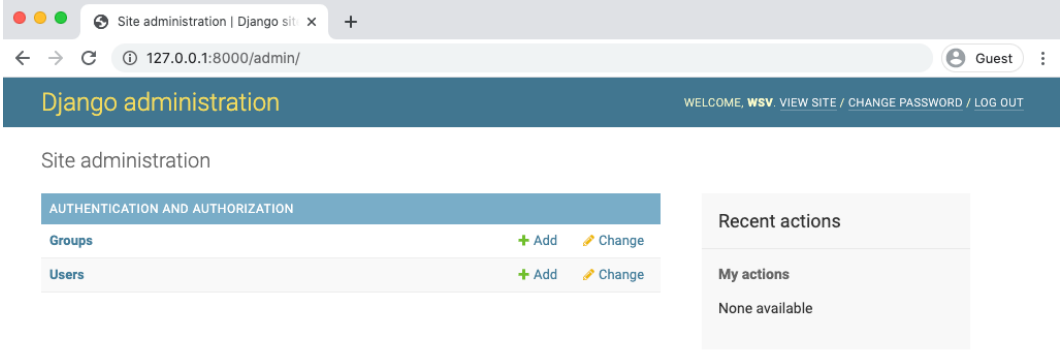
We need a way to access our data. Enter the Django admin! First, create a superuser account by typing the command below and following the prompts to set up an email and password. Note that when typing your password, it will not appear on the screen for security reasons.

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 navigate over to the admin at `127.0.0.1:8000/admin/`. Log in with your new superuser account.

Oops! Where's our new `Post` model?



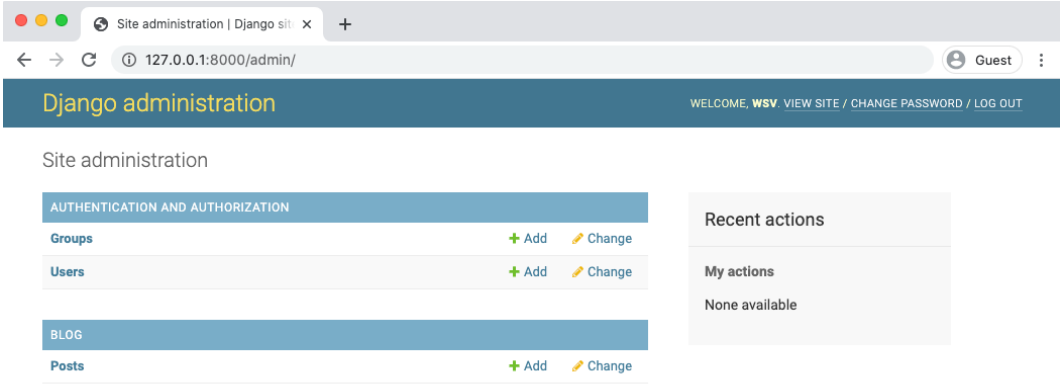
Admin homepage

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

Code

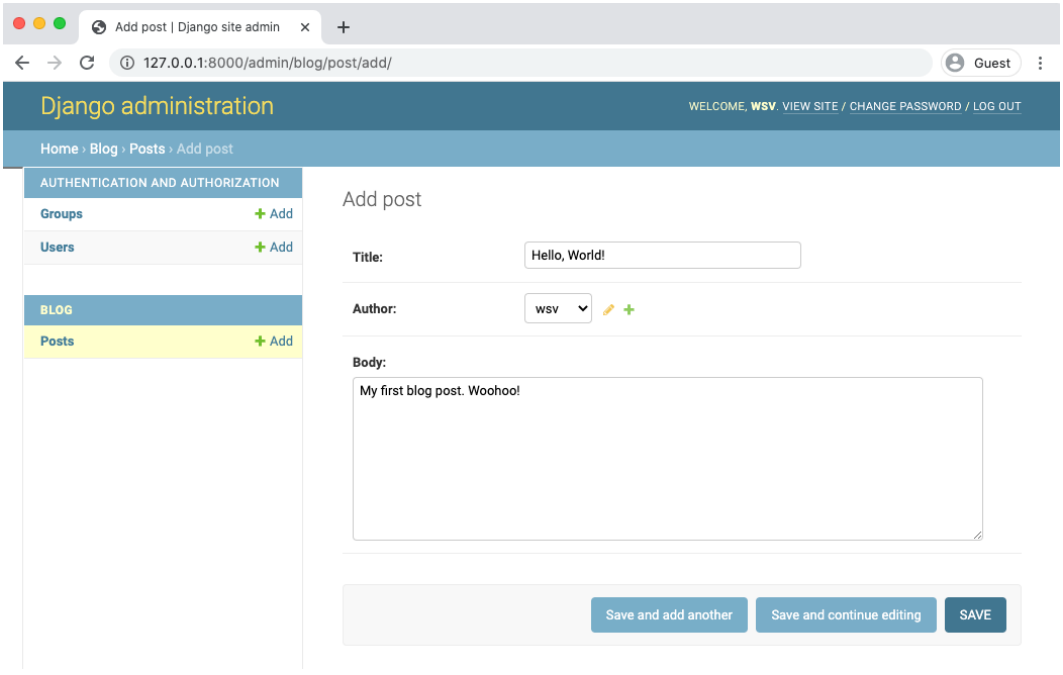
```
# blog/admin.py
from django.contrib import admin
from .models import Post

admin.site.register(Post)
```

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

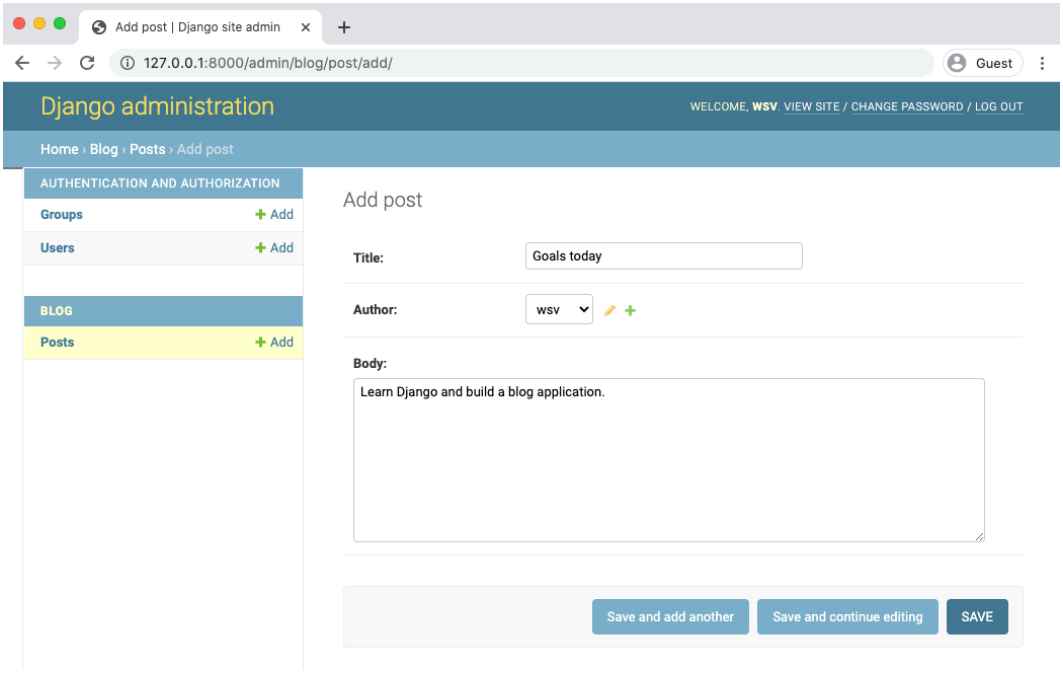


Admin homepage

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.

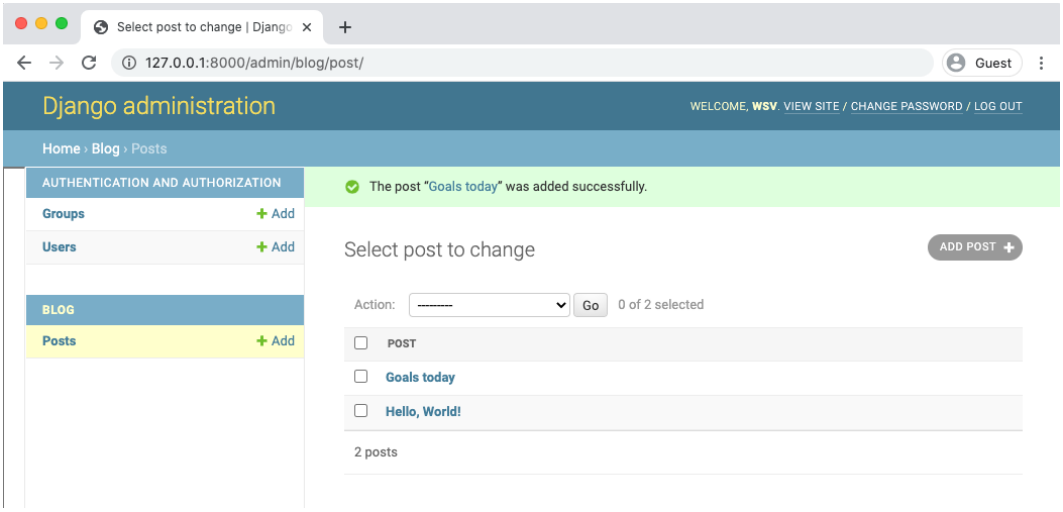


Admin first post



Admin second post

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.



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 `config/urls.py` file and then our app-level `blog/urls.py` file to achieve this.

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 .views import BlogListView

urlpatterns = [
    path('', 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 `config/urls.py` file so that it knows to forward all requests directly to the `blog` app.

Code

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

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

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 you 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 new `templates` directory with the two template files.

Command Line

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

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

Code

```
# config/settings.py
TEMPLATES = [
    {
        ...
        'DIRS': [str(BASE_DIR.joinpath('templates'))], # new
        ...
    },
]
```

Then update the `base.html` template as follows.

Code

```
<!-- templates/base.html -->
<html>
  <head>
    <title>Django blog</title>
  </head>
  <body>
    <header>
      <h1><a href="{% url 'home' %}">Django blog</a></h1>
    </header>
    <div>
      {% block content %}
      {% endblock content %}
    </div>
  </body>
</html>
```

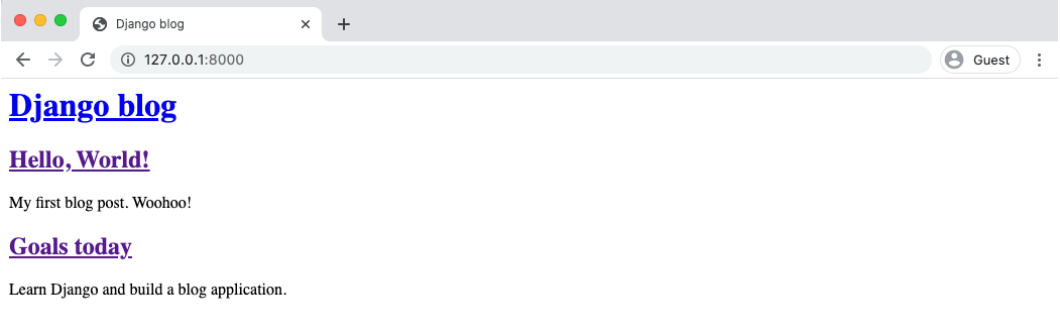
Note that code between `{% block content %}` and `{% endblock content %}` can be filled by other templates. Speaking of which, here is the code for `home.html`.

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 wrap 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 with `python manage.py runserver` and refresh the homepage we can see it is working.



Blog homepage with two posts

But it looks terrible. Let's fix that!

Static Files

We need to add some CSS to our project to improve the styling. CSS, JavaScript, and images are a core piece of any modern web application and within the Django world are referred to as "static

files." Django provides tremendous flexibility around *how* these files are used, but this can lead to quite a lot of confusion for newcomers.

By default, Django will look within each app for a folder called `static`. In other words, a folder called `blog/static/`. If you recall, this is similar to how `templates` are treated as well.

As Django projects grow in complexity over time and have multiple apps, it is often simpler to reason about static files if they are stored in a single, project-level directory instead. That is the approach we will take here.

Quit the local server with `Control+c` and create a new directory called `static` in the same folder as the `manage.py` file.

Command Line

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

Then we need to tell Django to look for this new folder when loading static files. If you look at the bottom of the `config/settings.py` file, there is already a single line of configuration:

Code

```
# config/settings.py
STATIC_URL = '/static/'
```

STATIC_URL is the *URL location* of static files in our project, aka at `/static/`.

By configuring STATICFILES_DIRS, we can tell Django where to look for static files beyond just `app/static` folder. In your `config/settings.py` file, at the bottom, add the following new line which tells Django to look within our newly-created `static` folder for static files.

Code

```
# config/settings.py
STATIC_URL = '/static/'
STATICFILES_DIRS = [str(BASE_DIR.joinpath('static'))] # new
```

Next create a `css` directory 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;
}
```

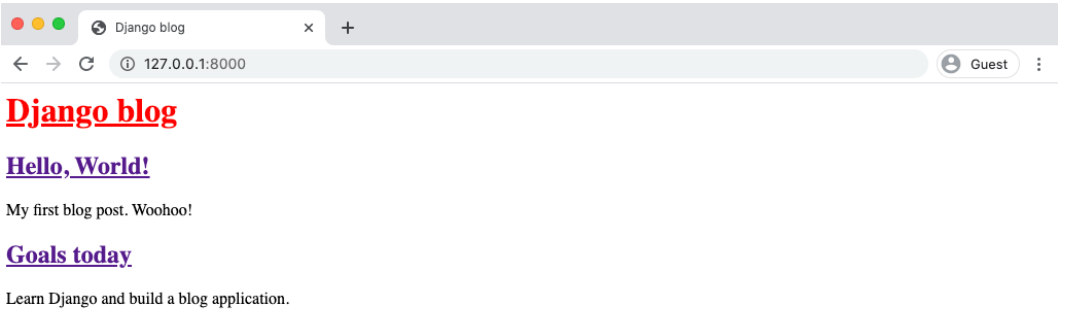
Last step now. We need to add the static files to our templates by adding `{% load static %}` to the top of `base.html`. Because our other templates inherit from `base.html`, we only have to add this once. Include a new line at the bottom of the `<head></head>` code that explicitly references our new `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 hassle. Now we can add static files to our `static` directory 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="styleshee\
t">
  <link href="{% static 'css/base.css' %}" rel="stylesheet">
</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.

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 # new
from .models import Post


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


class BlogDetailView(DetailView): # new
    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, which would be `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, it's possible to explicitly name the context object in our view using context_object_name.

The "magic" naming of the context object is a price you pay for the ease and simplicity of using generic views, which are great if you know what they're doing but take a little research in the official documentation to customize.

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

Code

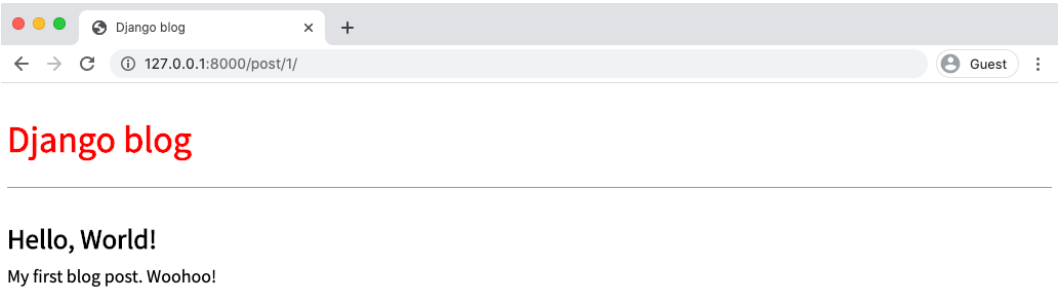
```
# blog/urls.py
from django.urls import path
from .views import BlogListView, BlogDetailView # new

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

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/`.

> 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` you'll see a dedicated page for our first blog post at `http://127.0.0.1:8000/post/1/`.



Blog post one detail

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: `<a href="">`. Update it to `<a href="{% url 'post_detail' post.pk %}">`.

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` and `DetailView`.

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 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/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')
```

There's a lot that's new in these tests so we'll walk through them slowly. At the top we import both get_user_model to reference our active `User` and `TestCase` which we've seen before.

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

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

Git

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

Command Line

```
(blog) $ 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

```
(blog) $ git status
(blog) $ git add -A
(blog) $ 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 a ubiquitous part of the modern web but they are very complicated to implement correctly. Any time you accept 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.

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 `<a href="{% url 'post_new' %}"></a>` where `post_new` is the name for our URL.

Your updated file should look as follows:

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 href="{% static 'css/base.css' %}" rel="stylesheet">
  </head>
  <body>
    <div>
      <header>
        <div class="nav-left">
          <h1><a href="{% url 'home' %}">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. Import our not-yet-created view called `BlogCreateView` at the top. And then make the URL which will start with `post/new/` and be named `post_new`.

Code

```
# blog/urls.py
from django.urls import path
from .views import BlogListView, BlogDetailView, BlogCreateView # new

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

Simple, right? It's the same url, views, template pattern we've seen before.

Now let's create our view by importing a new generic class called `CreateView` at the top 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 # new
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): # new
    model = Post
    template_name = 'post_new.html'
    fields = ['title', 'author', 'body']
```

Within `BlogCreateView` we specify our database model `Post`, the name of our template `post_new.html`. For `fields` we explicitly set the database fields we want to expose which are `title`, `author`, and `body`.

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 content %}
```

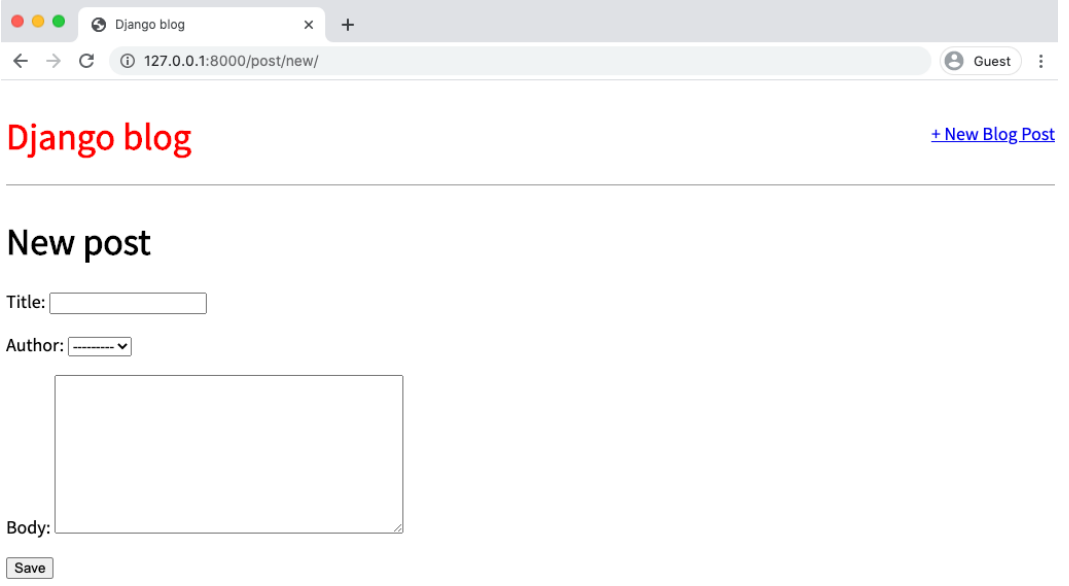
Let's breakdown what we've done:

- 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/`.

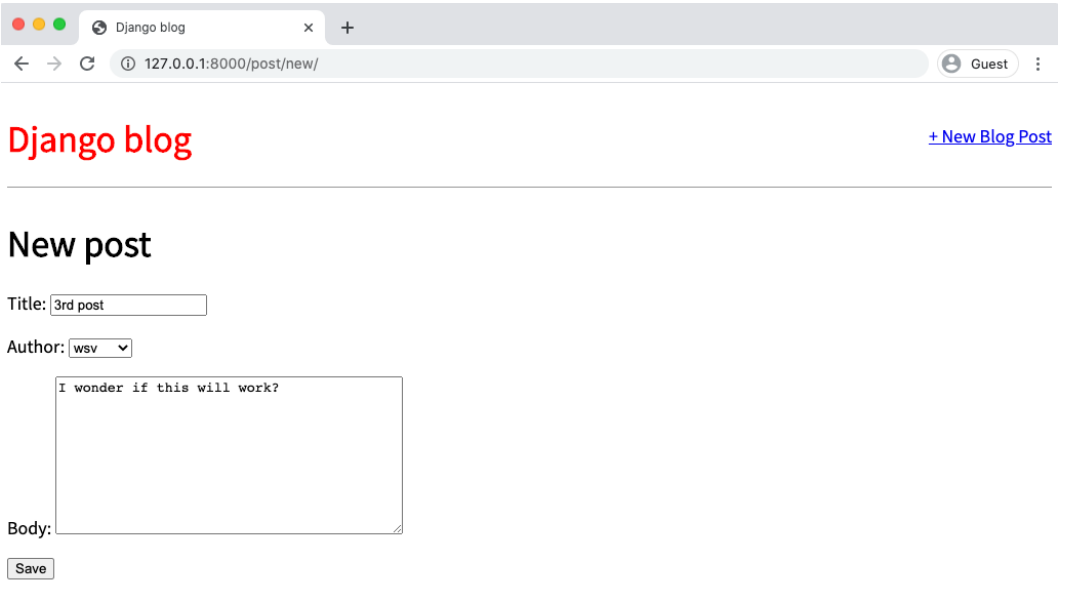
Homepage with new button

Click the "+ New Blog Post" link in the upper righthand corner. It will redirect to web page at `http://127.0.0.1:8000/post/new/`.



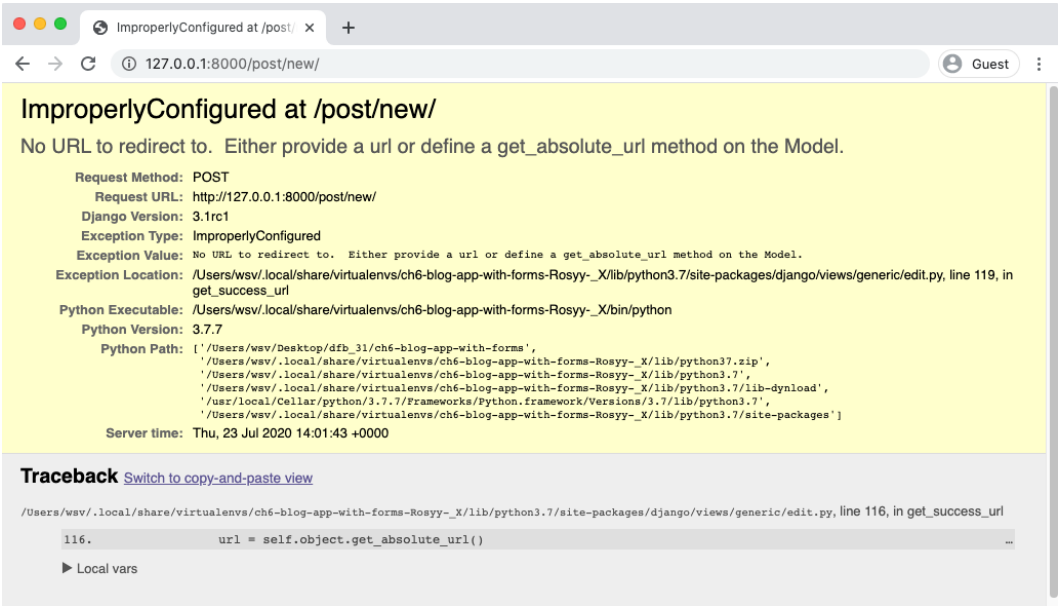
Blog new page

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



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.

Code

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


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

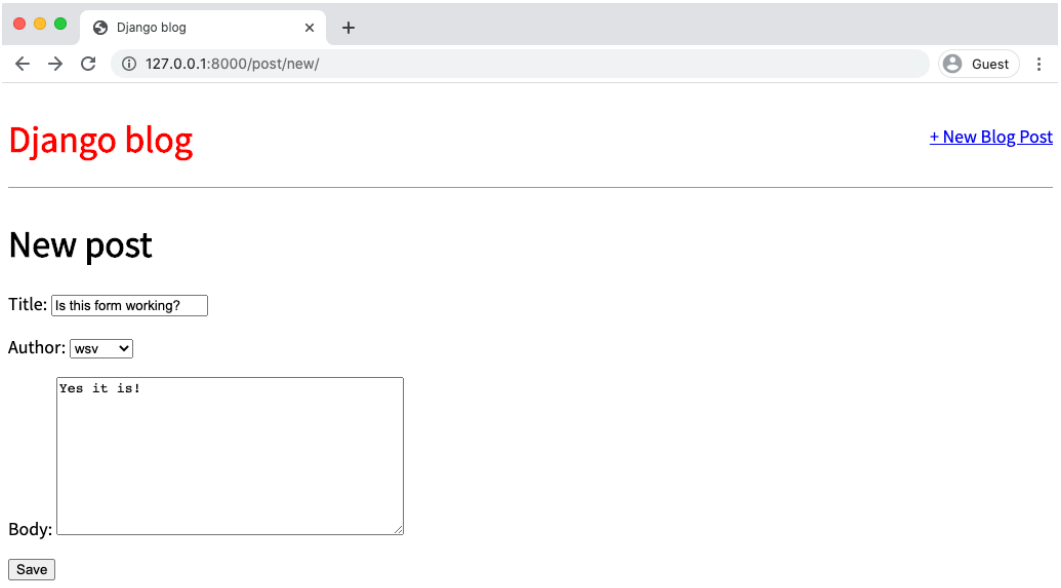
Reverse is a very handy utility function Django provides us to reference an object by its URL template name, in this case `post_detail`. If you recall, our URL pattern is the following:

Code

```
# blog/urls.py
path('post/<int:pk>/', BlogDetailView.as_view(), name='post_detail'),
```

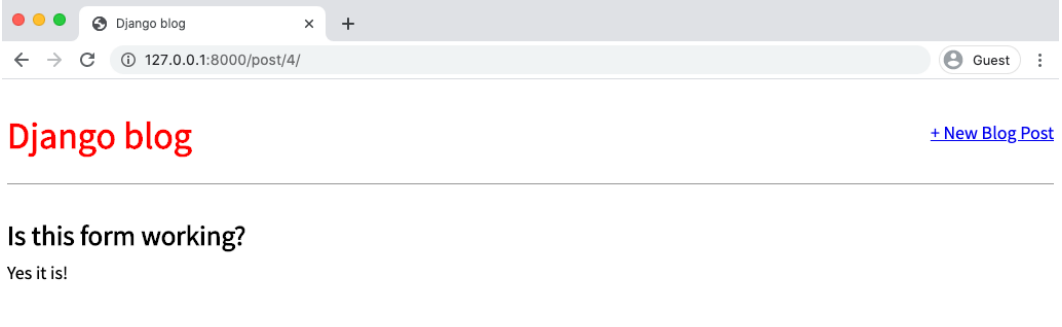
That means in order for this route to work we must *also* pass in an argument with the `pk` (primary key) of the object. Confusingly, `pk` and `id` are interchangeable in Django though the Django docs recommend using `self.id` with `get_absolute_url`. So we're telling Django that the ultimate location 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/`.



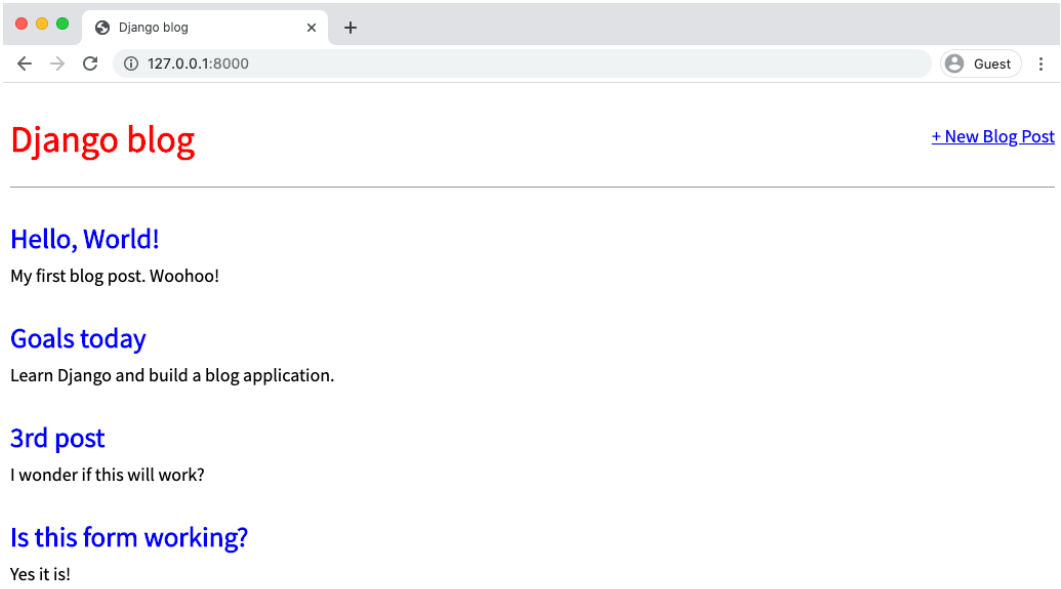
Blog new page with fourth post

Upon clicking the "Save" button you are now redirected to the detailed view page where the post appears.



Blog individual page

Go over to the homepage at http://127.0.0.1:8000/ and 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.



