

By **Leif Azzopardi** and **David Maxwell**

Python web
development
with Django

Tango with Django 2

- A beginner's guide to web development with **Django 2**.

Compatible with **Django 2.1 and 2.2**

Book Version 2020-01a



Tango With Django 2

A beginner's guide to web development with Django 2.

Leif Azzopardi and David Maxwell

This book is for sale at <http://leanpub.com/tangowithdjango2>

This version was published on 2020-01-12



This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2018 - 2020 Leif Azzopardi and David Maxwell



University
of Glasgow
Library

This copy of Tango with Django is licenced to:
[University of Glasgow Library](#).

Unauthorised access or downloading
of this material is not permitted.

2021-01-07

Tweet This Book!

Please help Leif Azzopardi and David Maxwell by spreading the word about this book on [Twitter](#)!

The suggested tweet for this book is:

I'm ready to @tangowithdjango too! Check out <https://www.tangowithdjango.com>

The suggested hashtag for this book is [#tangowithdjango](#).

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

[#tangowithdjango](#)

Also By These Authors

Books by [Leif Azzopardi](#)

[How to Tango with Django 1.9/1.10/1.11](#)

Books by [David Maxwell](#)

[How to Tango with Django 1.9/1.10/1.11](#)

Contents

1. Overview	1
1.1 Why Work with this Book?	1
1.2 What you will Learn	2
1.3 Technologies and Services	4
1.4 Rango: Initial Design and Specification	5
1.5 Summary	12
2. Getting Ready to Tango	14
2.1 Python 3	15
2.2 Virtual Environments	17
2.3 The Python Package Manager	18
2.4 Integrated Development Environment	20
2.5 Version Control	20
2.6 Testing your Implementation	22
3. Django Basics	25
3.1 Testing Your Setup	25
3.2 Creating Your Django Project	26
3.3 Creating a Django App	31
3.4 Creating a View	33
3.5 Mapping URLs	35
3.6 Basic Workflows	39
4. Templates and Media Files	42
4.1 Using Templates	42
4.2 Serving Static Media Files	50
4.3 Serving Media	58
4.4 Basic Workflow	61

CONTENTS

5. Models and Databases	65
5.1 Rango's Requirements	65
5.2 Telling Django about Your Database	66
5.3 Creating Models	67
5.4 Creating and Migrating the Database	70
5.5 Django Models and the Shell	73
5.6 Configuring the Admin Interface	75
5.7 Creating a Population Script	79
5.8 Workflow: Model Setup	85
6. Models, Templates and Views	91
6.1 Workflow: A Data-Driven Page	91
6.2 Showing Categories on Rango's Homepage	91
6.3 Creating a Details Page	95
7. Forms	112
7.1 Basic Workflow	112
7.2 Page and Category Forms	113
8. Working with Templates	131
8.1 Using Relative URLs in Templates	131
8.2 Dealing with Repetition	134
8.3 Template Inheritance	138
8.4 The <code>render()</code> Method and the <code>request</code> Context	142
8.5 Custom Template Tags	143
8.6 Summary	147
9. User Authentication	148
9.1 Setting up Authentication	148
9.2 Password Hashing	149
9.3 Password Validators	151
9.4 The <code>User</code> Model	151
9.5 Additional <code>User</code> Attributes	152
9.6 Creating a <i>User Registration</i> View and Template	154
9.7 Implementing Login Functionality	163
9.8 Restricting Access	169

CONTENTS

9.9	Logging Out	171
9.10	Tidying up the base.html Hyperlinks	173
9.11	Taking it Further	174
10.	Cookies and Sessions	176
10.1	Cookies, Cookies Everywhere!	176
10.2	Sessions and the Stateless Protocol	179
10.3	Setting up Sessions in Django	181
10.4	A Cookie Tasting Session	182
10.5	Client-Side Cookies: A Site Counter Example	183
10.6	Session Data	187
10.7	Browser-Length and Persistent Sessions	190
10.8	Clearing the Sessions Database	190
10.9	Basic Considerations and Workflow	191
11.	User Authentication with Django-Registration-Redux	194
11.1	Setting up Django Registration Redux	194
11.2	Functionality and URL mapping	196
11.3	Setting up the Templates	197
12.	Bootstrapping Rango	204
12.1	The Template	207
12.2	Quick Style Change	210
13.	Adding Search to Rango	224
13.1	The Bing Search API	224
13.2	Adding Search Functionality	228
13.3	Putting Search into Rango	233
14.	Making Rango Tango Exercises	239
14.1	Tracking Page Clickthroughs	240
14.2	Searching Within a Category Page	242
14.3	Create and View User Profiles	244
15.	Making Rango Tango Hints	246
15.1	Track Page Clickthroughs	246
15.2	Searching Within a Category Page	249

CONTENTS

15.3	Creating a UserProfile Instance	255
15.4	Class-Based Views	262
15.5	Viewing your Profile	268
15.6	Listing all Users	274
16.	JQuery Crash Course	280
16.1	Including JQuery	280
16.2	Testing your Setup	282
16.3	Further DOM Manipulation Examples	289
16.4	Debugging Hints	290
17.	AJAX in Django with JQuery	294
17.1	AJAX and Rango	294
17.2	Adding a “Like” Button	295
17.3	Adding Inline Category Suggestions	303
17.4	Further AJAX-ing	310
18.	Automated Testing	317
18.1	Running Tests	318
18.2	Examining Testing Coverage	324
19.	Deploying Your Project	328
19.1	Creating a PythonAnywhere Account	328
19.2	The PythonAnywhere Web Interface	329
19.3	Creating a Virtual Environment	330
19.4	Setting up your Web Application	335
19.5	Log Files	345
20.	Final Thoughts	347
20.1	Acknowledgements	348
Appendices		350
Setting up your System		351
Installing Python 3 and pip		351
Virtual Environments		359
Using pip		363

CONTENTS

Version Control System	365
A Crash Course in UNIX-based Commands	367
Using the Terminal	367
Core Commands	373
A Git Crash Course	375
Why Use Version Control?	375
How Git Works	376
Setting up Git	378
Basic Commands and Workflow	384
Recovering from Mistakes	393
A CSS Crash Course	397
Including Stylesheets	399
Basic CSS Selectors	400
Element Selectors	400
Fonts	402
Colours and Backgrounds	403
Containers, Block-Level and Inline Elements	408
Basic Positioning	410
The Box Model	420
Styling Lists	422
Styling Links	425
The Cascade	427
Additional Reading	428

1. Overview

This book aims to provide you with a practical guide to web development using *Django 2* and *Python 3*. The book is designed primarily for students, providing a walkthrough of the steps involved in getting a web application up and running with Django. However, anyone who's starting off with web development will find this book to be beneficial.

This book seeks to complement the [official Django Tutorials¹](#) and many of the other excellent tutorials available online. By putting everything together in one place, this book fills in many of the gaps in the official Django documentation by providing an example-based, design-driven approach to learning the Django framework. Furthermore, this book provides an introduction to many of the aspects required to master web application development (such as HTML, CSS and JavaScript).

1.1 Why Work with this Book?

This book will save you time. On many occasions we've seen clever students get stuck, spending hours trying to fight with Django and other aspects of web development. More often than not, the problem was usually because a key piece of information was not provided, or something was not made clear. While the occasional blip might set you back 10-15 minutes, sometimes they can take hours to resolve. We've tried to remove as many of these hurdles as possible. This will mean you can get on with developing your application instead of getting stuck.

This book will lower the learning curve. Web application frameworks can save you a lot of hassle and a lot of time. But that is only true if you know how to use them in the first place! Often the learning curve is steep. This book tries to get you going – and going fast – by explaining how all the pieces fit together and how to build your web app logically.

¹<https://docs.djangoproject.com/en/2.1/intro/tutorial01/>

This book will improve your workflow. Using web application frameworks requires you to pick up and run with particular design patterns – so you only have to fill in certain pieces in certain places. After working with many students, we heard lots of complaints about using web application frameworks – specifically about how they take control away from the software engineer (i.e. [inversion of control](#)²). To help you, we've created several *workflows* to focus your development process so that you can regain that sense of control and build your web application in a disciplined manner.

This book is not designed to be read. Whatever you do, *do not read this book!* It is a hands-on guide to building web applications in Django. Reading is not doing. To increase the value you gain from this experience, go through and develop the application. When you code up the application, *do not just cut and paste the code*. Type it in, think about what it does, then read the explanations we have provided. If you still do not understand, then check out the Django documentation, go to [Stack Overflow](#)³ or other helpful websites and fill in this gap in your knowledge. If you are stuck, get in touch with us, so that we can improve the book – we've already had contributions from [numerous other readers!](#)

1.2 What you will Learn

In this book, we will be taking an example-based approach to web application development. In the process, we will show you how to design a web application called *Rango* ([see the Design Brief below](#)), and take a step by step in setting up, developing and deploying the application. Along the way, we'll show you how to perform the following key tasks which are common to most software engineering and web-based projects.

- How to **configure your development environment** – including how to use the terminal, your virtual environment, the pip installer, and how to work with Git.
- How to **set up a Django project** and **create a basic Django application**.
- How to **configure the Django project** to serve static media and user-uploaded media files (such as profile images).

²https://en.wikipedia.org/wiki/Inversion_of_control

³<http://stackoverflow.com/questions/tagged/django>

- How to **work with Django's Model-View-Template design pattern**.
- How to **work with database models** and use the *object-relational mapping (ORM)*⁴ functionality provided by Django.
- How to **create forms** that can utilise your database models to create **dynamically-generated webpages**.
- How to use the **user authentication** services provided by Django.
- How to incorporate **external services** into your Django application.
- How to include **Cascading Styling Sheets (CSS)** and **JavaScript** within a web application to aid in styling and providing it with additional functionality.
- How to **apply CSS** to give your application a professional look and feel.
- How to work with **cookies and sessions** with Django.
- How to include more advanced functionality like **AJAX** into your application.
- How to **write class-based views** with Django.
- How to **Deploy your application** to a web server using *PythonAnywhere*.

At the end of each chapter, we have also included several exercises designed to push you to apply what you have learnt during the chapter. To push you harder, we've also included several open development challenges, which require you to use many of the lessons from the previous chapters – but don't worry, as we've also included solutions and explanations on these, too!



Exercises

In each chapter, we have added several exercises to test your knowledge and skill. Such exercises are denoted like this.

You will need to complete all of these exercises as subsequent chapters will assume that you have fully completed them.



Hints and Tips

For each set of exercises, we will provide a series of hints and tips that will assist you if you need a push. If you get stuck however, you can always check out our solutions to all the exercises on our *GitHub* repository⁵.

⁴https://en.wikipedia.org/wiki/Object-relational_mapping

⁵https://github.com/maxwelld90/tango_with_django_2_code

1.3 Technologies and Services

Through the course of this book, we will use various technologies and external services including:

- the [Python⁶](#) programming language;
- the [Pip package manager⁷](#);
- [Django⁸](#);
- [unit testing⁹](#);
- the [Git¹⁰](#) version control system;
- [GitHub¹¹](#);
- [HTML¹²](#);
- [CSS¹³](#);
- the [JavaScript¹⁴](#) programming language;
- the [JQuery¹⁵](#) library;
- the [Twitter Bootstrap¹⁶](#) framework;
- the [Bing Search API¹⁷](#); and
- the [PythonAnywhere¹⁸](#) hosting service.

We've selected all of these technologies and services as they are either fundamental to web development, and/or enable us to provide examples on how to integrate your web application with CSS toolkits like *Twitter Bootstrap*, external services like those provided by the *Microsoft Bing Search API* and deploy your application quickly and easily with *PythonAnywhere*. Let's get started!

⁶<https://www.python.org>

⁷<https://pip.pypa.io/en/stable/>

⁸<https://www.djangoproject.com>

⁹https://en.wikipedia.org/wiki/Unit_testing

¹⁰<https://git-scm.com>

¹¹<https://github.com>

¹²<https://www.w3.org/html/>

¹³<https://www.w3.org/Style/CSS/>

¹⁴<https://www.javascript.com/>

¹⁵<https://jquery.com>

¹⁶<https://getbootstrap.com/>

¹⁷<https://docs.microsoft.com/en-gb/rest/api/cognitiveservices/bing-web-api-v7-reference>

¹⁸<https://www.pythonanywhere.com>

1.4 Rango: Initial Design and Specification

The focus of this book will be to develop an application called *Rango*. As we develop this application, it will cover the core components that need to be developed when building any web application. To see a fully-functional version of the application, you can visit our [How to Tango with Django website](#)¹⁹.

Design Brief

Let's imagine that we would like to create a website called *Rango* that lets users browse through user-defined categories to access various web pages. In [Spanish, the word rango](#)²⁰ is used to mean “*a league ranked by quality*” or “*a position in a social hierarchy*” – so we can imagine that at some point, we will want to rank the web pages in Rango.

- For the **main page** of the Rango website, your client would like visitors to be able to see:
 - the *five most viewed pages*;
 - the *five most viewed (or rango'ed) categories*; and
 - *some way for visitors to browse and/or search through categories*.
- When a user views a **category page**, your client would like Rango to display:
 - the *category name, the number of visits, the number of likes*, along with the list of associated pages in that category (showing the page's title, and linking to its URL); and
 - *some search functionality (via the search API)* to find other pages that can be linked to this category.
- For a **particular category**, the client would like: the *name of the category to be recorded*; the *number of times each category page has been visited*; and how many users have *clicked a “like” button* (i.e. the page gets rango'ed, and voted up the social hierarchy).
- *Each category should be accessible via a readable URL* – for example, `/rango/books-about-django/`.

¹⁹<http://www.tangowithdjango.com/>

²⁰<https://www.vocabulary.com/dictionary/es/rango>

- Only *registered users will be able to search and add pages to categories*. Therefore, visitors to the site should be able to register for an account.

At first glance, the specified application to develop seems reasonably straightforward. In essence, it is just a list of categories that link to pages. However, there are several complexities and challenges that need to be addressed. First, let's try and build up a better picture of what needs to be developed by laying down some high-level designs.



Exercises

Before going any further, think about these specifications and draw up the following design artefacts.

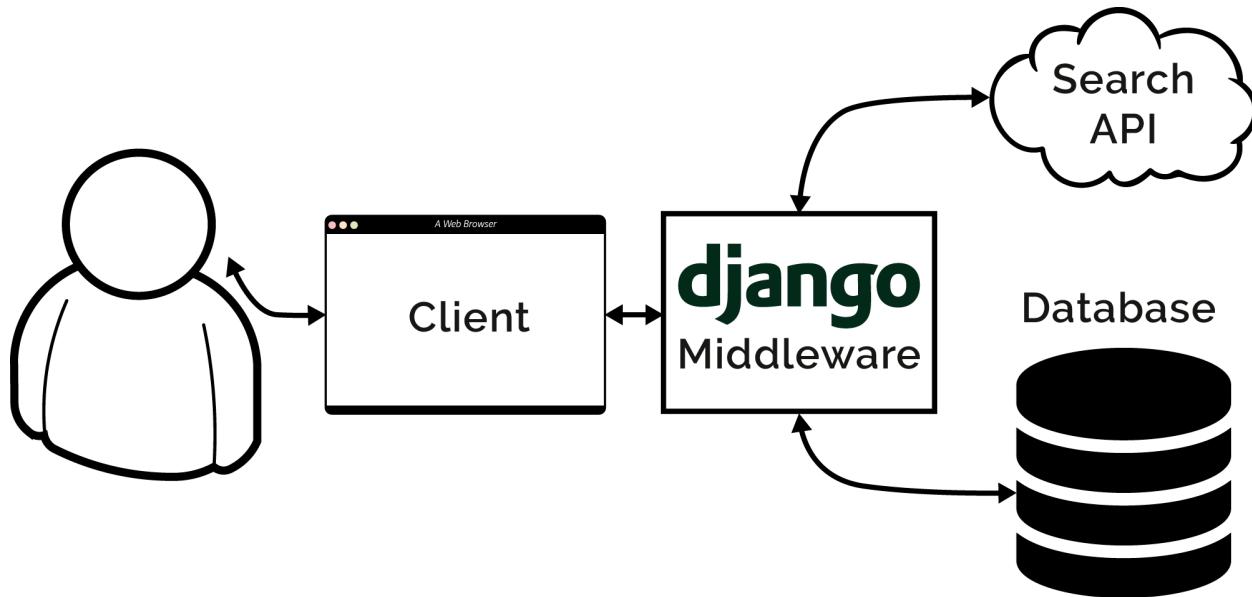
- What is the high-level architecture going be? Draw up a **N-Tier or System Architecture** diagram to represent the high-level system components.
- What is the interface going to look like? Draw up some **Wireframes** of the main and category pages.
- What are the URLs that users visit going to look like? Write down a series of **URL mappings** for the application.
- What data are we going to have to store or represent? Construct an **Entity-Relationship (ER)**²¹ diagram to describe the data model that we'll be implementing.

Try these exercises out before moving on – even if you aren't familiar with system architecture diagrams, wireframes or ER diagrams, how would you explain and describe, formally, what you are going to build so that someone else can understand it.

²¹https://en.wikipedia.org/wiki/Entity–relationship_model

N-Tier Architecture

The high-level architecture for most web applications is based around a *3-Tier architecture*. Rango will be a variant on this architecture as also interfaces with an external service.



Overview of the 3-tier system architecture for our Rango application.

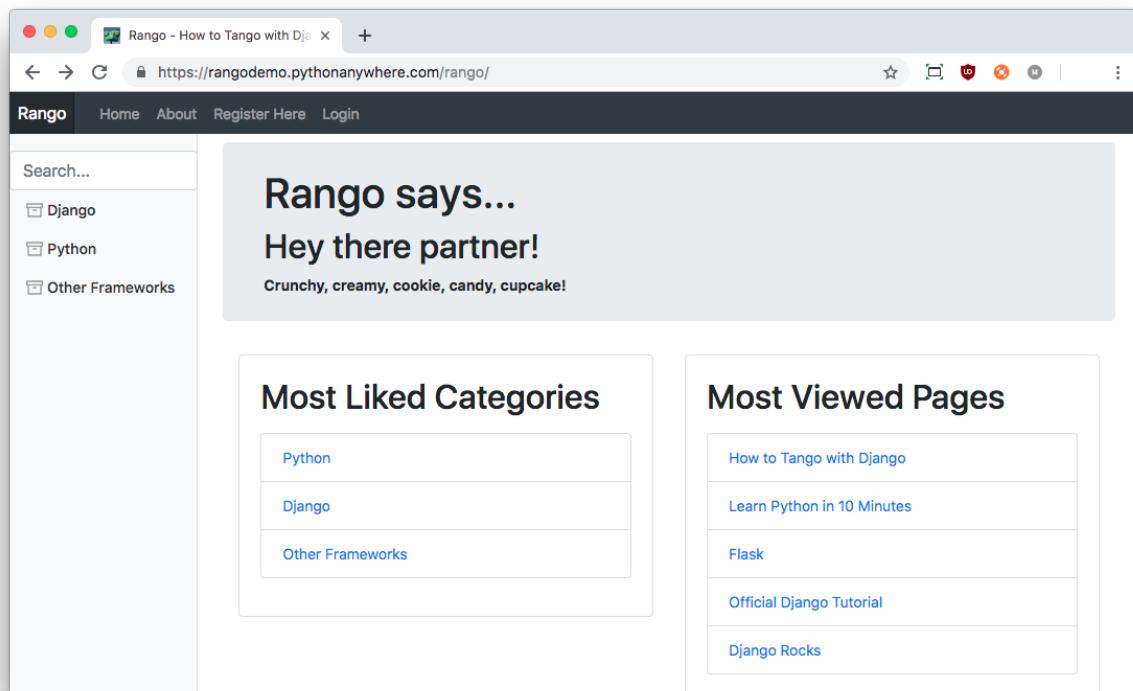
Given the different boxes within the high-level architecture, we need to start making some decisions about the technologies that will be going into each box. Since we are building a web application with Django, we will use the following technologies for the following tiers.