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>{{ post.title }}</h2>
    <p>{{ post.body }}</p>
  </div>

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

We've added a link using `<a href>...</a>` 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 content %}
```

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 # new
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 = ['title', 'author', 'body']


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

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.

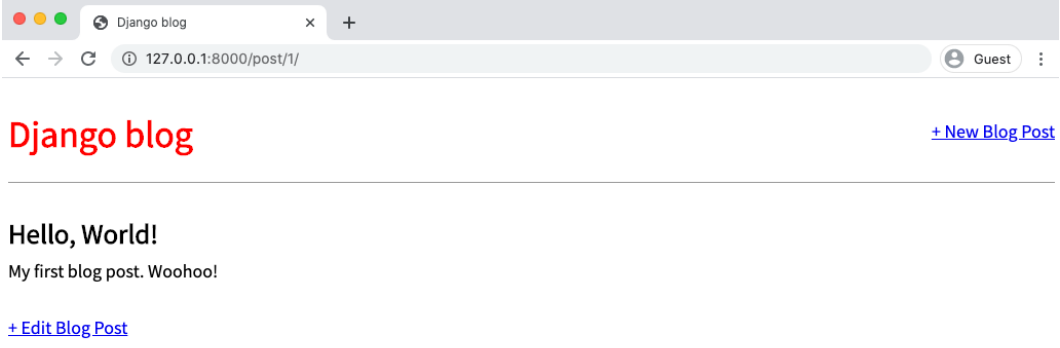
The final step is to update our `urls.py` file as follows. Add the `BlogUpdateView` up top and then the new route at the top of the existing `URLpatterns`.

Code

```
# blog/urls.py
from django.urls import path
from .views import (
    BlogListView,
    BlogDetailView,
    BlogCreateView,
    BlogUpdateView, # new
)

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

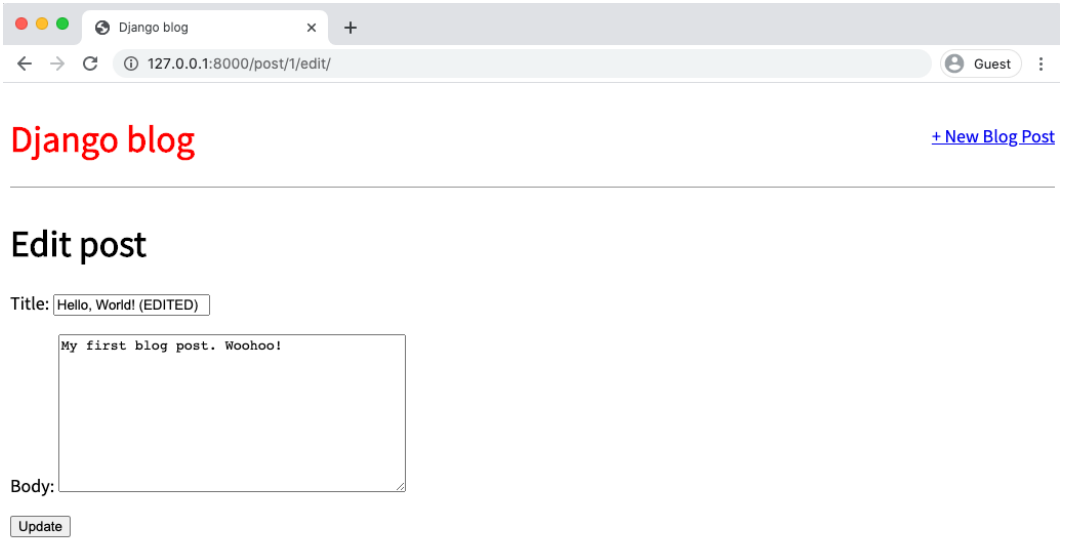
At the top we add our view `BlogUpdateView` to the list of imported views, then created a new url pattern for `/post/pk/edit` and given it the name `post_edit`.

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



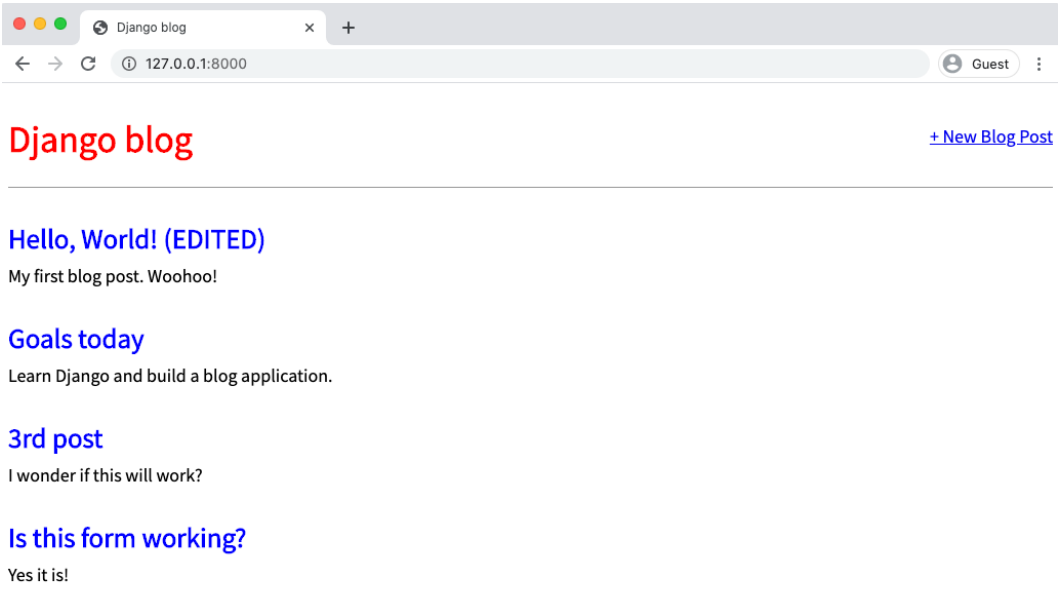
Blog page with edit button

If you click on "+ Edit Blog Post" you'll be redirected to `/post/1/edit/` if it is your first blog post, hence the `1` in the URL. Note that the form is pre-filled with our database's existing data for the post. Let's make a change...



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.



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, create the necessary view, url, and template.

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>{{ post.title }}</h2>
    <p>{{ post.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 content %}
```

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 the `blog/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
) # new
from django.urls import reverse_lazy # new

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 = ['title', 'author', 'body']


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


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

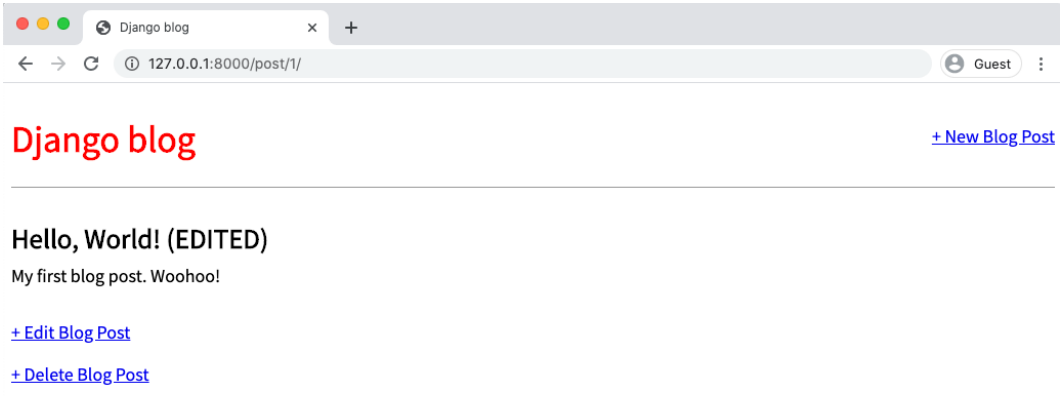
We use reverse_lazy as opposed to just reverse so that it won't execute the URL redirect until our view has finished deleting the blog post.

Finally, create a URL by importing our view `BlogDeleteView` and adding a new pattern:

Code

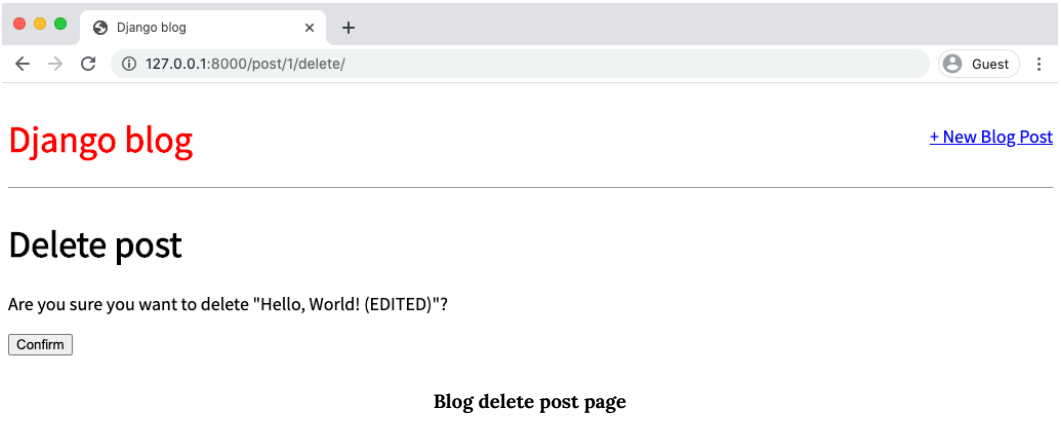
```
# blog/urls.py
from django.urls import path
from .views import (
    BlogListView,
    BlogDetailView,
    BlogCreateView,
    BlogUpdateView,
    BlogDeleteView, # new
)

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

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



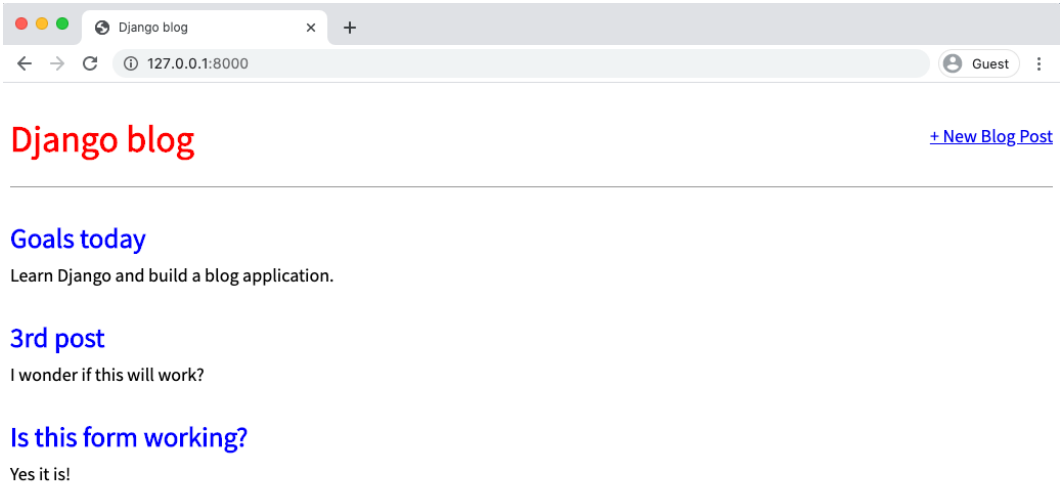
Blog delete post

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



Blog delete post page

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



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 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): # new
        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): # new
        response = self.client.post(reverse('post_new'), {
            'title': 'New title',
            'body': 'New text',
            'author': self.user.id,
        })
        self.assertEqual(response.status_code, 302)
        self.assertEqual(Post.objects.last().title, 'New title')
        self.assertEqual(Post.objects.last().body, 'New text')

    def test_post_update_view(self): # new
        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): # new
        response = self.client.post(
            reverse('post_delete', args='1'))
        self.assertEqual(response.status_code, 302)
```

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 302, a redirect since the item no longer exists.

There's always more tests that can be added but this at least has coverage on all our new functionality. Stop the local web server with `Control+c` and run these tests now. They should all pass.

Command Line

```
(blog) $ python manage.py test
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
........
----------------------------------------------------------------------
Ran 8 tests in 1.725s

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

Conclusion

With 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.

Note, however, a potential security concern: currently *any* user can update or delete blog entries, not just the creator! This is not ideal and indeed Django comes with built-in features to restrict access based on permissions, which we'll cover in-depth in Chapter 14.

But for now our Blog application is working and in the next chapter we'll add user accounts so users can sign up, log in, and log out of the web app.

Chapter 7: User Accounts

So far we've built a working blog application with 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 and customize as needed.

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 log in, log out, and sign up in our blog application.

Log In

Django provides us with a default view for a log in page via LoginView. All we need to add are a URLpattern for the auth system, a log in template, and a small update to our `config/settings.py` file.

First, update the `config/urls.py` file. We'll place our log in and log out pages at the `accounts/` URL. This is a one-line addition on the next-to-last line.

Code

```
# config/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')), # new
    path('', include('blog.urls')),
]
```

As the LoginView documentation notes, by default Django will look within a templates directory called `registration` for a file called `login.html` for a log in 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>Log In</h2>
<form method="post">
  {% csrf_token %}
  {{ form.as_p }}
  <button type="submit">Log In</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

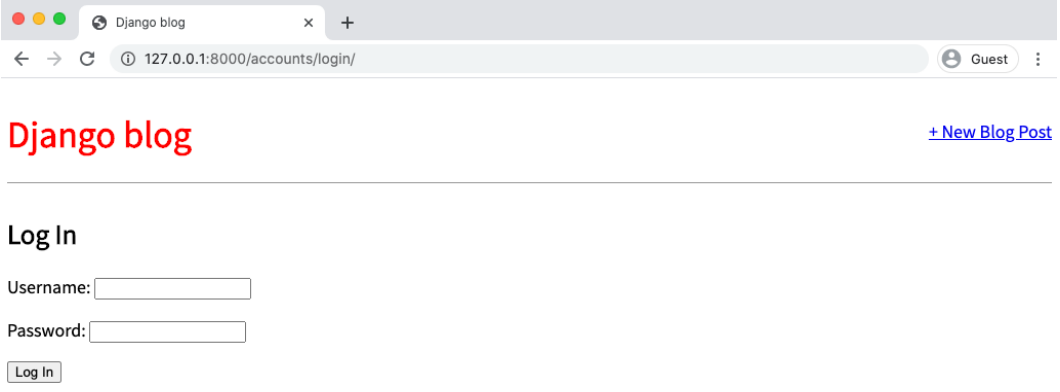
are outputted between paragraph tags thanks to `{{ form.as_p }}` and then we add a "submit" button.

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

Code

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

Now the user will be redirected to the `'home'` template which is our homepage. And 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 log in page at `http://127.0.0.1:8000/accounts/login/` you'll see the following:



Log in page

Upon entering the log in 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.

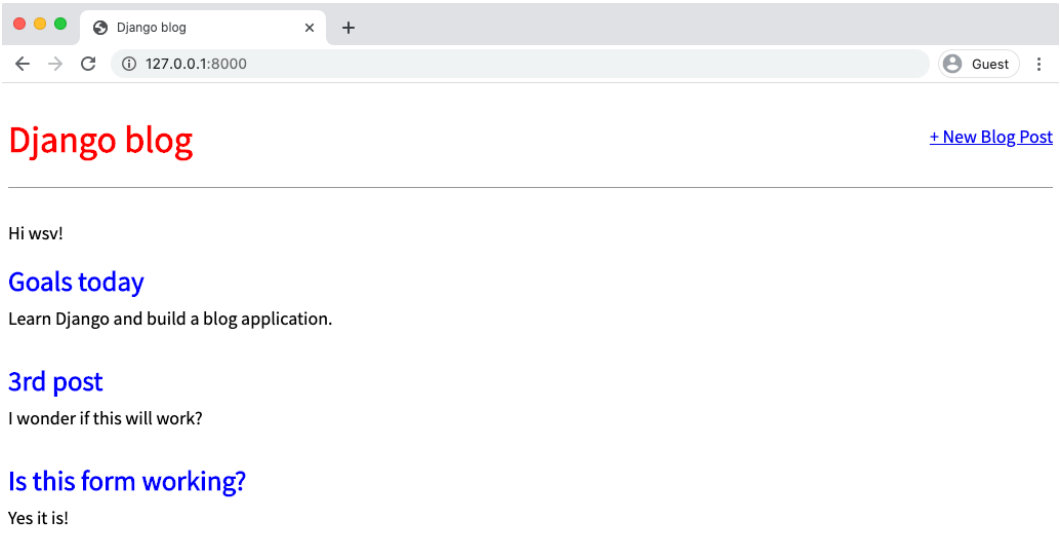
For now, we can simply place this code in a prominent position. Later on we can style it more appropriately. Update the `base.html` file with new code starting beneath the closing `</header>` tag.

Code

```
<!-- templates/base.html -->
{% load static %}
<html>
  <head>
    <title>Django blog</title>
    <link href="https://fonts.googleapis.com/css?family=Source+Sans+Pro:400" rel="stylesh\
eet">
    <link href="{% static 'css/base.css' %}" rel="stylesheet"s>
  </head>
  <body>
    <div>
      <header>
        <div class="nav-left">
          <h1><a href="{% url 'home' %}">Django blog</a></h1>
        </div>
        <div class="nav-right">
          <a href="{% url 'post_new' %}">+ New Blog Post</a>
        </div>
      </header>
      {% if user.is_authenticated %}
        <p>Hi {{ user.username }}!</p>
      {% else %}
        <p>You are not logged in.</p>
        <a href="{% url 'login' %}">Log In</a>
      {% endif %}
    {% block content %}
    {% endblock content %}
    </div>
  </body>
</html>
```

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

log in page.



Homepage logged in

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

Log Out 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 log out 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 just below our user greeting.

Command Line

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

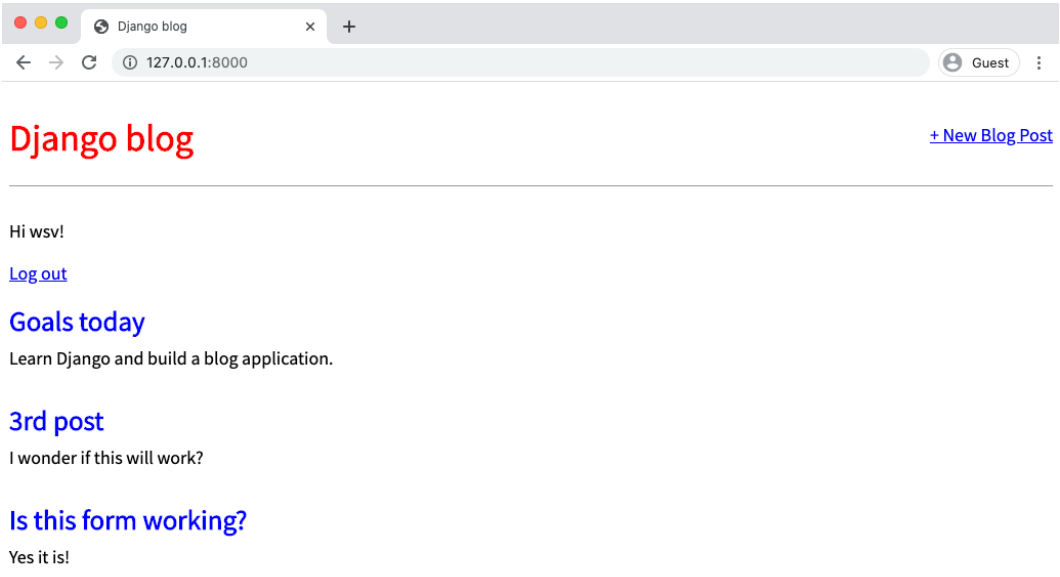
That's all we need to do as the necessary *view* is provided to us by the Django `auth` app. We do need to specify where to redirect a user upon log out though.

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

Code

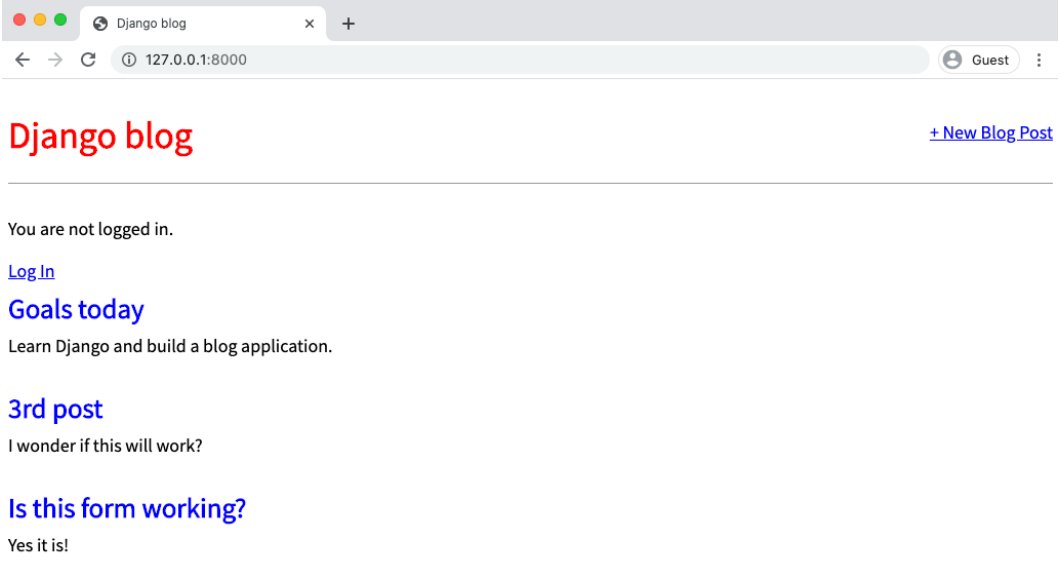
```
# config/settings.py
LOGIN_REDIRECT_URL = 'home'
LOGOUT_REDIRECT_URL = 'home' # new
```

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



Homepage log out 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.

Sign Up

We need to write our own view for a sign up 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. Stop the local server with `Control+c` and create a dedicated new app, `accounts`, for our sign up page.

Command Line

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

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

Code

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

Next add a new URL path in `config/urls.py` pointing to this new app directly **below** where we include the built-in `auth` app.

Code

```
# config/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 the `/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 .views import SignUpView

urlpatterns = [
    path('signup/', 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 URL 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 = 'registration/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 log in 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 the `templates/registration/` directory.

Command Line

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

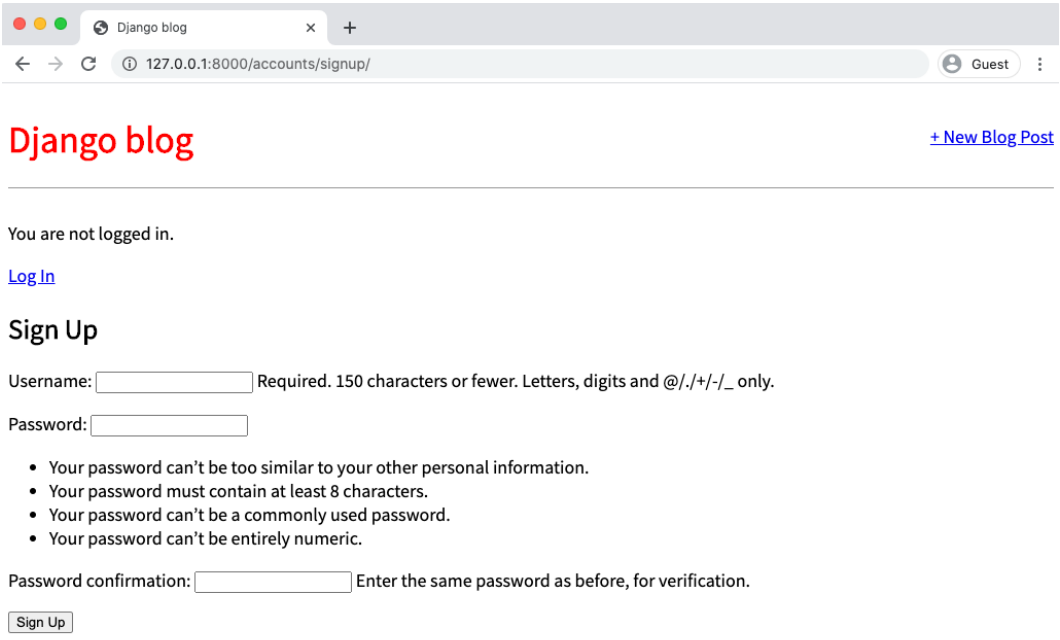
Add then populate it with the code below.

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 content %}
```

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.

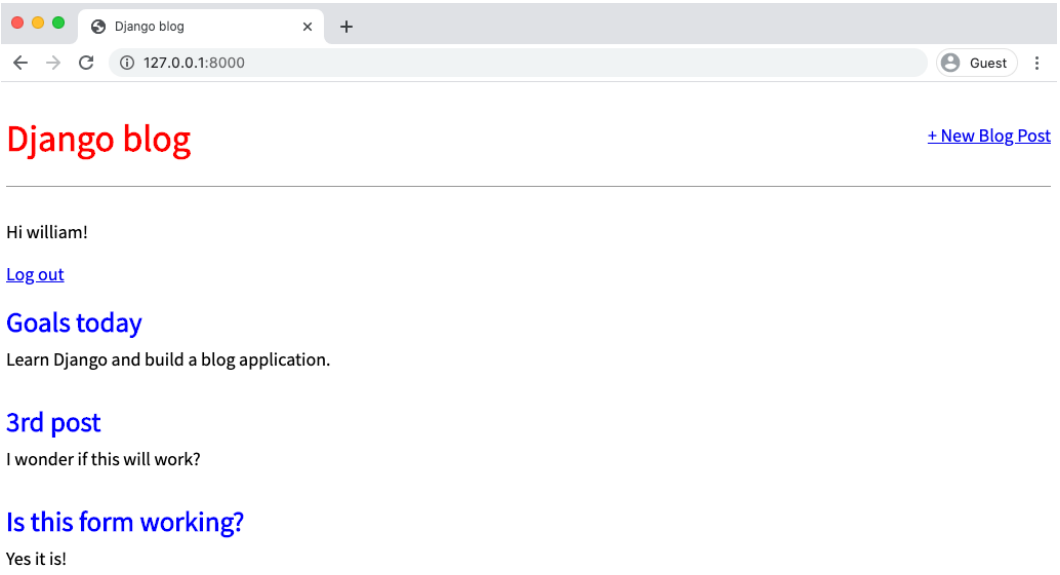
We're now done! To test it out, start up the local server with `python manage.py runserver` and navigate to `http://127.0.0.1:8000/accounts/signup/`.



Django sign up 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 log in page. Then after logging in successfully with my new user and password, I was redirected to the homepage with our personalized "Hi username" greeting.



Homepage for user william

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 `config/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.

GitHub

It's been a while since we made a `git` commit. Let's do that and then push a copy of our code onto GitHub. First check all the new work that we've done with `git status`.

Command Line

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

Then add the new content and enter a commit message.

Command Line

```
(blog) $ git add -A
(blog) $ git commit -m "forms and user accounts"
```

Create a new repo on GitHub which you can call anything you like. I'll choose the name `blog-app`. Therefore, *after creating the new repo on the GitHub site* ,I can type the following two commands. Make sure to replace my username `wsvincent` with your own from GitHub.

Command Line

```
(blog) $ git remote add origin https://github.com/wsvincent/blog-app.git
(blog) $ git push -u origin master
```

All done!

Static Files

Previously, we configured our static files by creating a dedicated `static` folder, pointing `STATICFILES_DIRS` to it in our `config/settings.py` file, and adding `{% load static %}` to our `base.html` template. But since Django won't serve static files in production, we need a few extra steps now.

The first change is to use Django's `collectstatic` command which compiles all static files throughout the project into a singe directory suitable for deployment. Second, we must set the STATIC_ROOT configuration, which is the *absolute location* of these collected files, to a folder called `staticfiles`. And third, we need to set STATICFILES_STORAGE, which is the *file storage engine* used by `collectstatic`.

Here is what the updated `config/settings.py` file should look like:

Code

```
# config/settings.py
STATIC_URL = '/static/'
STATICFILES_DIRS = [str(BASE_DIR.joinpath('static'))]
STATIC_ROOT = STATIC_ROOT = str(BASE_DIR.joinpath('staticfiles')) # new
STATICFILES_STORAGE =
    'django.contrib.staticfiles.storage.StaticFilesStorage' # new
```

Now run the command `python manage.py collectstatic`:

Command Line

```
(blog) $ python manage.py collectstatic
```

If you look at your project folder now you'll see there's a new `staticfiles` folder that contains `admin` and `css` folders. The `admin` is the built-in admin's static files, while the `css` is the one we created. Before each new deployment, the `collectstatic` command *must be run* to compile them into this `staticfiles` folder used in production. Since this is an easy step to forget it is often automated in larger projects though doing so is beyond the scope of our current project.

While there are multiple ways to serve these compiled static files in production, the most common approach–and the one we will use here–is to introduce the WhiteNoise package.

To start, install the latest version using Pipenv:

Command Line

```
(blog) $ pipenv install whitenoise==5.1.0
```

Then in `config/settings.py` there are three updates to make:

- add `whitenoise` to the `INSTALLED_APPS` **above** the built-in `staticfiles` app
- under `MIDDLEWARE` add a new line for `WhiteNoiseMiddleware`
- change `STATICFILES_STORAGE` to use `WhiteNoise`

The updated file should look as follows:

Code

```
# config/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',
    ...
]

STATIC_URL = '/static/'
STATICFILES_DIRS = [str(BASE_DIR.joinpath('static'))]
STATIC_ROOT = str(BASE_DIR.joinpath('staticfiles'))
STATICFILES_STORAGE =
    'whitenoise.storage.CompressedManifestStaticFilesStorage' # new
```

Since our `STATICFILES_STORAGE` method has changed, run `collectstatic` one more time to use `whitenoise` instead:

Command Line

```
(blog) $ python manage.py collectstatic
```

There will be a short warning, `This will overwrite existing files! Are you sure you want to do this?` Type "yes" and hit RETURN. The collected static files are now regenerated in the same `staticfiles` folder using WhiteNoise.

Static files are quite confusing to newcomers, so as a brief recap here are the steps we've executed so far in our *Blog* site. First, for local development back in **Chapter 5**, we created

a top-level `static` folder and updated `STATICFILES_DIRS` to point to it. In this chapter, we added configurations for `STATIC_ROOT` and `STATICFILES_STORAGE` before running `collectstatic` for the first time, which compiled *all* our static files across the entire project into a single `staticfiles` folder. Finally, we installed `whitenoise`, updated `INSTALLED_APPS`, `MIDDLEWARE`, and `STATICFILES_STORAGE`, and re-ran `collectstatic`.

Most developers, myself included, have trouble remembering all these steps properly and rely on notes as a friendly reminder!

Heroku Config

Now we come to Heroku for our third time deploying a website. Our deployment checklist is as follows:

- install `Gunicorn`
- add a `Procfile`
- update `ALLOWED_HOSTS`

Ready? Let's begin. Install `Gunicorn` as our production web server:

Command Line

```
(mb) $ pipenv install gunicorn==19.9.0
```

Create a new `Procfile` file.

Command Line

```
(blog) $ touch Procfile
```

Within your text editor, add the following line which tells Heroku to use `Gunicorn` rather than the local server for production.

Procfile

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

Then update the existing `ALLOWED_HOSTS` in `config/settings.py`.

Code

```
# config/settings.py
ALLOWED_HOSTS = ['.herokuapp.com', 'localhost', '127.0.0.1']
```

All set. We can commit our changes and push them up to GitHub.