- The **client** will be a web browser (such as *Chrome*, *Firefox*, and *Safari*) which will render HTML/CSS pages, and any interpret JavaScript code.
- The **middleware** will be a *Django* application and will be dispatched through Django's built-in development web server while we develop (and then later a web server like *Nginx* or *Apache web server*).
- The **database** will be the Python-based *SQLite3* Database engine.
- The **search API** will be the *Bing Search API*.

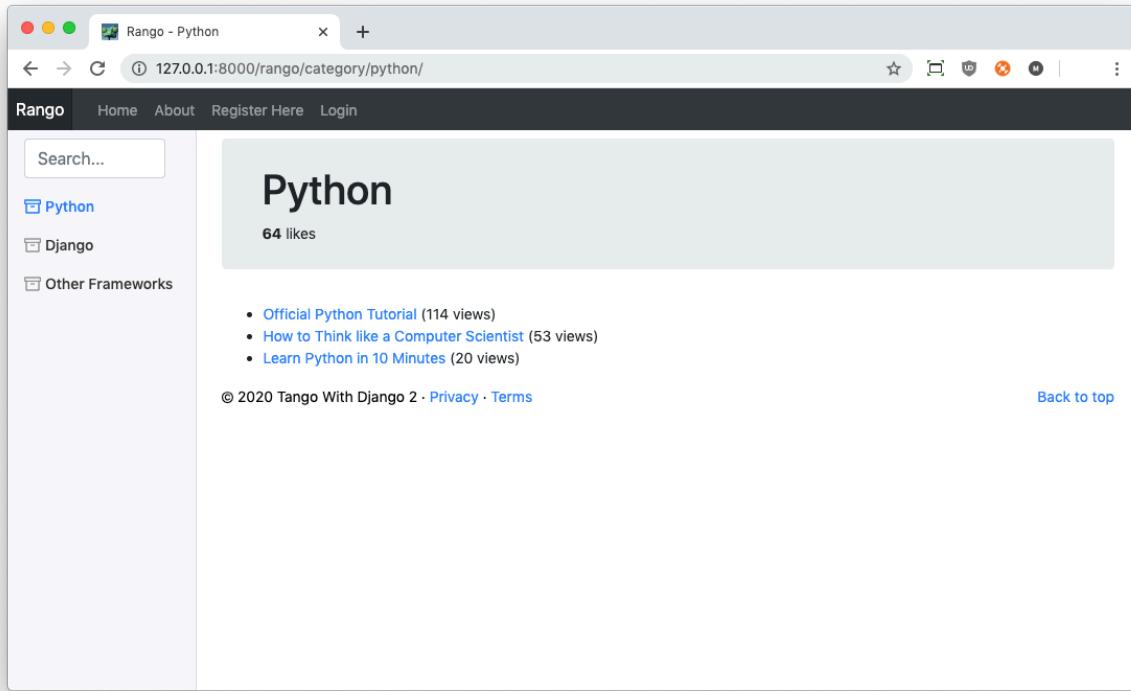
For the most part, this book will focus on developing middleware. However, it should be evident from the [system architecture diagram](#) that we will have to interface with all the other components.

Wireframes

Wireframes are a great way to provide clients with some idea of what the application is going to look like, and what features it will provide. They can vary from hand-drawn sketches to exact mockups depending on the tools that you have at your disposal. For our Rango application, we'd like to make the index page of the site look like the [screenshot below](#). Our category page is also [shown below](#).



The index page with a categories search bar on the left, also showing the top five pages and top five categories.



The category page showing the pages in the category (along with the number of views for the category and each page).

Pages and URL Mappings

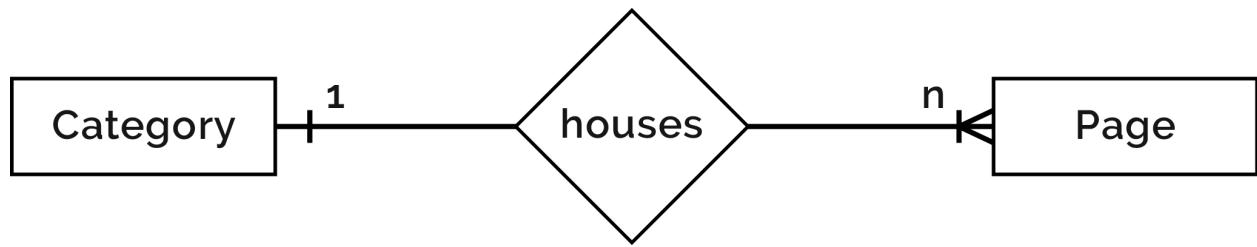
From the specification, we have already identified two pages that our application will present to the user at different points in time. To access each page, we will need to describe URL mappings. Think of a URL mapping as the text a user would have to enter into a browser's address bar to access a given page. The basic URL mappings for Rango are shown below.

- / or /rango/ will point to the main / index page.
- /rango/about/ will point to the about page.
- /rango/category/<category_name>/ will point to the category page for <category_name>, where the category might be:
 - games;
 - python-recipes; or
 - code-and-compilers.

As we build our application, we will probably need to create other URL mappings. However, the mappings listed above are enough for us to get started. As we progress through the book, we will flesh out how to construct all of these pages using the Django framework and use its [Model-View-Template²²](#) design pattern.

Entity-Relationship Diagram

Now that we have a gist of the URL mappings and what the pages are going to look like, we need to define the data model that will house the data for our web application. Given the specification, it should be clear that we have at least two entities: a *category* and a *page*. It should also be clear that a *category* can be associated with many *pages*. We can formulate the following ER Diagram to describe this simple data model.



The Entity Relationship Diagram of Rango's two main entities.

Note that this specification is rather vague. A single page could, in theory, exist in one or more categories. Working with this assumption, we could model the relationship between categories and pages as a [many-to-many relationship²³](#). However, this approach introduces several complexities.

We will make the simplifying assumption that **one category contains many pages, but one page is assigned to one category**. This does not preclude that the same page can be assigned to different categories – but the page would have to be entered twice. While this is not ideal, it does reduce the complexity of the models.

²²<https://docs.djangoproject.com/en/2.1/>

²³[https://en.wikipedia.org/wiki/Many-to-many_\(data_model\)](https://en.wikipedia.org/wiki/Many-to-many_(data_model))



Take Note!

Get into the habit of noting down any working assumptions that you make, just like the one-to-many relationship assumption that we assume above. You never know when they may come back to bite you later on! By noting them down, this means you can communicate it with your development team and make sure that the assumption is sensible, and that they are happy to proceed under such an assumption.

With this assumption, we can produce a series of tables that describe each entity in more detail. The tables contain information on what fields are contained within each entity. We use Django `ModelField` types to define the type of each field (i.e. `IntegerField`, `CharField`, `URLField` or `ForeignKey`). Note that in Django *primary keys* are implicit such that Django adds an `id` to each Model, but we will talk more about that later in the [Models and Databases chapter](#).

Category Model Fields and Data Types

Field	Data Type
name	CharField
views	IntegerField
likes	IntegerField

Page Model Fields and Data Types

Field	Data Type
category	ForeignKey
title	CharField
url	URLField
views	IntegerField

We will also have a model for the `User` so that they can register and login. We have not shown it here but shall introduce it later in the book when we discuss [user authentication](#). In subsequent chapters, we will see how to instantiate these models in Django, and how we can use the built-in ORM to interact with the database.

1.5 Summary

These high-level design and specifications will serve as a useful reference point when building our web application. While we will be focusing on using specific technologies, these steps are common to most database-driven websites. It's a good idea to become familiar with reading and producing such specifications and designs so that you can communicate your designs and ideas with others. Here, we will be focusing on using Django and the related technologies to implement this specification.



Cut and Paste Coding

As you progress through the tutorial, you'll most likely be tempted to cut and paste the code from the book to your code editor. **However, it is better to type in the code.** We know that this is a hassle, but it will help you to remember the process and get a feel for the commands that you will be using again and again.

Furthermore, cutting and pasting Python code is asking for trouble. Whitespace can end up being interpreted as spaces, tabs or a mixture of spaces and tabs. This will lead to all sorts of weird errors, and not necessarily indent errors. If you do cut and paste code be wary of this. Pay particular attention to this with regards to tabs and spaces – mixing these up will likely lead to a `TabError`.

Most code editors will show the ‘hidden characters’, which in turn will show whether whitespace is either a tab or a space. If you have this option, turn it on. You will likely save yourself a lot of confusion.



Representing Commands

As you work through this book, you'll encounter lots of text that will be entered into your computer's terminal or Command Prompt. Snippets starting with a dollar sign (\$) denotes a command that must be entered – the remainder of the line is the command. In a UNIX terminal, the dollar represents a separator between the *prompt* and the command that you enter.

```
david@seram:~ $ exit
```

In the example above, the prompt `david@seram:~` tells us our username (`david`), computer name (`seram`) and our current directory (~, or our home directory). After the \$, we have entered the command `exit`, which, when executed, will close the terminal. As such, the command that you would actually type here would simply be `exit`. Refer to the [UNIX chapter for more information](#).

Whenever you see `>>>`, the following is a command that should be entered into the interactive Python interpreter. This is launched by issuing `$ python`. See what we did there? Once inside the Python interpreter, you can exit it by typing `quit()` or `exit()`.

2. Getting Ready to Tango

Before we start coding, it's really important that we set your development environment up correctly so that you can *Tango with Django* with ease. You'll need to make sure that you have all of the necessary components installed on your computer, and that they are configured correctly. This chapter outlines the six key components you'll need to be aware of, setup and use. These are:

- the [terminal¹](#) (on macOS or UNIX/Linux systems), or the [Command Prompt²](#) (on Windows);
- *Python 3*, including how to code and run Python scripts;
- the Python Package Manager *pip*;
- *Virtual Environments*;
- your *Integrated Development Environment (IDE)*, if you choose to use one; and
- a *Version Control System (VCS)* called *Git*.

We also touch on the merits of *unit testing*, discussing how being able to test your implementation at different points can help keep you on track.

If you already have Python 3 and Django 2 installed on your computer and are familiar with the technologies listed above, you can skip straight ahead to the [Django Basics chapter](#). If you are not familiar with some or all of the technologies listed, we provide an overview of each below. These go hand in hand with later [supplementary chapter](#) that provides a series of pointers on how to set the different components up, if you need help doing so.

¹https://en.wikipedia.org/wiki/Terminal_emulator

²<https://en.wikipedia.org/wiki/Cmd.exe>



You Development Environment is Important!

Setting up your development environment can be a tedious and frustrating process. It's not something that you would do every day. The pointers we provide in this chapter (and the [additional supplementary chapter](#)) will help you in getting everything to a working state. The effort you expend now in making sure everything works will ensure that development can proceed unhindered, without you needing to go back and patch things up.

From experience, we can also say with confidence that as you set your environment up, it's a good idea to note down the steps that you took. You will probably need that workflow again one day – maybe you will purchase a new computer, or be asked to help a friend set their environment up, too.

Don't think short-term, think long-term!

2.1 Python 3

To work with Tango with Django, we require you to have installed on your computer a copy of the *Python 3* programming language. A Python version of 3.5 or greater should work fine with Django 2.0, 2.1 and 2.2 – although the official Django website recommends that you have the most recent version of Python installed. As such, we recommend you install *Python 3.7*. At the time of writing, the most recent release is *Python 3.7.2*. If you're not sure how to install Python and would like some assistance, have a look at [our quick guide on how to install Python](#).



Running macOS, Linux or UNIX?

On installations of macOS, Linux or UNIX, you will find that Python is already installed on your computer – albeit a much older version, typically 2.x. This version is required by your operating system to perform essential tasks such as downloading and installing updates. While you can use this version, it won't be compatible with Django 2, and you'll need to install a newer version of Python to run *side-by-side* with the old installation. *Do not uninstall or hack away at deleting Python 2.x* if it is already present on your system; you may break your operating system!

Django 2.0, 2.1 or 2.2?



In this book, we explicitly use Django version 2.1.5. However, we have also tested the instructions provided with versions 2.0.13, 2.1.10, and 2.2.3. Therefore, you will be able to use version 2.2 if you wish! If you do use a different version, substitute 2.1.5 with the version you are using.

We'll be regularly checking the compatibility of the instructions provided with future Django releases. If you notice any issues with later versions, feel free to get in touch with us. You can [send us a tweet³](#), [raise an issue on GitHub⁴](#), or e-mail us – our addresses are available on the [www.tangowithdjango.com⁵](http://www.tangowithdjango.com) website.

You must however make sure you are using *at least* Python version 3.5. Version 3.4 and below are incompatible with these releases of Django.

Python Skills Rusty?



If you haven't used Python before – or you simply want to brush up on your skills – then we highly recommend that you check out and work through one or more of the following guides:

- [The Official Python Tutorial⁶](#);
- [Think Python: How to Think like a Computer Scientist⁷](#) by Allen B. Downey; or
- [Learn Python in 10 Minutes⁸](#) by Stavros;
- [Learn to Program⁹](#) by Jennifer Campbell and Paul Gries.

These guides will help you familiarise yourself with the basics of Python so you can start developing with Django. Note you don't need to be an expert in Python to work with Django – Python is straightforward to use, and you can pick it up as you go, especially if you already know the ins and outs of at least one other programming language.

³<https://twitter.com/tangowithdjango>

⁴https://github.com/leifos/tango_with_django_2/issues

⁵<https://www.tangowithdjango.com>

⁶<https://docs.python.org/3/tutorial/>

⁷<https://greenteapress.com/wp/think-python-2e/>

⁸<https://www.stavros.io/tutorials/python/>

⁹<https://www.coursera.org/course/programming1>

2.2 Virtual Environments

With a working installation of Python 3 (and the basic programming skills to go with it), we can now setup our environment for the Django project (called Rango) we'll be creating in this tutorial. One super useful tool we *strongly* encourage you to use is a virtual environment. Although not strictly necessary, it provides a useful separation between your computer's Python installation and the environment you'll be using to develop Rango with.

A virtual environment allows for multiple installations of Python packages to exist in harmony, within unique *Python environments*. Why is this useful? Say you have a project, `projectA` that you want to run in Django 1.11, and a further project, `projectB` written for Django 2.1. This presents a problem as you would normally only be able to install one version of the required software at a time. By creating virtual environments for each project, you can then install the respective versions of Django (and any other required Python software) within each unique environment. This ensures that the software installed in one environment does not tamper with the software installed on another.

You'll want to create a virtual environment using Python 3 for your Rango development environment. Call the environment `rangoenv`. If you are unsure as to how to do this, go to the supplementary chapter detailing [how to set up virtual environments before continuing](#). If you do choose to use a virtual environment, remember to activate the virtual environment by issuing the following command.

```
$ workon rangoenv
```

From then on, all of your prompts with the terminal or Command Prompt will precede with the name of your virtual environment to remind you that it is switched on. Check out the following example to know what we are discussing.

```
$ workon rangoenv  
(rangoenv) $ pip install django==2.1.5  
...  
(rangoenv) $ deactivate  
$
```

The penultimate line of the example above demonstrates how to switch your virtual environment off after you have finished with it – note the lack of (rangoenv) before the prompt. Again, [refer to the system setup chapter in the appendices of this book](#) for more information on how to setup and use virtual environments.

2.3 The Python Package Manager

Going hand in hand with virtual environments, we'll also be making use of the Python package manager, *pip*, to install several different Python software packages – including Django – to our development environment. Specifically, we'll need to install two packages: Django 2 and *Pillow*. Pillow is a Python package providing support for handling image files (e.g. .jpg and .png files), something we'll be doing later in this tutorial.

A package manager, whether for Python, your [operating system¹⁰](#) or [some other environment¹¹](#), is a software tool that automates the process of installing, upgrading, configuring and removing *packages* – that is, a package of software which you can use on your computer that provides some functionality. This is opposed to downloading, installing and maintaining software manually.

Maintaining Python packages is pretty painful. Most packages often have *dependencies* – additional packages that are required for your package to work! This can get very complex very quickly. A package manager handles all of this for you, along with issues such as conflicts regarding different versions of a package. Luckily, *pip* handles all this for you.

Try and run the command `$ pip` to execute the package manager. Make sure you do this with your virtual environment activated. Globally, you may have to use the

¹⁰https://en.wikipedia.org/wiki/Advanced_Packaging_Tool

¹¹<https://docs.npmjs.com/cli/install>

command pip3. If these don't work, you have a setup issue – refer to our [pip setup guide](#) for help.

With your virtual environment switched on, execute the following two commands to install Django and Pillow.

```
$ pip install django==2.1.5  
$ pip install pillow==5.4.1
```

Installing these two packages will be sufficient to get you started. As you work through the tutorial, there will be a couple more packages that we will require. We'll tell you to install them as we require them. For now, you're good to go.



Problems Installing pillow?

When installing Pillow, you may receive an error stating that the installation failed due to a lack of JPEG support. This error is shown as the following:

```
ValueError: jpeg is required unless explicitly disabled using  
--disable-jpeg, aborting
```

If you receive this error, try installing Pillow *without* JPEG support enabled, with the following command.

```
pip install pillow==5.4.1 --global-option="build_ext"  
--global-option="--disable-jpeg"
```

While you obviously will have a lack of support for handling JPEG images, Pillow should then install without problem. Getting Pillow installed is enough for you to get started with this tutorial. For further information, check out the [Pillow documentation](#)¹².



Working within in a Virtual Environment

Substitute pip3 with pip when working within your virtual environment. The command pip is aliased to the correct one for your virtual environment.

¹²<https://pillow.readthedocs.io/en/stable/installation.html>

2.4 Integrated Development Environment

While not necessary, a good Python-based IDE can be very helpful to you during the development process. Several exist, with perhaps *PyCharm*¹³ by JetBrains and *PyDev* (a plugin of the *Eclipse IDE*¹⁴) standing out as popular choices. The *Python Wiki*¹⁵ provides an up-to-date list of Python IDEs.

Research which one is right for you, and be aware that some may require you to purchase a licence. Ideally, you'll want to select an IDE that supports integration with Django. Of course, if you prefer not to use an IDE, using a simple text editor like *Sublime Text*¹⁶, *TextMate*¹⁷ or *Atom*¹⁸ will do just fine. Many modern text editors support Python syntax highlighting, which makes things much easier!

We use PyCharm as it supports virtual environments and Django integration – though you will have to configure the IDE accordingly. We don't cover that here – although JetBrains does provide a [guide on setting PyCharm up](#)¹⁹.

2.5 Version Control

We should also point out that when you develop code, you should always house your code within a version-controlled repository such as *SVN*²⁰ or *Git*²¹. We won't be explaining this right now so that we can get stuck into developing an application in Django. We have however written a [chapter providing a crash course on Git](#) for your reference that you can refer to later on. **We highly recommend that you set up a Git repository for your projects.**

¹³<http://www.jetbrains.com/pycharm/>

¹⁴<http://www.eclipse.org/downloads/>

¹⁵<http://wiki.python.org/moin/IntegratedDevelopmentEnvironments>

¹⁶<https://www.sublimetext.com/>

¹⁷<https://macromates.com/>

¹⁸<https://atom.io/>

¹⁹<https://www.jetbrains.com/help/pycharm/2016.1/creating-and-running-your-first-django-project.html>

²⁰<http://subversion.tigris.org/>

²¹<http://git-scm.com/>



Exercises

To get comfortable with your environment, try out the following exercises.

- Get up to speed with Python if you're new to the language. Try out one or more of the tutorials we listed earlier.
- Install Python 3.7. Make sure `pip3` (or `pip` within your virtual environment) is also installed and works on your computer.
- Play around with your *command line interface (CLI)*, whether it be the Command Prompt (Windows) or a terminal (macOS, Linux, UNIX, etc.).
- Create a new virtual environment using Python 3.7. This is optional, but we *strongly encourage you to use virtual environments*.
- Within your environment, install Django 2 and Pillow 5.4.1.
- Set up an account on a Git repository site like [GitHub²²](#) or [BitBucket²³](#) if you haven't already done so.
- Download and set up an IDE like [PyCharm²⁴](#), or set up your favourite text editor for working with Python files.

As previously stated, we've made the code for the application available on our [GitHub repository²⁵](#).

- If you spot any errors or problems in the book, please let us know by making an [issue on GitHub²⁶](#).
- If you have any problems with the exercises, you can check out the repository to see how we completed them.
- If you spot any errors or problems with the unit tests and/or sample solutions we provide, you can also let us know by raising an issue in the [relevant repository²⁷](#).

²²<https://github.com/>

²³<https://bitbucket.org/>

²⁴<https://www.jetbrains.com/pycharm/>

²⁵https://github.com/maxwelld90/tango_with_django_2_code

²⁶https://github.com/leifos/tango_with_django_2/issues

²⁷https://github.com/maxwelld90/tango_with_django_2_code/issues

2.6 Testing your Implementation

As you work through your implementation of the requirements for the Rango app, we want you to have the confidence to know that *what you are coding up is correct*. We can't physically sit next to you, so we've gone and done the next best thing – **we've implemented a series of different tests that you can run against your codebase to see what's correct, and what can be improved**. We've got tests for the core chapters (up to Chapter 10).

These are available from our sample codebase repository, [available on GitHub²⁸](#). The `progress_tests` directory on this repository contains a number of different Python modules, each containing a series of different test modules you can run against your Rango implementation. Note that they are for individual chapters – for example, you should run the module `tests_chapter3.py` against your implementation *after* completion of Chapter 3, but before starting Chapter 4. Note that not every chapter will have tests at the end of it – some portions of the book are very difficult to write automated tests for without resorting to having to download sizable support libraries.



Complete the Exercises!

These tests assume that you complete all of the exercises for a chapter! If you don't do this, it's likely some tests will not pass.

We check the basic functionality that should be working up to the point you are testing at. We also check what is returned from the server when a particular URL is accessed – and if the response doesn't match *exactly* what we requested in the book, *the test will fail*. This might seem overly harsh, but we want to drill into your head that *you must satisfy requirements exactly as they are laid out – no deviation is acceptable*. This also drills into your head the idea of *test-driven development*, something that we outline [at the start of the testing chapter](#).

²⁸https://github.com/maxwelld90/tango_with_django_2_code/tree/master/progress_tests

Running the Unit Tests

How do you run the tests, though? This step-by-step process demonstrates the basic process on what you have to do. We will assume that you want to run the tests for [Chapter 3, Django Basics](#).

1. First, identify what chapter's tests you want to run.
2. Either make a clone of our [sample code repository²⁹](#) on your computer, or access the individual test module that you want from the [GitHub web interface³⁰](#).
 - To do the latter, click the module you require (i.e. `tests_chapter3.py`). When you see the code on the GitHub website, click the Raw button and save the page that then loads.
3. Move the `tests_chapter3.py` module to your project's `rango` directory. This step does not make sense right now; as you progress through the book and come back here to refresh your memory on what to do, this will make sense.
4. Run the command `$ python manage.py test rango.tests_chapter3`. This will start the tests.

You will also need to ensure that when these tests run, your `rangoenv` virtual environment is active.

Once the tests all complete, you should see `OK`. This means they all passed! If you don't see `OK`, something failed – look through the output of the tests to see what test failed, and why. Sometimes, you might have missed something which causes an exception to be raised before the test can be carried out. In instances like this, you'll need to look at what is expected, and go back and fill it in. You can tweak your code and re-run the tests to see if they then pass.



Test your Implementation

When you have completed enough of the book to reach another round of tests, we'll denote the prompt for you to do this like so. We'll tell you what module to run, and always point you back to here so you can refresh your memory if you forget how to run them.

²⁹https://github.com/maxwelld90/tango_with_django_2_code

³⁰https://github.com/maxwelld90/tango_with_django_2_code/tree/master/progress_tests



Delete when Complete!

When you have finished with the tests for a particular chapter, we **highly recommend** that you delete the module that you moved over to your `rango` directory. In the example above, we'd be looking to delete `tests_chapter3.py`. Once you have confirmed your solution passes the tests we provide, there's no need for the module anymore. Just delete it – don't clutter your repository up with these modules!



Test Updates

Over time, we may release updates to the unit test files if an issue is raised, or we decide to add extra tests if there is sufficient demand for us to do so. You should keep an eye on the repository for any changes – if you cloned it to your computer, you can simply `git pull` to retrieve updates.

3. Django Basics

Let's get started with Django! In this chapter, we'll be giving you an overview of the creation process. You'll be setting up a new project and a new web application. By the end of this chapter, you will have a simple Django powered website running!

3.1 Testing Your Setup

Let's start by checking that your Python and Django installations are correct for this tutorial. To do this, open a new terminal/Command Prompt window, and activate your `rangoenv` virtual environment.

Once activated, issue the following command. The output will tell what Python version you have.

```
$ python --version
```

The response should be something like `3.7.2`, but any `3.5+` versions of Python should work fine. If you need to upgrade or install Python, go to the chapter on [setting up your system](#).

If you are using a virtual environment, then ensure that you have activated it – if you don't remember how then have a look at our chapter on [virtual environments](#).

After verifying your Python installation, check your Django installation. In your terminal window, run the Python interpreter by issuing the following command.

```
$ python
Python 3.7.2 (default, Mar 30 2019, 05:40:15)
[Clang 9.0.0 (clang-900.0.39.2)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

At the prompt, enter the following commands:

```
>>> import django
>>> django.get_version()
'2.1.5'
>>> exit()
```

All going well you should see the correct version of Django, and then can use `exit()` to leave the Python interpreter. If `import django` fails to import, then check that you are in your virtual environment, and check what packages are installed with `pip list` at the terminal window.

If you have problems with installing the packages or have a different version installed, go to [System Setup](#) chapter or consult the [Django Documentation on Installing Django](#)¹.

3.2 Creating Your Django Project

To create a new Django Project, go to your workspace directory, and issue the following command:

```
$ django-admin.py startproject tango_with_django_project
```

If you don't have a workspace directory, we recommend that you create one. This means that you can house your Django projects (and other code projects) within this directory. It keeps things organised, without you placing directories containing code in random places, such as your Desktop directory!

We will refer to your workspace directory throughout this book as `<workspace>`. You will have to substitute this with the path to your workspace directory. For example,

¹<https://docs.djangoproject.com/en/2.1/topics/install/>

we recommend that you create a `workspace` directory in your home folder. The path `/Users/maxwelld90/Workspace/` would then constitute as a valid directory for the user `maxwelld90` on a Mac.



Can't find `django-admin.py`?

Try entering `django-admin` instead. Depending on your setup, some systems may not recognise `django-admin.py`. This is especially true on Windows computers – you may have to use the full path to the `django-admin.py` script, for example:

```
python c:\Users\maxwelld90\.virtualenvs\rangoenv\bin\django-admin.py  
    startproject tango_with_django_project
```

as suggested on [StackOverflow²](#). Note that the path will likely vary on your own computer.

This command will invoke the `django-admin.py` script, which will set up a new Django project called `tango_with_django_project` for you. Naming conventions dictate that we would typically append `_project` to the end of our Django project directories so we know exactly what they contain – but naming this is really entirely up to you.

You'll now notice within your workspace is a directory set to the name of your new project, `tango_with_django_project`. Within this newly created directory, you should see two items:

- `tango_with_django_project`, another *nested directory* with the same name as your main project directory (argh!); and
- a Python script called `manage.py`.

For the purposes of this tutorial, we call the `tango_with_django_project` nested directory the *project configuration directory*. Within this directory, you will find four Python scripts. We will discuss these scripts in detail later on, but for now, you should see:

²<http://stackoverflow.com/questions/8112630/cant-create-django-project-using-command-prompt>

- `__init__.py`, a blank Python script whose presence indicates to the Python interpreter that the directory is a Python package;
- `settings.py`, the place to store all of your Django project's settings;
- `urls.py`, a Python script to store URL patterns for your project; and
- `wsgi.py`, a Python script used to help run your development server and deploy your project to a production environment.

In the project directory, you will see there is a file called `manage.py`. We will be calling this script time and time again as we develop our project. It provides you with a series of commands you can run to maintain your Django project. For example, `manage.py` allows you to run the built-in Django development server, test your application, and run various database commands. We will be using the script for virtually every Django command we want to run.



The Django Admin and Manage Scripts

For Further Information on Django admin script, see the Django documentation for more details about the [Admin and Manage scripts³](#).

Note that if you run `python manage.py help` you can see the list of commands available.

You can try using the `manage.py` script now, by issuing the following command.

```
$ python manage.py runserver
```

Executing this command will launch Python, and instruct Django to initiate its lightweight development server. You should see the output in your terminal window similar to the example shown below:

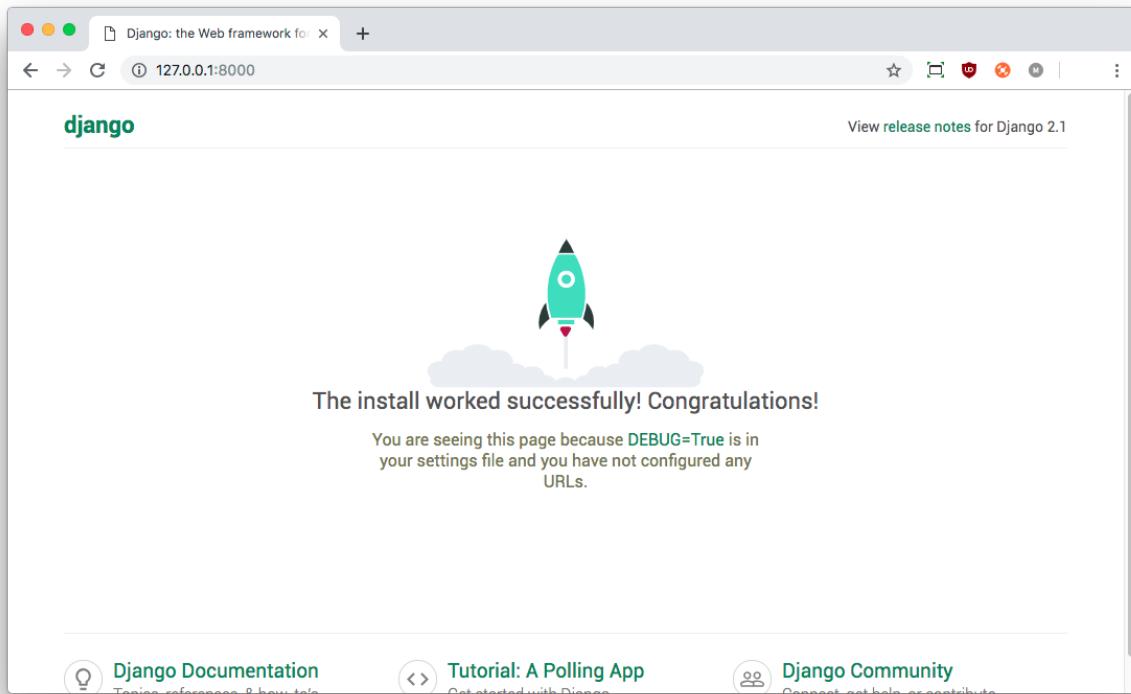
³<https://docs.djangoproject.com/en/2.1/ref/django-admin/#django-admin-py-and-manage-py>

```
$ python manage.py runserver  
Performing system checks...  
  
System check identified no issues (0 silenced).  
  
You have 14 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.  
  
July 23, 2019 - 17:12:34  
Django version 2.1.5, using settings 'tango_with_django_project.settings'  
Starting development server at http://127.0.0.1:8000/  
Quit the server with CONTROL-C.
```

In the output, you can see several things. First, there are no issues that stop the application from running. However, you will notice that a warning is raised – unapplied migration(s). We will talk about this in more detail when we set up our database, but for now we can ignore it. Third, and most importantly, you can see that a URL has been specified: <http://127.0.0.1:8000/>, which is the address that the Django development server is running at.

Now open up your web browser and enter the URL mentioned above into your browser's address bar – <http://127.0.0.1:8000/>⁴. You should see a webpage similar to [the one shown below](#).

⁴<http://127.0.0.1:8000/>



A screenshot of the initial Django page you will see when running the development server for the first time.

You can stop the development server at any time by pushing **CTRL + C** in your terminal or Command Prompt window. This applies to both Macs and PCs! If you wish to run the development server on a different port or allow users from other machines to access it, you can do so by supplying optional arguments. Consider the following command.

```
$ python manage.py runserver <your_machines_ip_address>:5555
```

Executing this command will force the development server to respond to incoming requests on TCP port 5555. You will need to replace `<your_machines_ip_address>` with your computer's IP address or `127.0.0.1`.



Don't know your IP Address?

If you use `0.0.0.0`, Django figures out what your IP address is. Go ahead and try:

```
python manage.py runserver 0.0.0.0:5555
```

When setting ports, it is unlikely that you will be able to use TCP port 80 or 8080 as these are traditionally reserved for HTTP traffic. Also, any port below 1024 is considered to be **privileged⁵** by your operating system.

While you won't be using the lightweight development server to deploy your application, it's nice to be able to demo your application on another machine in your network. Running the server with your machine's IP address will enable others to enter in `http://<your_machines_ip_address>:<port>/` and view your web application. **Of course, this will depend on how your network is configured. There may be proxy servers or firewalls in the way that would need to be configured before this would work. Check with the administrator of the network you are using if you can't view the development server remotely.**

3.3 Creating a Django App

A Django project is a collection of *configurations* and *apps* that together make up a given web application or website. One of the intended outcomes of using this approach is to promote good software engineering practices. By developing a series of small applications, the idea is that you can theoretically drop an existing application into a different Django project and have it working with minimal effort.

A Django application exists to perform a particular task. You need to create specific apps that are responsible for providing your site with particular kinds of functionality. For example, we could imagine that a project might consist of several apps including a polling app, a registration app, and a specific content related app. In another project, we may wish to re-use the polling and registration apps, and so can include them in other projects. We will talk about this later. For now, we are going to create the app for the *Rango* app.

To achieve this, you need to make sure you're in your Django project's directory (e.g. `<workspace>/tango_with_django_project`). From there, run the following command.

```
$ python manage.py startapp rango
```

⁵<http://www.w3.org/Daemon/User/Installation/PrivilegedPorts.html>

The `startapp` command creates a new directory within your project's root. Unsurprisingly, this directory is called `rango` – and contained within it are several Python scripts:

- another `__init__.py`, serving the same purpose as discussed previously;
- `admin.py`, where you can register your models so that you can benefit from some Django machinery which creates an admin interface for you;
- `apps.py`, that provides a place for any app-specific configuration;
- `models.py`, a place to store your app's data models – where you specify the entities and relationships between data;
- `tests.py`, where you can store a series of functions to test your implementation;
- `views.py`, where you can store a series of functions that handle requests and return responses; and
- the `migrations` directory, which stores database specific information related to your models.

`views.py` and `models.py` are the two files you will use for any given app and form part of the main architectural design pattern employed by Django, i.e. the *Model-View-Template* pattern. You can check out [the official Django documentation](#)⁶ to see how models, views and templates relate to each other in more detail.

Before you can get started with creating your models and views, you must first tell your Django project about your new app's existence. To do this, you need to modify the `settings.py` file, contained within your project's configuration directory. Open the file and find the `INSTALLED_APPS` list. Add the `rango` app to the end of the tuple, which should then look like the following example.

⁶<https://docs.djangoproject.com/en/2.1/intro/overview/>

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'rango',
]
```

Verify that Django picked up your new app by running the development server again. If you can start the server without errors, your app was picked up and you will be ready to proceed to the next step.

3.4 Creating a View

With our Rango app created, let's now create a simple view. Views handle a *request* that comes from the client, *executes some code*, and provides a *response* to the client. To fulfil the request, it may contact other services or query for data from other sources. The job of a view is to collate and package the data required to handle the request, as we outlined above. For our first view, given a request, the view will simply send some text back to the client. For the time being, we won't concern ourselves about using models (i.e. getting data from other sources) or templates (i.e. which help us package our responses nicely).

In your editor, open the file `views.py`, located within your newly created `rango` app directory. Remove the comment `# Create your views here.`, but keep the following line line – you'll use it later on!

```
from django.shortcuts import render
```

You can now add in the following code underneath the aforementioned `import` statement.

```
from django.http import HttpResponse

def index(request):
    return HttpResponse("Rango says hey there partner!")
```

Breaking down the three lines of code, we observe the following points about creating this simple view.

- We first import the `HttpResponse`⁷ object from the `django.http` module.
- Each view exists within the `views.py` file as a series of individual functions. In this instance, we only created one view – called `index`.
- Each view takes in at least one argument – a `HttpRequest`⁸ object, which also lives in the `django.http` module. Convention dictates that this is named `request`, but you can rename this to whatever you want if you so desire.
- Each view must return a `HttpResponse` object. A simple `HttpResponse` object takes a string parameter representing the content of the page we wish to send to the client requesting the view.

With the view created, you're only part of the way to allowing a user to access it. For a user to see your view, you must map a **Uniform Resource Locator (URL)**⁹ to the view.

To create an initial mapping, open `urls.py` located in your project configuration directory (i.e. `<workspace>/tango_with_django_project/tango_with_django_project` – the second `tango_with_django_project` directory!) and add the following import to the top of the file – underneath `from django.urls import path`.

```
from rango import views
```

Next, change the `urlpatterns` list to look like the example below.

⁷<https://docs.djangoproject.com/en/2.1/ref/request-response/#django.http.HttpResponse>

⁸<https://docs.djangoproject.com/en/2.1/ref/request-response/#django.http.HttpRequest>

⁹http://en.wikipedia.org/wiki/Uniform_resource_locator

```
urlpatterns = [
    path('', views.index, name='index'),
    path('admin/', admin.site.urls),
]
```

This maps the basic URL to the `index` view in the `rango` app. Run the development server (e.g. `python manage.py runserver`) and visit `http://127.0.0.1:8000` or whatever address your development server is running on. You'll then see the rendered output of the `index` view.

3.5 Mapping URLs

Rather than directly mapping URLs from the project to the app, we can make our app more modular (and thus re-usable) by changing how we route the incoming URL to a view. To do this, we first need to modify the project's `urls.py` and have it point to the app to handle any specific Rango app requests. We then need to specify how Rango deals with such requests.

First, open the project's `urls.py` file which is located inside your project configuration directory. As a relative path from your workspace directory, this would be the file `<workspace>/tango_with_django_project/tango_with_django_project/urls.py`. Update module to look like the example below.

```
from django.contrib import admin
from django.urls import path
from django.urls import include
from rango import views

urlpatterns = [
    path('', views.index, name='index'),
    path('rango/', include('rango.urls')),
    # The above maps any URLs starting with rango/ to be handled by rango.
    path('admin/', admin.site.urls),
]
```

You will see that the `urlpatterns` is a Python list, which is expected by the Django framework. The added mapping looks for URL strings that match the patterns

rango/. When a match is made, the remainder of the URL string is then passed onto and handled by `rango.urls` through the use of the `include()` function from within the `django.urls` package.

`http://www.servername.com/rango/about/`

Protocol and Domain Name

`http://www.servername.com/`

Project Configuration's `urls.py`

`rango/`

Rango's (your app's) `urls.py`

`about/`

An illustration of a URL, represented as a chain, showing how different parts of the URL following the domain are the responsibility of different `url.py` files.

Think of this as a chain that processes the URL string – as illustrated in the [URL chain figure](#). In this chain, the domain is stripped out and the remainder of the URL string (`rango/`) is passed on to `tango_with_django` project, where it finds a match and strips away `rango/`, leaving an empty string to be passed on to the app `rango` for it to handle.