Command Line

```
(blog) $ git status
(blog) $ git add -A
(blog) $ git commit -m "Heroku config"
(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
```

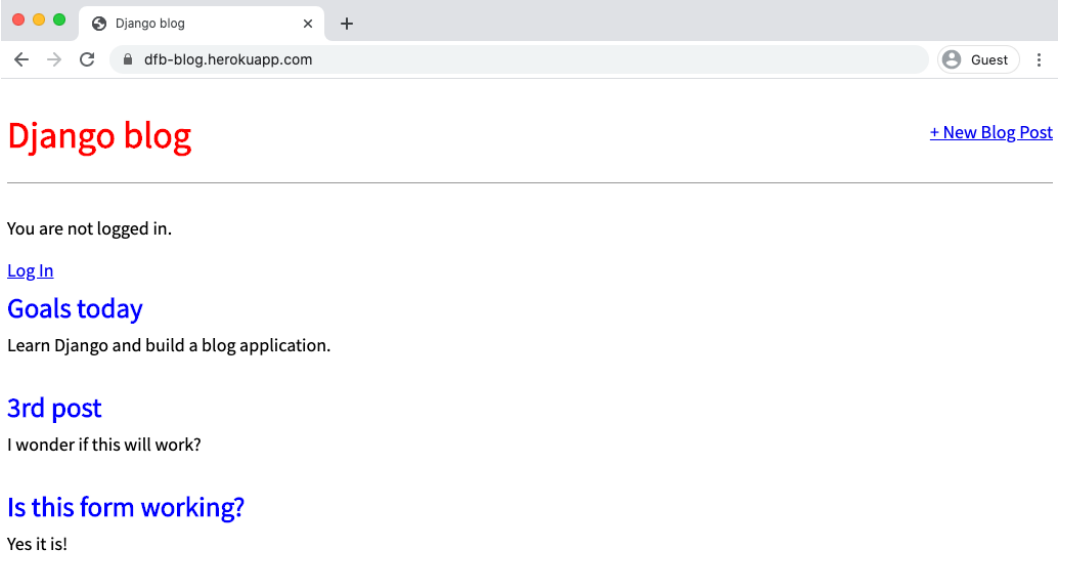
Heroku runs Django's `collectstatic` command automatically, which is why in the previous apps, where we had not configured static files, we told Heroku to disable this step with `heroku config:set DISABLE_COLLECTSTATIC=1`. But since we *have* configured static files, we'll happily let this happen now as part of the deployment process.

The final step is to 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, we have added log in, log out, and sign up to our *Blog* website. Under-the-hood Django has taken care of the many security gotchas that can crop up if you try to create your own user authentication flow from scratch. We properly configured static files for production and deployed our website, yet again, to Heroku. Good job!

In the next chapter, we'll embark on the final major project of the book, a *Newspaper* site which uses a custom user model, advanced user registration flow, enhanced styling via Bootstrap, and email configuration. It also includes proper permissions and authorizations, environment variables, and more security improvements to our deployment process.

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. But the approach demonstrated in the official documentation example is actually not what many Django experts recommend. It uses the quite complex `AbstractBaseUser` when if we just use `AbstractUser` instead things are far simpler and still customizable.

Thus we will use `AbstractUser` in this chapter where we start a new *Newspaper* app properly with environment variables and 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.

Initial Set Up

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 `config`
- make a new app `accounts`

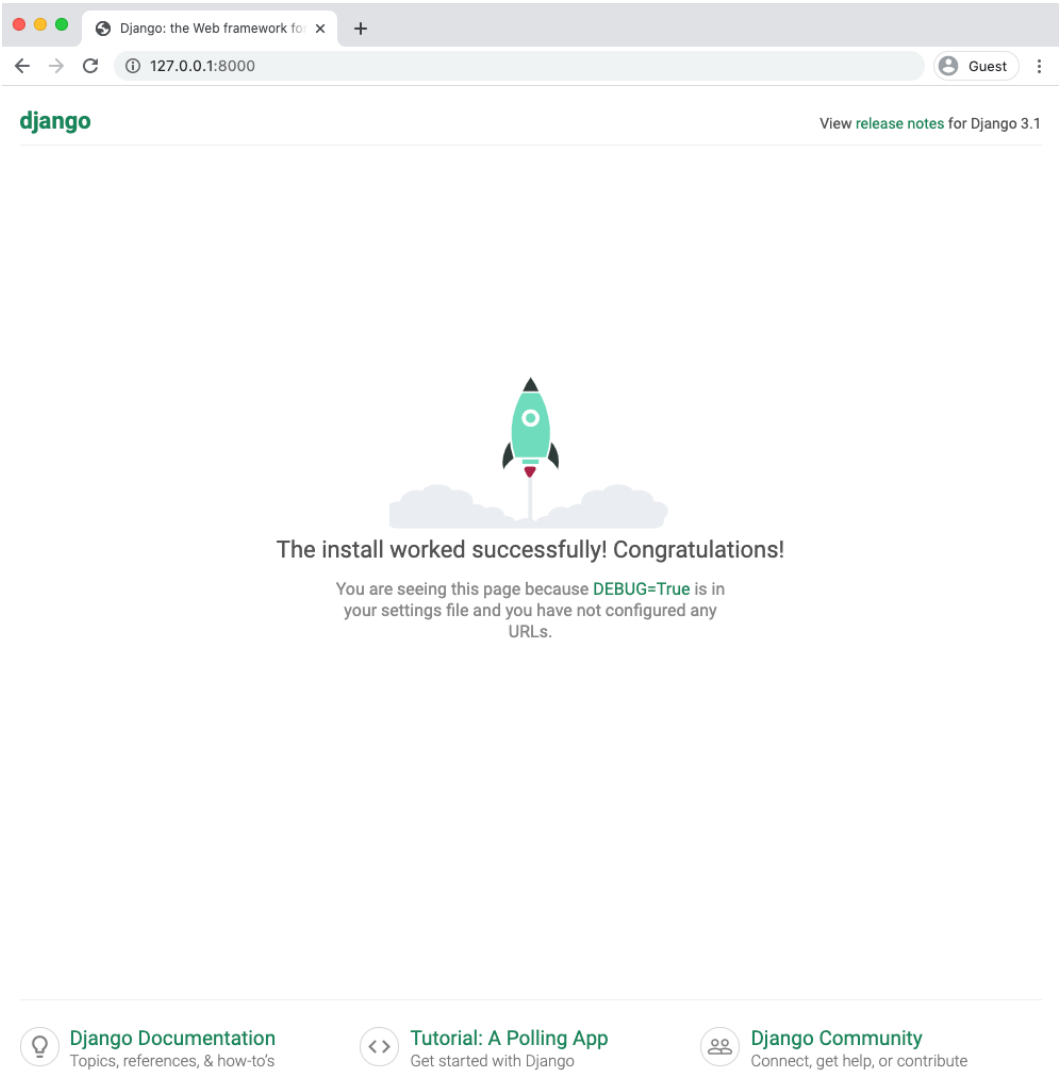
Here are the commands to run:

Command Line

```
$ cd ~/Desktop
$ mkdir news
$ cd news
$ pipenv install django~=3.1.0
$ pipenv shell
(news) $ django-admin startproject config .
(news) $ python manage.py startapp accounts
(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.



Welcome page

Custom User Model

Creating our custom user model requires four steps:

- update `config/settings.py`
- create a new `CustomUser` model
- create new forms for `UserCreationForm` and `UserChangeForm`
- update `accounts/admin.py`

In `config/settings.py` we'll add the `accounts` 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 `accounts` app we refer to it as `accounts.CustomUser`.

Code

```
# config/settings.py
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'accounts', # new
]
...
AUTH_USER_MODEL = 'accounts.CustomUser' # new
```

Now update `accounts/models.py` with a new User model which we'll call `CustomUser` that extends the existing `AbstractUser`. We also include our first custom field, `age`, here.

Code

```
# accounts/models.py
from django.contrib.auth.models import AbstractUser
from django.db import models


class CustomUser(AbstractUser):
    age = models.PositiveIntegerField(null=True, blank=True)
```

If you read the official documentation on custom user models it recommends using `AbstractBaseUser` not `AbstractUser`. This needlessly complicates things in my opinion, especially for beginners.

> ## AbstractBaseUser vs AbstractUser
>
> `AbstractBaseUser` requires a very fine level of control and customization. We essentially rewrite Django. This *can be* helpful, but if we just want a custom user model that can be updated with additional fields, the better choice is `AbstractUser` which subclasses `AbstractBaseUser`. In other words, we write much less code and have less opportunity to mess things up. It's the better choice unless you really know what you're doing with Django!

Note that we use both null and blank with our `age` field. These two terms are easy to confuse but quite distinct:

- `null` is **database-related**. When a field has `null=True` it can store a database entry as `NULL`, meaning no value.
- `blank` is **validation-related**, if `blank=True` then a form will allow an empty value, whereas if `blank=False` then a value is required.

In practice, `null` and `blank` are commonly used together in this fashion so that a form allows an empty value and the database stores that value as `NULL`.

A common gotcha to be aware of is that the **field type** dictates how to use these values. Whenever you have a string-based field like `CharField` or `TextField`, setting both `null` and `blank` as we've done will result in two possible values for "no data" in the database. Which is a bad idea. The Django convention is instead to use the empty string `''`, not `NULL`.

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 `accounts` app called `forms.py`.

Command Line

```
(news) $ touch accounts/forms.py
```

We'll update it with the following code to extend the existing `UserCreationForm` and `UserChangeForm` forms.

Code

```
# accounts/forms.py
from django import forms
from django.contrib.auth.forms import UserCreationForm, UserChangeForm
from .models import CustomUser


class CustomUserCreationForm(UserCreationForm):

    class Meta(UserCreationForm):
        model = CustomUser
        fields = UserCreationForm.Meta.fields + ('age',)


class CustomUserChangeForm(UserChangeForm):

    class Meta:
        model = CustomUser
        fields = UserChangeForm.Meta.fields
```

For both new forms we are using the Meta class to override the default fields by setting the `model` to our `CustomUser` and using the default fields via `Meta.fields` which includes *all* default fields.

To add our custom `age` field we simply tack it on at the end and it will display automatically on our future sign up page. Pretty slick, no?

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 might not click for you since understanding forms and models properly takes some time. In the next chapter we will create our own sign up, log in, and log out pages which will tie together our `CustomUser` model and forms more clearly. So hang tight!

The only other step we need 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.

Code

```
# accounts/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
    model = CustomUser


admin.site.register(CustomUser, CustomUserAdmin)
```

Ok we're done! Type `Control+c` to stop the local server and 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 accounts
Migrations for 'accounts':
  accounts/migrations/0001_initial.py
    - Create model CustomUser
(news) $ python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, sessions, users
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying contenttypes.0002_remove_content_type_name... OK
  Applying auth.0001_initial... OK
  Applying auth.0002_alter_permission_name_max_length... OK
  Applying auth.0003_alter_user_email_max_length... OK
  Applying auth.0004_alter_user_username_opts... OK
  Applying auth.0005_alter_user_last_login_null... OK
  Applying auth.0006_require_contenttypes_0002... OK
  Applying auth.0007_alter_validators_add_error_messages... OK
  Applying auth.0008_alter_user_username_max_length... OK
  Applying auth.0009_alter_user_last_name_max_length... OK
  Applying auth.0010_alter_group_name_max_length... OK
  Applying auth.0011_update_proxy_permissions... OK
  Applying auth.0012_alter_user_first_name_max_length... OK
  Applying accounts.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... OK
  Applying admin.0003_logentry_add_action_flag_choices... OK
  Applying sessions.0001_initial... OK
```

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
```

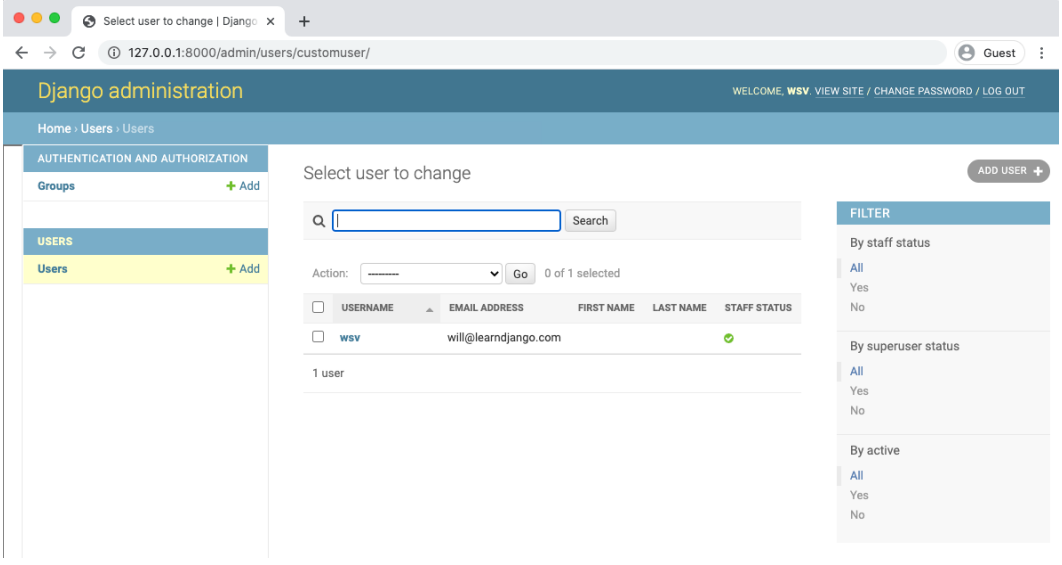
The fact that this works is the first proof our custom user model works as expected. Let's view things in the admin too to be extra sure.

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. If you click on the link for "Users," you should see your superuser account as well as the default fields of Username, Email Address, First Name, Last Name, and Staff Status.



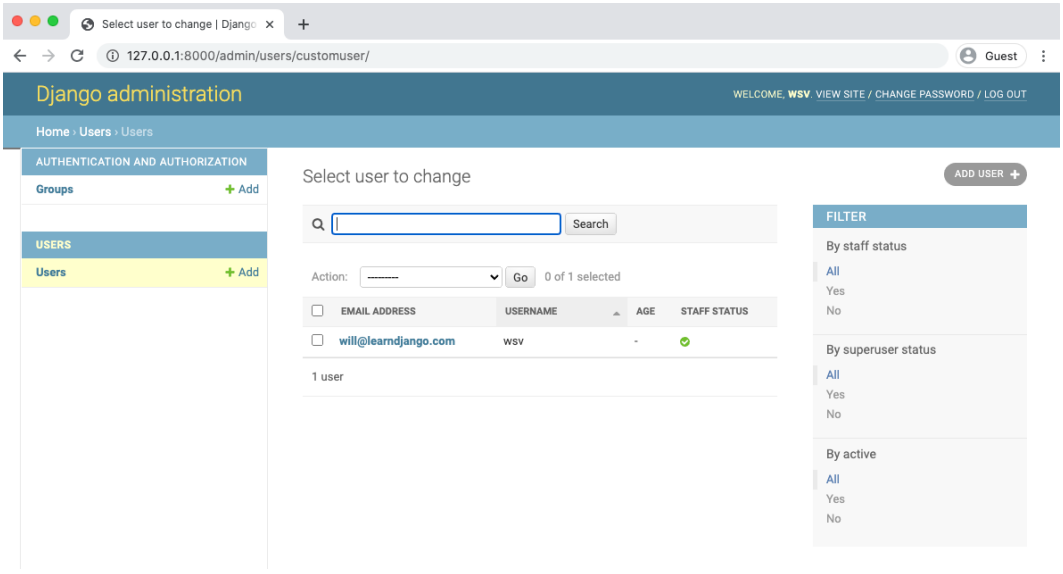
Admin one user

To control the fields listed here we use list_display. However, to actually edit and add new custom fields, like `age`, we must also add fieldsets.

Code

```
# accounts/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
    model = CustomUser
    list_display = ['email', 'username', 'age', 'is_staff', ] # new
    fieldsets = UserAdmin.fieldsets + ( # new
        (None, {'fields': ('age',)}),
    )
    add_fieldsets = UserAdmin.add_fieldsets + ( # new
        (None, {'fields': ('age',)}),
    )


admin.site.register(CustomUser, CustomUserAdmin)
```

Refresh the page and you should see the update.



Admin custom list display

Conclusion

With our custom user model complete, we can now focus on building out the rest of our *Newspaper* website. In the next chapter we will configure and customize sign up, log in, and log out 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 sign up, log in, and log out users. Django provides everything we need for log in and log out 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. The structure `accounts/templates/accounts/home.html` would be needed for a `home.html` template within the `accounts` app. But a single `templates` directory within `config` 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` directory as that's where Django will look for the log in 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 `config/settings.py`. This is a one-line change.

Code

```
# config/settings.py
TEMPLATES = [
    {
        ...
        'DIRS': [str(BASE_DIR.joinpath('templates'))], # new
        ...
    }
]
```

If you think about what happens when you log in or log out 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 to call it `'home'`.

Add these two lines at the bottom of the `config/settings.py` file.

Code

```
# config/settings.py
LOGIN_REDIRECT_URL = 'home' # new
LOGOUT_REDIRECT_URL = 'home' # new
```

Now we can create four new templates:

Command Line

```
(news) $ touch templates/base.html
(news) $ touch templates/home.html
(news) $ touch templates/registration/login.html
(news) $ touch templates/registration/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>{% block title %}Newspaper App{% endblock title %}</title>
</head>
<body>
  <main>
    {% block content %}
    {% endblock content %}
  </main>
</body>
</html>
```

Code

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

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

{% block content %}
{% if user.is_authenticated %}
  Hi {{ user.username }}!
  <p><a href="{% url 'logout' %}">Log Out</a></p>
{% else %}
  <p>You are not logged in</p>
  <a href="{% url 'login' %}">Log In</a> |
  <a href="{% url 'signup' %}">Sign Up</a>
{% endif %}
{% endblock content %}
```

Code

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

{% block title %}Log In{% endblock title %}

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

Code

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

{% block title %}Sign Up{% endblock title %}

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

Our templates are now all set. Still to go are the related URLs and views.

URLs

Let's start with the URL routes. In our `config/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. 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 `accounts` app and the built-in `auth` app. The reason is that the built-in `auth` app already provides views and urls for log in and log out. But for sign up we will need to create our own view and url. To ensure that our URL routes are consistent we place them *both* at `accounts/` so the eventual URLS will be `/accounts/login`, `/accounts/logout`, and `/accounts/signup`.

Code

```
# config/urls.py
from django.contrib import admin
from django.urls import path, include # new
from django.views.generic.base import TemplateView # new

urlpatterns = [
    path('admin/', admin.site.urls),
    path('accounts/', include('accounts.urls')), # new
    path('accounts/', include('django.contrib.auth.urls')), # new
    path('', TemplateView.as_view(template_name='home.html'),
      name='home'), # new
]
```

Now create a `urls.py` file in the `accounts` app.

Command Line

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

Update `accounts/urls.py` with the following code:

Code

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

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

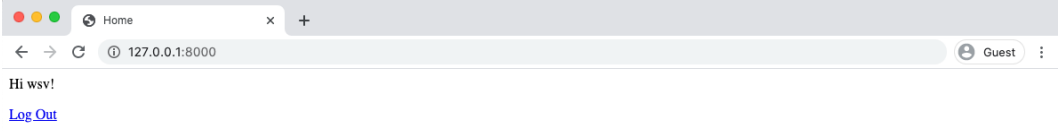
The last step is our `views.py` file which will contain the logic for our sign up form. We're using Django's generic `CreateView` here and telling it to use our `CustomUserCreationForm`, to redirect to `login` once a user signs up successfully, and that our template is named `signup.html`.

Code

```
# accounts/views.py
from django.urls import reverse_lazy
from django.views.generic import CreateView
from .forms import CustomUserCreationForm


class SignUpView(CreateView):
    form_class = CustomUserCreationForm
    success_url = reverse_lazy('login')
    template_name = 'registration/signup.html'
```

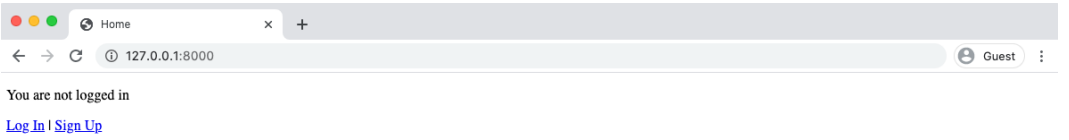
Ok, phew! We're done. Let's test things out.

Start up the server with `python manage.py runserver` and go to the homepage.



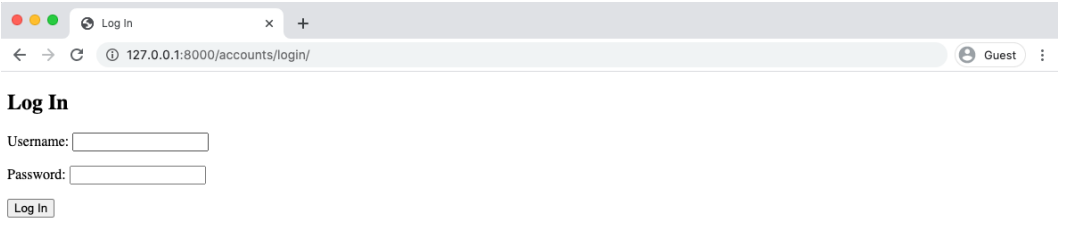
Homepage logged in

We logged in to the admin in the previous chapter so you should see a personalized greeting here. Click on the "Log Out" link.



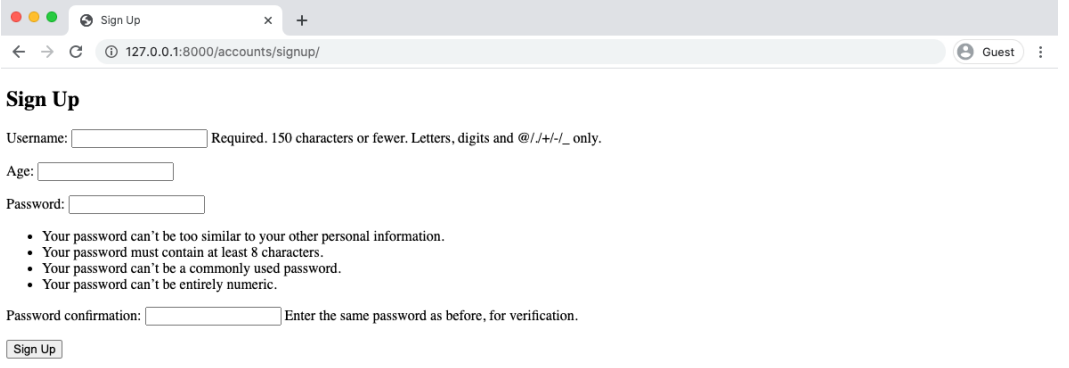
Homepage logged out

Now we're on the logged out homepage. Go ahead and click on *login* link and use your **superuser** credentials.



Log in

Upon successfully logging in you'll be redirected back to the homepage and see the same personalized greeting. It works! Now use the "Log Out" link to return to the homepage and this time click on the "Sign Up" link. You'll be redirected to our signup page. See that the `age` field is included!



Sign up page

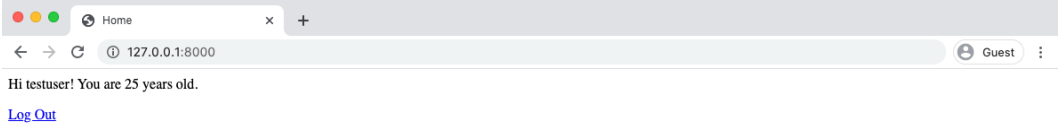
Create a new user. Mine is called `testuser` and I've set the age to `25`. After successfully submitting the form you'll be redirected to the log in page. Log in with your new user and you'll again be

redirected to the homepage with a personalized greeting for the new user. But since we have the new `age` field, let's add that to the `home.html` template. It is a field on the `user` model, so to display it we only need to use `{{ user.age }}`.

Code

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

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

{% block content %}
{% if user.is_authenticated %}
  Hi {{ user.username }}! You are {{ user.age }} years old.
  <p><a href="{% url 'logout' %}">Log Out</a></p>
{% else %}
  <p>You are not logged in</p>
  <a href="{% url 'login' %}">Log In</a> |
  <a href="{% url 'signup' %}">Sign Up</a>
{% endif %}
{% endblock content %}
```

Save the file and refresh the homepage.

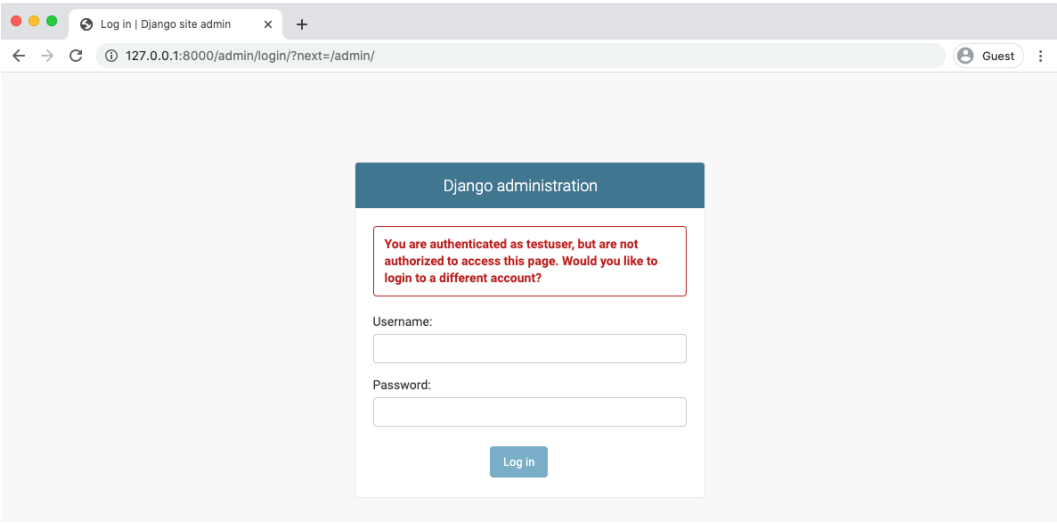


Homepage for testuser

Everything works as expected.

Admin

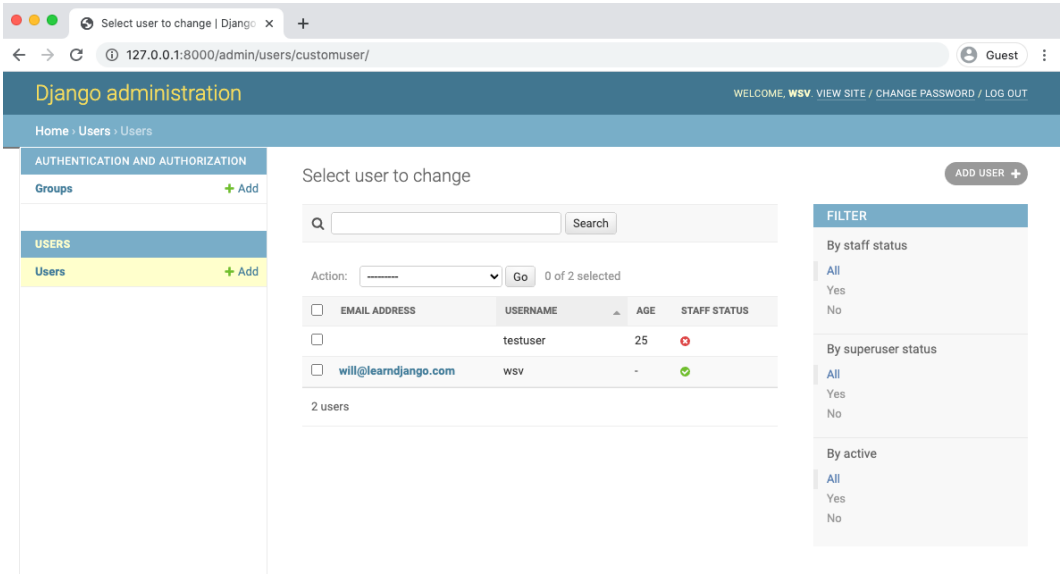
Let's also log in to the admin to view our two user accounts. Navigate to `http://127.0.0.1:8000/admin` and ...



Admin log in 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 `testuser` account we just created and your previous superuser name (mine is `wsv`).



Users in the Admin

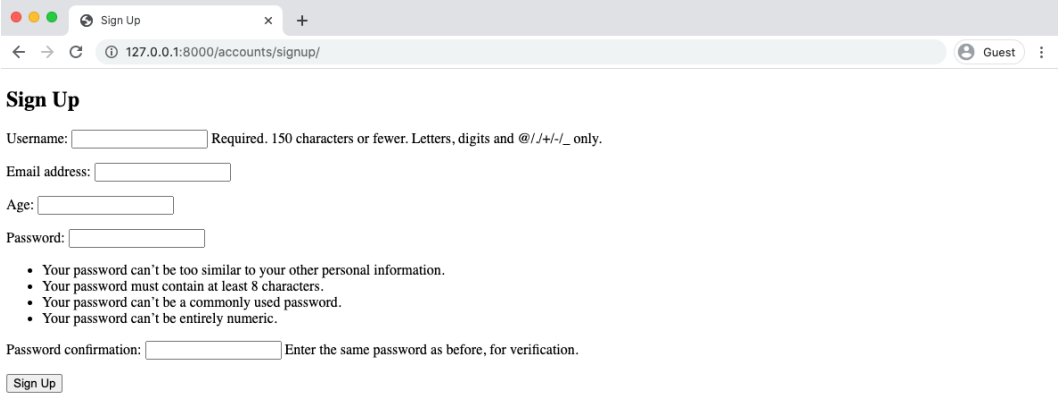
Everything is working but you may notice that there is no "Email address" for our `testuser`. Why is that? Well, our sign up page has no email field because it was not included in `accounts/forms.py`. This is an important point: just because the `user` model has a field, it will not be included in our custom sign up form unless it is explicitly added. Let's do so now.

Currently, in `accounts/forms.py` under `fields` we're using `Meta.fields`, which just displays the default settings of username/age/password. But we can also explicitly set which fields we want displayed so let's update it to ask for a username/email/age/password by setting it to `('username', 'email', 'age',)`. We don't need to include the `password` fields because they are required! All the other fields can be configured however we choose.

Code

```
# accounts/forms.py
from django import forms
from django.contrib.auth.forms import UserCreationForm, UserChangeForm

from .models import CustomUser


class CustomUserCreationForm(UserCreationForm):

    class Meta(UserCreationForm):
        model = CustomUser
        fields = ('username', 'email', 'age',) # new


class CustomUserChangeForm(UserChangeForm):

    class Meta:
        model = CustomUser
        fields = ('username', 'email', 'age',) # new
```

Now if you try `http://127.0.0.1:8000/accounts/signup/` again you can see the additional "Email address" field is there.

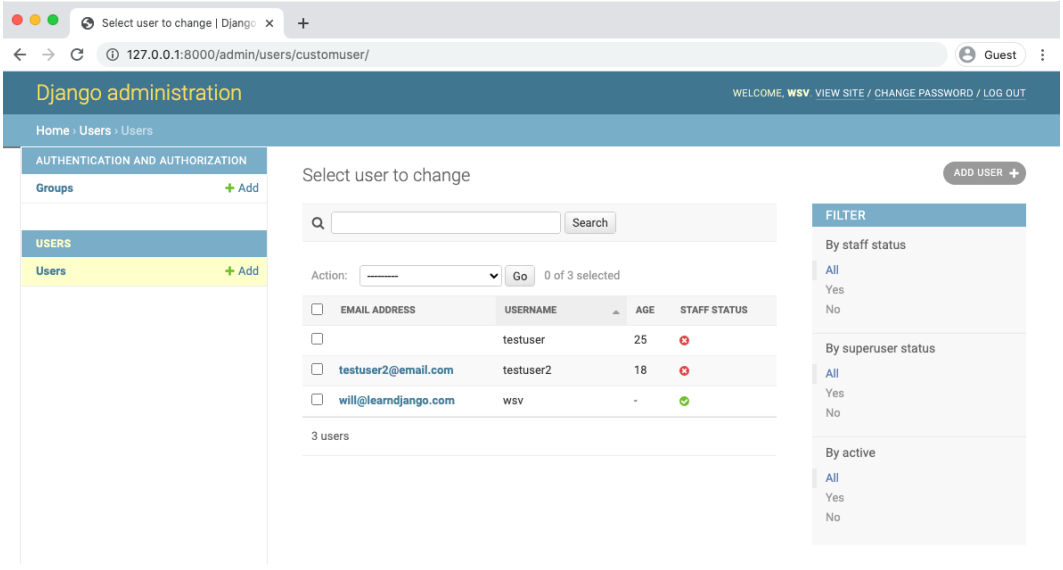


New sign up page

Sign up with a new user account. I've named mine `testuser2` with an age of `18` and an email address of `testuser2@email.com`. Continue to log in and you'll see a personalized greeting on the homepage.

testuser2 homepage greeting

Then switch back to the admin page–log in using our superuser account to do so–and all three users are on display.



Three users in the Admin

Django's user authentication flow requires a little bit of set up but you should be starting to see that it also provides us incredible flexibility to configure sign up and log in *exactly* how we want.

Conclusion

So far our *Newspaper* app has a custom user model and working sign up, log in, and log out 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.

While it's possible to hand-code all the required CSS and JavaScript for a modern-looking website, in practice most developers user a framework like Bootstrap or TailwindCSS. For our project, we'll use Bootstrap which can be extended and customized as needed.

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, such as the homepage, a future about page, and so on. 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 `config/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

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

Now we can update our `urls.py` file inside the `config` directory by adding the `pages` app, removing the import of `TemplateView`, and removing the previous URL path for the older homepage.

Code

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

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

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

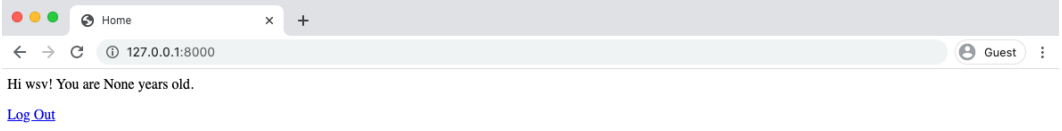
```
# pages/urls.py
from django.urls import path
from .views import HomePageView

urlpatterns = [
    path('', 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. Start up the local server `python manage.py runserver` and navigate to the homepage at `http://127.0.0.1:8000/` to confirm it remains unchanged.