Consequently, we need to create a new file called `urls.py` in the `rango` app directory, to handle the remaining URL string (and map the empty string to the `index` view):

```
from django.urls import path
from rango import views

app_name = 'rango'

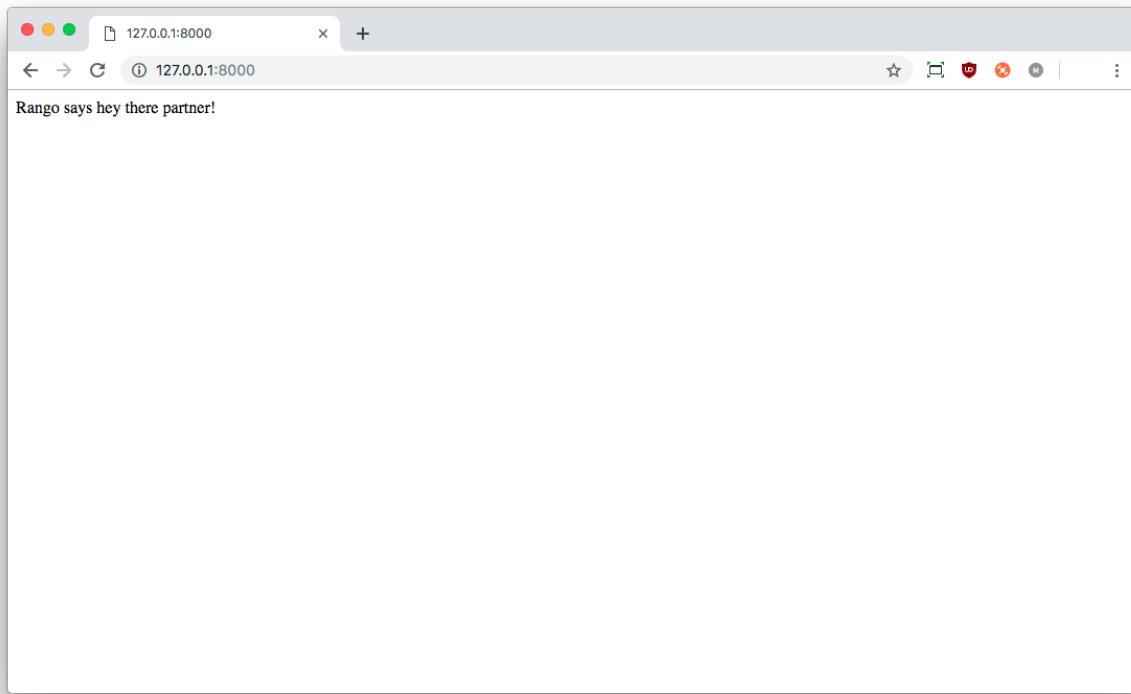
urlpatterns = [
    path('', views.index, name='index'),
]
```

This code imports the relevant Django machinery for URL mappings and the `views` module from `rango`. This allows us to call the function `url` and point to the `index` view for the mapping in `urlpatterns`.

When we talk about URL strings, we assume that the host portion of a given URL has *already been stripped away*. The host portion of a URL denotes the host address or domain name that maps to the webserver, such as `http://127.0.0.1:8000` or `http://www.tangowithdjango.com`. Stripping the host portion away means that the Django machinery needs to only handle the remainder of the URL string. For example, given the URL `http://127.0.0.1:8000/rango/about/`, Django will handle the `/rango/about/` part of the URL string.

The URL mapping we have created above calls Django's `path()` function, where the first parameter is the string to match. As we have used an empty string `''`, Django will then only find a match if there is nothing after `http://127.0.0.1:8000/`. The second parameter tells Django what view to call if the pattern `''` is matched. In this case, `views.index()` will be called. The third and optional parameter is called `name`. It provides a convenient way to reference the view, and by naming our URL mappings we can employ *reverse URL matching*. That is we can reference the URL mapping by name rather than by the URL. [Later, we will explain and show why this is incredibly useful](#). It can save you time and hassle as your application becomes more complex. This will go hand-in-hand with the `app_name` variable we've also placed in the new `urls.py` module.

Now restart the Django development server and visit `http://127.0.0.1:8000/rango/`. If all went well, you should see the text `Rango says hey there partner!`. It should look just like the screenshot shown below.



A screenshot of a web browser displaying our first Django powered webpage. Hello, Rango!

Within each app, you will create several URL mappings. The initial mapping is quite simple, but as we progress through the book we will create more sophisticated and parameterised URL mappings.

It's also important to have a good understanding of how URLs are handled in Django. It may seem a bit confusing right now, but as we progress through the book, we will be creating more and more URL mappings, so you'll soon be a pro. To find out more about them, check out the [official Django documentation on URLs¹⁰](#) for further details and further examples.

If you are using version control, now is a good time to commit the changes you have made to your workspace. Refer to the [chapter providing a crash course on Git](#) if you can't remember the commands and steps involved in doing this.

¹⁰<https://docs.djangoproject.com/en/2.1/topics/http/urls/>

3.6 Basic Workflows

What you've just learnt in this chapter can be succinctly summarised into a list of actions. Here, we provide these lists for the two distinct tasks you have performed. You can use this section for a quick reference if you need to remind yourself about particular actions later on.

Creating a new Django Project

1. To create the project run, `python django-admin.py startproject <name>`, where `<name>` is the name of the project you wish to create.

Creating a new Django App

1. To create a new app, run `$ python manage.py startapp <appname>`, where `<appname>` is the name of the app you wish to create.
2. Tell your Django project about the new app by adding it to the `INSTALLED_APPS` tuple in your project's `settings.py` file.
3. In your project `urls.py` file, add a mapping to the app.
4. In your app's directory, create a `urls.py` file to direct incoming URL strings to views.
5. In your app's `view.py`, create the required views ensuring that they return a `HttpResponse` object.



Exercises

Now that you have got Django and your new app up and running, try out the following exercises to reinforce what you've learnt. Getting to this stage is a significant landmark in working with Django. Creating views and mapping URLs to views is the first step towards developing more complex and usable web applications.

- Revise the procedure and make sure you follow how the URLs are mapped to views.
- Create a new view method called `about` which returns the following `HttpResponse`: 'Rango says here is the about page.'
- Map this view to `/rango/about/`. For this step, you'll only need to edit the `urls.py` of the Rango app. Remember the `/rango/` part is handled by the projects `urls.py`. This new mapping will have a name of `about`.
- Revise the `HttpResponse` in the `index()` view to include a hyperlink (or *anchor*) to the about page. **This is part of the response – meaning that it also lives within the same string that says Rango says hey there partner!.**
- Include a link back to the index page in the about view's response.
- Now that you have started the book, you can follow us on Twitter if you have it – our handle is [@tangowithdjango¹¹](https://twitter.com/tangowithdjango). Let us know how you are getting on!

¹¹<https://twitter.com/tangowithdjango>



Hints

If you're struggling to get the exercises done, the following hints will provide you with some inspiration on how to progress.

- In your `views.py`, create a function called `def about(request)`, and have the function return a `HttpResponse()`. Within this `HttpResponse()`, insert the message that you want to return.
- The expression to use for matching the second view is `'about/'`. This means that in `rango/urls.py`, you would add in a new path mapping to the `about()` view.
- Within the `index()` view, you will want to include an HTML [hyperlink¹²](#) to provide a link to the about page. A `a` tag, complete with an `href` attribute (`About`) will suffice.
- The same will also be added to the `about()` view, although this time it will point to `/rango/`, the homepage – not `/rango/about/`. An example would look like: `Index`.
- If you haven't done so already, now's a good time to head off and complete part one of the official [Django Tutorial¹³](#).



Test your Implementation

If you have completed everything in this chapter up to and including the exercises, you can test your implementation so far. [Follow the guide we provided earlier](#), using the test module `tests_chapter3.py`. Do the tests pass when run against your implementation?

¹²<https://en.wikipedia.org/wiki/Hyperlink>

¹³<https://docs.djangoproject.com/en/2.1/intro/tutorial01/>

4. Templates and Media Files

In this chapter, we'll be introducing the Django template engine, as well as showing you how to serve both *static* files and *media* files. Rather than crafting each page by returning strings as our response, we can use *templates* to provide the skeleton structure of the page from a separate file. From the view that generates the response, we can provide the template with the necessary data to render that page in its entirety. To incorporate JavaScript and CSS (along with images and other media content) we will use the machinery provided by Django to include and dispatch such files to clients, which will in turn allow us to provide added functionality (in the case of JavaScript), or to provide styling to our pages.

4.1 Using Templates

Up until this point, we have only connected a URL mapping to a view. However, the Django framework is based around the *Model-View-Template* architecture. In this section, we will go through the mechanics of how *Templates* work with *Views*. In subsequent chapters, we will put these together with *Models*.

Why templates? The layout from page to page within a website is often the same. Whether you see a common header or footer on a website's pages, the [repetition of page layouts¹](#) aids users with navigation and reinforces a sense of continuity. [Django provides templates²](#) to make it easier for developers to achieve this design goal, as well as separating application logic (code within your views) from presentational concerns (look and feel of your app).

In this chapter, you'll create a basic template that will be used to generate an HTML page. This will then be dispatched via a Django view. In the [chapter concerning databases and models](#), we will take this a step further by using templates in conjunction with models to dispatch dynamically generated data.

¹<http://www.techrepublic.com/blog/web-designer/effective-design-principles-for-web-designers-repetition/>

²<https://docs.djangoproject.com/en/2.1/ref/templates/>



Summary: What is a Template?

In the world of Django, think of a *template* as the scaffolding that is required to build a complete HTML webpage. A template contains the *static parts* of a webpage (that is, parts that never change), complete with special syntax (or *template tags*) which can be overridden and replaced with *dynamic content* that your Django app's views can replace to produce a final HTML response.

Configuring the Templates Directory

To get templates up and running with your Django app, you'll need to create two directories in which template files are stored.

In your Django project's directory (e.g. <workspace>/tango_with_django_project/), create a new directory called `templates`. To clarify, this should be at the same level as your project's `manage.py` script. Within the new `templates` directory, you will then want to create further directory called `rango`. This means that the path `<workspace>/tango_with_django_project/templates/rango/` is the location where we will store templates associated with our `rango` application.



Keep your Templates Organised

It's good practice to separate your templates into subdirectories for each app you have. This is why we've created a `rango` directory within our `templates` directory. If you package your app up to distribute to other developers, it'll be much easier to know which templates belong to which app!

To tell the Django project where templates will be stored, open your project's `settings.py` file. Next, locate the `TEMPLATES` data structure. By default, when you create a new Django project, it will look like the following.

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debug',
                'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',
                'django.contrib.messages.context_processors.messages',
            ],
        },
    },
]
```

What we need to do is tell Django where our templates will be stored by modifying the `DIRS` list, which is set to an empty list by default. Change the dictionary key/value pair to look like the following.

```
'DIRS': ['<workspace>/tango_with_django_project/templates']
```

Note that you are *required to use absolute paths* to locate the `templates` directory. If you are collaborating with team members or working on different computers, then this will become a problem. You'll have different usernames and different drive structures, meaning the paths to the `<workspace>` directory will be different. One solution would be to add the path for each different configuration. For example:

```
'DIRS': [ '/Users/leifos/templates',
           '/Users/maxwelld90/templates',
           '/Users/davidm/templates', ]
```

However, there are several problems with this. First, you have to add in the path for each setting, each time. Second, if you are running the app on different operating systems the backslashes have to be constructed differently.



Don't hard code Paths!

The road to hell is paved with hard-coded paths. Hard-coding paths³ is a software engineering anti-pattern⁴, and will make your project less portable⁵ – meaning that when you run it on another computer, it probably won't work! (Or it will work, just with a lot of effort!)

Dynamic Paths

A better solution is to make use of built-in Python functions to work out the path of your templates directory automatically. This way, an absolute path can be obtained regardless of where you place your Django project's code. This, in turn, means that your project becomes more *portable*.

At the top of your `settings.py` file, there is a variable called `BASE_DIR`. This variable stores the path to the directory in which your project's `settings.py` module is contained. This is obtained by using the special Python `__file__` attribute, which is set to the path of your settings module⁶. Using this as a parameter to `os.path.abspath()` guarantees the *absolute path* to the `settings.py` module. The call to `os.path.dirname()` then provides the reference to the absolute path of the *directory containing* the `settings.py` module. Calling `os.path.dirname()` again removes another directory layer, so that `BASE_DIR` then points to your project directory, or `<workspace>/tango_with_django_project/`. If you are curious, you can see how this works by adding the following lines to your `settings.py` file.

```
print(__file__)
print(os.path.dirname(__file__))
print(os.path.dirname(os.path.dirname(os.path.abspath(__file__))))
```

Having access to the value of `BASE_DIR` makes it easy for you to reference other aspects of your Django project. Using the `BASE_DIR` variable, we can now create a new variable called `TEMPLATE_DIR` that will reference your new `templates` directory. We can make use of the `os.path.join()` function to join up multiple paths, leading

³http://en.wikipedia.org/wiki/Hard_coding

⁴<http://sourcemaking.com/antipatterns>

⁵http://en.wikipedia.org/wiki/Software_portability

⁶<http://stackoverflow.com/a/9271479>

to a variable definition like the example below. Make sure you put this underneath the definition of `BASE_DIR`!

Delete those Lines!

If you included the three `print()` statements above to see what's going on, make sure you remove them once you understand. Don't just leave them lying there. They will clutter your settings module, and clutter the output of the Django development server!

```
TEMPLATE_DIR = os.path.join(BASE_DIR, 'templates')
```

Here, we make use of the `os.path.join()` function to join together (or *concatenate*) the value of the `BASE_DIR` variable, and the string '`templates`'. Upon completion, this concatenation yields `<workspace>/tango_with_django_project/templates/`. From here, we can then use our new `TEMPLATE_DIR` variable to replace the hard-coded path we defined earlier in `TEMPLATES`. Update the `DIRS` key/value pairing to look like the following code snippet.

```
'DIRS': [TEMPLATE_DIR, ]
```

Why `TEMPLATE_DIR`?

You've created a new variable called `TEMPLATE_DIR` at the top of your `settings.py` file because it's easier to access should you ever need to change it. For more complex Django projects, the `DIRS` list allows you to specify more than one template directory to draw templates from. For this book however, one location is sufficient to get everything working.

Concatenating Paths

When concatenating system paths together, always use `os.path.join()`.

Using this built-in function ensures that the correct path separators are used. On a UNIX operating system (or derivative of), forward slashes (/) would be used to separate directories, whereas a Windows operating system would use backward slashes (\). If you manually append slashes to paths, you may end up with path errors when attempting to run your code on a different operating system, thus reducing your project's portability.

Adding a Template

With your template directory and path now set up, create a file called `index.html` and place it in the `templates/rango/` directory. Within this new file, add the following HTML markup and Django template code.

```
<!DOCTYPE html>
<html>

    <head>
        <title>Rango</title>
    </head>

    <body>
        <h1>Rango says...</h1>
        <div>
            hey there partner! <br />
            <strong>{{ boldmessage }}</strong><br />
        </div>
        <div>
            <a href="/rango/about/">About</a><br />
        </div>
    </body>

</html>
```

From this HTML code, it should be clear that a simple HTML page is going to be generated that greets a user with a *hello world* message. You might also notice some non-HTML in the form of `{{ boldmessage }}`. This is a *Django template variable*. We can set values to these variables so they are replaced with whatever we want when the template is rendered. We'll get to that in a moment.

To use this template, we need to reconfigure the `index()` view that we created earlier. Instead of dispatching a simple response, we will change the view to dispatch our template.

In `rango/views.py`, check to see if the following `import` statement exists at the top of the file. Django should have added it for you when you created the Rango app. If it is not present, add it, but you should have kept it from earlier.

```
from django.shortcuts import render
```

You can then update the `index()` view function as follows. Check out the inline commentary to see what each line does.

```
def index(request):
    # Construct a dictionary to pass to the template engine as its context.
    # Note the key boldmessage matches to {{ boldmessage }} in the template!
    context_dict = {'boldmessage': 'Crunchy, creamy, cookie, candy, cupcake!'}

    # Return a rendered response to send to the client.
    # We make use of the shortcut function to make our lives easier.
    # Note that the first parameter is the template we wish to use.
    return render(request, 'rango/index.html', context=context_dict)
```

First, we construct a dictionary of key/value pairs that we want to use within the template. Then, we call the `render()` helper function. This function takes as input the user's request, the template filename, and the context dictionary. The `render()` function will take this data and mash it together with the template to produce a complete HTML page that is returned with a `HttpResponse`. This response is then returned and dispatched to the user's web browser.



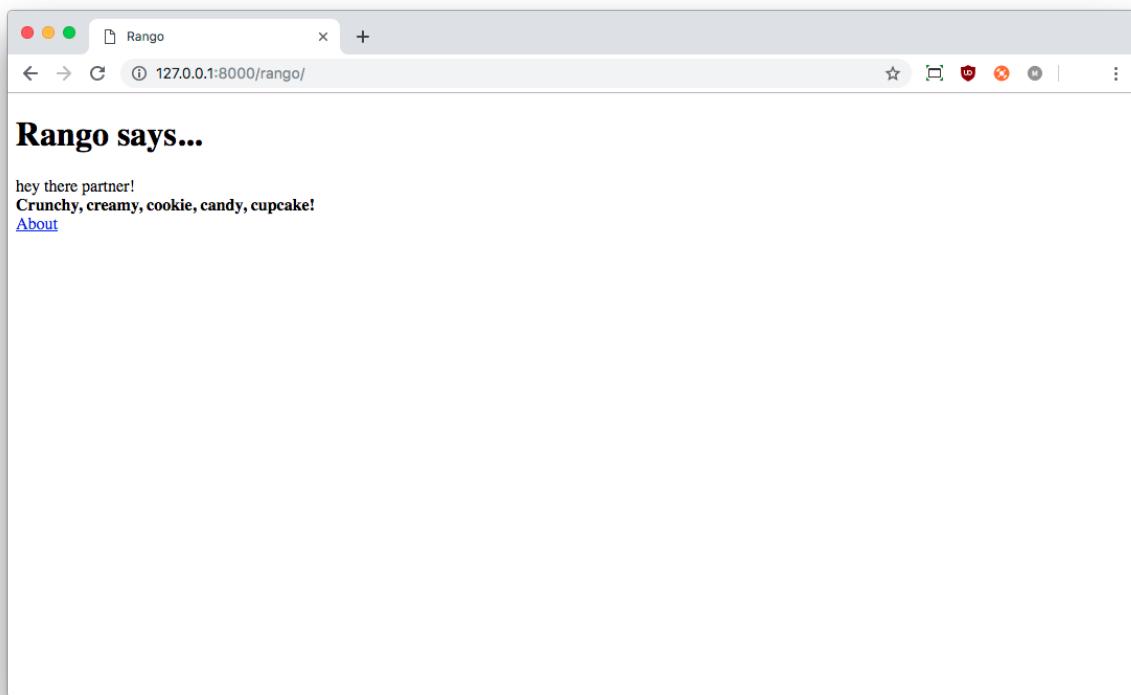
What is the Template Context?

When a template file is loaded with the Django templating system, a *template context* is created. In simple terms, a template context is a Python dictionary that maps template variable names with Python variables. In the template we created above, we included a template variable name called `boldmessage`. In our updated `index(request)` view example, the string `Crunchy, creamy, cookie, candy, cupcake!` is mapped to template variable `boldmessage`. The string `Crunchy, creamy, cookie, candy, cupcake!` therefore replaces *any* instance of `{{ boldmessage }}` within the template.

Now that you have updated the view to employ the use of your template, start the Django development server and visit `http://127.0.0.1:8000/rango/`. You should see your simple HTML template rendered in your browser – and it should look just like the [example screenshot shown below](#).

If you don't, read the error message presented to see what the problem is, and then double-check all the changes that you have made. One of the most common issues people have with templates is that the path is set incorrectly in `settings.py`. Sometimes it's worth adding a `print` statement to `settings.py` to report the `BASE_DIR` and `TEMPLATE_DIR` to make sure everything is correct.

This example demonstrates how to use templates within your views. However, we have only touched on a fraction of the functionality provided by the Django templating engine. We will use templates in more sophisticated ways as you progress through this book. In the meantime, you can find out more about [templates from the official Django documentation⁷](#).



What you should see when your first template is working correctly. Note the bold text – `crunchy`, `creamy`, `cookie`, `candy`, `cupcake`! – which originates from the view, but is rendered in the template.

⁷<https://docs.djangoproject.com/en/2.1/ref/templates/>

4.2 Serving Static Media Files

While you've got templates working, your Rango app is admittedly looking a bit plain right now – there's no styling or imagery. We can add references to other files in our HTML template such as *Cascading Style Sheets (CSS)*⁸, *JavaScript*⁹ and images to improve the presentation. These are called *static files*, because they are not generated dynamically by a web server; they are simply sent as is to a client's web browser. This section shows you how to set Django up to serve static files, and shows you how to include an image within your simple template.

Configuring the Static Media Directory

To start, you will need to set up a directory in which static media files are stored. In your project directory (e.g. <workspace>/tango_with_django_project/), create a new directory called `static` and a new directory called `images` inside `static`. Check that the new `static` directory is at the same level as the `templates` directory you created earlier in this chapter.

Next, place an image inside the `images` directory. As shown in below, we chose a picture of the chameleon Rango¹⁰ – a fitting mascot, if ever there was one.

⁸http://en.wikipedia.org/wiki/Cascading_Style_Sheets

⁹<https://en.wikipedia.org/wiki/JavaScript>

¹⁰<http://www.imdb.com/title/tt1192628/>



Rango the chameleon within our static/images media directory.

Just like the templates directory we created earlier, we need to tell Django about our new static directory. To do this, we once again need to edit our project's settings.py module. Within this file, we need to add a new variable pointing to our static directory, and a data structure that Django can parse to work out where our new directory is.

First of all, create a variable called STATIC_DIR at the top of settings.py, preferably underneath BASE_DIR and TEMPLATES_DIR to keep your paths all in the same place. STATIC_DIR should make use of the same os.path.join trick – but point to static this time around, just as shown below.

```
STATIC_DIR = os.path.join(BASE_DIR, 'static')
```

This results in the path <workspace>/tango_with_django_project/static/. We then need to create a new data structure called STATICFILES_DIRS. This is essentially a list of paths with which Django will expect to find static files that can be served. By default, this list does not exist – **check** it doesn't before you create it. If you define it twice, you can start to confuse Django, and yourself.

For this book, we're only going to be using a single location to store our project's static files – the path defined in STATIC_DIR. As such, we can simply set up the list STATICFILES_DIRS with the following.

```
STATICFILES_DIRS = [STATIC_DIR, ]
```



Keep settings.py Tidy!

It's in your best interests to keep your `settings.py` module tidy and in good order. Don't just put things in random places; keep it organised. Keep your `DIRS` variables at the top of the module so they are easy to find, and place `STATICFILES_DIRS` in the portion of the module responsible for static media (close to the bottom). When you come back to edit the file later, it'll be easier for you or other collaborators to find the necessary variables.

Finally, check that the `STATIC_URL` variable is defined within your `settings.py` module. If it has not been defined, then add it in, as shown below. Note that this variable by default appears close to the end of the module, so you may have to scroll down to the bottom of `settings.py` to find it (if it's already there).

```
STATIC_URL = '/static/'
```

With everything required now entered, what does it all mean? Put simply, the first two variables `STATIC_DIR` and `STATICFILES_DIRS` refers to the locations on your computer where static files are stored. The final variable `STATIC_URL` then allows us to specify the URL with which static files can be accessed when we run our Django development server. For example, with `STATIC_URL` set to `/static/`, we would be able to access static content at `http://127.0.0.1:8000/static/`. *Think of the first two variables as server-side locations, with the third variable as the location with which clients can access static content.*



Test your Configuration

As a small exercise, test to see if everything is working correctly. Try and view the `rango.jpg` image in your browser when the Django development server is running. If your `STATIC_URL` is set to `/static/` and `rango.jpg` can be found at `images/rango.jpg`, what is the complete URL that you would enter into your web browser's window to access this resource?

Try to figure this out before you move on! It'll help you understand how to interpret static URLs. The answer is coming up if you are struggling to figure it out.



Don't Forget the Slashes!

When setting `STATIC_URL`, check that you end the URL you specify with a forward slash (e.g. `/static/`, not `/static`). As per the [official Django documentation¹¹](#), not doing so can open you up to a world of pain. The extra slash at the end ensures that the root of the URL (e.g. `/static/`) is separated from the static content you want to serve (e.g. `images/rango.jpg`).



Serving Static Content

While using the Django development server to serve your static media files is fine for a development environment, it's highly unsuitable for a production environment. The [official Django documentation on deployment¹²](#) provides further information about deploying static files in a production environment. We'll look at this issue in more detail however when we [deploy Rango](#).

If you haven't managed to figure out where the image should be accessible from, point your web browser to `http://127.0.0.1:8000/static/images/rango.jpg`.

Static Media Files and Templates

Now that you have your Django project set up to handle static files, you can now make use of these files within your templates to improve their appearance and add additional functionality.

¹¹https://docs.djangoproject.com/en/2.1/ref/settings/#std:setting-STATIC_URL

¹²<https://docs.djangoproject.com/en/2.1/howto/static-files/deployment/>

To demonstrate how to include static files, open up the `index.html` templates you created earlier, located in the `<workspace>/templates/rango/` directory. Modify the HTML source code as follows. The two lines that we add are shown with an HTML comment next to them for easy identification.

```
<!DOCTYPE html>

{% load staticfiles %} <!-- New line -->

<html>
    <head>
        <title>Rango</title>
    </head>

    <body>
        <h1>Rango says...</h1>

        <div>
            hey there partner! <br />
            <strong>{{ boldmessage }}</strong><br />
        </div>

        <div>
            <a href="/rango/about/">About</a><br />
             <!-- New line -->
        </div>
    </body>

</html>
```

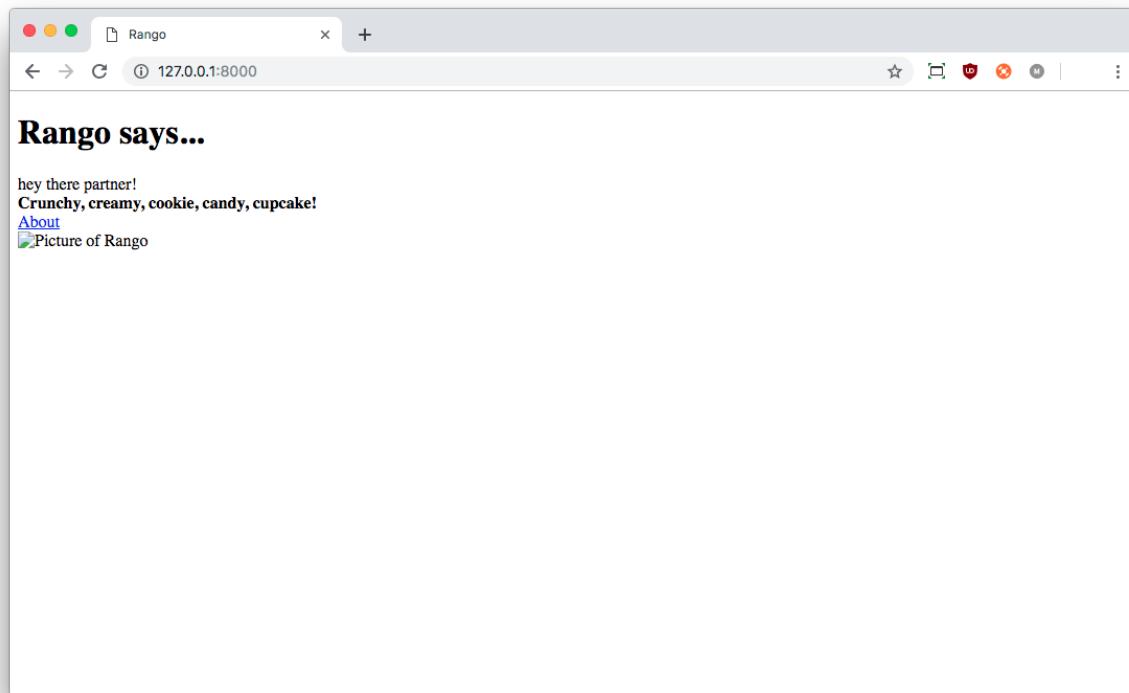
The first new line added (`{% load staticfiles %}`) informs Django's template engine that we will be using static files within the template. This then enables us to access the media in the static directories via the use of the `static` template tag¹³. This indicates to Django that we wish to show the image located in the static media directory called `images/rango.jpg`. Template tags are denoted by curly brackets (e.g. `{% %}`), and calling `static` will combine the URL specified in `STATIC_URL` with `images/rango.jpg` to yield `/static/images/rango.jpg`. The HTML generated by the Django template engine would be:

¹³<https://docs.djangoproject.com/en/2.1/ref/templates/builtins/>

```

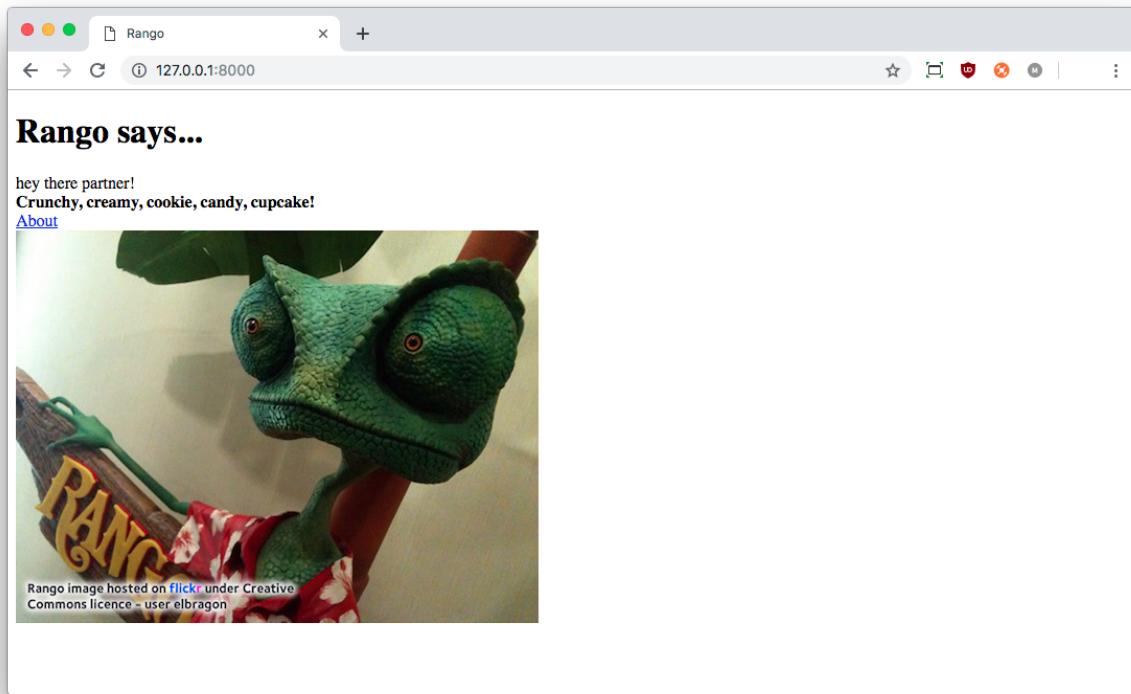
```

If for some reason the image cannot be loaded, it is always a good idea to specify an alternative text tagline. This is what the `alt` attribute provides inside the `img` tag. You can see what happens in the [image below](#).



The image of Rango couldn't be found, and is instead replaced with a placeholder containing the text from the `img alt` attribute.

With these minor changes in place, start the Django development server once more and navigate to `http://127.0.0.1:8000/rango`. If everything has been done correctly, you will see a webpage that looks similar to the [screenshot shown below](#).



Our first Rango template, complete with a picture of Rango the chameleon.



Always put <!DOCTYPE> First!

When creating the HTML templates, always ensure that the [DOCTYPE declaration](#)¹⁴ appears on the **first line**. If you put the `{% load staticfiles %}` template command first, then whitespace will be added to the rendered template before the DOCTYPE declaration. This whitespace will lead to your HTML markup [failing validation](#)¹⁵.

¹⁴http://www.w3schools.com/tags/tag_doctype.asp

¹⁵<https://validator.w3.org/>



Loading other Static Files

The `{% static %}` template tag can be used whenever you wish to reference static files within a template. The code example below demonstrates how you could include JavaScript, CSS and images into your templates with correct HTML markup.

```
<!DOCTYPE html>
{% load staticfiles %}

<html>

    <head>
        <title>Rango</title>
        <!-- CSS -->
        <link rel="stylesheet" href="{% static "css/base.css" %}" />
        <!-- JavaScript -->
        <script src="{% static "js/jquery.js" %}"></script>
    </head>

    <body>
        <!-- Image -->
        
    </body>

</html>
```

Don't update the `index.html` template here – this is merely a demonstration to show you how the `{% static %}` template function works. You'll be adding CSS and JavaScript later on in this tutorial.

Static files you reference will obviously need to be present within your static directory. If a requested file is not present or you have referenced it incorrectly, the console output provided by Django's development server will show a [HTTP 404 error¹⁶](#). Try referencing a non-existent file and see what happens. Looking at the output snippet below, notice how the last entry's HTTP status code is 404.

```
[24/Mar/2019 17:05:54] "GET /rango/ HTTP/1.1" 200 366
[24/Mar/2019 17:05:55] "GET /static/images/rango.jpg HTTP/1.1" 200 0
[24/Mar/2019 17:05:55] "GET /static/images/not-here.jpg HTTP/1.1" 404 0
```

For further information about including static media you can read through the official [Django documentation on working with static files in templates¹⁷](#).

¹⁶https://en.wikipedia.org/wiki/HTTP_404
www.tangowithdjango.com

Licenced to University of Glasgow Library

4.3 Serving Media

Static media files can be considered files that don't change and are essential to your application. However, often you will have to store *media files* which are dynamic. These files can be uploaded by your users or administrators, and so they may change. As an example, a media file would be a user's profile picture. If you run an e-commerce website, a series of media files would be used as images for the different products that your online shop has.

To serve media files successfully, we need to update the Django project's settings. This section details what you need to add – [but we won't be fully testing it out until later](#) where we implement the functionality for users to upload profile pictures.



Serving Media Files

Like serving static content, Django provides the ability to serve media files in your development environment. This allows you to test and make sure everything is working. The methods that Django uses to serve this content are highly unsuitable for a production environment, so you should be looking to host your app's media files by some other means (through a production-grade web server like Apache). The [deployment chapter](#) will discuss this in more detail.

Modifying `settings.py`

First, open your Django project's `settings.py` module. In here, we'll be adding a couple more things. Like static files, media files are uploaded to a specified directory on your filesystem. We need to tell Django where to store these files.

At the top of your `settings.py` module, locate your existing `BASE_DIR`, `TEMPLATE_DIR` and `STATIC_DIR` variables – they should be close to the top. Underneath, add a further variable, `MEDIA_DIR`.

¹⁷<https://docs.djangoproject.com/en/2.1/howto/static-files/#staticfiles-in-templates>

```
MEDIA_DIR = os.path.join(BASE_DIR, 'media')
```

This line instructs Django that media files will be uploaded to your Django project's root, plus '/media' – or <workspace>/tango_with_django_project/media/. As we previously mentioned, keeping these path variables at the top of your `settings.py` module makes it easy to change paths later on if necessary.

Now find a blank spot in `settings.py`, and add two more variables. The variables `MEDIA_ROOT` and `MEDIA_URL` will be picked up and used by Django to set up media file hosting¹⁸.

```
MEDIA_ROOT = MEDIA_DIR  
MEDIA_URL = '/media/'
```



Once again, don't Forget the Forward Slashes!

Like the `STATIC_URL` variable, ensure that `MEDIA_URL` ends with a forward slash. Make sure you use something like `/media/`, not `/media`. The extra slash at the end ensures that the root of the URL (e.g. `/media/`) is separated from the filenames of content uploaded by your app's users.

The two variables tell Django where to look in your filesystem for media files (`MEDIA_ROOT`) that have been uploaded and stored, and what URL to serve them from (`MEDIA_URL`). With the configuration we defined above, a user uploading the file `sample.pdf` will, for example, be then made available on your Django development server through the URL `http://localhost:8000/media/sample.pdf`.

When we come to working with templates [later on in this book](#), it'll be handy for us to obtain a reference to the `MEDIA_URL` path when we need to reference uploaded content. Django provides a [*template context processor*](#)¹⁹ that'll make it easy for us to do. While we don't strictly need this set up now, it's a good time to add it in.

To do this, find the `TEMPLATES` list that resides within your project's `settings.py` module. The list contains a dictionary; look for the `context_processors` list within the nested dictionary. Within the `context_processors` list, add a new string to include

¹⁸<https://docs.djangoproject.com/en/2.1/howto/static-files/#serving-files-uploaded-by-a-user-during-development>

¹⁹<https://docs.djangoproject.com/en/2.1/ref/templates/api/#django-template-context-processors-media>

an additional context processor: 'django.template.context_processors.media'. Your context_processors list should then look like the example below.

```
'context_processors': [
    'django.template.context_processors.debug',
    'django.template.context_processors.request',
    'django.contrib.auth.context_processors.auth',
    'django.contrib.messages.context_processors.messages',
    'django.template.context_processors.media', # Check/add this line!
],
```

Tweaking your URLs

The final step for setting up the serving of media in a development environment is to tell Django to serve static content from `MEDIA_URL`. This can be achieved by opening your **project's** `urls.py` module. Remember, your **project's** `urls.py` module is the one that lives within the `tango_with_django_project/tango_with_django_project` directory – **not** the `tango_with_django_project/rango/urls.py` module!

In the `urls.py` module, begin by adding the following `import` statements at the top. If you find that one of these lines already exists, you don't need to enter it again.

```
from django.conf import settings
from django.conf.urls.static import static
```

Once you have the imports, you need to modify the `urlpatterns` list underneath. Modify it by appending a call to the `static()` function that you just imported, complete with the settings telling the function where the files are stored on your filesystem (`MEDIA_URL`), and what URL they should be served from (`MEDIA_URL`).