Homepage logged in

It should show the name of your logged in superuser account which we used at the end of the last chapter. And, interestingly, since we have not added an `age` field to our superuser, Django defaults to displaying `None`.

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
- sign up
- log in
- log out

We only need tests, however, for the first two. Log in and log out are part of Django and rely on internal views, URL routes, templates, and tests. We do not need to re-test core Django functionality. If we made substantial changes to either of 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 sign up 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('/accounts/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, 'registration/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 sign up 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.

Note that there are two ways to specify a page: either hardcoded as in `test_signup_page_status_code` where we set the `response` to `/accounts/signup/` or via the URL name of `signup` which is done for `test_view_url_by_name` and `test_view_uses_correct_template`.

Quit the local server with `Control+c` and then run our tests to confirm everything passes.

Command Line

```
(news) $ python manage.py test
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
.......
----------------------------------------------------------------------
Ran 7 tests in 0.043s

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

Bootstrap

If you've never used Bootstrap before you're in for a real treat. Much like Django, 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. You can see the four lines to add on the Bootstrap introduction page:

- `Bootstrap.css`
- `jQuery.js`
- `Popper.js`
- `Bootstrap.js`

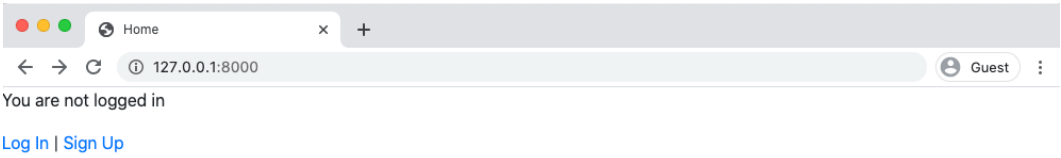
We'll also need to add a `meta name="viewport"` line to `<head></head>` for it all to work properly. In general, typing out all code yourself is the recommended approach but adding the Bootstrap CDN is an exception since it is lengthy and easy to miss-type. I recommend copy and pasting the Bootstrap CSS and JavaScript links into the `base.html` file.

Code

```
<!-- templates/base.html -->
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>{% block title %}Newspaper App{% endblock title %}</title>
  <meta name="viewport" content="width=device-width,
  initial-scale=1, shrink-to-fit=no">

  <!-- Bootstrap CSS -->
  <link href="https://stackpath.bootstrapcdn.com/..." rel="stylesheet">
</head>
<body>
  <main>
    {% block content %}
    {% endblock content %}
  </main>

  <!-- Optional JavaScript -->
  <!-- jQuery first, then Popper.js, then Bootstrap JS -->
  <script src="https://code.jquery.com/..."></script>
  <script src="https://cdn.jsdelivr.net/npm/popper.js..."></script>
  <script src="https://stackpath.bootstrapcdn.com/bootstrap/..."></script>
</body>
</html>
```

This code snippet **does not** include the full links for Bootstrap CSS and JavaScript. It is abbreviated. Copy and paste the full links for Bootstrap 4.5 from the quick start docs. {/aside}

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 the font size has changed as well as the link colors.



Homepage with Bootstrap

Let's add a navigation bar at the top of the page which contains our links for the homepage, log in, log out, and sign up. 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.

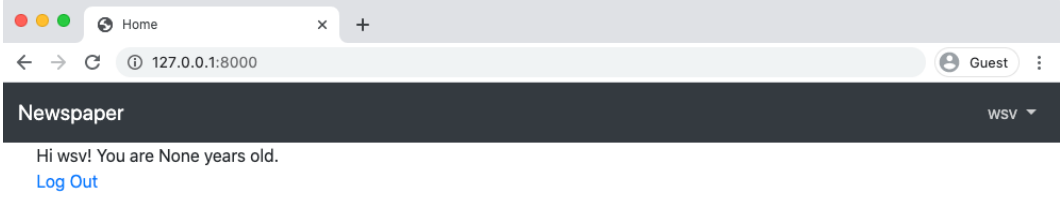
Again, it's ok to copy/paste here since the focus of this book is on learning Django not HTML, CSS, and Bootstrap.

Code

```
<!-- templates/base.html -->
...
<body>
  <nav class="navbar navbar-expand-lg navbar-dark bg-dark">
    <a class="navbar-brand" href="{% url 'home' %}">Newspaper</a>
    <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>
    <div class="collapse navbar-collapse" id="navbarSupportedContent">
      {% if user.is_authenticated %}
        <ul class="navbar-nav ml-auto">
          <li class="nav-item">
            <a class="nav-link dropdown-toggle" href="#"
            id="navbarDropdown" role="button" data-toggle="dropdown"
            aria-haspopup="true" aria-expanded="false">
              {{ user.username }}
            </a>
            <div class="dropdown-menu dropdown-menu-lg-right"
            aria-labelledby="navbarDropdown">
              <a class="dropdown-item"
              href="{% url 'password_change'%}">Change password</a>
              <div class="dropdown-divider"></div>
              <a class="dropdown-item" href="{% url 'logout' %}">
```

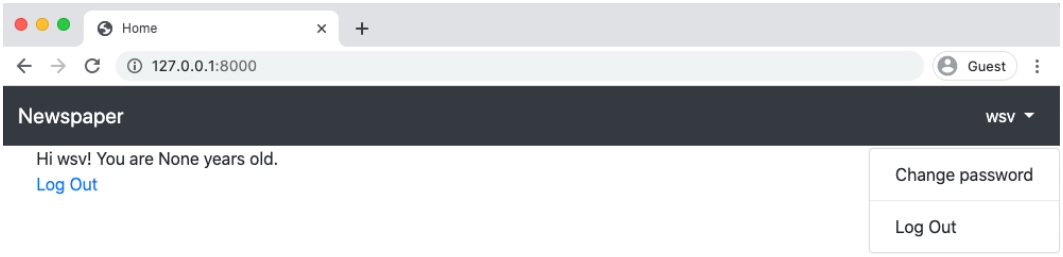
```
            Log Out</a>
          </div>
        </li>
      </ul>
    {% else %}
      <form class="form-inline ml-auto">
        <a href="{% url 'login' %}" class="btn btn-outline-secondary">
        Log In</a>
        <a href="{% url 'signup' %}" class="btn btn-primary ml-2">
        Sign up</a>
      </form>
    {% endif %}
  </div>
 </nav>
 <main>
  <div class="container">
    {% block content %}
    {% endblock content %}
  </div>
 </main>
</body>
...
```

If you refresh the homepage at `http://127.0.0.1:8000/` our new nav has magically appeared! We've also added `"container"` divs around our content, which is a Bootstrap way of providing automatic side padding to a site.



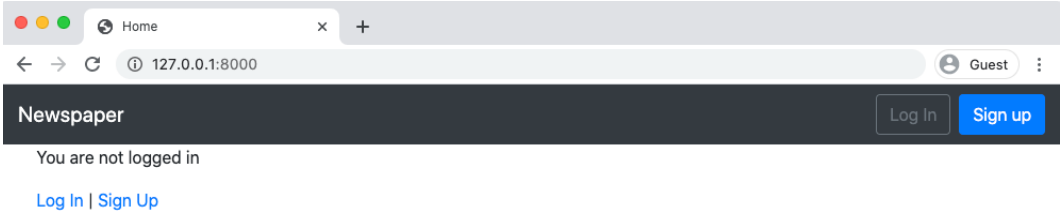
Homepage with Bootstrap nav logged in

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

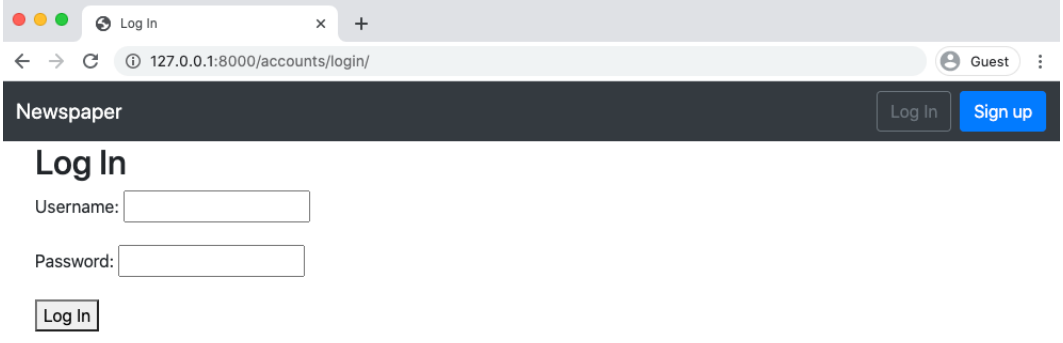
If you click on the "Log Out" the nav bar changes to button links for either "Log In" or "Sign Up."



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. 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 "Log Out" button and then "Log In" from the top nav you can also see that our log in page `http://127.0.0.1:8000/accounts/login` looks better too.



Bootstrap login

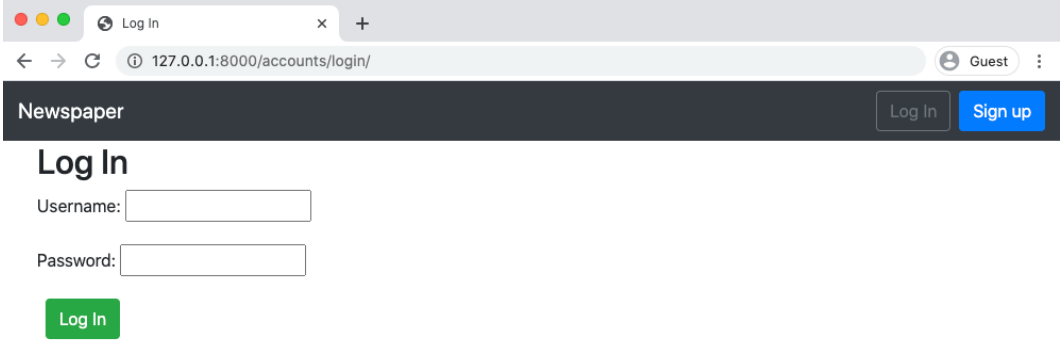
The only thing that looks off is our "Login" button. We can use Bootstrap to add some nice styling such as making it green and inviting.

Change the "button" line in `templates/registration/login.html` as follows.

Code

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

{% block title %}Log In{% endblock title %}

{% block content %}
<h2>Log In</h2>
<form method="post">
  {% csrf_token %}
  {{ form.as_p }}
  <button class="btn btn-success ml-2" type="submit">Log In</button>
</form>
{% endblock content %}
```

Now refresh the page to see our new button.

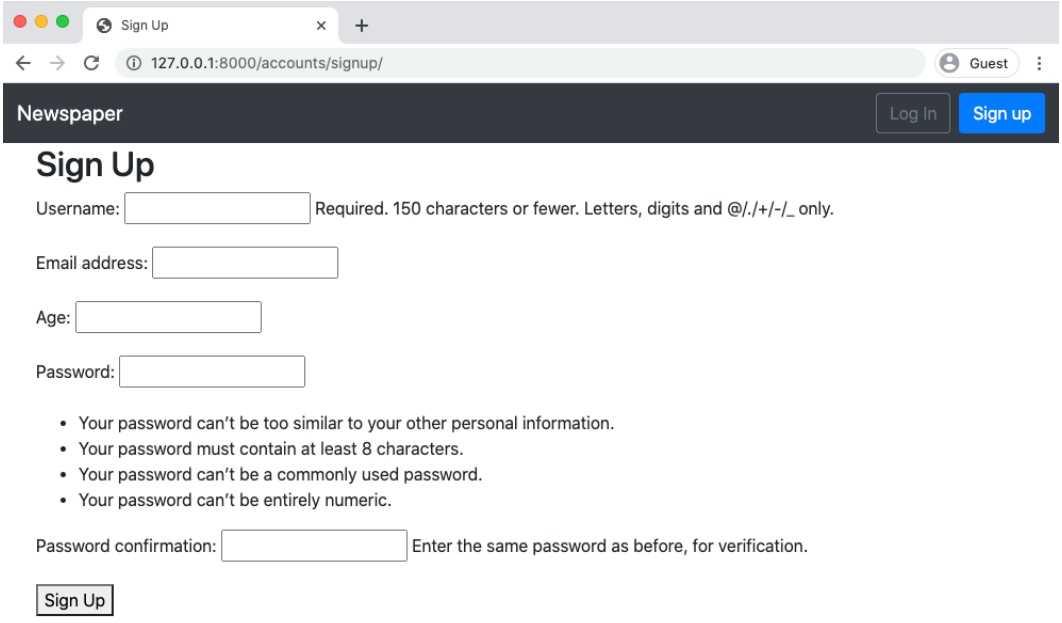


Bootstrap log in with new button

Sign Up Form

Our sign up page at `http://127.0.0.1:8000/accounts/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 page for `django/contrib/auth/models.py`. As of this writing, this is the link and the specific line is on line 334. 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 (Don't-Repeat-Yourself) 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==1.9.2
```

Add the new app to our `INSTALLED_APPS` list in the `config/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

```
# config/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', # new

    # Local
    'accounts',
    'pages',
]
```

Add a new setting for `CRISPY_TEMPLATE_PACK`, set to `bootstrap4`, to the bottom of our `config/settings.py` file.

Code

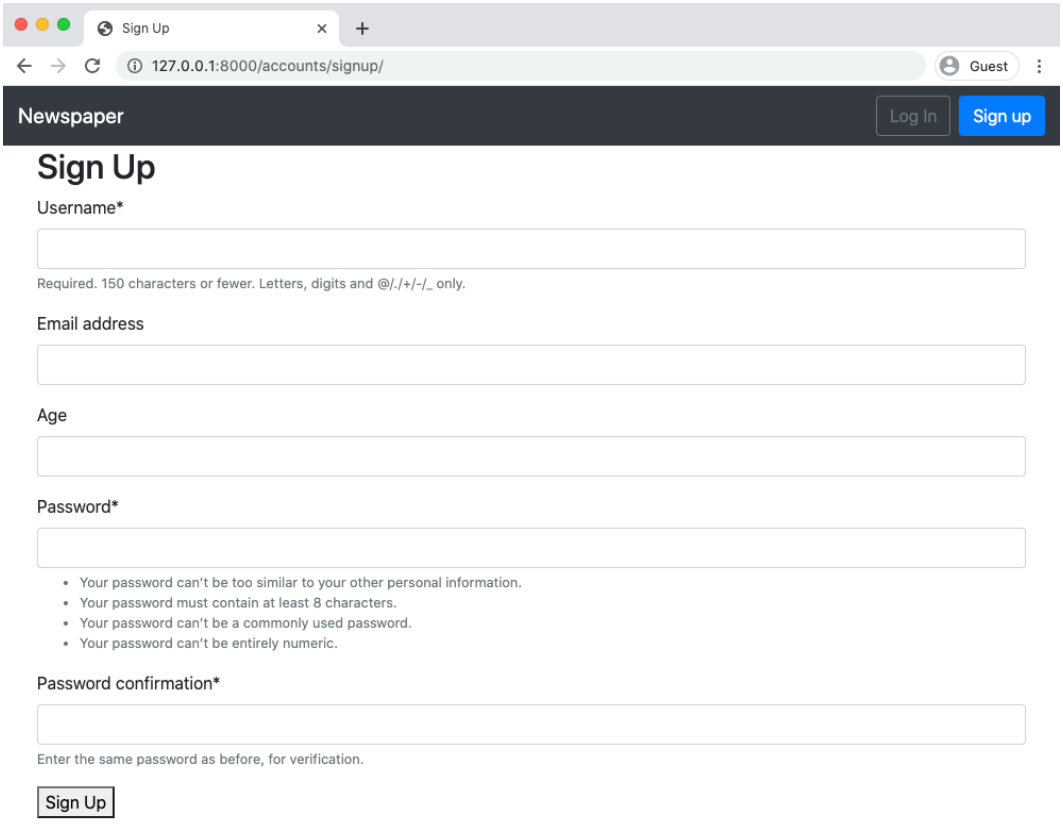
```
# config/settings.py
CRISPY_TEMPLATE_PACK = 'bootstrap4'
```

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 title%}

{% block content %}
  <h2>Sign Up</h2>
  <form method="post">
    {% csrf_token %}
    {{ form|crispy }}
    <button type="submit">Sign Up</button>
  </form>
{% endblock content %}
```

If you start up the server again with `python manage.py runserver` and refresh the sign up page we can see the new changes.



Crispy sign up page

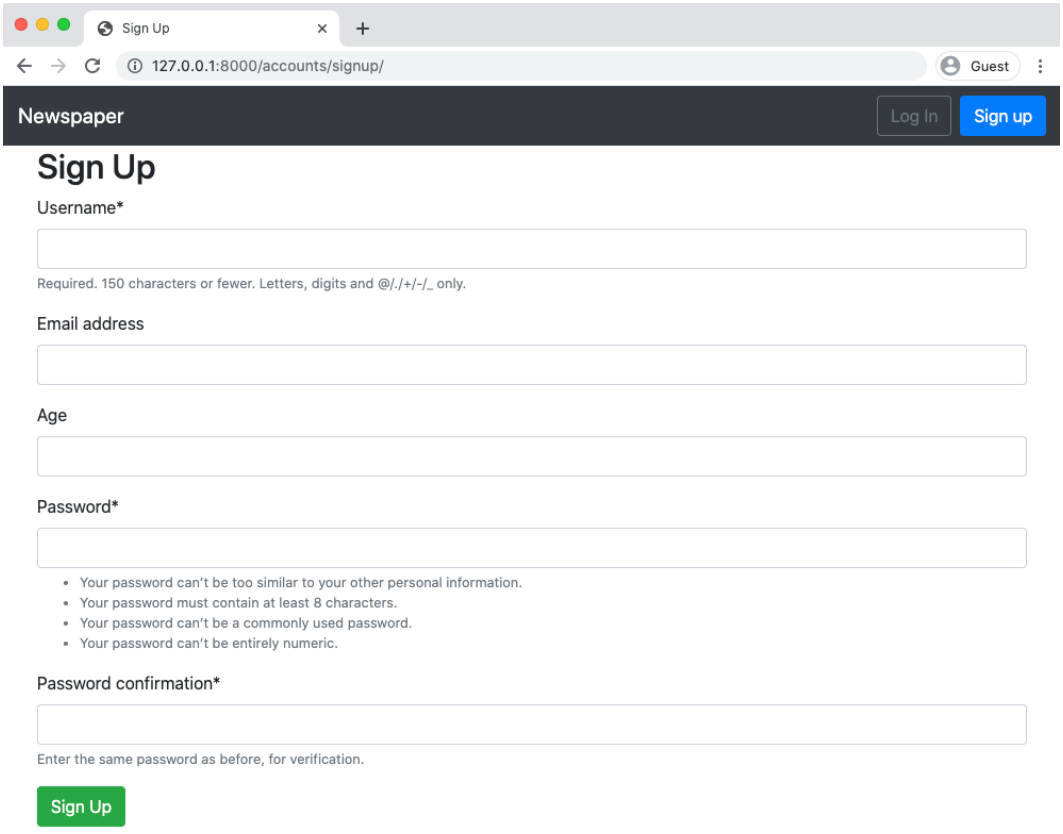
Much better. Although how about if our "Sign Up" button was a little more inviting? Maybe make it green? Bootstrap has all sorts of button styling options we can choose from. Let's use the "success" one which has a green background and white text.

Update the `signup.html` file on the line for the sign up button.

Code

```
<!-- templates/registration/signup.html -->
{% extends 'base.html' %}
{% load crispy_forms_tags %}
{% block title %}Sign Up{% endblock title%}

{% block content %}
  <h2>Sign Up</h2>
  <form method="post">
    {% csrf_token %}
    {{ form|crispy }}
    <button class="btn btn-success" type="submit">Sign Up</button>
  </form>
{% endblock content %}
```

Refresh the page and you can see our updated work.



Crispy sign up page green button

Conclusion

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

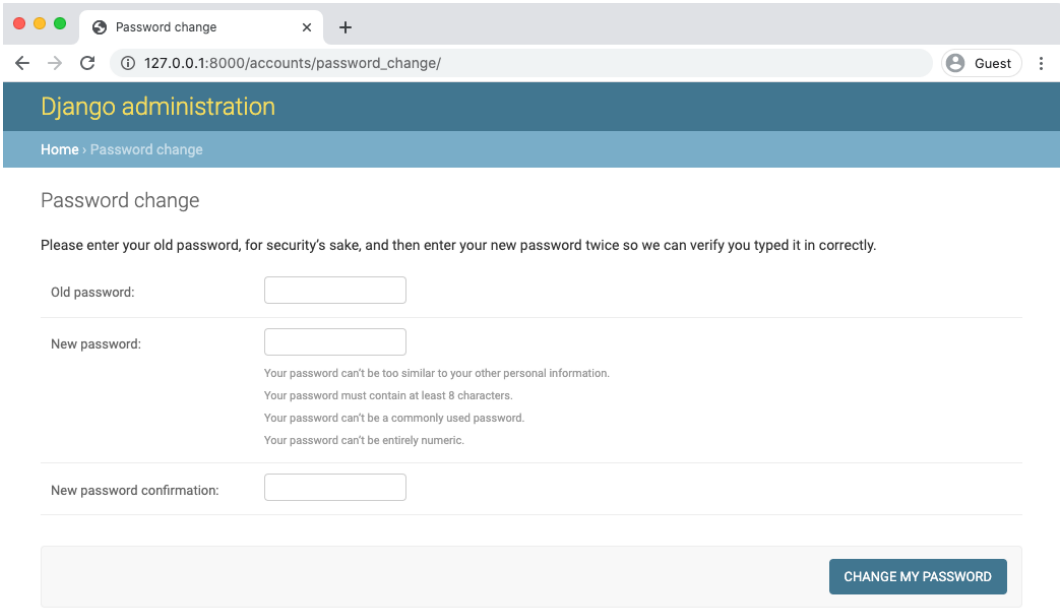
In this chapter we will complete the authorization flow of our *Newspaper* app by adding password change and reset functionality. Users will be able to change their current password or, if they've forgotten it, to reset it via email.

Initially we will implement Django's built-in views and URLs for both password change and password reset before then customizing 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, which is located at:

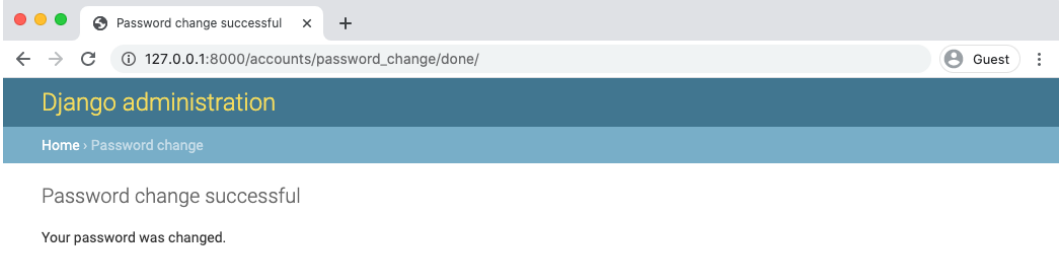
```
http://127.0.0.1:8000/accounts/password_change/
```



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/accounts/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 change the templates.

On the command line stop the local server `Control+c` and create two new template files in the `registration` directory.

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.