```
urlpatterns = [
    ...
    ...
] + static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
```

Once these changes have been made, you should be able to serve content from the `media` directory of your project from the `/media/` URL.



Create the media Directory

Did you create the `media` directory within the `tango_with_django_project` directory? It should be at the same level as the `static` directory and the `manage.py` module.

4.4 Basic Workflow

With the chapter complete, you should now know how to set up and create templates, use templates within your views, setup and use the Django development server to serve static media files, *and* include images within your templates. We've covered quite a lot!

Creating a template and integrating it within a Django view is a key concept for you to understand. It takes several steps but will become second nature to you after a few attempts.

1. First, create the template you wish to use and save it within the `templates` directory you specified in your project's `settings.py` module. You may wish to use Django template variables (e.g. `{{ variable_name }}`) or [template tags](#)²⁰ within your template. You'll be able to replace these with whatever you like within the corresponding view.
2. Find or create a new view within an application's `views.py` file.
3. Add your view specific logic (if you have any) to the view. For example, this may involve extracting data from a database and storing it within a list.
4. Within the view, construct a dictionary object which you can pass to the template engine as part of the [template's context](#).
5. Make use of the `render()` helper function to generate the rendered response. Ensure you reference the request, then the template file, followed by the context dictionary.
6. Finally, map the view to a URL by modifying your project's `urls.py` file (or the application-specific `urls.py` file if you have one). This step is only required if you're creating a new view, or you are using an existing view that hasn't yet been mapped!

²⁰<https://docs.djangoproject.com/en/2.1/ref/templates/builtins/>

The steps involved in getting a static media file onto one of your pages are part of another important process that you should be familiar with. Check out the steps below on how to do this.

1. Take the static media file you wish to use and place it within your project's `static` directory. This directory is defined in `STATICFILES_DIRS` – one of the variables that you set up in `settings.py`.
2. Add a reference to the static media file to a template. For example, an image would be inserted into an HTML page through the use of the `` tag.
3. Remember to use the `{% load staticfiles %}` and `{% static "<filename>" %}` commands within the template to access the static files. Replace `<filename>` with the path to the image or resource you wish to reference. **Whenever you wish to refer to a static file, use the `static` template tag!**

The steps for serving media files are similar to those for serving static media.

1. Place a file within your project's `media` directory. The `media` directory is specified by your project's `MEDIA_ROOT` variable.
2. Link to the media file in a template through the use of the `{{ MEDIA_URL }}` context variable. For example, referencing an uploaded image `cat.jpg` would have an `` tag like ``.



Exercises

Give the following exercises a go to reinforce what you've learnt from this chapter.

- Convert the about page to use a template too. Use a template called `about.html` for this purpose. Base the contents of this file on `index.html`. In the new template's `<h1>` element, keep Rango says... – but on the line underneath, have the text 'here is the about page.'.
- Remember to update the `about()` view in `views.py`! Do you think you need a context dictionary for this view?
- Your new `about.html` must include a link back to the index page. If you copied your existing `index.html` template, a small change is required to achieve this.
- Within the new `about.html` template, add a picture stored within your project's static files. Please reuse the `rango.jpg` image you used in the `index()` view. Make sure you keep the same alt text as before!
- On the about page, include a line that says This tutorial has been put together by <your-name>. If you copied over from `index.html`, replacing `{{ boldmessage }}` would be the perfect place for this.
- In your Django project directory, create a new directory called `media` (if you have not done so already). Download a JPEG image of a cat, and save it to the `media` directory as `cat.jpg`.
- In your `about.html` template, add in an `` tag to display the picture of the cat to ensure that your media is being served correctly. **Keep the static image of Rango in your index page** so that your about page has working examples of both static and media files. The cat image should have alternative text of Picture of a Cat. **This means you should have an image of both Rango (from static) and a cat (from media) in your rendered about page.** You don't need to touch the `about()` view – only `about.html` needs to be modified for this final task.



Static and Media Files

Remember, **static files, as the name implies, do not change**. These files form the core components of your website. **Media files are user-defined; and as such, they may change often!**

An example of a static file could be a stylesheet file (CSS), which determines the appearance of your app's webpages. An example of a media file could be a user profile image, which is uploaded by the user when they create an account on your app.



Test your Implementation

If you have completed everything in this chapter up to and including the exercises, you can test your implementation so far. [Follow the guide we provided earlier](#), using the test module `tests_chapter4.py`. How does your implementation stack up against our tests?

5. Models and Databases

Typically, web applications require a backend to store the dynamic content that appears on the app's webpages. For Rango, we need to store pages and categories that are created, along with other details. The most convenient way to do this is by employing the services of a relational database. These will likely use the *Structured Query Language (SQL)* to allow you to query the data store.

However, Django provides a convenient way in which to access data stored in databases by using an *Object Relational Mapper (ORM)*¹. In essence, data stored within a database table is encapsulated via Django *models*. A model is a Python object that describes the database table's data. Instead of directly working on the database via SQL, Django provides methods that let you manipulate the data via the corresponding Python model object. Any commands that you issue to the ORM are automatically converted to the corresponding SQL statement on your behalf.

This chapter walks you through the basics of data management with Django and its ORM. You'll find it's incredibly easy to add, modify and delete data within your app's underlying database, and see how straightforward it is to get data from the database to the web browsers of your users.

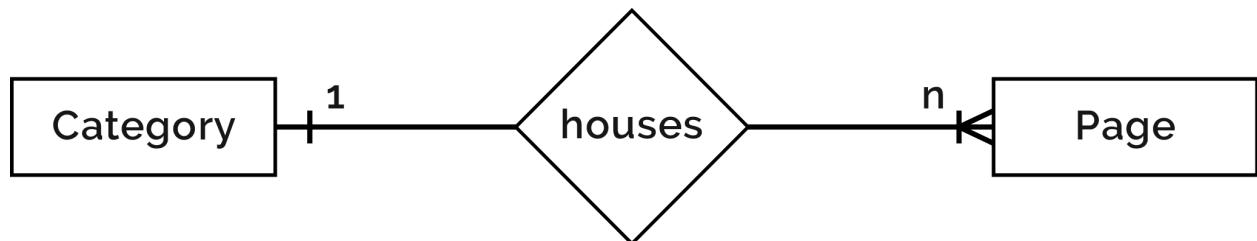
5.1 Rango's Requirements

Before we get started, let's go over the data requirements for the Rango app that we are developing. Full requirements for the application are [provided in detail earlier on](#). Let's look at the database requirements again here.

- Rango is essentially a *web page directory* – it is a website that houses links to other, external websites.

¹https://en.wikipedia.org/wiki/Object-relational_mapping

- There are several different *webpage categories* with each category housing none, one or many links. We assumed in the overview chapter that this is a one-to-many relationship. Check out the Entity Relationship diagram below.
- A category has a name, several visits, and several likes.
- A page belongs to a particular category, has a title, a URL, and several views.



The Entity Relationship Diagram of Rango's two main entities.

5.2 Telling Django about Your Database

Before we can create any models, we need to set up our database to work with Django. In Django, a `DATABASES` variable is automatically created in your `settings.py` module when you set up a new project. Unless you changed it, it should look like the following example.

```

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
    }
}
  
```

We can pretty much leave this as-is for our Rango app. You can see a default database that is powered by a lightweight database engine, [SQLite²](#) (see the `ENGINE` option). The `NAME` entry for this database is the path to the database file, which is by default `<workspace>/tango_with_django_project/db.sqlite3` – or, in other words, the `db.sqlite3` file that lives in the root of your Django project.

²<https://www.sqlite.org/>



Don't git push your Database!

If you are using Git, you might be tempted to add and commit the database file. However, we don't think that this is a particularly good idea. What if you are working on your app with other people? It would be highly likely that they will change the database as they test features. Different versions of database files (which ultimately do not matter in development) **will** cause endless conflicts.

Instead, add `db.sqlite3` to your `.gitignore` file so that it won't be added when you `git commit` and `git push`. You can also do this for other files like `*.pyc` and machine specific files. For more information on how to set your `.gitignore` file up, you can refer to our [Git familiarisation chapter](#) in the appendices.



Using other Database Engines

The Django database framework has been created to cater for a variety of different database backends, such as [PostgreSQL³](#), [MySQL⁴](#) and [Microsoft's SQL Server⁵](#). For other database engines, other keys like `USER`, `PASSWORD`, `HOST` and `PORT` exist for you to configure the database with Django.

While we don't cover how to use other database engines in this book, there are guides online which show you how to do this. A good starting point is the [official Django documentation⁶](#).

Note that SQLite is sufficient for demonstrating the functionality of the Django ORM. When you find your app has become viral and has accumulated thousands of users, you may want to consider [switching the database backend to something more robust⁷](#).

5.3 Creating Models

With your database configured in `settings.py`, let's create the two initial data models for the Rango application. Models for a Django app are stored in the

³<http://www.postgresql.org/>

⁴<https://www.mysql.com/>

⁵https://en.wikipedia.org/wiki/Microsoft_SQL_Server

⁶<https://docs.djangoproject.com/en/2.1/ref/databases/#storage-engines>

⁷<http://www.sqlite.org/whentouse.html>

respective `models.py` module. This means that for Rango, models are stored within `<workspace>/tango_with_django_project/rango/models.py`.

For the models themselves, we will create two classes – one class representing each model. Both must `inherit8` from the `Model` base class, `django.db.models.Model`. The two Python classes will be the definitions for models representing *categories* and *pages*. Define the `Category` and `Page` model as follows.

```
class Category(models.Model):
    name = models.CharField(max_length=128, unique=True)

    def __str__(self):
        return self.name

class Page(models.Model):
    category = models.ForeignKey(Category, on_delete=models.CASCADE)
    title = models.CharField(max_length=128)
    url = models.URLField()
    views = models.IntegerField(default=0)

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



Check import Statements

At the top of the `models.py` module, you should see `from django.db import models`. If you don't see it, add it in.

When you define a model, you need to specify the list of fields and their associated types, along with any required or optional parameters. By default, all models have an auto-increment integer field called `id` which is automatically assigned and acts a primary key.

Django provides a [comprehensive series of built-in field types⁹](#). Some of the most commonly used are detailed below.

- `CharField`, a field for storing character data (e.g. strings). Specify `max_length` to provide a maximum number of characters that a `CharField` field can store.

⁸[https://en.wikipedia.org/wiki/Inheritance_\(object-oriented_programming\)](https://en.wikipedia.org/wiki/Inheritance_(object-oriented_programming))

⁹<https://docs.djangoproject.com/en/2.1/ref/models/fields/#model-field-types>

- `URLField`, much like a `CharField`, but designed for storing resource URLs. You may also specify a `max_length` parameter.
- `IntegerField`, which stores integers.
- `DateField`, which stores a Python `datetime.date` object.



Other Field Types

Check out the [Django documentation on model fields¹⁰](#) for a full listing of the Django field types you can use, along with details on the required and optional parameters that each has.

For each field, you can specify the `unique` attribute. If set to `True`, the given field's value must be unique throughout the underlying database table that is mapped to the associated model. For example, take a look at our `Category` model defined above. The field `name` has been set to `unique`, meaning that every category name must be unique. This means that you can use the field as a primary key.

You can also specify additional attributes for each field, such as stating a default value with the syntax `default='value'`, and whether the value for a field can be blank (or `NULL11`) (`null=True`) or not (`null=False`).

Django provides three types of fields for forging relationships between models in your database. These are:

- `ForeignKey`, a field type that allows us to create a [one-to-many relationship¹²](#);
- `OneToOneField`, a field type that allows us to define a strict [one-to-one relationship¹³](#); and
- `ManyToManyField`, a field type which allows us to define a [many-to-many relationship¹⁴](#).

From our model examples above, the field `category` in model `Page` is a `ForeignKey`. This allows us to create a one-to-many relationship with model/table `Category`,

¹⁰<https://docs.djangoproject.com/en/2.1/ref/models/fields/#model-field-types>

¹¹https://en.wikipedia.org/wiki/Nullable_type

¹²[https://en.wikipedia.org/wiki/One-to-many_\(data_model\)](https://en.wikipedia.org/wiki/One-to-many_(data_model))

¹³[https://en.wikipedia.org/wiki/One-to-one_\(data_model\)](https://en.wikipedia.org/wiki/One-to-one_(data_model))

¹⁴[https://en.wikipedia.org/wiki/Many-to-many_\(data_model\)](https://en.wikipedia.org/wiki/Many-to-many_(data_model))

which is specified as an argument to the field's constructor. When specifying the foreign key, we also need to include instructions to Django on how to handle the situation when the category that the page belongs to is deleted. `CASCADE` instructs Django to delete the pages associated with the category when the category is deleted. However, there are other settings which will provide Django with other instructions on how to handle this situation. See the [Django documentation on Foreign Keys¹⁵](#) for more details.

Finally, it is good practice to implement the `__str__()` method. Without this method implemented it will show as `<Category: Category object>` if you were to `print()` the object (perhaps in the Django shell, [as we discuss later in this chapter](#)). This isn't very useful when debugging or accessing the object. How do you know what category is being shown? When including `__str__()` as defined above, you will see `<Category: Python>` (as an example) for the Python category. It is also helpful when we go to use the admin interface later because Django will display the string representation of the object, derived from `__str__()`.



Always Implement `__str__()` in your Classes

Implementing the `__str__()` method in your classes will make debugging so much easier – and also permit you to take advantage of other built-in features of Django (such as the admin interface). If you've used a programming language like Java, `__str__()` is the Python equivalent of the `toString()` method!

5.4 Creating and Migrating the Database

With our models defined in `models.py`, we can now let Django work its magic and create the tables in the underlying database. Django provides what is called a [*migration tool¹⁶*](#) to help us set up and update the database to reflect any changes to your models. For example, if you were to add a new field, then you can use the migration tools to update the database.

¹⁵<https://docs.djangoproject.com/en/2.1/ref/models/fields/#django.db.models.ForeignKey>

¹⁶https://en.wikipedia.org/wiki/Data_migration

Setting up

First of all, the database must be *initialised*. This means that creating the database and all the associated tables so that data can then be stored within it/them. To do this, you must open a terminal or Command Prompt, and navigate to your project's root directory – where the `manage.py` module is stored. Run the following command, *bearing in mind that the output may vary slightly from what you see below*.

```
$ 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 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 sessions.0001_initial... OK
```

All apps installed in your Django project (check `INSTALLED_APPS` in `settings.py`) will be called to update their database representations when this command is issued. After this command is issued, you should then see a `db.sqlite3` file in your Django project's root.

Next, create a superuser to manage the database. Run the following command.

```
$ python manage.py createsuperuser
```

The superuser account will be used to access the Django admin interface, used later on in this chapter. Enter a username for the account, e-mail address and

provide a password when prompted. Once completed, the script should finish successfully. Make sure you take note of the username and password for your superuser account.

Creating and Updating Models/Tables

Whenever you make changes to your app's models, you need to *register* the changes via the `makemigrations` command in `manage.py`. Specifying the `rango` app as our target, we then issue the following command from our Django project's root directory.

```
$ python manage.py makemigrations rango
```

```
Migrations for 'rango':  
    rango/migrations/0001_initial.py  
        - Create model Category  
        - Create model Page
```

Upon the completion of this command, check the `rango/migrations` directory to see that a Python script has been created. It's called `0001_initial.py`, which contains all the necessary details to create your database schema for that particular migration.



Checking the Underlying SQL

If you want to check out the underlying SQL that the Django ORM issues to the database engine for a given migration, you can issue the following command.

```
$ python manage.py sqlmigrate rango 0001
```

In this example, `rango` is the name of your app, and `0001` is the migration you wish to view the SQL code for. Doing this allows you to get a better understanding of what exactly is going on at the database layer, such as what tables are created. You will find for complex database schemas including a many-to-many relationship that additional tables are created for you.

After you have created migrations for your app, you need to commit them to the database. Do so by once again issuing the `migrate` command.

```
$ python manage.py migrate

Operations to perform:
  Apply all migrations: admin, auth, contenttypes, rango, sessions
Running migrations:
  Applying rango.0001_initial... OK
```

This output confirms that the database tables have been created in your database, and you are then ready to start using the new models and tables.

However, you may have noticed that our `Category` model is currently lacking some fields that [were specified in Rango's requirements](#). **Don't worry about this, as these will be added in later, allowing you to work through the migration process once more.**

5.5 Django Models and the Shell

Before we turn our attention to demonstrating the Django admin interface, it's worth noting that you can interact with Django models directly from the Django shell – a very useful tool for debugging purposes. We'll demonstrate how to create a `Category` instance using this method.

To access the shell, we need to call `manage.py` from within your Django project's root directory once more. Run the following command.

```
$ python manage.py shell
```

This will start an instance of the Python interpreter and load in your project's settings for you. You can then interact with the models, with the following terminal session demonstrating this functionality. Check out the inline commentary that we added to see what each command achieves.

```
# Import the Category model from the Rango application
>>> from rango.models import Category

# Show all the current categories
>>> print(Category.objects.all())
# Since no categories have been defined we get an empty QuerySet object.
<QuerySet []>

# Create a new category object, and save it to the database.
>>> c = Category(name='Test')
>>> c.save()

# Now list all the category objects stored once more.
>>> print(Category.objects.all())
# You'll now see a 'Test' category.
<QuerySet [<Category: Test>]

# Quit the Django shell.
>>> quit()
```

In the example, we first import the model that we want to manipulate. We then print out all the existing categories. As our underlying Category table is empty, an empty list is returned. Then we create and save a Category, before printing out all the categories again. This second print then shows the new Category just added. Note the name `Test` appears in the second print – this is the output of the `__str__()`, and neatly demonstrates why including these methods is important!



Complete the Official Tutorial

The example above is only a very basic taster on database related activities you can perform in the Django shell. If you have not done so already, it's now a good time to complete [part two of the official Django Tutorial](#) to learn more about interacting with models¹⁷. In addition, have a look at the [official Django documentation](#) on the list of available commands¹⁸ for working with models.

¹⁷<https://docs.djangoproject.com/en/2.1/intro/tutorial02/>

¹⁸<https://docs.djangoproject.com/en/2.1/ref/django-admin/#available-commands>

5.6 Configuring the Admin Interface

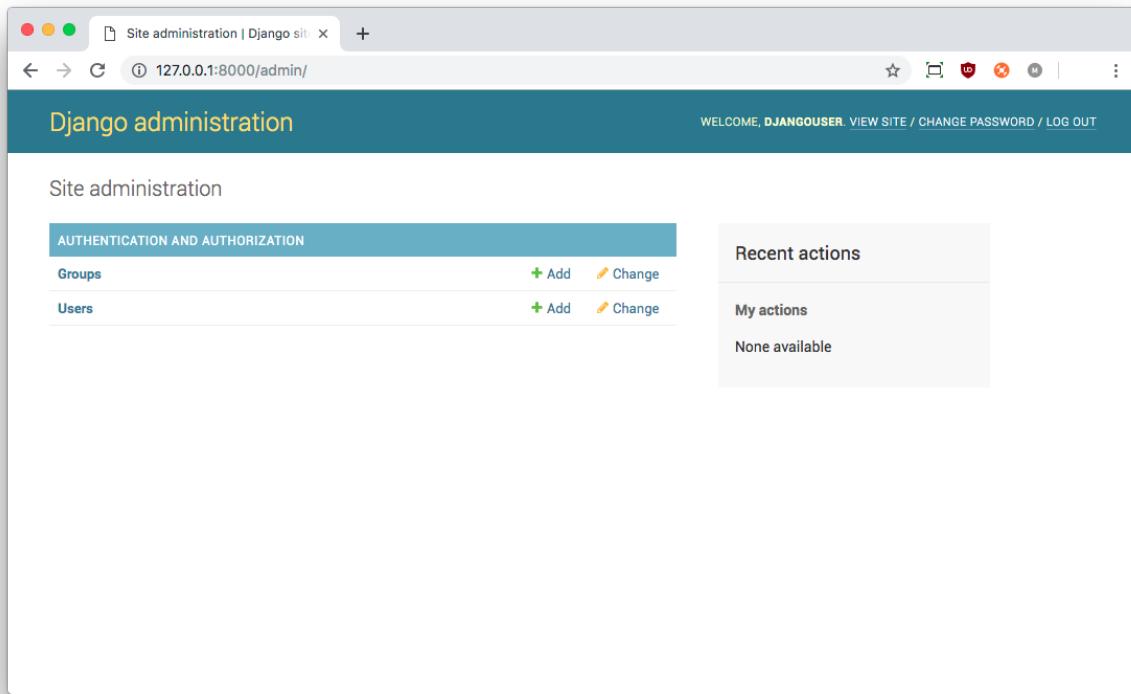
One of the standout features of Django is the built-in, web-based administrative (or *admin*) interface that allows you to browse, edit and delete data represented as model instances (from the corresponding database tables). In this section, we'll be setting the admin interface up so you can see the two Rango models you have created so far.

Setting everything up is relatively straightforward. In your project's `settings.py` module, you will notice that one of the preinstalled apps (within the `INSTALLED_APPS` list) is `django.contrib.admin`. Furthermore, there is a `urlpattern` that matches `admin/` within your project's `urls.py` module.

By default, things are pretty much ready to go. Start the Django development server in the usual way with the following command.

```
$ python manage.py runserver
```

Navigate your web browser to `http://127.0.0.1:8000/admin/`. You are then presented with a login prompt. Login using the credentials you created previously with the `$ python manage.py createsuperuser` command. You are then presented with an interface looking [similar to the one shown below](#).



The Django admin interface, sans Rango models.

While this looks good, we are missing the `Category` and `Page` models that were defined for the Rango app. To include these models, we need to let Django know that we want to include them.

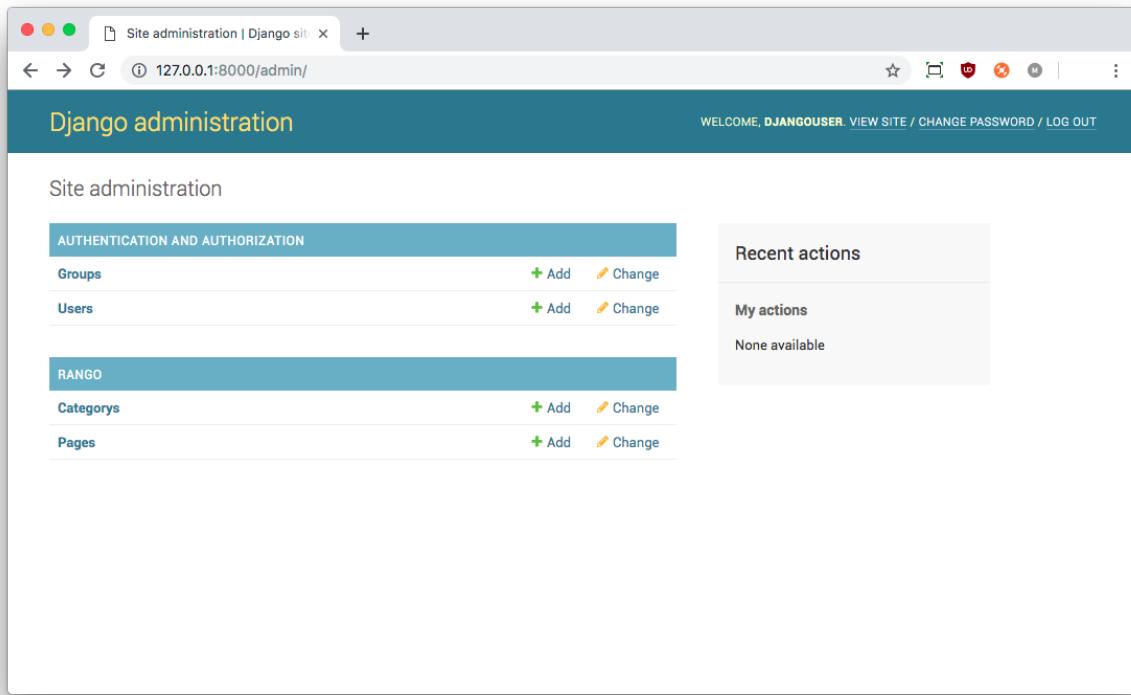
To do this, open the file `rango/admin.py`. With an `include` statement already present, modify the module so that you register each class you want to include. The example below registers both the `Category` and `Page` class to the admin interface.

```
from django.contrib import admin
from rango.models import Category, Page

admin.site.register(Category)
admin.site.register(Page)
```

Adding further classes which may be created in the future is as simple as adding another call to the `admin.site.register()` method, making sure that the model is imported at the top of the module.

With these changes saved, restart the Django development server and revisit the admin interface at `http://127.0.0.1:8000/admin/`. You will now see the Category and Page models, as shown below.



The Django admin interface, complete with Rango models.

Try clicking the Categories link within the Rango section. From here, you should see the Test category that we created earlier via the Django shell.



Experiment with the Admin Interface

As you move forward with Rango's development, you'll be using the admin interface extensively to verify data is stored correctly. Experiment with it, and see how it all works. The interface is self-explanatory and straightforward to understand.

Delete the Test category that was previously created. We'll be populating the database shortly with example data. You can delete the Test category from the admin interface by clicking the checkbox beside it, and selecting Delete selected categories from the dropdown menu at the top of the page. Confirm your intentions by clicking the big red button that appears!



User Management

The Django admin interface is also your port of call for user management through the Authentication and Authorisation section. Here, you can create, modify and delete user accounts, and vary privilege levels. [More on this later.](#)



Plural vs. Singular Spellings

Note the typo within the admin interface (Categorys, not Categories). This typo can be fixed by adding a nested `Meta` class into your model definitions with the `verbose_name_plural` attribute. Check out a modified version of the `Category` model below for an example, and [Django's official documentation on models¹⁹](#) for more information about what can be stored within the `Meta` class.

```
class Category(models.Model):
    name = models.CharField(max_length=128, unique=True)

    class Meta:
        verbose_name_plural = 'Categories'

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



Expanding `admin.py`

It should be noted that the example `admin.py` module for your Rango app is the most simple, functional example available. However, you can customise the Admin interface in several ways. Check out the [official Django documentation on the admin interface²⁰](#) for more information if you're interested. We'll be working towards manipulating the `admin.py` module later on in the tutorial.

¹⁹<https://docs.djangoproject.com/en/2.1/topics/db/models/#meta-options>

²⁰<https://docs.djangoproject.com/en/2.1/ref/contrib/admin/>

5.7 Creating a Population Script

Entering test data into your database tends to be a hassle. Many developers will add in some bogus test data by randomly hitting keys, like wTFzmN00bz7. Rather than do this, it is better to write a script to automatically populate the database with **realistic and credible data**. This is because when you go to demo or test your app, you'll need to be able to see some credible examples in the database. If you're working in a team, an automated script will mean each collaborator can simply run that script to initialise the database on their computer with the same sample data as you. It's therefore good practice to create what we call a *population script*.

To create a population script, create a new file called `populate_rango.py`. Create this file in `<workspace>/tango_with_django_project/`, or in other words, your Django project's root directory. When the file has been created, add the following code.

```
1 import os
2 os.environ.setdefault('DJANGO_SETTINGS_MODULE',
3                       'tango_with_django_project.settings')
4
5 import django
6 django.setup()
7 from rango.models import Category, Page
8
9 def populate():
10     # First, we will create lists of dictionaries containing the pages
11     # we want to add into each category.
12     # Then we will create a dictionary of dictionaries for our categories.
13     # This might seem a little bit confusing, but it allows us to iterate
14     # through each data structure, and add the data to our models.
15
16     python_pages = [
17         {'title': 'Official Python Tutorial',
18          'url':'http://docs.python.org/3/tutorial/'},
19         {'title':'How to Think like a Computer Scientist',
20          'url':'http://www.greenteapress.com/thinkpython/'},
21         {'title':'Learn Python in 10 Minutes',
22          'url':'http://www.korokithakis.net/tutorials/python/'} ]
23
24     django_pages = [
```

```
25      {'title':'Official Django Tutorial',
26       'url':'https://docs.djangoproject.com/en/2.1/intro/tutorial01/'},
27      {'title':'Django Rocks',
28       'url':'http://www.djangorocks.com/'},
29      {'title':'How to Tango with Django',
30       'url':'http://www.tangowithdjango.com/'} ]
31
32 other_pages = [
33     {'title':'Bottle',
34      'url':'http://bottlepy.org/docs/dev/'},
35     {'title':'Flask',
36      'url':'http://flask.pocoo.org'} ]
37
38 cats = {'Python': {'pages': python_pages},
39          'Django': {'pages': django_pages},
40          'Other Frameworks': {'pages': other_pages} }
41
42 # If you want to add more categories or pages,
43 # add them to the dictionaries above.
44
45 # The code below goes through the cats dictionary, then adds each category,
46 # and then adds all the associated pages for that category.
47 for cat, cat_data in cats.items():
48     c = add_cat(cat)
49     for p in cat_data['pages']:
50         add_page(c, p['title'], p['url'])
51
52 # Print out the categories we have added.
53 for c in Category.objects.all():
54     for p in Page.objects.filter(category=c):
55         print(f'- {c}: {p}')
56
57 def add_page(cat, title, url, views=0):
58     p = Page.objects.get_or_create(category=cat, title=title)[0]
59     p.url=url
60     p.views=views
61     p.save()
62     return p
63
64 def add_cat(name):
65     c = Category.objects.get_or_create(name=name)[0]
66     c.save()
67     return c
```

```
68  
69 # Start execution here!  
70 if __name__ == '__main__':  
71     print('Starting Rango population script...')  
72     populate()
```



Understand this Code!

To reiterate, don't simply copy, paste and leave. Add the code to your new module, and then step through line by line to work out what is going on. It'll help with your understanding.

We've provided explanations below to help you learn from our code!

You should also note that when you see line numbers alongside the code. We've included these to make copying and pasting a laborious chore – why not just type it out yourself and think about each line instead?

While this looks like a lot of code, what is going on is essentially a series of function calls to two small functions, `add_page()` and `add_cat()`, both defined towards the end of the module. Reading through the code, we find that execution starts at the *bottom* of the module – look at lines 71 and 72. This is because, above this point, we define functions; these are not executed *unless* we call them. When the interpreter hits `if __name__ == '__main__'`²¹, we call and begin execution of the `populate()` function.



What does `__name__ == '__main__'` Represent?

The `__name__ == '__main__'` trick is a useful one that allows a Python module to act as either a reusable module or a standalone Python script. Consider a reusable module as one that can be imported into other modules (e.g. through an `import` statement), while a standalone Python script would be executed from a terminal/Command Prompt by entering `python module.py`.

Code within a conditional `if __name__ == '__main__'` statement will therefore only be executed when the module is run as a standalone Python script. Importing the module will not run this code; any classes or functions will however be fully accessible to you.

²¹<http://stackoverflow.com/a/419185>



Importing Models

When importing Django models, make sure you have imported your project's settings by importing `django` and setting the environment variable `DJANGO_SETTINGS_MODULE` to be your project's setting file, as demonstrated in lines 1 to 6 above. You then call `django.setup()` to import your Django project's settings.

If you don't perform this crucial step, you'll **get an exception when attempting to import your models. This is because the necessary Django infrastructure has not yet been initialised.** This is why we import `Category` and `Page` *after* the settings have been loaded on the seventh line.

The `for` loop occupying lines 47-50 is responsible for calling the `add_cat()` and `add_page()` functions repeatedly. These functions are in turn responsible for the creation of new categories and pages. `populate()` keeps tabs on categories that are created. As an example, a reference to a new category is stored in local variable `c` – check line 48 above. This is stored because a `Page` requires a `Category` reference. After `add_cat()` and `add_page()` are called in `populate()`, the function concludes by looping through all-new `Category` and associated `Page` objects, displaying their names on the terminal.



Creating Model Instances

We make use of the convenience `get_or_create()` method for creating model instances in the population script above. As we don't want to create duplicates of the same entry, we can use `get_or_create()` to check if the entry exists in the database for us. If it doesn't exist, the method creates it. If it does, then a reference to the specific model instance is returned.

This helper method can remove a lot of repetitive code for us. Rather than doing this laborious check ourselves, we can make use of code that does exactly this for us.

The `get_or_create()` method returns a tuple of `(object, created)`. The first element `object` is a reference to the model instance that the `get_or_create()` method creates if the database entry was not found. The entry is created using the parameters you pass to the method – just like `category`, `title`, `url` and `views` in the example above. If the entry already exists in the database, the method simply returns the model instance corresponding to the entry. `created` is a boolean value; `True` is returned if `get_or_create()` had to create a model instance.

This explanation means that the `[0]` after `get_or_create()` returns the object reference only. Like most other programming language data structures, Python tuples use **zero-based numbering**²².

You can check out the [official Django documentation](#)²³ for more information on the handy `get_or_create()` method. We'll be using this extensively throughout the rest of the tutorial.

When saved, you can then run your new populations script by changing the present working directory in a terminal to the Django project's root. It's then a simple case of executing the command `$ python populate_rango.py`. You should then see an output to the command similar to that shown below. The order in which categories are added may vary; don't worry about that – as long as they are all listed!

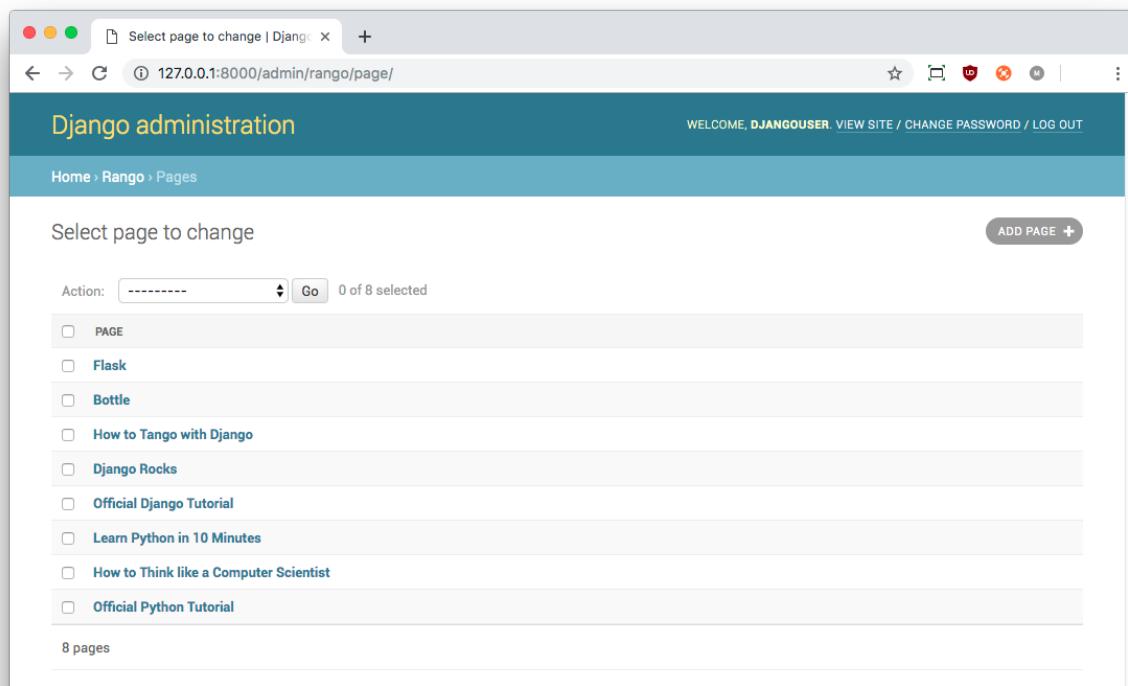
²²http://en.wikipedia.org/wiki/Zero-based_numbering

²³<https://docs.djangoproject.com/en/2.1/ref/models/querysets/#get-or-create>