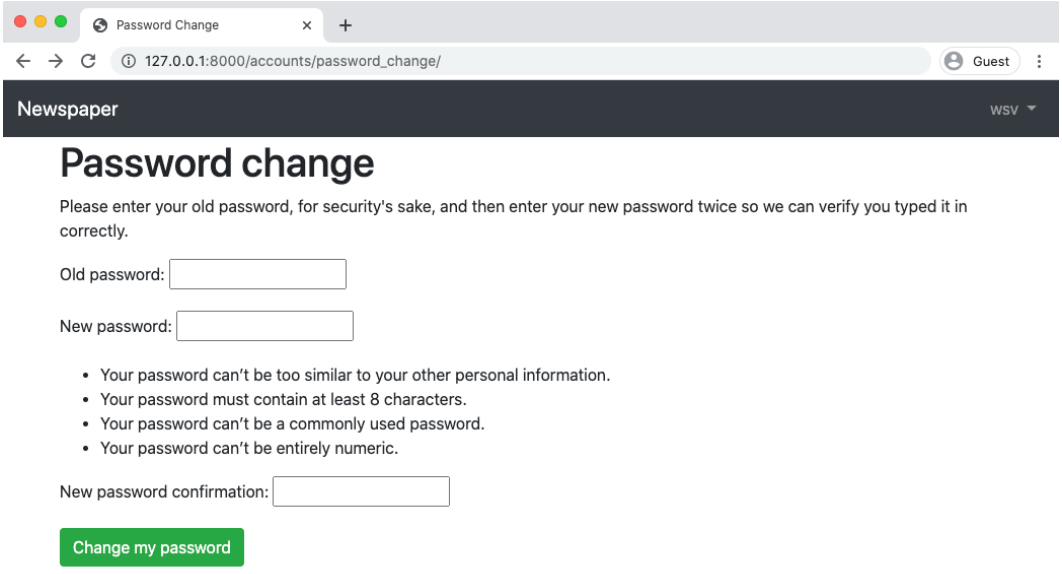
Code

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

{% block title %}Password Change{% endblock title %}

{% block content %}
  <h1>Password change</h1>
  <p>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.</p>

  <form method="POST">
    {% csrf_token %}
    {{ form.as_p }}
    <input class="btn btn-success" type="submit"
      value="Change my password">
  </form>
{% endblock content %}
```

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/accounts/password_change/`) to see our changes.



New password change form

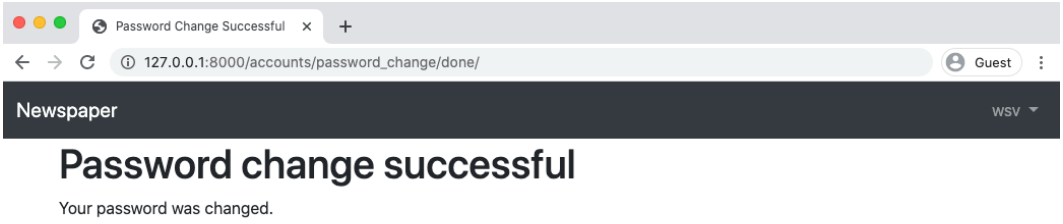
Next up is the `password_change_done` template.

Code

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

{% block title %}Password Change Successful{% endblock title %}

{% 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. This updated page is at:

`http://127.0.0.1:8000/accounts/password_change/done/`



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 the 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 look and feel of the rest of our site.

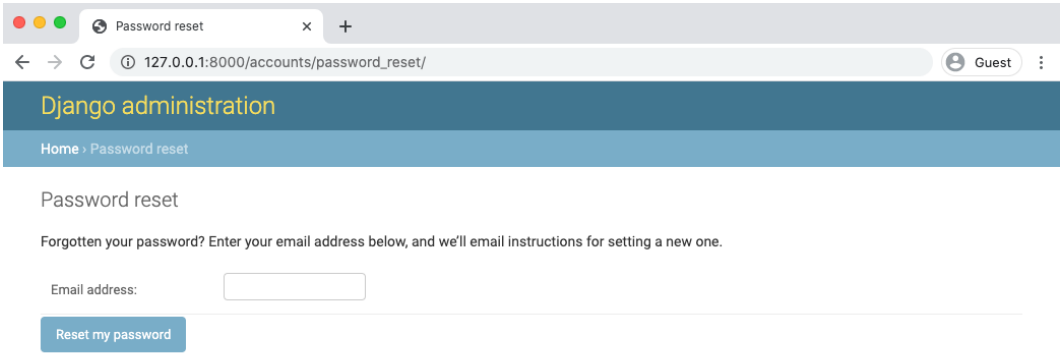
The only configuration required is telling Django **how** to send emails. After all, a user can only reset a password if they have access to the email linked to the account. In production, we'll use the email service SendGrid to actually send the emails but for testing purposes we can rely on Django's console backend setting which outputs the email text to our command line console instead.

At the bottom of the `config/settings.py` file make the following one-line change.

Code

```
# config/settings.py
EMAIL_BACKEND = 'django.core.mail.backends.console.EmailBackend'
```

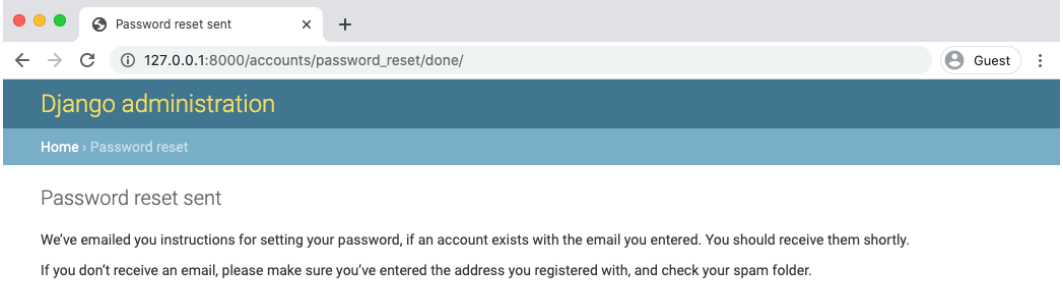
And we're all set! Django will take care of all the rest for us. Let's try it out. Navigate to `http://127.0.0.1:8000/accounts/password_reset/` to view the default password reset page.



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/accounts/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.

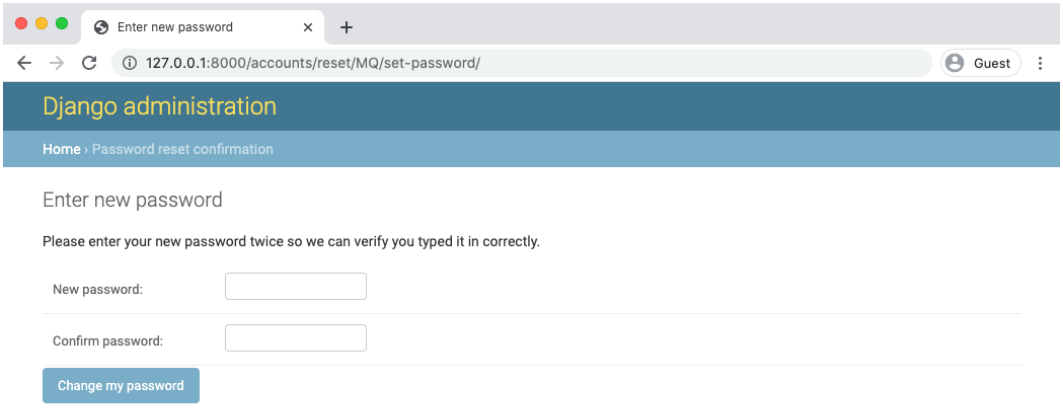
Command Line

```
Content-Type: text/plain; charset="utf-8"
MIME-Version: 1.0
Content-Transfer-Encoding: 8bit
Subject: Password reset on 127.0.0.1:8000
From: webmaster@localhost
To: will@learndjango.com
Date: Fri, 03 Aug 2020 22:41:00 -0000
Message-ID: <159563046041.20868.9904958624780076281@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/accounts/reset/MQ/a7jdoc-d05b03730b\
6586d76fc80d82c43de5a9/

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:

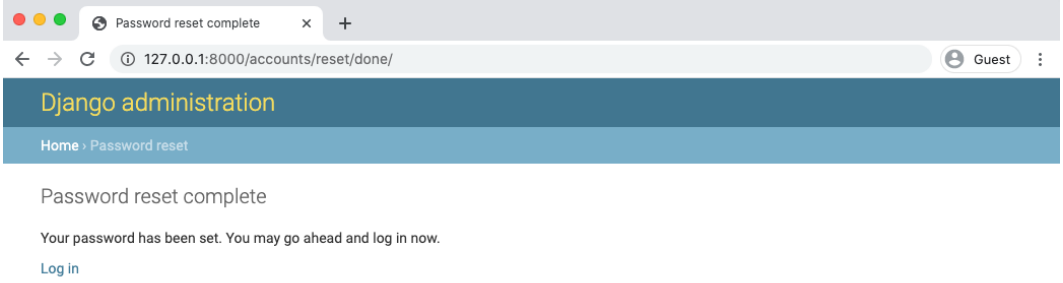
- the “To” on the sixth line contains the email address of the user
- the URL link contains a secure token that Django randomly generates for us and can be used only once
- Django helpfully reminds us of our username

We will customize all of the email default text shortly but for now focus on finding the link provided and enter it into your web browser and you’ll be redirected to the “change password page”.



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.



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. Stop the local server with `Control+c` and then 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

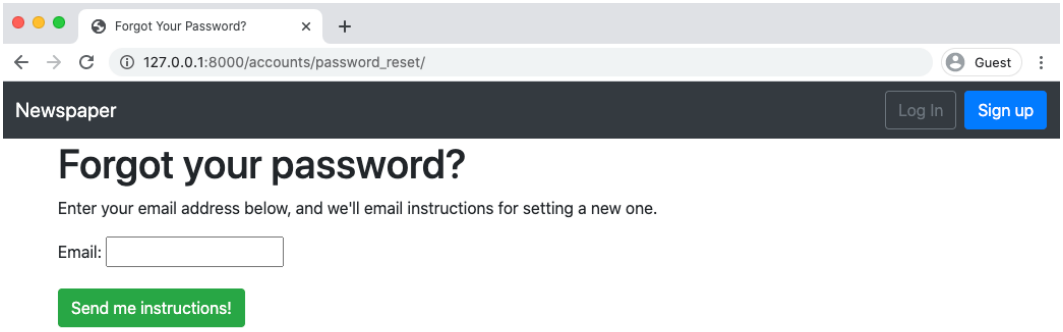
```
<!-- templates/registration/password_reset_form.html -->
{% extends 'base.html' %}

{% block title %}Forgot Your Password?{% endblock title %}

{% block content %}
<h1>Forgot your password?</h1>
<p>Enter your email address below, and we'll email instructions
for setting a new one.</p>

<form method="POST">
  {% csrf_token %}
  {{ form.as_p }}
  <input class="btn btn-success" type="submit"
    value="Send me instructions!">
</form>
{% endblock content %}
```

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.

Start up the server again with `python manage.py runserver` and navigate to:

`http://127.0.0.1:8000/accounts/password_reset/`

Refresh the page you can see our new page.



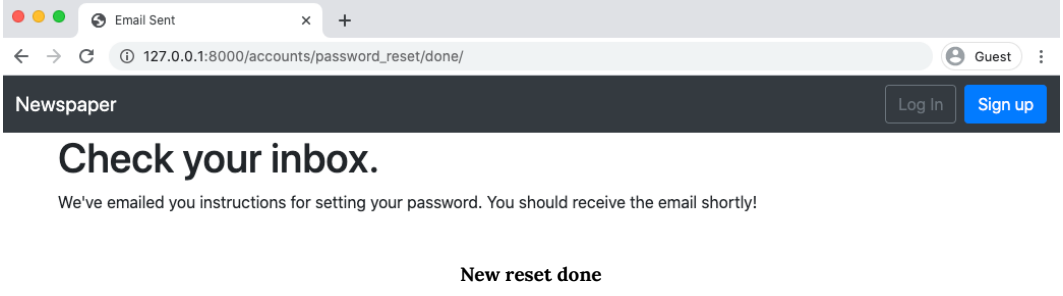
New password reset

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.html` an updated form as well.

Code

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

{% block title %}Email Sent{% endblock title %}

{% block content %}
  <h1>Check your inbox.</h1>
  <p>We've emailed you instructions for setting your password.
  You should receive the email shortly!</p>
{% endblock content %}
```

Confirm the changes by going to `http://127.0.0.1:8000/accounts/password_reset/done/`.



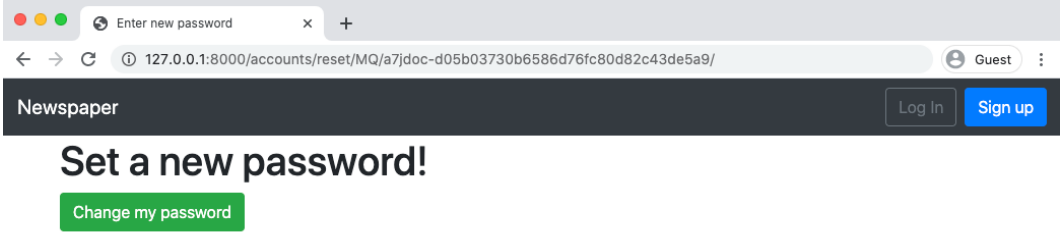
New reset done

Next the password reset confirm page.

Code

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

{% block title %}Enter new password{% endblock title %}

{% 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 content %}
```

In the command line grab the URL link from the email outputted to the console and you'll see the following.



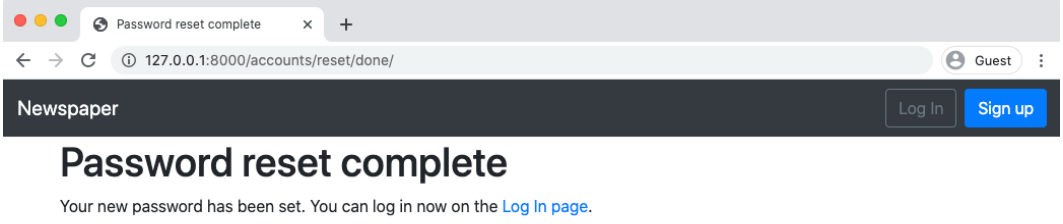
New set password

Finally here is the password reset complete code.

Code

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

{% block title %}Password reset complete{% endblock title %}

{% 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 content %}
```

You can view it at `http://127.0.0.1:8000/accounts/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 so our automated emails are actually sent to users as opposed to outputting them in the 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 that 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.

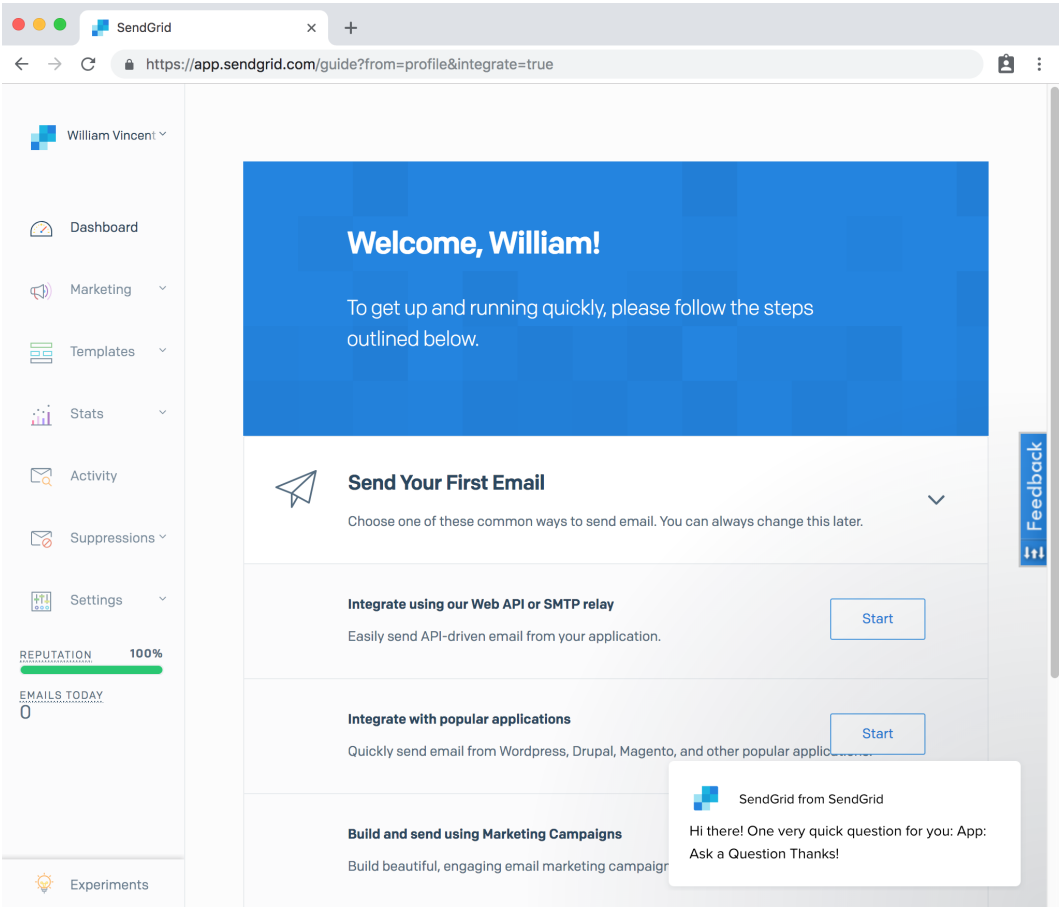
Currently, emails are outputted to our command line console, they are not actually sent to users. Let's change that! First we need to sign up for an account at SendGrid and then update our `config/settings.py` files. Django will take care of the rest. Ready?

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 the "Try for free" button in the upper right corner. Enter in your email address, username, and password to create a free 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 weird errors may result. Finally, complete the "Tell Us About Yourself" page. The only tricky part might be the "Company Website" section. I recommend using the URL of a Heroku deployment from a previous chapter here as this setting can later be changed. Then on the bottom of the page click the "Get Started" button.

SendGrid then presents us with a welcome screen that provides three different ways to send our first email. Select the first option, "Integrate using our Web API or SMTP relay" and click on the "Start" button next to it.

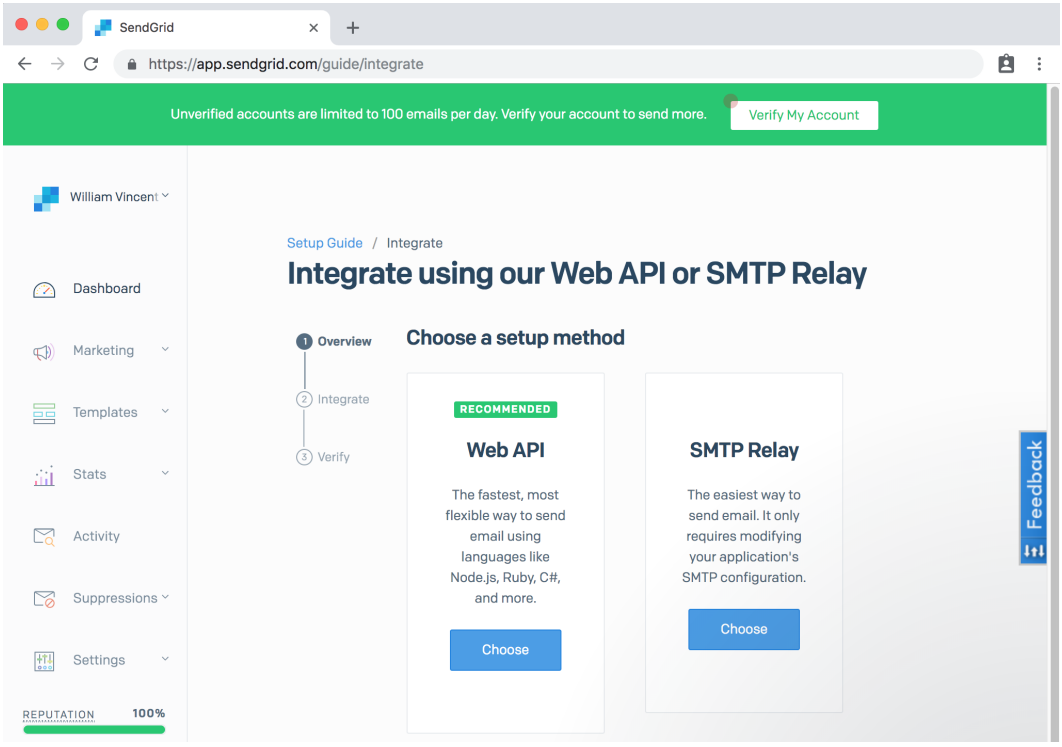


SendGrid welcome screen

Now we have one more choice to make: Web API or SMTP Relay. We'll use SMTP since it is the simplest and works well for our basic needs here. In a large-scale website you likely would want to use the Web API instead but ... one thing at a time.

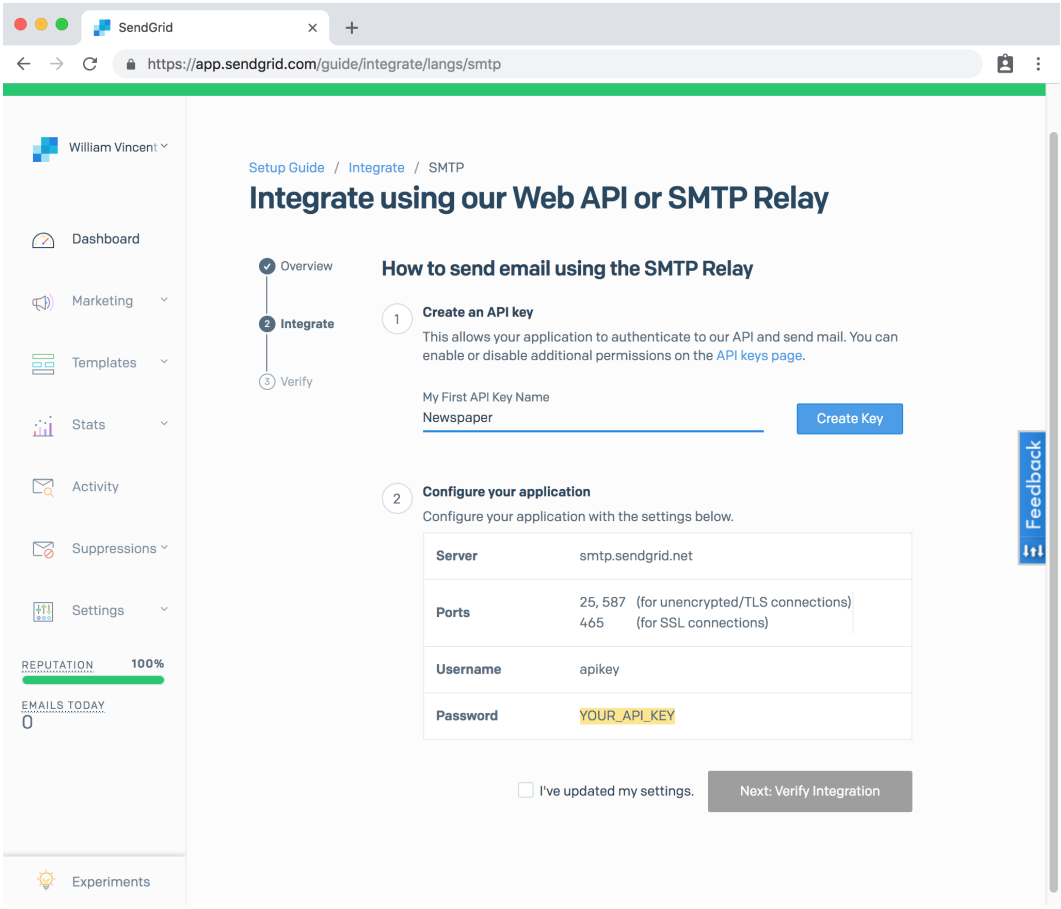
You'll also note the "Verify My Account" banner on the top of the page. If you want that to go away, log in to the email account you used for the account and confirm your account.

Click on the "Choose" button under "SMTP Relay" to proceed.



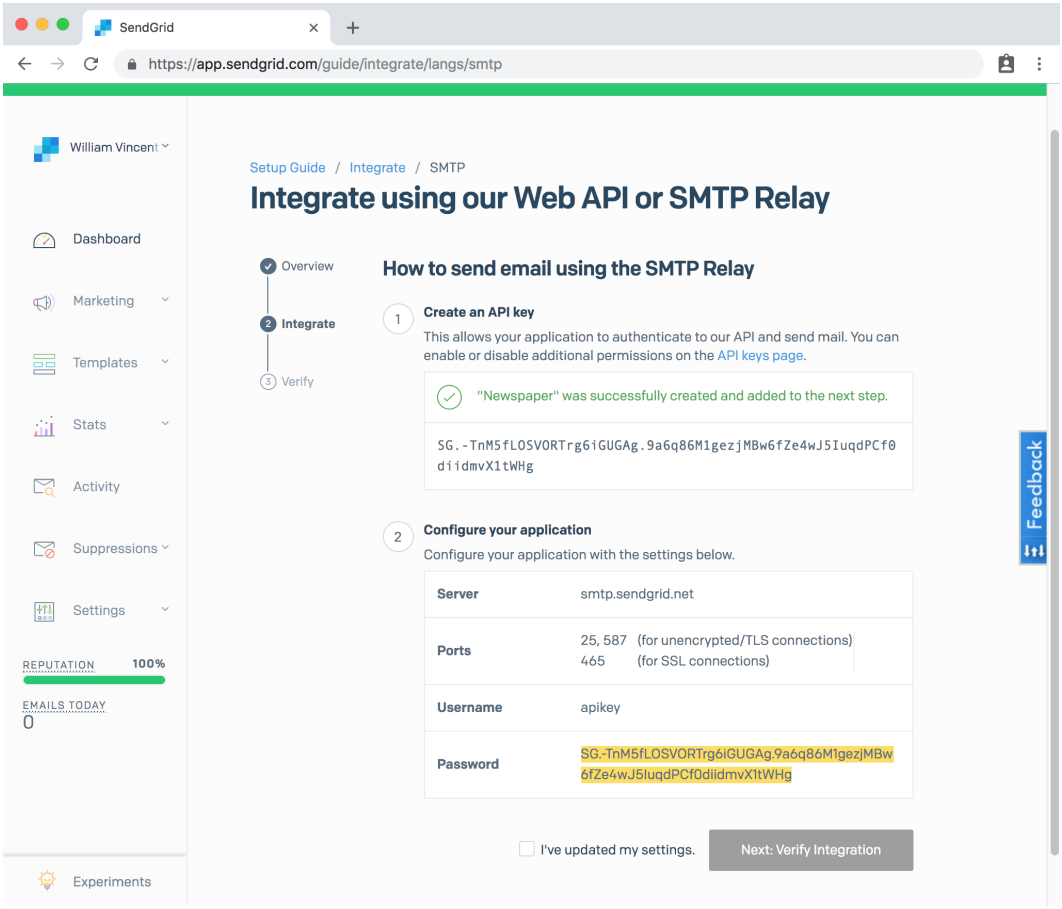
SendGrid Web API vs SMTP Relay

Ok, one more screen to navigate. Under step 1, "Create an API key," enter in a name for your first API Key. I've chosen the name "Newspaper" here. Then click on the blue "Create Key" button next to it.



SendGrid Integrate

The page will update and generate a custom API key in part 1. SendGrid is really pushing us to use API keys, no? But that's ok, it will *also*, under part 2, create a username and password for us that we can use with an SMTP relay. This is what we want.



SendGrid username and password

The username here, `apikey`, is the same for everyone but the password will be different for each account. Now, time to add the new username and password into our Django project. This won't take long!

First, in the `config/settings.py` file update the email backend to use SMTP. We already configured this once before; the line should be at the bottom of the file. Instead of outputting emails to the console we want to instead send them for real using SMTP.

Code

```
# config/settings.py
EMAIL_BACKEND = 'django.core.mail.backends.smtp.EmailBackend' # new
```

Then, right below it, add the following six lines of email configuration. The DEFAULT_FROM_EMAIL field is set, by default, to `webmaster@localhost`. You should update it with your intended email account. Make sure to enter your own SendGrid `EMAIL_HOST_PASSWORD` here; `sendgrid_password` is just a placeholder!

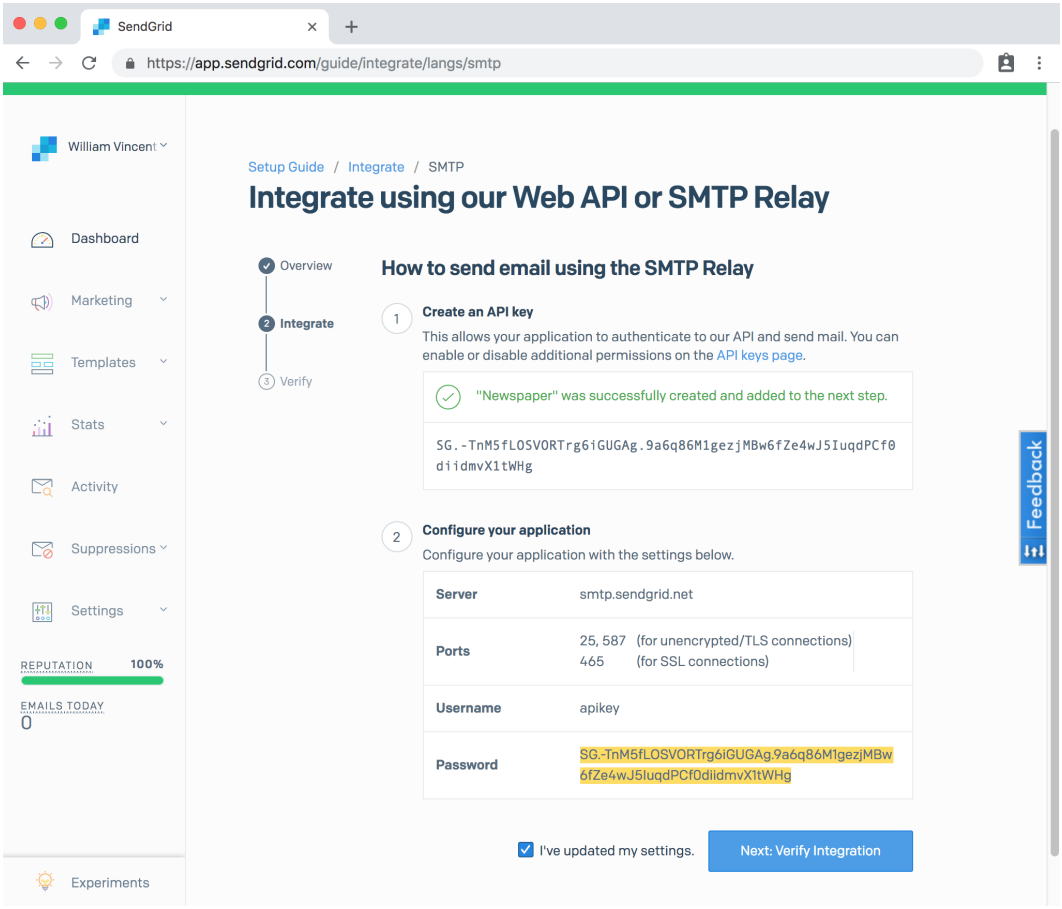
Code

```
# config/settings.py
DEFAULT_FROM_EMAIL = 'your_custom_email_account'
EMAIL_HOST = 'smtp.sendgrid.net'
EMAIL_HOST_USER = 'apikey'
EMAIL_HOST_PASSWORD = 'sendgrid_password'
EMAIL_PORT = 587
EMAIL_USE_TLS = True
```

Also, note that ideally you would store secure information like your password in environment variables, not in plain text. But, to keep things simple, we won't do that here. However, in a proper production environment you should.

Once complete, we're ready to confirm everything is working. The local server should be already running at this point but if not, type `python manage.py runserver` to ensure that it is.

Go *back* to the SendGrid "Integrate using our Web API or SMTP Relay" page and select the checkbox next to "I've updated my settings." Then click on "Next: Verify Integration."

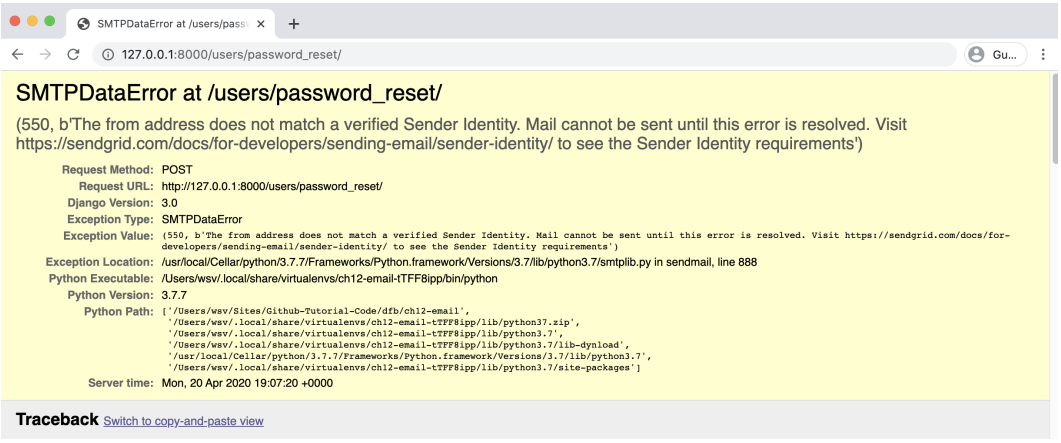


SendGrid updated settings

Navigate to the password reset form in your web browser, which should be located at:

```
http://127.0.0.1:8000/accounts/password_reset/
```

Type in the email address for your superuser account. Do not use the email for your SendGrid account, which should be different. Fill in the form and submit.



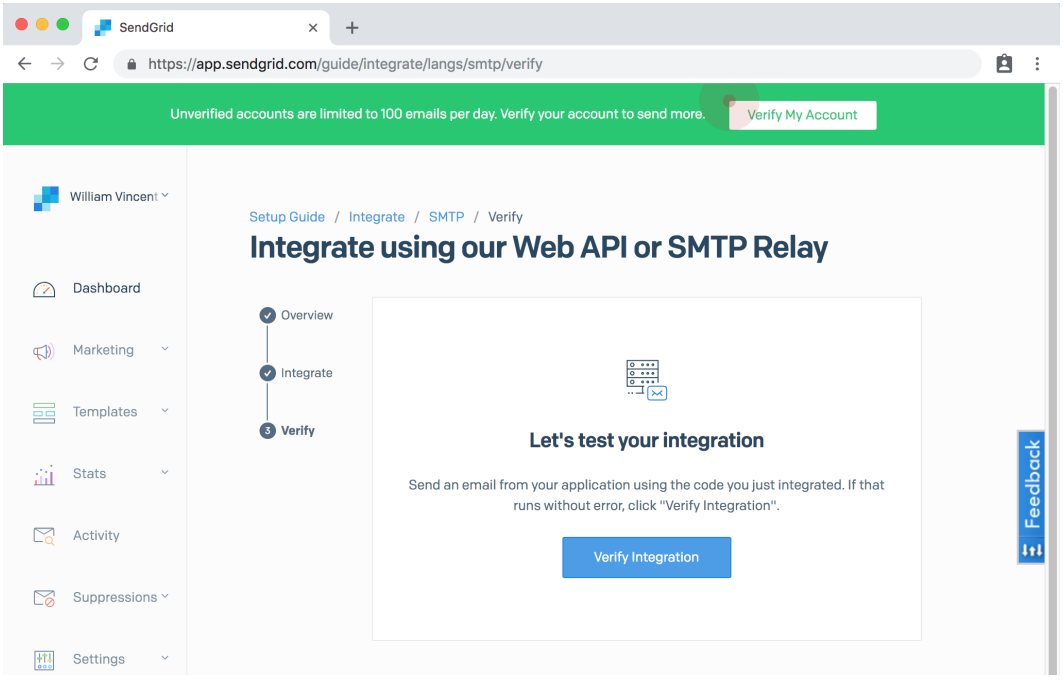
SMTPDataError

Ack, what's this? If you created a free SendGrid account after April 6, 2020, then single sender verification is required. Essentially, this is an additional step to help SendGrid comply with anti-spam laws. To fix it, we'll need to follow SendGrid's instructions to verify an email account. And while previously it was possible to send emails from a free address at services like `gmail.com` or `yahoo.com`, that is no longer the case due to the DMARC email authentication protocol. So to send actual emails now you must use a custom, non-free email account which you can verify ownership of.

After completing this additional step, stop the local web server with `Control+c` and start it up again with our handy `runserver` command. Then navigate back to the password reset page and fill out the form again.

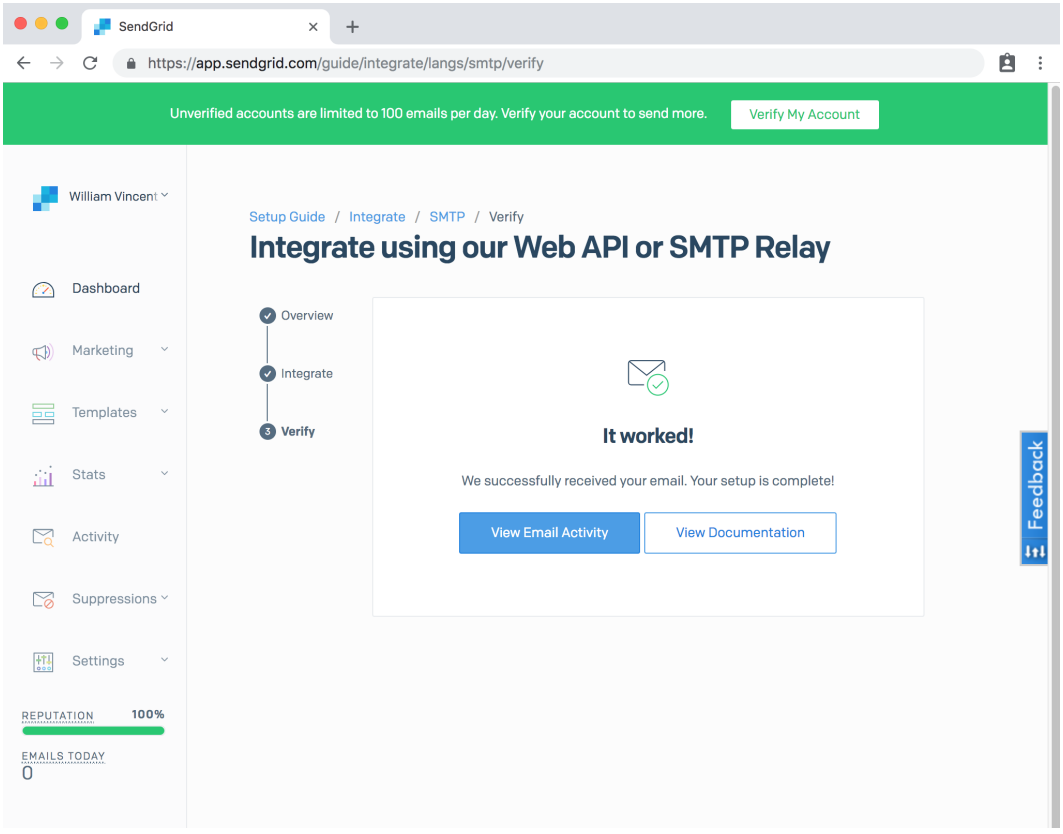
Now check your email inbox. You should see a new email there from your `DEFAULT_FROM_EMAIL` email address which was just verified. The text will be exactly the same as that outputted to our command line console previously.

The final step is to return to SendGrid and click on the blue button to "Verify Integration."



SendGrid verify integration

The button will turn grey and display “Checking…” for a moment until displaying “It worked!”



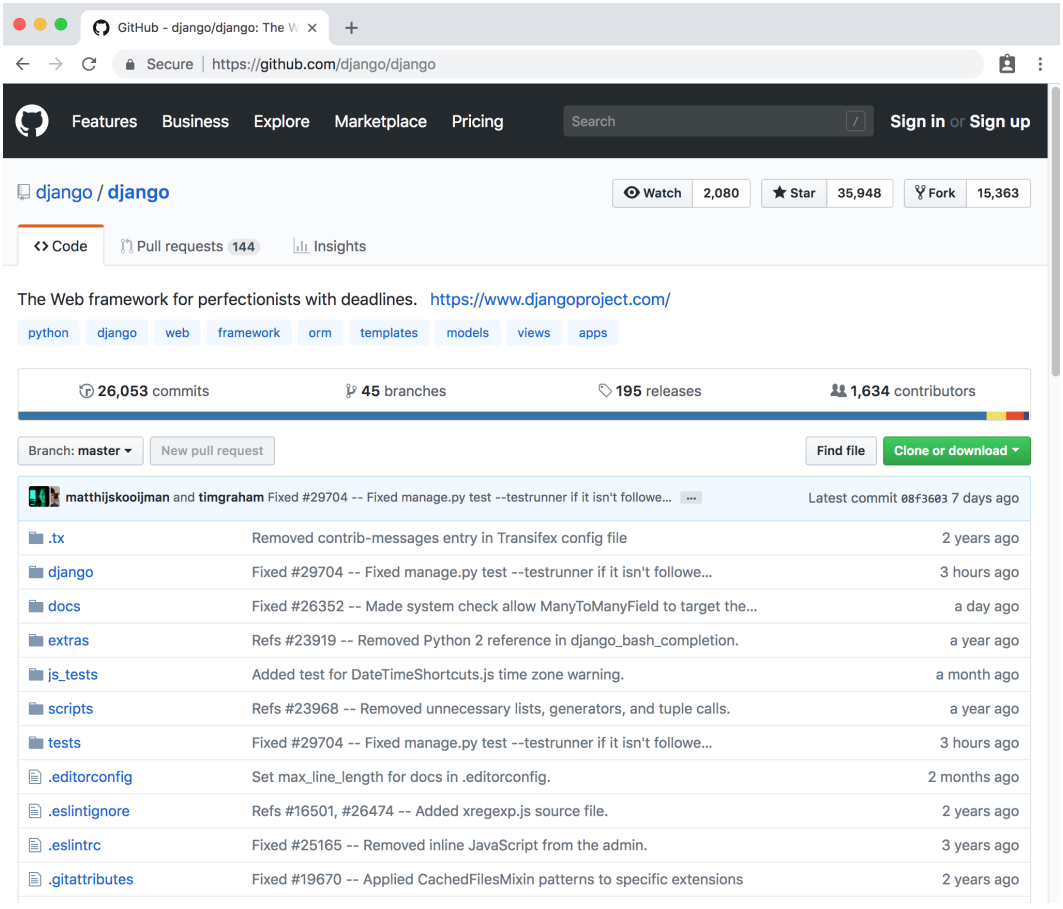
SendGrid it worked

Phew. We're done! That was a lot of steps but our real-world email integration is now working.

Custom Emails

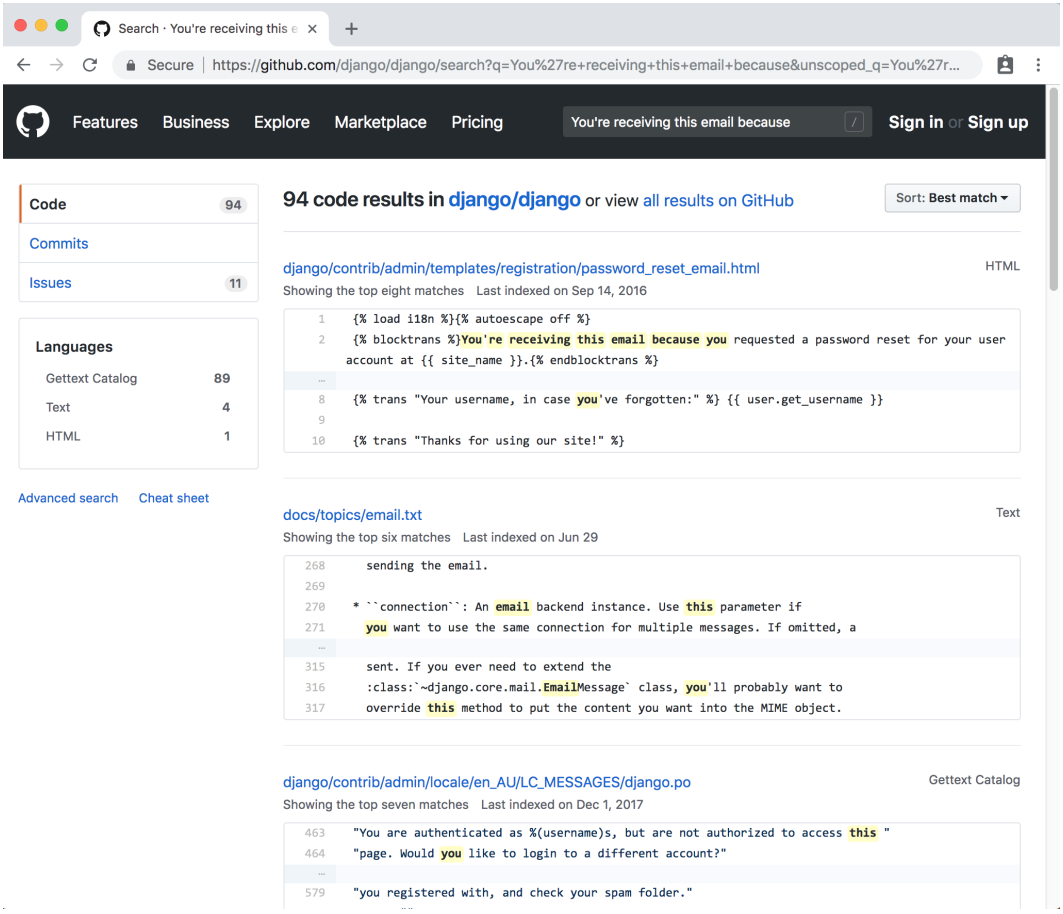
The current email text isn't very personal, is it? Let's change things. At this point I could just show you what steps to take, but I think it's helpful if I can explain **how** I figured out how to do this. After all, you want to be able to customize all parts of Django as needed.

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 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 in the `contrib` app in a file called `password_reset_email.html`.

`django/contrib/admin/templates/registration/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 %}
```

To make changes create a `password_reset_email.html` file in our `registration` directory. Stop the server with `Control+c` and use `touch` for the new file.

Command Line

```
(news) $ touch templates/registration/password_reset_email.html
```

Then copy and paste the code from the Django repo into it. If you want to customize the text, you can.

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 `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 another new file called `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.

password_reset_subject.txt

```
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/accounts/password_reset/`. Then check your email and it will have the desired content and updated subject.

Conclusion

We've now finished implementing a complete user authentication flow. Users can sign up for a new account, log in, log out, 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 `config/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

```
# config/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
    'accounts',
    'pages',
    'articles', # new
]

TIME_ZONE = 'America/New_York' # new
```

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 `'accounts.CustomUser'` which we set in the `config/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 `articles/admin.py` so our new app is displayed.

Code

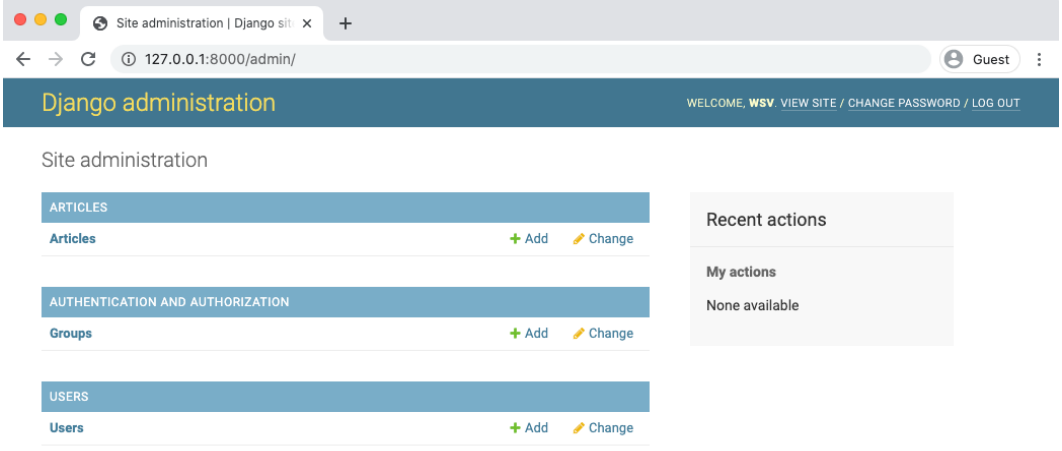
```
# articles/admin.py
from django.contrib import admin
from .models import Article

admin.site.register(Article)
```

Now we start the server.

Command Line

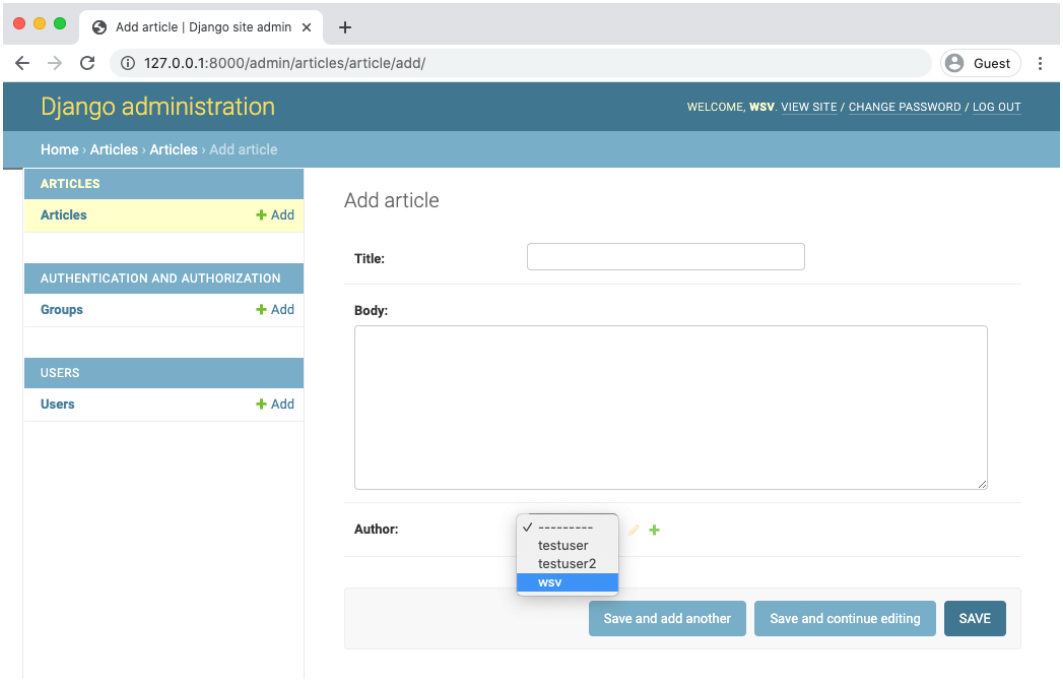
```
(news) $ python manage.py runserver
```

Navigate to `http://127.0.0.1:8000/admin/` and log in.



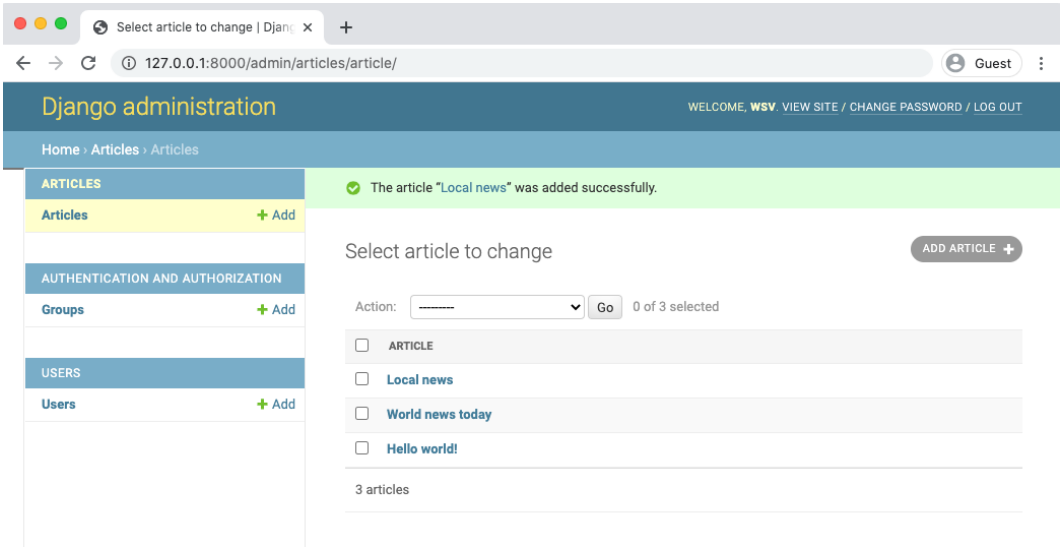
Admin page

If you click on "+ Add" next to "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.



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/`. Add a URL pattern for `articles` in our `config/urls.py` file.

Code

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

urlpatterns = [
    path('admin/', admin.site.urls),
    path('accounts/', include('accounts.urls')),
    path('accounts/', include('django.contrib.auth.urls')),
    path('articles/', include('articles.urls')), # new
    path('', include('pages.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 .views import ArticleListView

urlpatterns = [
    path('', 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 .models import Article


class ArticleListView(ListView):
    model = 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.

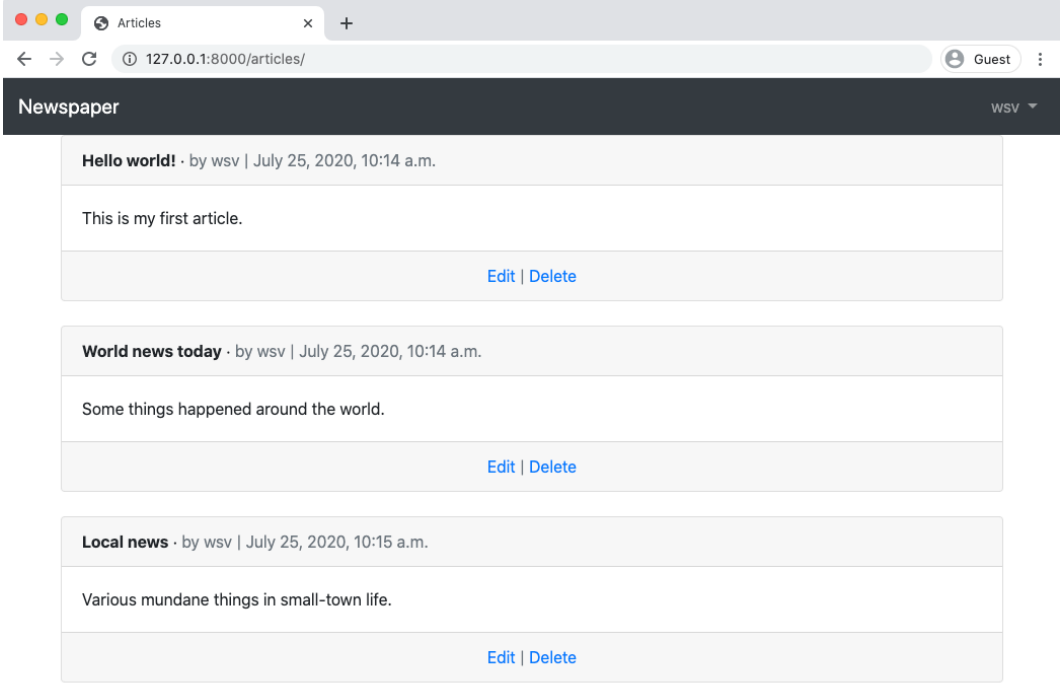
Within each `article` we display the title, body, author, and date. We can even provide links to "edit" and "delete" functionality that we haven't built yet.

Code

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

{% block title %}Articles{% endblock title %}

{% 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>
    </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 .views import (
    ArticleListView,
    ArticleUpdateView, # new
    ArticleDetailView, # new
    ArticleDeleteView, # new
)

urlpatterns = [
    path('<int:pk>/edit/',
         ArticleUpdateView.as_view(), name='article_edit'), # new
    path('<int:pk>/',
         ArticleDetailView.as_view(), name='article_detail'), # new
    path('<int:pk>/delete/',
         ArticleDeleteView.as_view(), name='article_delete'), # new
    path('', ArticleListView.as_view(), name='article_list'),
]
```

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 # new
from django.views.generic.edit import UpdateView, DeleteView # new
from django.urls import reverse_lazy # new
from .models import Article


class ArticleListView(ListView):
    model = Article
    template_name = 'article_list.html'


class ArticleDetailView(DetailView): # new
    model = Article
    template_name = 'article_detail.html'


class ArticleUpdateView(UpdateView): # new
    model = Article
    fields = ('title', 'body',)
    template_name = 'article_edit.html'


class ArticleDeleteView(DeleteView): # new
    model = 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 %}
    {{ form.as_p }}
    <button class="btn btn-info ml-2" type="submit">Update</button>
  </form>
{% endblock content %}
```

Code

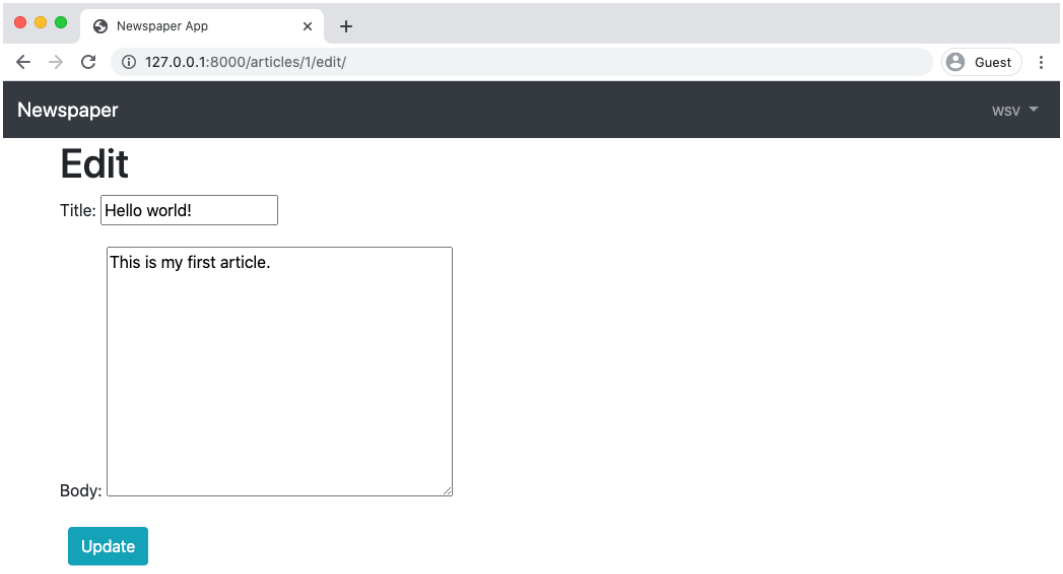
```
<!-- templates/article_delete.html -->
{% extends 'base.html' %}

{% 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 content %}
```

As a final step, in the `card-footer` section of `article_list.html` we can swap out the `"` placeholder `a hrefs` in place of the actual URL routes, using the url template tag, the URL name, and as a parameter the `pk` of each.

Code

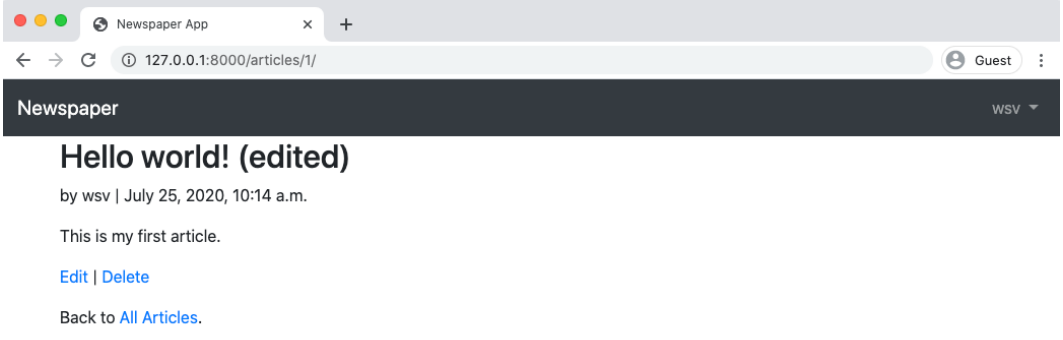
```
<!-- templates/article_list.html -->
...
<div class="card-footer text-center text-muted">
  <a href="{% url 'article_edit' article.pk %}">Edit</a> |
  <a href="{% url 'article_delete' article.pk %}">Delete</a>
</div>
...
```

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/`.



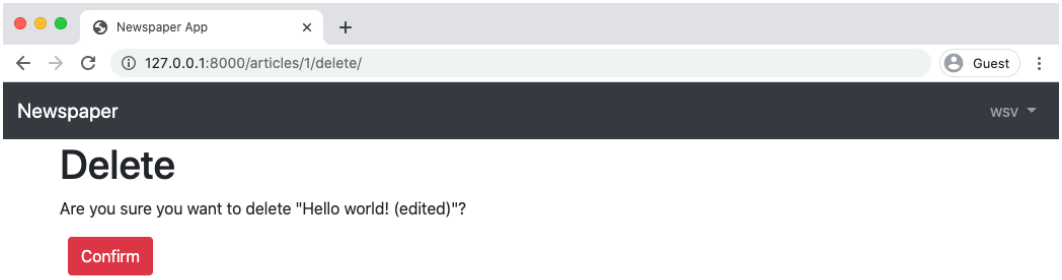
Edit page

If you update the "title" field by adding "(edited") at the end and click update you'll be redirected to the detail page which shows the new change.



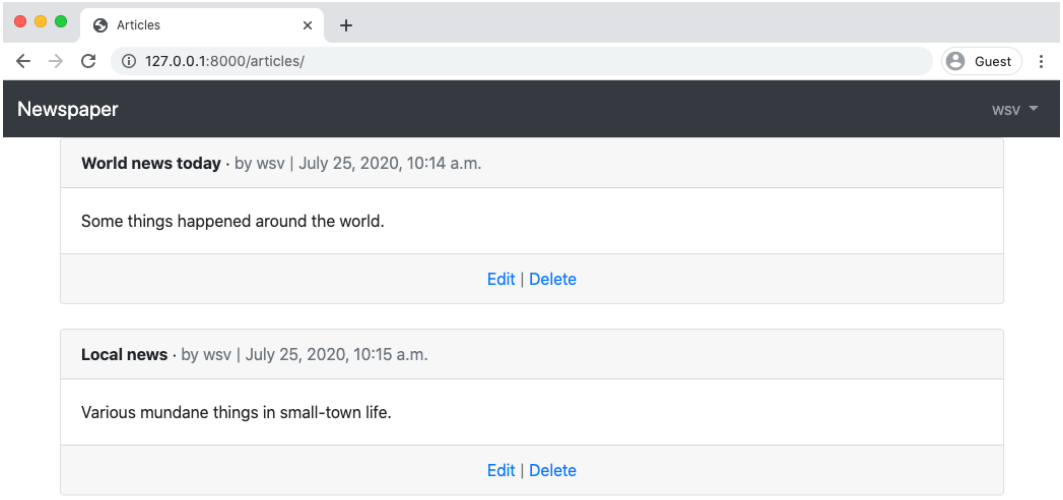
Detail page

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 at the bottom of the file `ArticleCreateView` that specifies our model, template, and the fields available.

Code

```
# articles/views.py
...
from django.views.generic.edit import (
  UpdateView, DeleteView, CreateView # new
)
...
class ArticleCreateView(CreateView): # new
    model = 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
from django.urls import path

from .views import (
    ArticleListView,
    ArticleUpdateView,
    ArticleDetailView,
    ArticleDeleteView,
    ArticleCreateView, # new
)

urlpatterns = [
    path('<int:pk>/edit/',
         ArticleUpdateView.as_view(), name='article_edit'),
    path('<int:pk>/',
         ArticleDetailView.as_view(), name='article_detail'),
    path('<int:pk>/delete/',
         ArticleDeleteView.as_view(), name='article_delete'),
    path('new/', ArticleCreateView.as_view(), name='article_new'), # new
    path('', ArticleListView.as_view(), name='article_list'),
]
```

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 content %}
```

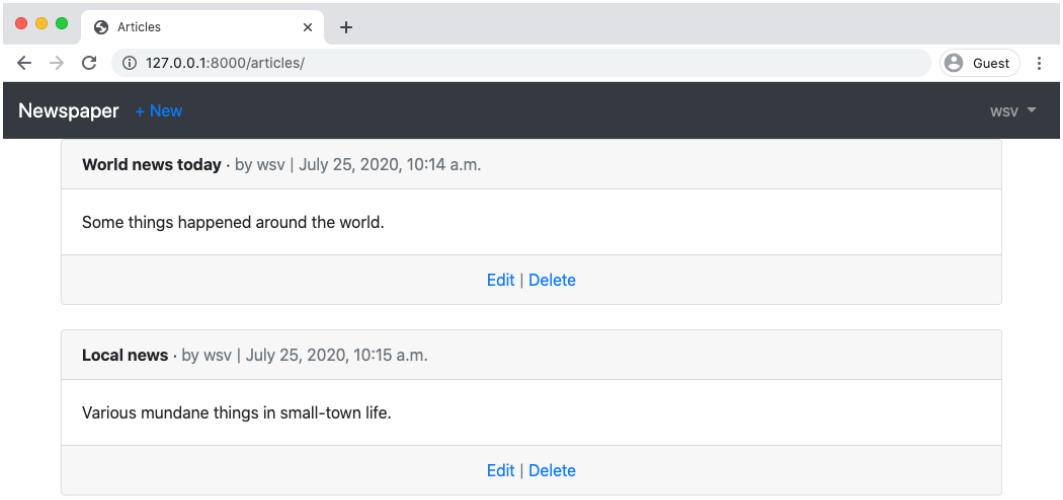
Finally, we should add a link to creating new articles in our navbar so it is accessible everywhere on the site to logged-in users.

Code

```
<!-- templates/base.html -->
<body>
  <nav class="navbar navbar-expand-lg navbar-dark bg-dark">
    <a class="navbar-brand" href="{% url 'home' %}">Newspaper</a>
    {% if user.is_authenticated %}
      <ul class="navbar-nav mr-auto">
        <li class="nav-item">
          <a href="{% url 'article_new' %}">+ New</a>
        </li>
      </ul>
    {% endif %}
    ...
```

If you need help to make sure your HTML file is accurate now, please refer to the official source code.

Refresh the articles page and the change is evident in the top navbar:



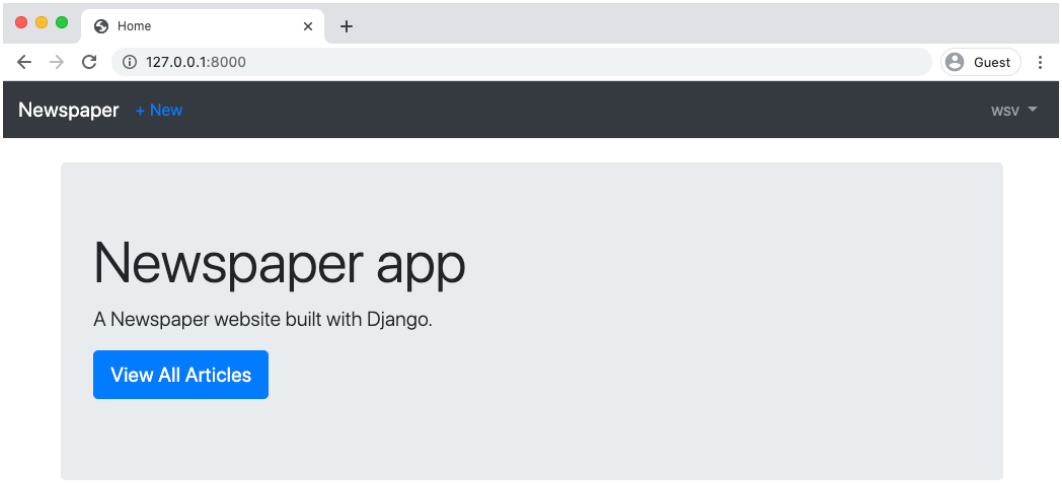
Navbar new link

Why not use Bootstrap to improve our original homepage now, too? We can update the template for `home.html` as follows.

Code

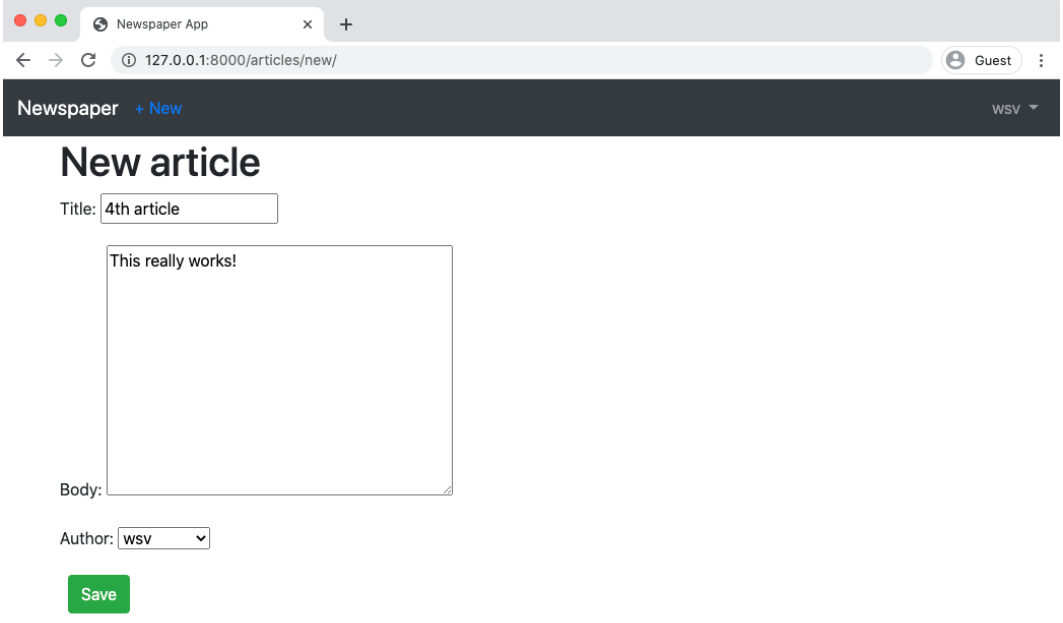
```
<!-- templates/home.html -->
{% extends 'base.html' %}

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

{% block content %}
  <br/>
  <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">View All Articles</a>
    </p>
  </div>
{% endblock content %}
```

We're all done. Let's just confirm everything works as expected. Navigate to our homepage at `http://127.0.0.1:8000/`.



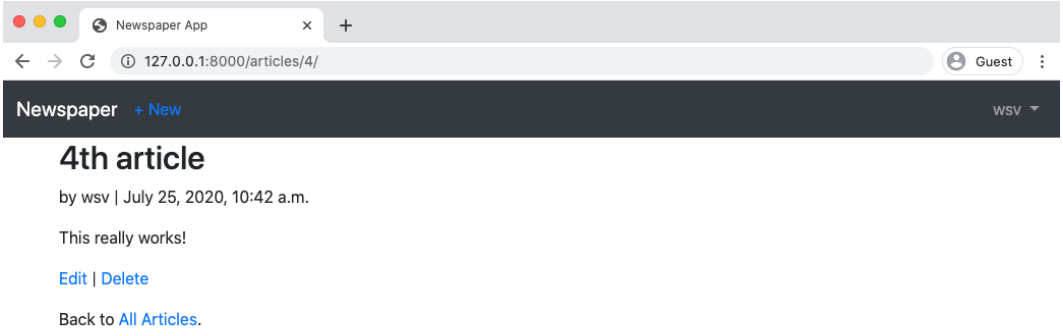
Homepage with new link in nav

Click on the link for "+ New" in the top navbar and you'll be redirected to our create page.



Create page

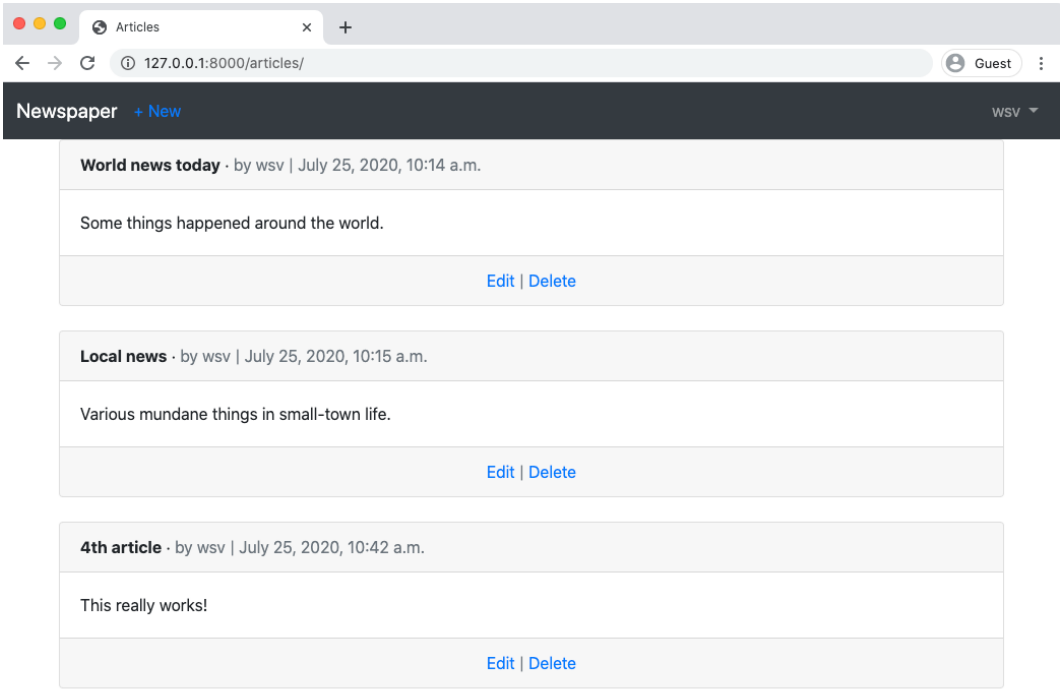
Go ahead and create a new article. Then click on the "Save" button. You will be redirected to the detail page. Why? Because in our `models.py` file we set the `get_absolute_url` method to `article_detail`. This is a good approach because if we later change the url pattern for the detail page to, say, `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.



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 sign up and log in 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

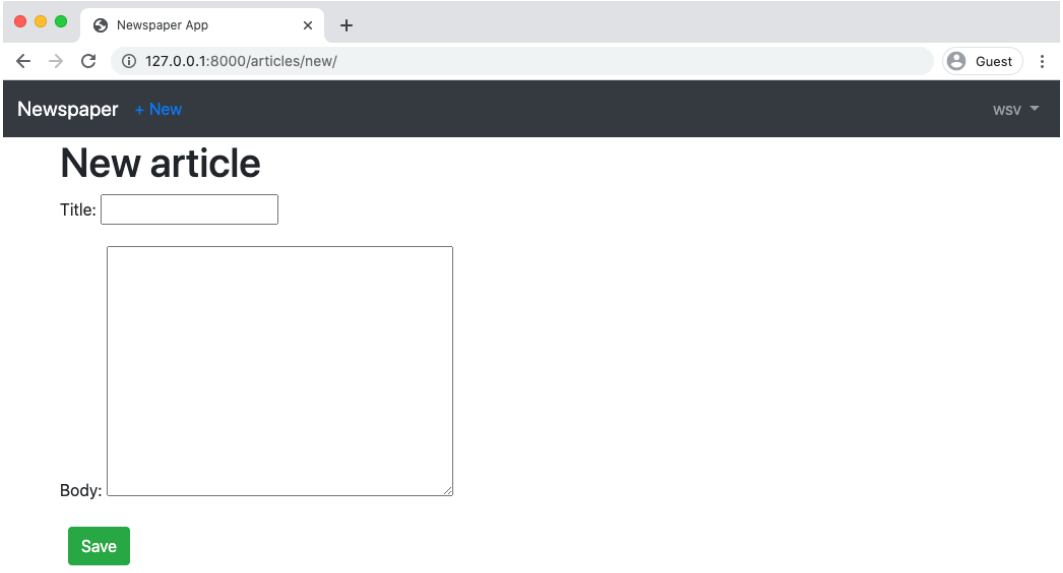
At present the `author` on a new article can be set to any user. Instead it should be automatically set to the current user. The default `CreateView` provides a lot of functionality for us but in order to set the current user to `author` we need to customize it. We will remove `author` from the `fields` and instead set it automatically via the `form_valid` method.

Code

```
# articles/views.py
...
class ArticleCreateView(CreateView):
    model = Article
    template_name = 'article_new.html'
    fields = ('title', 'body') # new

    def form_valid(self, form): # new
        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.

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.



New article link

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.

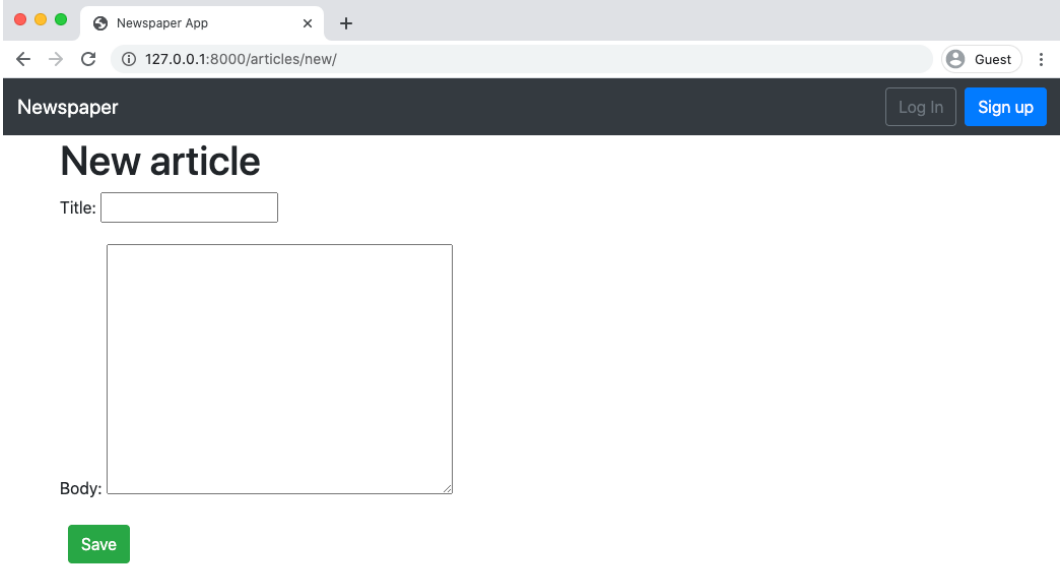
Authorizations

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.

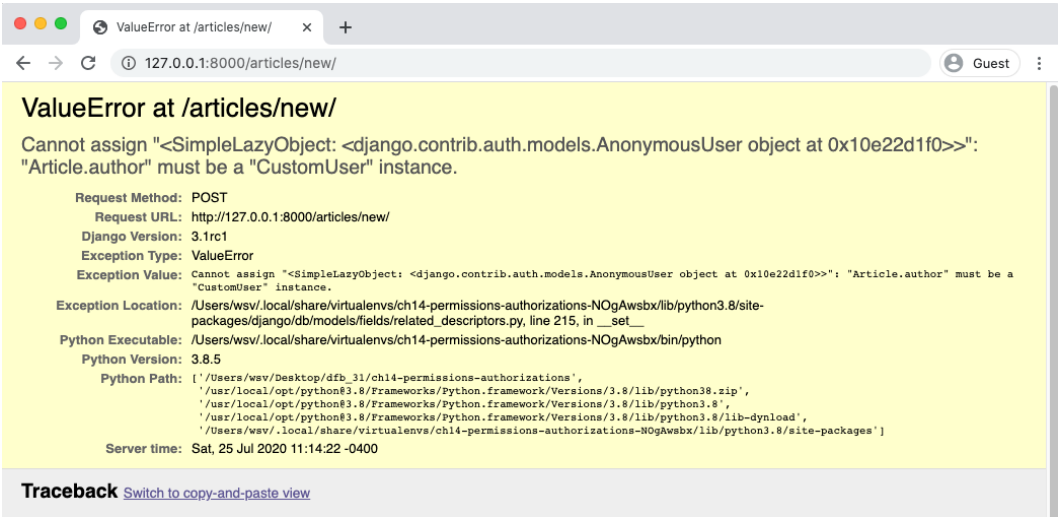
Consider what would happen if a logged-out user tried to create a new article? To try it out, click on your username in the upper right corner of the nav bar, then select "Log out" from the dropdown options. The "+ New" link disappears from the nav bar but what happens if you go to it directly: `http://127.0.0.1:8000/articles/new/`?

The page is still there.



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. What to do?

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 still allows 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 LoginRequired mixin that we can use. It's powerful and extremely concise.

In the `articles/views.py` file, import `LoginRequiredMixin` at the top and then it add to our `ArticleCreateView`. Make sure that the mixin is to the left of `CreateView` so it will be read first.

We want the `CreateView` 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
from django.views.generic import ListView, DetailView
from django.views.generic.edit import UpdateView, DeleteView, CreateView
from django.urls import reverse_lazy

from .models import Article

...

class ArticleCreateView(LoginRequiredMixin, CreateView): # new
    ...
```

Now return to the homepage briefly at `http://127.0.0.1:8000/` so we avoid resubmitting the form. Then go to our new message URL directly again at `http://127.0.0.1:8000/articles/new/`. You'll see the following "Page not found" error:

Error page

What's happening? Django has automatically redirected users to the log in page, just as we desired!

LoginRequiredMixin

Now we see that restricting view access requires 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 .models import Article


class ArticleListView(LoginRequiredMixin, ListView): # new
    model = Article
    template_name = 'article_list.html'


class ArticleDetailView(LoginRequiredMixin, DetailView): # new
    model = Article
    template_name = 'article_detail.html'


class ArticleUpdateView(LoginRequiredMixin, UpdateView): # new
    model = Article
    fields = ('title', 'body',)
    template_name = 'article_edit.html'


class ArticleDeleteView(LoginRequiredMixin, DeleteView): # new
    model = Article
    template_name = 'article_delete.html'
    success_url = reverse_lazy('article_list')


class ArticleCreateView(LoginRequiredMixin, CreateView):
    model = 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)
```

Go ahead and play around with the site to confirm that the log in 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.

UpdateView and DeleteView

We're making progress but there's still the issue of our edit and delete views. Any *logged in* user can make changes to any article. What we want is to restrict this access so that only the author of an article has this permission.

We could add permissions logic to each view for this but a more elegant solution is to create a dedicated mixin, a class with a particular feature that we want to reuse in our Django code. And better yet, Django ships with a built-in mixin, UserPassesTestMixin, just for this purpose!

To use `UserPassesTestMixin`, first import it at the top of the `articles/views.py` file and then add it to both the update and delete views where we want this restriction.

The `test_func` method is used by `UserPassesTestMixin` for our logic. We need to override it. In this case we set the variable `obj` to the current object returned by the view using `get_object()`. Then we say, if the `author` on the current object matches the current user on the webpage (whoever is logged in and trying to make the change), then allow it. If false, an error will automatically be thrown.

The code looks like this:

Code

```
# articles/views.py
from django.contrib.auth.mixins import (
    LoginRequiredMixin,
    UserPassesTestMixin # new
)
from django.views.generic import ListView, DetailView
from django.views.generic.edit import UpdateView, DeleteView, CreateView
from django.urls import reverse_lazy
from .models import Article

...

class ArticleUpdateView(
  LoginRequiredMixin, UserPassesTestMixin, UpdateView
  ): # new
    model = Article
    fields = ('title', 'body',)
    template_name = 'article_edit.html'

    def test_func(self): # new
        obj = self.get_object()
        return obj.author == self.request.user


class ArticleDeleteView(
  LoginRequiredMixin, UserPassesTestMixin, DeleteView
  ): # new
    model = Article
    template_name = 'article_delete.html'
    success_url = reverse_lazy('article_list')

    def test_func(self): # new
        obj = self.get_object()
        return obj.author == self.request.user
```

Now log out of your superuser account and log in with `testuser`. If the code works, then you should not be able to edit or delete any posts written by your superuser, which is all of them right now. Instead you will see a Permission Denied 403 error page.

403 error page

Conclusion

Our *Newspaper* app is almost done. There are further steps we could take at this point, such as only displaying edit and delete links to the appropriate users, which would involve custom template tags but overall the app is in good shape. We have our articles properly configured, set permissions and authorizations, and user authentication is in order. The last item needed is 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. We will take the simpler approach since it's always easy to add more complexity later. 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): # new
    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 the `makemigrations` 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
```

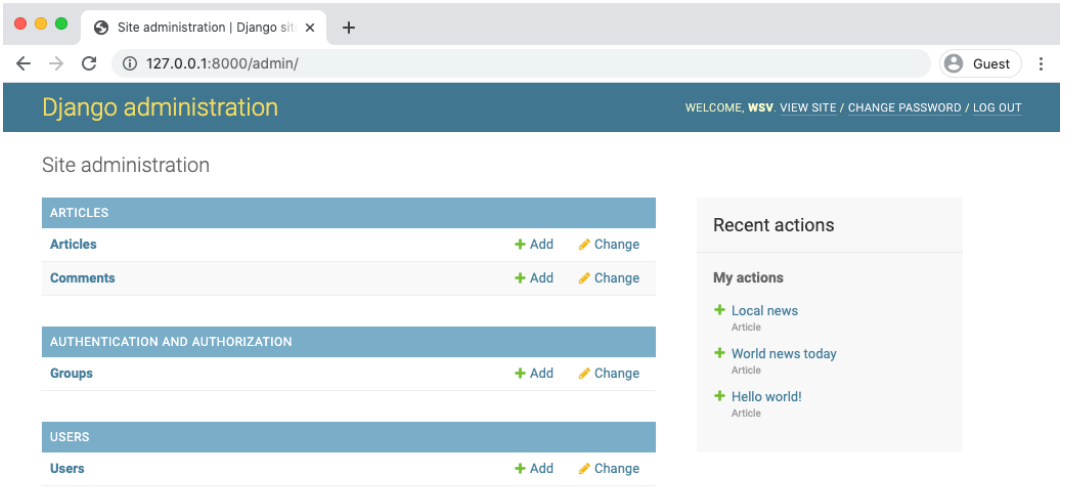
Admin

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

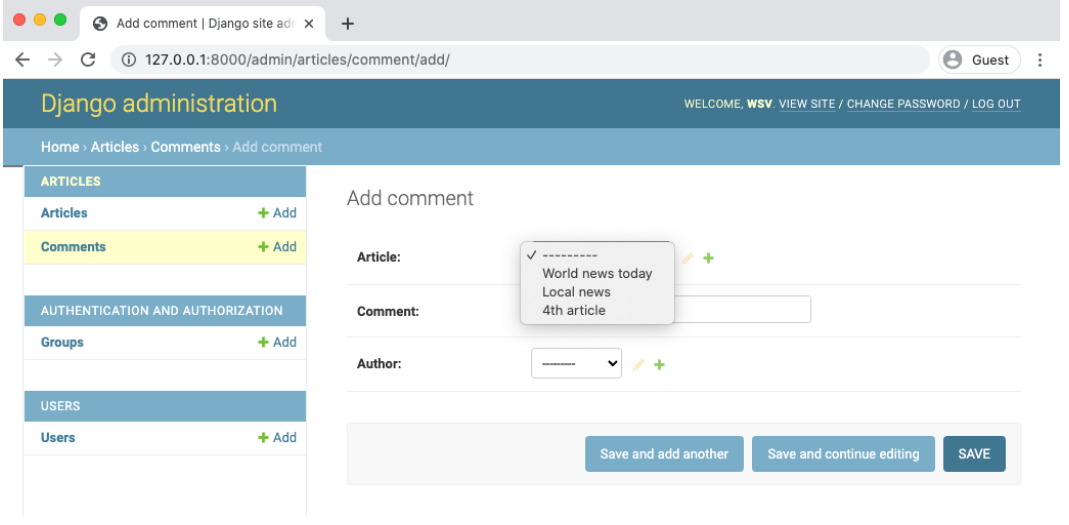
```
# articles/admin.py
from django.contrib import admin
from .models import Article, Comment # new

admin.site.register(Article)
admin.site.register(Comment) # new
```

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

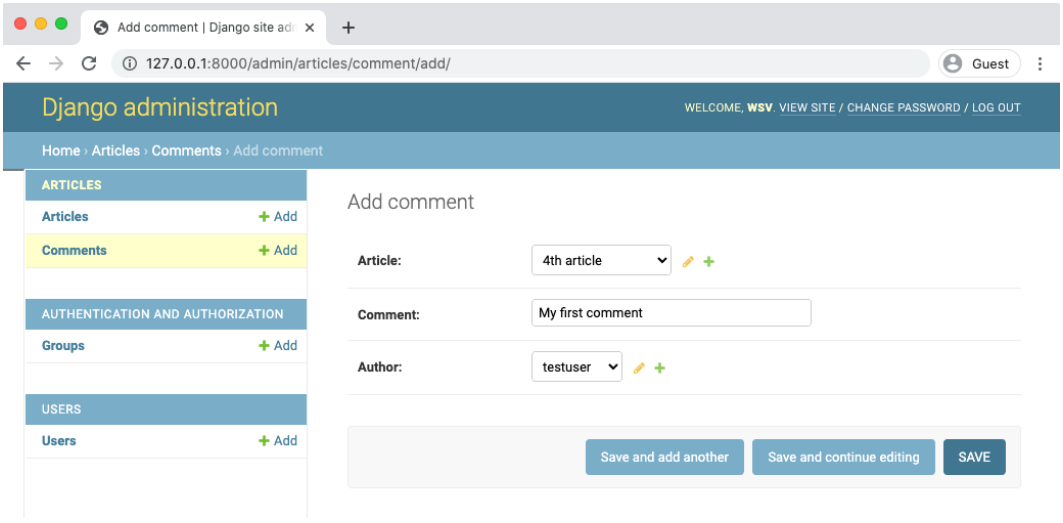
Under our app "Articles" you'll see our two tables: Comments and Articles. Click on the "+ Add" next to Comments. You'll see that under Article is a dropdown of existing articles, same thing for Author, and there is a text field next to Comment.



Admin Comments

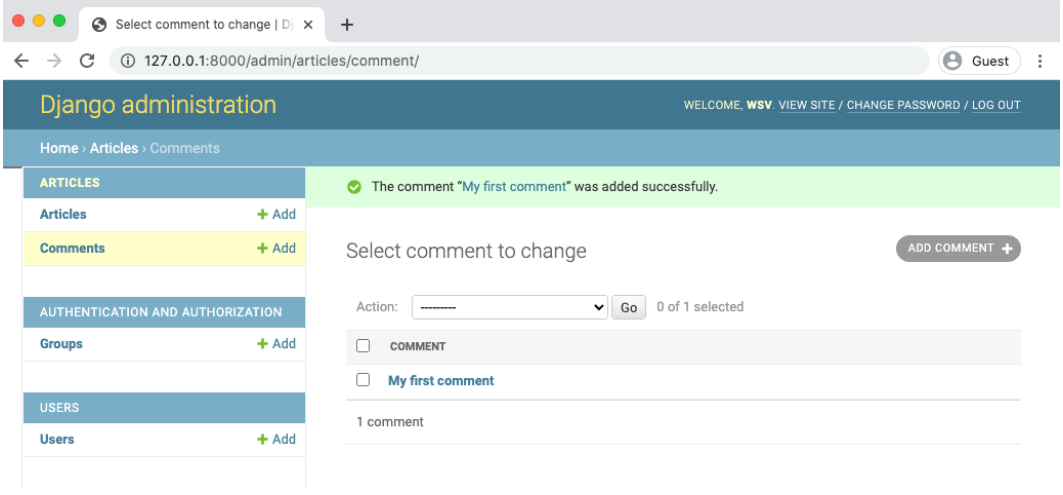
Select an Article, write a comment, and then select an author that is not your superuser, perhaps

`testuser` as I've done in the picture. Then click on the "Save" button.



Admin testuser comment

You should next see your comment on the "Comments" page.



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.

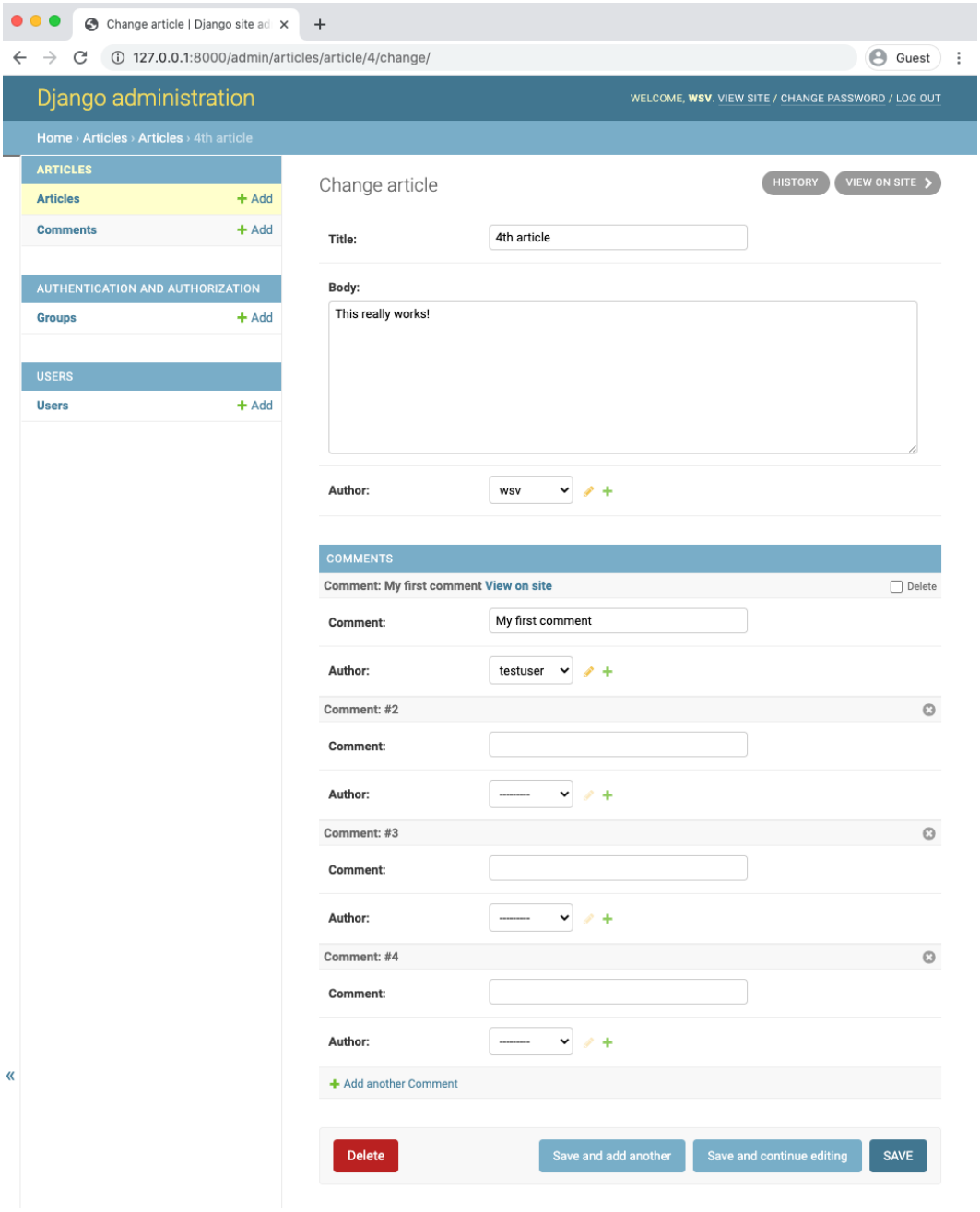
There are two main inline views used: TabularInline and StackedInline. The only difference between the two is the template for displaying information. In a TabularInline all model fields appear on one line while in a StackedInline each field has its own line. We'll implement both so you can decide which one you prefer.

Update `articles/admin.py` as follows in your text editor.

Code

```
# articles/admin.py
from django.contrib import admin
from .models import Article, Comment


class CommentInline(admin.StackedInline): # new
    model = Comment


class ArticleAdmin(admin.ModelAdmin): # new
    inlines = [
        CommentInline,
    ]


admin.site.register(Article, ArticleAdmin) # new
admin.site.register(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 "4th article" in my case.



Admin change page

Better, right? We can see and modify all our related articles and comments in one place. Note that by default, the Django admin will display 3 empty rows here. You can change the default number that appear with the `extra` field. So if you wanted no fields by default, the code would look like this:

Code

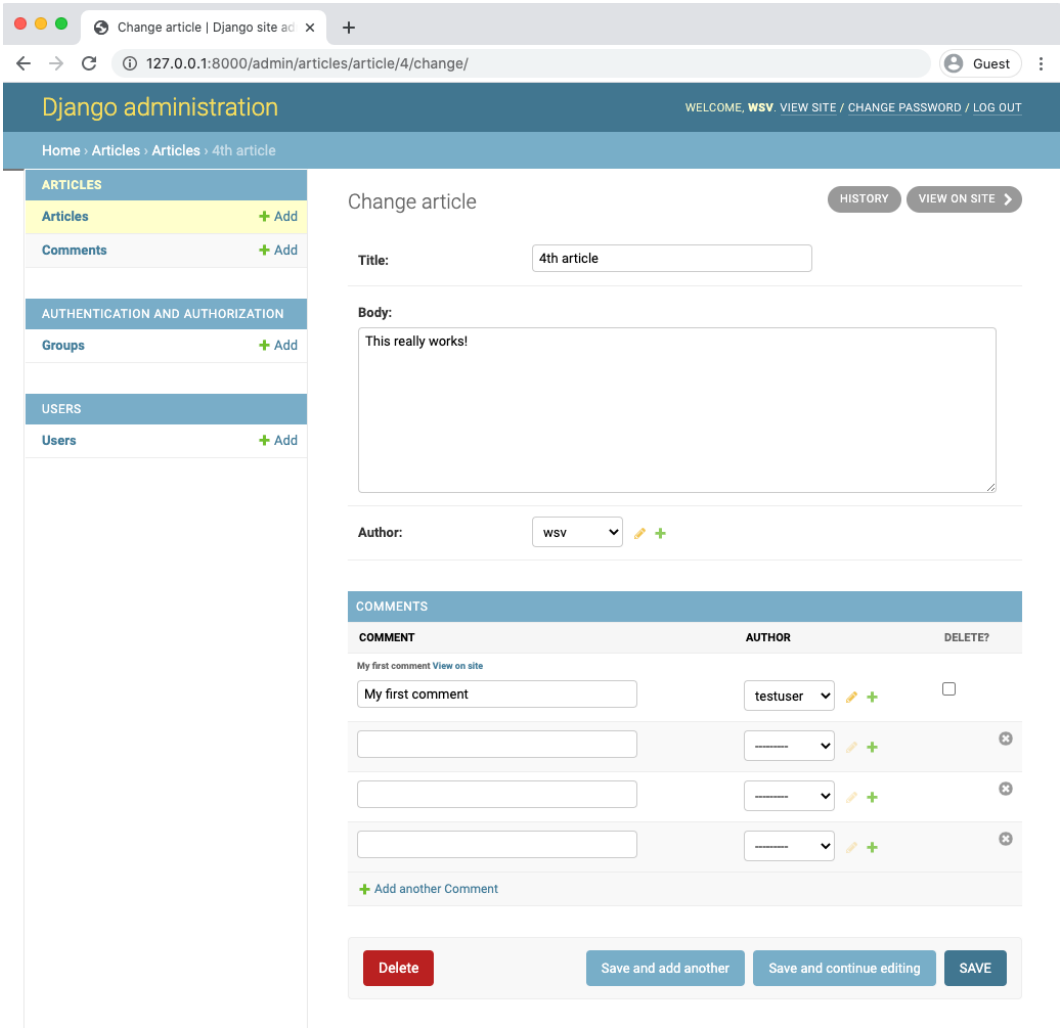
```
# articles/admin.py
...
class CommentInline(admin.StackedInline):
    model = Comment
    extra = 0 # new
```

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 .models import Article, Comment


class CommentInline(admin.TabularInline): # new
    model = Comment


class ArticleAdmin(admin.ModelAdmin):
    inlines = [
        CommentInline,
    ]


admin.site.register(Article, ArticleAdmin)
admin.site.register(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 for following relationships "backward" 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
    )
    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')
```

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
Migrations for 'articles':
  articles/migrations/0003_auto_20200726_1405.py
    - Alter field article on comment
(news) $ python manage.py migrate
Operations to perform:
  Apply all migrations: admin, articles, auth, contenttypes, sessions, users
Running migrations:
  Applying articles.0003_auto_20200726_1405... OK
(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.

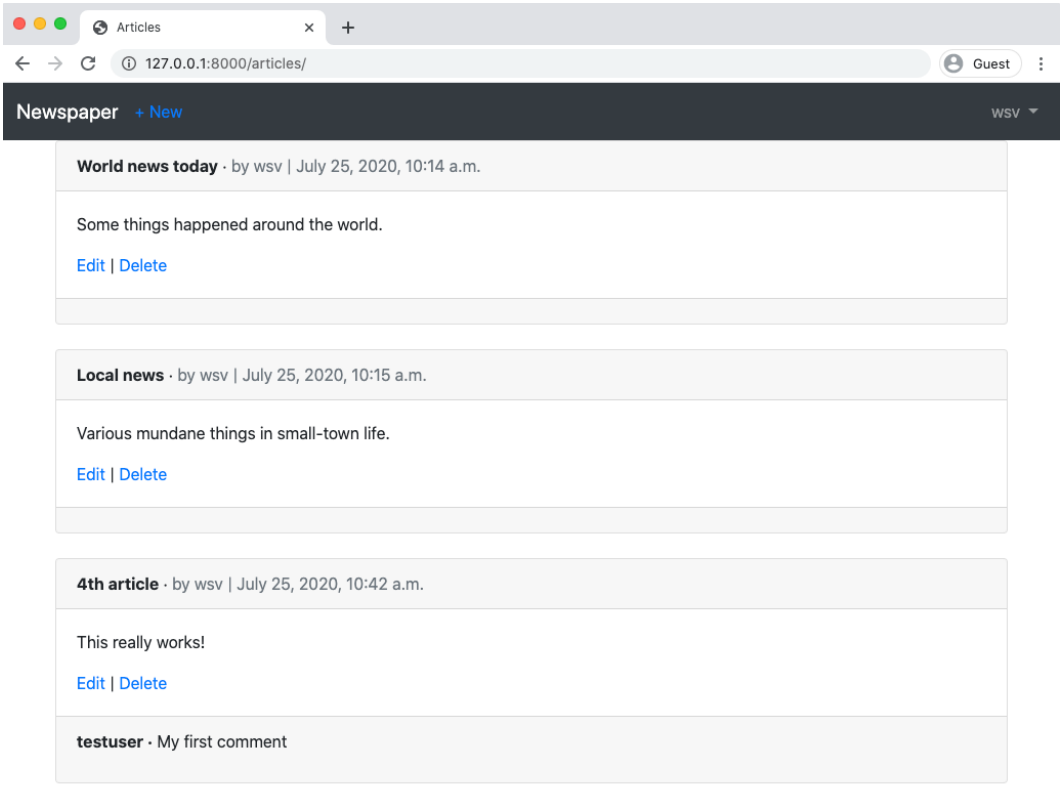
In our `article_list.html` file we can add our comments to the `card-footer`. Note that I've moved our edit and delete links up into `card-body`. To access each comment we're calling `article.comments.all` which means first look at the `article` model, then `comments` which is the related name of the 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 title %}

{% 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">
        <!-- Changes start here! -->
        <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 }} &middot;
            </span>
            {{ comment }}
          </p>
        {% endfor %}
      </div>
      <!-- Changes end here! -->
    </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

Yoohoo! It works. We can see comments listed underneath the initial message.

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.

Conclusion

Our *Newspaper* app is now complete. It has a robust user authentication flow that uses a custom user model and email. Improved styling thanks to Bootstrap. And both articles and comments. We even tipped our toes into permissions and authorizations.

Our remaining task is to deploy it online. Our deployment process has grown in complexity with

each successive application, but we're still taking shortcuts around security and performance. In the next chapter, we'll see how to properly deploy a Django site by using environment variables, PostgreSQL, and additional settings.

Chapter 16: Deployment

Fundamentally, there is a tension between the ease-of-use desired in a local Django development environment and the security and performance necessary in a production environment. Django is designed to make web developers' lives easier and it therefore defaults to a local configuration when the `startproject` command is first run.

But as we've already seen in our projects, deployment requires additional packages and configurations. Here is the complete deployment checklist last used for the *Blog* app back in Chapter 7:

- configure static files, install `whitenoise`, and run `collectstatic`
- update `ALLOWED_HOSTS`
- install `Gunicorn` as the production web server
- create a `Procfile`
- create a new Heroku project and push the code to Heroku
- run `heroku ps:scale web=1` to start a dyno web process

In truth, though, this list is far from complete. We've still been making a number of shortcuts in our deployments that need to be fixed. As the Django deployment checklist notes, at the very minimum, a production environment should also have:

- `DEBUG` set to `False`
- `SECRET_KEY` actually kept secret
- a production database, not SQLite

The question is: how do we balance both of these needs? One environment for local development and another for production? Today, the best practice is to use environment variables, which can be loaded into the codebase at runtime yet not stored in the source code. In other words, even if someone had access to your Github repo and all the source code, they couldn't do much damage

because the environment variables representing the most important details would be stored elsewhere!

In this final chapter we will switch over to environment variables and create a deployment checklist suitable for a professional website.

Environment Variables

There are multiple ways to work with environment variables in Python but for this project we'll use the environs package. It allows us to create a dedicated `.env` file for environment variables as well as load a number of Django-specific additional packages which help with configuration.

On the command line, install `environs[django]`. Note that you'll probably need to add single quotes `''` around the package if you're using Zsh as your terminal shell, so run `pipenv install 'environs[django]==8.0.0'`.

Command Line

```
(news) $ pipenv install 'environs[django]==8.0.0'
```

Then, in the `config/settings.py` file, there are three lines of imports to add at the top of the file.

Code

```
# config/settings.py
from environs import Env # new

env = Env() # new
env.read_env() # new
```

Next up, create a new file called `.env` in the same folder that contains `.manage.py`.

Command Line

```
(news) $ touch .env
```

Any file that starts with a dot, such as `.env`, is treated as a hidden file, meaning it won't be displayed by default during a directory listing. To prove this, try typing the `ls` command. You'll see all the files and directories listed, but not the "hidden" `.env` file.

.gitignore

We're using Git for source control, but we don't want it to track *every* file in our project. For example, the reason we create a dedicated `.env` file is so that it will be secret, not stored in our source code.

Fortunately, there is a way to identify which files and folders we want Git to ignore. Create a new file called `.gitignore` file in the same folder that contains `.env` and `manage.py`.

Command Line

```
(news) $ touch .gitignore
```

While we're at it, there are several other files and folders that are convenient to ignore. This includes the `__pycache__` directory, containing `.pyc` files created automatically whenever Django runs `.py` files. The `db.sqlite3` file, since it's a bad practice to track our SQLite database as it might accidentally be pushed to production. And as a final step, if you're on a Mac, there's no need to track `.DS_Store`, which stores information about folder settings on MacOS.

Here is what the final `.gitignore` file should contain:

.gitignore

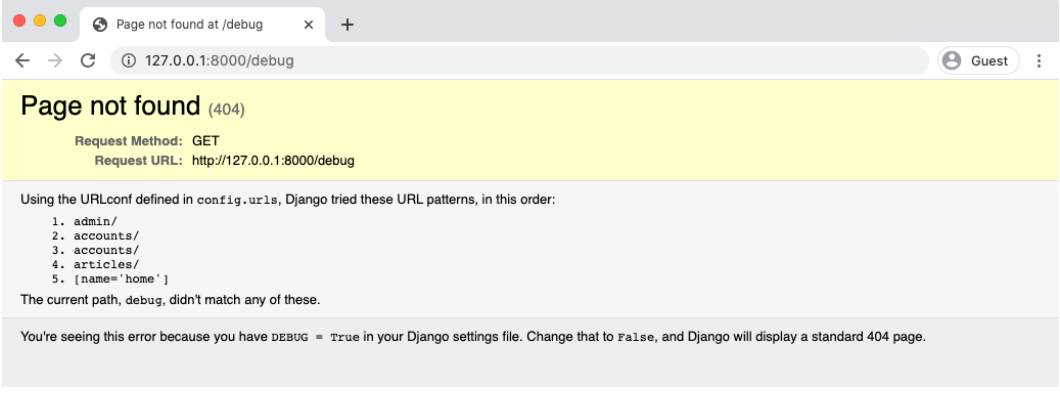
```
.env
__pycache__/
db.sqlite3
.DS_Store # Mac only
```

If you're curious, Github maintains an official Python gitignore file containing additional configurations.

If you run `git status` now you'll see that `.env` is no longer tracked. That's the behavior we want!

DEBUG & ALLOWED HOSTS

It's time to configure our environment variable for `DEBUG`, which by default is set to `True`. This is helpful for local development, but a major security issue if deployed into production. For example, if you start up the local server with `python manage.py runserver` and navigate to a page that does not exist, like `http://127.0.0.1:8000/debug`, you'll see the following:



Page Not Found

This page lists all the URLconfs tried and apps loaded, which is a treasure map for any hacker attempting to break into your site. You'll even see that on the bottom of the error page, it says that Django will display a standard 404 page if `DEBUG=False`. Within the `config/settings.py` file, change `DEBUG` to be `False`.

Code

```
# config/settings.py
DEBUG = False
```

Oops! If you look at the command line, Django is complaining about a `CommandError` and has automatically stopped the local server.

Command Line

```
CommandError: You must set settings.ALLOWED_HOSTS if DEBUG is False.
```

Because `DEBUG` is set to `False`, Django assumes we're trying to push the project into production and comes with a number of warnings like this. `ALLOWED_HOSTS` should be set to accept both local ports (`localhost` and `127.0.0.1`) as well as `.herokuapp.com` for its Heroku deployment. We can add all three routes to our config.

Code

```
# config/settings.py
ALLOWED_HOSTS = ['.herokuapp.com', 'localhost', '127.0.0.1']
```

Re-run the `python manage.py runserver` command and refresh the page. The descriptive error message is no longer there! Instead, it has been replaced by a generic "Not Found" message.

Not Found

Our goal is for `DEBUG` to be `True` for local development, but set to `False` in production. The two-step process for adding any environment variable is to first add the it to `.env` and then to `config/settings.py`.

Within the `.env` file, add the line `export DEBUG=True`. This syntax of exporting both the variable name and its value will be used for all environment variables.

.env

```
export DEBUG=True
```

Then in `config/settings.py`, change the `DEBUG` setting to read the variable `"DEBUG"` from the `.env` file.

Code

```
# config/settings.py
DEBUG = env.bool("DEBUG")
```

It's easy to be confused here. Our environment variable is named `DEBUG`, the same as the setting it replaces. But we could have named our environment variable `ANYTHING` instead. That would have looked like this:

.env

```
export ANYTHING=True
```

Code

```
# config/settings.py
DEBUG = env.bool("ANYTHING")
```

`ANYTHING` is a variable so it can have almost any name we desire. In practice, however, most developers will name the environment variable to match the name of the setting it replaces. We will do the same so `export DEBUG=True`. {/aside}

One more best practice we will adopt is to set a `default` value, in this case `False`, meaning that if an environment variable can't be found, our production setting will be used. It's a best practice to default to production settings since they are more secure and if something goes wrong in our code, we won't default to exposing all our secrets out in the open.

The final `config/settings.py` line therefore looks as follows:

Code

```
# config/settings.py
DEBUG = env.bool("DEBUG", default=False)
```

If you refresh the webpage at `http://127.0.0.1:8000/debug`, you'll see the full error page is back again. Everything is working properly.

SECRET_KEY

The next environment variable to set is our `SECRET_KEY`, a random 50 character string generated each time `startproject` is run. Your value will differ from mine below, but not that the enclosing single quotes that make it a string `''` should **not be** included. In other words, here is the `config/settings.py` value of the `SECRET_KEY`.

Code

```
# config/settings.py
SECRET_KEY = 'yq@=cjw8z@ssx7_3ukfoi9j3di)m-km8=^x9a))p2)3y-24g%*'
```

And here it is, without single quotes, in the `.env` file.

.env

```
export DEBUG=True
export SECRET_KEY=yq@=cjw8z@ssx7_3ukfoi9j3di)m-km8=^x9a))p2)3y-24g%*
```

Since we deliberately haven't made any Git commits yet, this value is "safe" and not stored in source control. If you have made previous Git commits, you'll need to generate a new `SECRET_KEY` for security reasons. One way to do so is by invoking Python's built-in secrets module by running `python -c 'import secrets; print(secrets.token_urlsafe())'` on the command line. {/aside}

Update `config/settings.py` so that `SECRET_KEY` points to this new environment variable. We won't set a default value.

Code

```
# config/settings.py
SECRET_KEY = env.str("SECRET_KEY")
```

Now restart the local server with `python manage.py runserver` and refresh your website. It will work with `SECRET_KEY` loaded from our `.env` file.

DATABASES

Our current `DATABASES` configuration is for SQLite, but we want to be able to switch to PostgreSQL for production on Heroku. When we installed `environs[django]` earlier, the Django "goodies" included the elegant dj-database-url package, which takes all the database configurations needed for our database, SQLite or PostgreSQL, and creates a `DATABASE_URL` environment variable.

Our updated `DATABASES` configuration uses `dj_db_url` from `environs[django]` to help parse `DATABASE_URL` and looks as follows:

Code

```
# config/settings.py
DATABASES = {
    "default": env.dj_db_url("DATABASE_URL")
}
```

That's it! All we need to do now is specify SQL as the local `DATABASE_URL` value in the `.env` file.

.env

```
export DEBUG=True
export SECRET_KEY=yq@=cjw8z@ssx7_3ukfoi9j3di)m-km8=^x9a))p2)3y-24g%*
export DATABASE_URL=sqlite:///db.sqlite3
```

I hope you're wondering now: how do we set `DATABASE_URL` in production on Heroku? It turns out that when Heroku provisions a new PostgreSQL database, it automatically creates a configuration variable for it named ... `DATABASE_URL`. Since the `.env` file is not committed to

production, our Django project on Heroku will instead use this PostgreSQL configuration. Pretty elegant, no?

The last step is to install Psycopg, a database adapter that lets Python apps talk to PostgreSQL databases. Heroku needs it in deployment so we can just install it now.

Command Line

```
(news) $ pipenv install psycopg2-binary==2.8.5
```

We can use this approach because Django's ORM (Object Relational Mapper) translates our `models.py` code from Python into the database backend of choice. This works *almost* all the time without error. However, it is possible for weird bugs to creep up and it is recommended to install PostgreSQL locally, too, on professional projects however doing so is beyond the scope of this book.

Static Files

Surprisingly, we actually don't have any static files in our *Newspaper* app at this point. We've relied entirely on hosted Bootstrap rather than own CSS, JavaScript, or images as we did in the *Blog* app. That is likely to change as the site grows in the future so we may as well set up static files properly now.

Stop the local server with `Control+c` and create a new `static` folder in the same directory as `manage.py`. Then add folders for `css`, `javascript`, and `images`.

Command Line

```
(news) $ mkdir static
(news) $ mkdir static/css
(news) $ mkdir static/js
(news) $ mkdir static/images
```

We'll also need to install the WhiteNoise package since Django does not support serving static files in production itself.

Command Line

```
(news) $ pipenv install whitenoise==5.1.0
```

WhiteNoise must be added to `config/settings.py` in the following locations:

- `whitenoise` above `django.contrib.staticfiles` in `INSTALLED_APPS`
- `WhiteNoiseMiddleware` above `CommonMiddleware`
- `STATICFILES_STORAGE` configuration pointing to WhiteNoise

Code

```
# config/settings.py
INSTALLED_APPS = [
    ...
    'whitenoise.runserver_nostatic', # new
    'django.contrib.staticfiles',
]

MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'whitenoise.middleware.WhiteNoiseMiddleware', # new
    ...
]

STATIC_URL = '/static/'
STATICFILES_DIRS = [str(BASE_DIR.joinpath('static'))] # new
STATIC_ROOT = str(BASE_DIR.joinpath('staticfiles')) # new
STATICFILES_STORAGE =
    'whitenoise.storage.CompressedManifestStaticFilesStorage' # new
```

Run the `collectstatic` command for the first time to compile all the static file directories and files into one self-contained unit suitable for deployment.

Command Line

```
(news) $ python manage.py collectstatic
```

As a final step, in order for our templates to display any static files, they must be loaded in so add `{% load static %}` to the top of the `base.html` file.

Code

```
<!-- templates/base.html -->
{% load static %}
<html>
...
```

Deployment Checklist

It's easy to lose track of all the steps required for readying a Django website for production. There's one more we must do: install `Gunicorn` as the production web server.

Command Line

```
(news) $ pipenv install gunicorn==19.9.0
```

And then tell Heroku to use it via a new `Procfile` file created in the same folder as `manage.py`.

Command Line

```
(news) $ touch Procfile
```

Add the following single line to `Procfile`.

Procfile

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

Here is a recap of what we've done so far:

- add environment variables via `environs[django]`

- set `DEBUG` to `False`
- set `ALLOWED_HOSTS`
- use environment variable for `SECRET_KEY`
- update `DATABASES` to use SQLite locally and PostgreSQL in production
- configure static files
- install `whitenoise` for static file hosting
- install `gunicorn` for production web server

These steps apply to Django deployments on any server or platform; they are not Heroku-specific.

Git & GitHub

Amazingly we haven't used Git in our project yet. In general, initializing Git and regularly committing work should be a regular part of your work flow even though we haven't done that so far in this project, in part to avoid the extra step of updating the `SECRET_KEY` value.

Initialize a new Git repository and commit the code changes we've made.

Command Line

```
(news) $ git status
(news) $ git add -A
(news) $ git commit -m "initial commit"
```

As a best practice, create a repository on GitHub to store the code as well. Create a new GitHub repo called `news-app`. Make sure to select the "Private" radio button and then click on the "Create repository" button. On the next page, scroll down to where it says "â€¦or push an existing repository from the command line." Copy and paste the two commands there into your terminal.

It should look like the below albeit instead of `wsvincent` as the username it will be your GitHub username.

Command Line

```
(news) $ git remote add origin https://github.com/wsvincent/news-app.git
(news) $ git push -u origin master
```

All set! Now we can configure Heroku and finally see our Newspaper project live.

Heroku Deployment

Make sure that you are already logged into your Heroku account via the command line.

Command Line

```
(news) $ heroku login
```

The command `heroku create` makes a new container for our app to live in and by default, Heroku will assign a random name. You can specify a custom name, as we are doing here, but it must be *unique on Heroku*. Mine is called `dfb-news` so that name is already taken; you need another combination of letters and numbers!

Command Line

```
(news) $ heroku create dfb-news
```

Now configure Git so that when you push to Heroku, it goes to your new app name (replacing `dfb-news` with your custom name).

Command Line

```
(news) $ heroku git:remote -a dfb-news
```

So far so good. A new step at this point is creating a PostgreSQL database on Heroku itself, which we haven't done before. Heroku has its own hosted PostgreSQL databases we can use which come in multiple tiers. For a learning project like this, the free `hobby-dev` tier is more than adequate. Run the following command to create this new database:

Command Line

```
(news) $ heroku addons:create heroku-postgresql:hobby-dev
Creating heroku-postgresql:hobby-dev on â¬¢ dfb-news... free
Database has been created and is available
 ! This database is empty. If upgrading, you can transfer
 ! data from another database with pg:copy
Created postgresql-symmetrical-16853 as DATABASE_URL
Use heroku addons:docs heroku-postgresql to view documentation
```

Did you see that Heroku has created a custom `DATABASE_URL` to access the database? For mine here, it is `postgresql-symmetrical-16853`. This is automatically available as a configuration variable within Heroku once we deploy. That's why we don't need to set an environment variable for `DATABASE_URL` in production. We also don't need to set `DEBUG` to `False` because that is the default value in our `config/settings.py` file. The only environment variable to manually add to Heroku is `SECRET_KEY`, so copy its value from your `.env` file and run the `config:set` command, placing the value of the `SECRET_KEY` itself within single quotes `''`.

Command Line

```
(news) $ heroku config:set SECRET_KEY='yq@=cjw8z@ssx7_3ukfoi9j3di)m-km8=^x9a))p2)3y-24g%*'
```

Now it's time to push our code up to Heroku itself and start a web process so our Heroku dyno is running.

Command Line

```
(news) $ git push heroku master
(news) $ 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.

But if you go to this URL now though you'll see a 500 Server Error message! That's because the PostgreSQL database exists but has not been setup yet! Previously we used SQLite in production, which is file-based, and was already configured locally and then pushed up to Heroku. But this PostgreSQL database of ours is brand new! Heroku has all our code but we haven't configured this production database yet.

The same process used locally of running `migrate`, creating a `superuser` account, and entering blog posts in the admin must be followed again. To run a command with Heroku, as opposed to locally, prefix it with `heroku run`.

Command Line

```
(news) $ heroku run python manage.py migrate
(news) $ heroku run python manage.py createsuperuser
```

You will need to log into the live admin site to add newspaper entries and comments since, again, this is a brand-new database and not related to our local SQLite one.

For larger sites, there are techniques such as using fixtures to load data into a local database but doing so is beyond our scope here.

Refresh your live website and it should work correctly. Note that since the production server will run constantly in the background, you do *not* need to use the `runserver` command on Heroku.

Conclusion

Phew! We just covered *a ton* of material so it's likely you feel overwhelmed right now. That's normal. There are many steps involved to configure a website for proper deployment. The good news is that this same list of production settings will hold true for almost every Django project. Don't worry about memorizing all the steps.

After you've built and deployed several Django websites, these steps will soon feel very familiar. And in fact, we've only scratched the surface of additional security measures that can be configured. Django comes with its own deployment checklist that can be run via the command line to highlight additional security issues.

Command Line

```
(news) $ heroku run python manage.py check --deploy
```

The other big stumbling block for newcomers is becoming comfortable with the difference between local and production environments. It's very likely you will forget at some point to push code changes into production and spend minutes or hours wondering why the change isn't live

on your site. Or even worse, you'll make changes to your local SQLite database and expect them to magically appear in the production PostgreSQL database. It's part of the learning process. But Django really does make it much smoother than it otherwise would be. And now you know enough to confidently deploy any Django project online.

Conclusion

Congratulations on finishing *Django for Beginners*! After starting from absolute zero we've now built five different web applications from scratch and 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 (Create-Read-Update-Delete) 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? A Twitter or Facebook clone? You already have all the tools you need. When you're starting out I believe the best approach is to build as many small projects as possible and incrementally add complexity and research new things.

Django For Professionals

Web development is a very deep field and there's always something new to learn. This is especially true for large websites that must handle thousands or millions of visitors at a time. Django itself is more than capable of this. If you'd like to learn more, I've written a follow-up book called Django for Professionals. It tackles many of the challenges around building truly *production-ready* websites such as using Docker, a production database locally like PostgreSQL, handling payments, advanced user registration, security, performance, and much more.

Django for APIs

In practice, most professional Django developers rarely build full-stack websites from scratch. Instead, they work on teams and focus on the back-end creating web APIs that can be consumed by mobile apps like iOS and Android or websites that use dedicated JavaScript front-end framework such as Vue, React, or Angular.

Thanks to the power of Django REST Framework, a third-party app that is tightly coupled with Django itself, it is possible to transform any existing Django website into an API with a minimal amount of code. If you'd like to learn more, I've written an entire book on the topic, Django for APIs.

3rd Party Packages

As we've seen in this book, 3rd party packages are a vital part of the Django ecosystem especially when it comes to deployment or improvements around user registration. It's not uncommon for a professional Django website to rely on literally dozens of such packages.

However, a word of caution is in order. Don't blindly install and use 3rd party packages just because it saves a small amount of time now. Every additional package introduces another dependency, another risk that its maintainer won't fix every bug or won't keep up to date with the latest version of Django. Take the time to understand what it is doing.

If you'd like to view more packages, the Django Packages website is a comprehensive resource of all available third party apps. Or, if you find that overwhelming, the awesome-django repo contains a curated list of the most popular packages, which are worth a closer look.

Learning Resources

As you become more comfortable with Django and web development in general, you'll find the official Django documentation and source code increasingly valuable. I refer to both on an almost daily basis. There is also the official Django forum, a great resource albeit underutilized resource for Django-specific questions.

To continue on your Django journey, a good source of additional tutorials and courses is the website LearnDjango.com, which I maintain. You might also look at the DjangoX and DRFX starter projects to speed up the development of new projects.

If you're interested in a weekly podcast on Django, I co-host Django Chat, which features interviews with leading developers and topic deep-dives. And I co-write a weekly newsletter, Django News, filled with news, articles, tutorials, and more all about Django.

Python Books

Django is, ultimately, just Python so if your Python skills could use improvement there are two books in particular I recommend. For beginners and those new to Python, it doesn't get much better than Eric Matthes's Python Crash Course. For intermediate to advanced developers, Fluent Python, Effective Python, and Python Tricks are worthy of additional study.

Feedback

As a final note, I'd love to hear your thoughts about the book. It is a constant work-in-progress and the detailed feedback from readers helps me continue to improve it. I respond to every email and can be reached at will@learndjango.com.

If you purchased this book on Amazon, please consider leaving an honest review. These reviews make an enormous impact on book sales.

Thank you for reading the book and good luck on your journey with Django!