```
$ python populate_rango.py
```

```
Starting Rango population script...
- Python: Official Python Tutorial
- Python: How to Think like a Computer Scientist
- Python: Learn Python in 10 Minutes
- Django: Official Django Tutorial
- Django: Django Rocks
- Django: How to Tango with Django
- Other Frameworks: Bottle
- Other Frameworks: Flask
```

Next, verify that the population script worked as it should have and populated the database. To do this, restart the Django development server, and navigate to the admin interface (at <http://127.0.0.1:8000/admin/>). Once there, check that you have some new categories and pages. Do you see all the pages if you click Pages, like in the figure shown below?



The Django admin interface, showing the Page model populated with the new population script. Success!

While creating a population script may take time initially, you will save yourself heaps of time in the long run. When deploying your app elsewhere, running the population script after setting everything up means you can start demonstrating your app straight away. You'll also find it very handy when it comes to [unit testing your code](#).

5.8 Workflow: Model Setup

Now that we've covered the core principles of dealing with Django's ORM, now is a good time to summarise the processes involved in setting everything up. We've split the core tasks into separate sections for you. Check this section out when you need to quickly refresh your mind of the different steps.

Setting up your Database

With a new Django project, you should first [tell Django about the database you intend to use](#) (i.e. configure DATABASES in settings.py). You can also register any models in the admin.py module of your app to then make them accessible via the web-based admin interface.

Adding a Model

The workflow for adding models can be broken down into five steps.

1. First, create your new model(s) in your Django application's models.py file.
2. Update admin.py to include and register your new model(s) if you want to make them accessible to the admin interface.
3. Perform the migration \$ python manage.py makemigrations <app_name>.
4. Apply the changes \$ python manage.py migrate. This will create the necessary infrastructure (tables) within the database for your new model(s).
5. Create/edit your population script for your new model(s).

There will be times when you will have to delete your database. Sometimes it's easier to just start afresh. Perhaps you might end up caught in a loop when trying to make further migrations, and something goes wrong.

When you encounter the need to refresh the database, you can go through the following steps. Note that for this tutorial, you are using an SQLite database – Django does support a [variety of other database engines²⁴](#).

1. If you're running it, stop your Django development server.
2. For an SQLite database, delete the `db.sqlite3` file in your Django project's directory. It'll be in the same directory as the `manage.py` file.
3. If you have changed your app's models, you'll want to run the `$ python manage.py makemigrations <app_name>` command, replacing `<app_name>` with the name of your Django app (i.e. `rango`). Skip this if your models have not changed.
4. Run the `$ python manage.py migrate` command to create a new database file (if you are running SQLite), and migrate database tables to the database.
5. Create a new admin account with the `$ python manage.py createsuperuser` command.
6. Finally, run your population script again to insert credible test data into your new database.

²⁴<https://docs.djangoproject.com/en/2.1/ref/databases/>



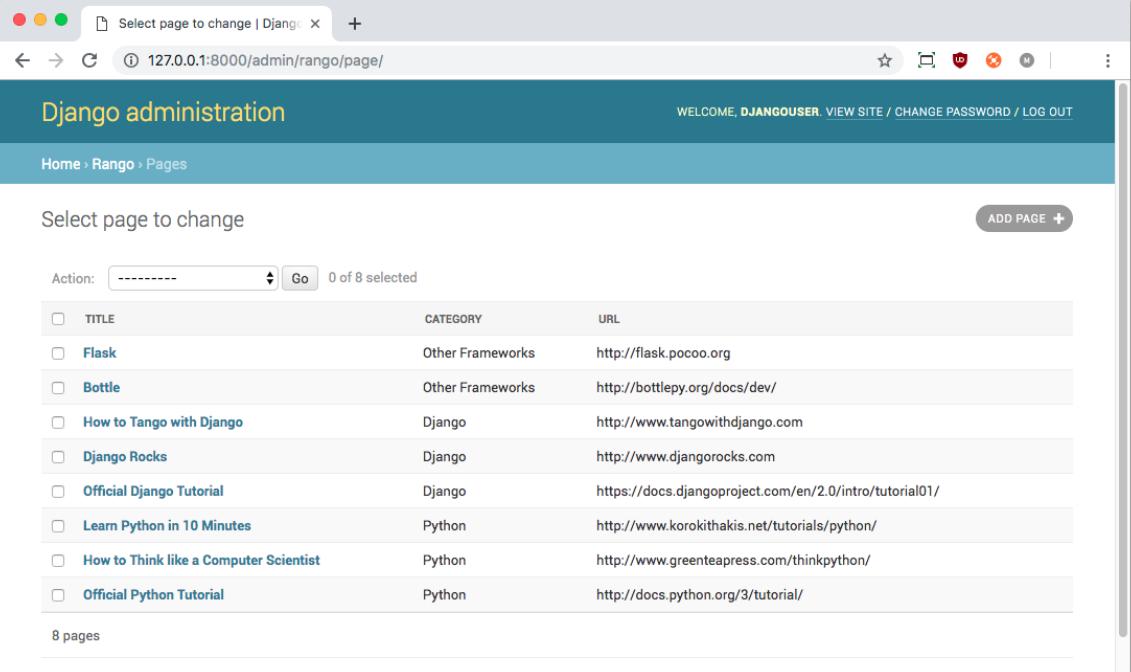
Exercises

Now that you've completed this chapter, try out these exercises to reinforce and practice what you have learnt. **Once again, note that the following chapters will have expected you to have completed these exercises!**

- Update the Category model to include the additional attributes views and likes where the default values for each are both zero (0).
- As you have changed your models, make the migrations for your Rango app. After making migrations, commit the changes to the database.
- Next update your population script so that the Python category has 128 views and 64 likes, the Django category has 64 views and 32 likes, and the Other Frameworks category has 32 views and 16 likes.
- Delete and recreate your database, populating it with your updated population script.
- Complete parts [two²⁵](#) and [seven²⁶](#) of the official Django tutorial. The two sections will reinforce what you've learnt on handling databases in Django, and will also provide you with additional techniques to customising the Django admin interface. This knowledge will help you complete the final exercise below.
- Customise the admin interface. Change it in such a way so that when you view the Page model, the table displays the category, the title of the page, and the url – just like in the screenshot shown below. **Make sure the three fields appear in that order, too.** Complete the official Django tutorial or look at the tip below to complete this particular exercise.

²⁵<https://docs.djangoproject.com/en/2.1/intro/tutorial02/>

²⁶<https://docs.djangoproject.com/en/2.1/intro/tutorial07/>



The screenshot shows the Django administration interface for the 'Pages' model. The page title is 'Django administration' and the sub-page title is 'Home > Rango > Pages'. The main content is a table titled 'Select page to change' with the following data:

Action:	TITLE	CATEGORY	URL
<input type="checkbox"/>	Flask	Other Frameworks	http://flask.pocoo.org
<input type="checkbox"/>	Bottle	Other Frameworks	http://bottlepy.org/docs/dev/
<input type="checkbox"/>	How to Tango with Django	Django	http://www.tangowithdjango.com
<input type="checkbox"/>	Django Rocks	Django	http://www.djangorocks.com
<input type="checkbox"/>	Official Django Tutorial	Django	https://docs.djangoproject.com/en/2.0/intro/tutorial01/
<input type="checkbox"/>	Learn Python in 10 Minutes	Python	http://www.korokithakis.net/tutorials/python/
<input type="checkbox"/>	How to Think like a Computer Scientist	Python	http://www.greenteapress.com/thinkpython/
<input type="checkbox"/>	Official Python Tutorial	Python	http://docs.python.org/3/tutorial/

At the bottom left of the table, it says '8 pages'.

The updated admin interface Page view, complete with columns for category and URL.



Exercise Hints

If you require some help or inspiration to complete these exercises done, here are some hints.

- Modify the `Category` model by adding two `IntegerField`s: `views` and `likes`. Both should have a parameter `default` set to 0.
- In your population script, you can then modify the `add_cat()` function to take the values of the `views` and `likes`.
 - You'll need to add two parameters to the definition of `add_cat()` so that `views` and `likes` values can be passed to the function, as well as a `name` for the category. Give them default values of 0 (i.e. `views=0`).
 - You can then use these parameters to set the `views` and `likes` fields within the new `Category` model instance you create within the `add_cat()` function. The model instance is assigned to variable `c` in the population script, as defined earlier in this chapter. As an example, you can access the `likes` field using the notation `c.likes`. Don't forget to `save()` the instance!
 - You then need to update the `cats` dictionary in the `populate()` function of your population script. Look at the dictionary. Each **key/value pairing**²⁷ represents the *name* of the category as the key, and an additional dictionary containing additional information relating to the category as the *value*. You'll want to modify this second, nested dictionary to include `views` and `likes` for each category.
 - The final step involves you modifying how you call the `add_cat()` function. You now have three parameters to pass (`name`, `views` and `likes`); your code currently provides only the `name`. You need to add the additional two parameters to the function call. If you aren't sure how the `for` loop works over dictionaries, check out [this online Python tutorial](#)²⁸. From here, you can figure out how to access the `views` and `likes` values from your dictionary (hint: `cat_data`).
- After your population script has been updated, you can move on to customising the admin interface. You will need to edit `rango/admin.py` and create a `PageAdmin` class that inherits from `admin.ModelAdmin`.
 - Within your new `PageAdmin` class, add `list_display = ('title', 'category', 'url')`.
 - Finally, register the `PageAdmin` class with Django's admin interface. You should modify the line `admin.site.register(Page)`. Change it to

²⁷https://www.tutorialspoint.com/python/python_dictionary.htm



Test your Implementation

Like in the previous chapter, we've implemented a series of unit tests to allow you to check your implementation up until this point. [Follow the guide we provided earlier](#), using the test module `tests_chapter5.py`. How does your implementation stack up against our tests?

²⁸https://www.tutorialspoint.com/python/python_dictionary.htm

6. Models, Templates and Views

Now that we have the models set up and populated the database with some sample data, we can now start connecting the models, views and templates to serve up dynamic content. In this chapter, we will go through the process of showing categories on the main page, and then create dedicated category pages which will show the associated list of links.

6.1 Workflow: A Data-Driven Page

To do this, there are five main steps that you must undertake to create a data-driven webpage in Django.

1. In the `views.py` module, import the models you wish to use.
2. In the view function, query the model to get the data you want to present.
3. Also in the view, pass the results from your model into the template's context.
4. Create/modify the template so that it displays the data from the context.
5. If you have not done so already, map a URL to your view.

These steps highlight how we need to work within Django's framework to bind models, views and templates together.

6.2 Showing Categories on Rango's Homepage

One of the requirements regarding the main page was to show the top five categories present within your app's database. To fulfil this requirement, we will go through each of the above steps.

Importing Required Models

First, we need to complete step one. Open `rango/views.py` and at the top of the file, after the other imports, import the `Category` model from Rango's `models.py` file.

```
# Import the Category model
from rango.models import Category
```

Modifying the Index View

Here we will complete steps two and step three, where we need to modify the view `index()` function. Remember that the `index()` function is responsible for the main page view. Modify `index()` as follows:

```
def index(request):
    # Query the database for a list of ALL categories currently stored.
    # Order the categories by the number of likes in descending order.
    # Retrieve the top 5 only -- or all if less than 5.
    # Place the list in our context_dict dictionary (with our boldmessage!)
    # that will be passed to the template engine.
    category_list = Category.objects.order_by('-likes')[:5]

    context_dict = {}
    context_dict['boldmessage'] = 'Crunchy, creamy, cookie, candy, cupcake!'
    context_dict['categories'] = category_list

    # Render the response and send it back!
    return render(request, 'rango/index.html', context=context_dict)
```

Here, the expression `Category.objects.order_by('-likes')[:5]` queries the `Category` model to retrieve the top five categories. You can see that it uses the `order_by()` method to sort by the number of `likes` in descending order. The `-` in `-likes` denotes that we would like them in *descending* order (if we removed the `-` then the results would be returned in *ascending* order). Since a list of `Category` objects will be returned, we used Python's [list operators¹](#) to take the first five objects from the list `([:5])` to return a subset of `Category` objects.

¹https://www.quackit.com/python/reference/python_3_list_methods.cfm

With the query complete, we passed a reference to the list (stored as variable `category_list`) to the dictionary, `context_dict`. This dictionary is then passed as part of the context for the template engine in the `render()` call. Note that above, we still include our `boldmessage` in the `context_dict` – this is still required for the existing template to work! This means our context dictionary now contains two key/value pairs: `boldmessage`, representing our `Crunchy`, `creamy`, `cookie`, `candy`, `cupcake!` message, and `categories`, representing our top five categories that have been extracted from the database.



Warning

For this to work, you will have had to complete the exercises in the previous chapter where you need to add the field `likes` to the `Category` model. Like we have said already, we assume that you complete all exercises as you progress through this book.

Modifying the Index Template

With the view updated, we can complete the fourth step and update the template `rango/index.html`, located within your project's templates directory. Change the HTML and Django template code so that it looks like the example shown below. Note that the major changes start at line 15.

```
1  <!DOCTYPE html>
2
3  {% load staticfiles %}
4
5  <html>
6
7      <head>
8          <title>Rango</title>
9      </head>
10
11     <body>
12         <h1>Rango says...</h1>
13         <div>
14             hey there partner! <br/>
15             <strong>{{ boldmessage }}</strong>
```

```
16      </div>
17
18      <div>
19          {% if categories %}
20              <ul>
21                  {% for category in categories %}
22                      <li>{{ category.name }}</li>
23                  {% endfor %}
24              </ul>
25          {% else %}
26              <strong>There are no categories present.</strong>
27          {% endif %}
28      </div>
29
30      <div>
31          <a href="/rango/about/">About Rango</a><br />
32          
34      </div>
35  </body>
36
37 </html>
```

Here, we make use of Django's template language to present the data using `if` and `for` control statements. Within the `<body>` of the page, we test to see if `categories` – the name of the context variable containing our list of (a maximum of) five categories – actually contains any categories (`{% if categories %}`).

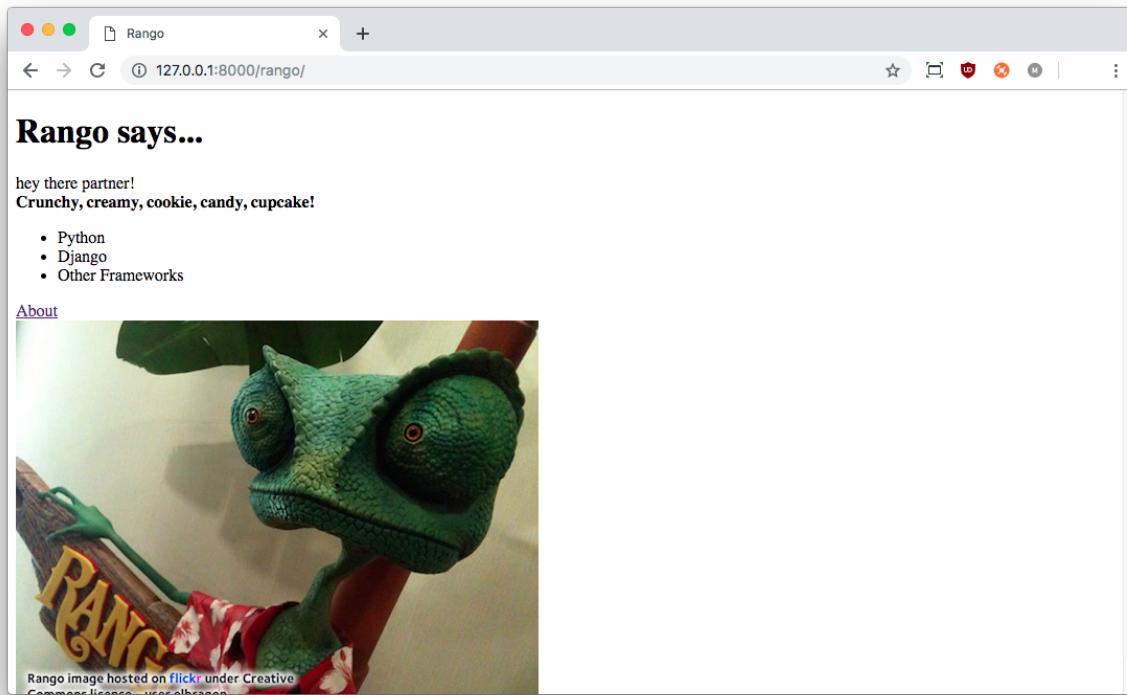
If so, we proceed to construct an unordered HTML list (within the `` tags). The `for` loop (`{% for category in categories %}`) then iterates through the list of results, and outputs each category's name (`{{ category.name }}`) within an `` element to denote that it is a *list element*.

If no categories exist, a message is displayed instead indicating that no categories are present – 'There are no categories present.'. As this is wrapped in a `` element, it will appear in **bold**.

As the example also demonstrates Django's template language, all template commands are enclosed within the tags `{%` and `%}`, while variables whose values are to be placed on the page are referenced within `{{` and `}}` brackets. *Everything* within these

tags and brackets is interpreted by the Django templating engine before sending a completed response back to the client.

Now, save the template file and head over to your web browser. Refresh Rango's homepage at `http://127.0.0.1:8000/rango/`, and you should then see a list of categories underneath the page title and your bold message, just like in the figure below. Well done, this is your first *data-driven webpage!*



The Rango homepage – now dynamically generated – showing a list of categories.

6.3 Creating a Details Page

According to the [specifications for Rango](#), we also need to show a list of pages that are associated with each category. To accomplish this, there are several different challenges that we need to overcome. First, a new view must be created. *This new view will have to be parameterised*. We also need to create URL patterns and URL strings that encode category names.

URL Design and Mapping

Let's start by considering the URL problem. One way we could handle this problem is to use the unique ID for each category within the URL. For example, we could create URLs like `/rango/category/1/` or `/rango/category/2/`, where the numbers correspond to the categories with unique IDs 1 and 2 respectively. However, it is not possible to infer what the category is about just from the ID.

Instead, we could use the category name as part of the URL. For example, we can imagine that the URL `/rango/category/python/` would lead us to a list of pages related to Python. This is a simple, readable and meaningful URL. If we go with this approach, we'll also have to handle categories that have multiple words, like `Other Frameworks`, for example. **Putting spaces in URLs is generally regarded as bad practice** (as we describe below). Spaces need to be *percent-encoded*². For example, the percent-encoded string for `Other Frameworks` would read `Other%20Frameworks`. This looks messy and confusing!



Clean your URLs

Designing clean and readable URLs is an important aspect of web design.

See [Wikipedia's article on Clean URLs³](#) for more details.

To solve this problem, we will make use of the so-called `slugify()` function provided by Django.

Update the Category Table with a Slug Field

To make readable URLs, we need to include a `slug`⁴ field in the `Category` model. First, we need to import the function `slugify` from Django that will replace whitespace with hyphens, circumnavigating the percent-encoded problem. For example, "how do i create a slug in django" turns into "how-do-i-create-a-slug-in-django".

²https://www.w3schools.com/tags/ref_urlencode.asp

³http://en.wikipedia.org/wiki/Clean_URL

⁴<https://prettylinks.com/2018/03/url-slugs/>



Unsafe URLs

While you can use spaces in URLs, it is considered to be unsafe to use them.

Check out the [Internet Engineering Task Force Memo on URLs⁵](http://www.ietf.org/rfc/rfc1738.txt) to read more.

Next, we need to override the `save()` method of the `Category` model. This overridden function will call the `slugify()` function and update the new `slug` field with it. Note that every time the category name changes, the slug will also change – the `save()` method is always called when creating or updating an instance of a Django model. Update your `Category` model as shown below.

```
class Category(models.Model):
    name = models.CharField(max_length=128, unique=True)
    views = models.IntegerField(default=0)
    likes = models.IntegerField(default=0)
    slug = models.SlugField()

    def save(self, *args, **kwargs):
        self.slug = slugify(self.name)
        super(Category, self).save(*args, **kwargs)

    class Meta:
        verbose_name_plural = 'categories'

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

Don't forget to add in the following `import` at the top of the module, either.

```
from django.template.defaultfilters import slugify
```

The overriden `save()` method is relatively straightforward to understand. When called, the `slug` field is set by using the output of the `slugify()` function as the new field's value. Once set, the overriden `save()` method then calls the parent (or `super`) `save()` method defined in the base `django.db.models.Model` class. It is this call that performs the necessary logic to take your changes and save the said changes to the correct database table.

⁵<http://www.ietf.org/rfc/rfc1738.txt>

Now that the model has been updated, the changes must now be propagated to the database. However, since data already exists within the database from previous chapters, we need to consider the implications of the change. Essentially, for all the existing category names, we want to turn them into slugs (which is performed when the record is initially saved). When we update the models via the migration tool, it will add the `slug` field and provide the option of populating the field with a default value. Of course, we want a specific value for each entry – so we will first need to perform the migration, and then re-run the population script. This is because the population script will explicitly call the `save()` method on each entry, triggering the `save()` as implemented above, and thus update the slug accordingly for each entry.

To perform the migration, issue the following commands (as detailed in the [Models and Databases Workflow](#)).

```
$ python manage.py makemigrations rango
```

Since we did not provide a default value for the slug and we already have existing data in the model, the `makemigrations` command will give you two options. Select the option to provide a default (option 1), and enter an empty string – denoted by two quote marks (i.e. '').

You should then see output that confirms that the migrations have been created.

```
You are trying to add a non-nullable field 'slug' to category without a default;
      we can't do that (the database needs something to populate existing rows).
Please select a fix:
 1) Provide a one-off default now (will be set on all existing rows with a null
      value for this column)
 2) Quit, and let me add a default in models.py
Select an option: 1
Please enter the default value now, as valid Python
The datetime and django.utils.timezone modules are available, so you can do
  e.g. timezone.now
Type 'exit' to exit this prompt
>>> ''
Migrations for 'rango':
  rango/migrations/0003_category_slug.py
    - Add field slug to category
```

From there, you can then migrate the changes, and run the population script again to update the new slug fields.

```
$ python manage.py migrate  
$ python populate_rango.py
```

Now run the development server with the command `$ python manage.py runserver`, and inspect the data in the models with the admin interface. Remember that the admin interface is reached by pointing your browser to `http://127.0.0.1:8000/admin/`.

If you go to add in a new category via the admin interface you may encounter a problem – or two!

1. Let's say we added in the category Python User Groups. If you try to save the record, Django will not let you save it unless you also fill in the slug field too. While we could type in `python-user-groups`, relying on users to fill out fields will always be error-prone, and wouldn't provide a good user experience. It would be better to have the slug automatically generated.
2. The next problem arises if we have one category called `Django` and one called `django`. Since the `slugify()` makes the slugs lower case it will not be possible to identify which category corresponds to the `django` slug.

To solve the first problem, there are two possible solutions. The easiest solution would be to update our model so that the slug field allows blank entries, i.e.:

```
slug = models.SlugField(blank=True)
```

However, this is less than desirable. A blank slug would probably not make any sense, and at that, you could define multiple blank slugs! How could we then identify what category a user is referring to? **A much better solution** would be to customise the admin interface so that it automatically pre-populates the slug field as you type in the category name. To do this, update `rango/admin.py` with the following code.

```
from django.contrib import admin
from rango.models import Category, Page
...
# Add in this class to customise the Admin Interface
class CategoryAdmin(admin.ModelAdmin):
    prepopulated_fields = {'slug':('name',)}

# Update the registration to include this customised interface
admin.site.register(Category, CategoryAdmin)
...
```

Note that with the above `admin.site.register()` call, you should ensure that you replace the existing call to `admin.site.register(Category)` with the one provided above. Once completed, try out the admin interface and add in a new category. Note that the `slug` field automatically populates as you type into the `name` field. Try adding a space and see what happens!

Now that we have addressed the first problem, we can ensure that the `slug` field is also unique by adding the constraint to the `slug` field. Update your `models.py` module to reflect this change with the inclusion of `unique=True`.

```
slug = models.SlugField(unique=True)
```

Now that we have added in the `slug` field, we can now use the slugs to guarantee a unique match to an associated category. We could have added the `unique` constraint earlier. However, if we had done that and then performed the migration (and thus setting everything to be an empty string by default), the migration would have failed. This is because the `unique` constraint would have been violated. We could have deleted the database and then recreated everything – but that is not always desirable.



Migration Woes

It's always best to plan out your database in advance and avoid changing it. Making a population script means that you easily recreate your database if you find that you need to delete it and start again.

If you find yourself completely stuck, it is sometimes just more straightforward to delete the database and recreate everything from scratch, rather than trying to work out where the issue is coming from.

Category Page Workflow

Now we need to figure out how to create a page for individual categories. This will allow us to then be able to see the pages associated with a category. To implement the category pages so that they can be accessed using the URL pattern `/rango/category/<category-name-slug>/`, we need to undertake the following steps.

1. Import the `Page` model into `rango/views.py`.
2. Create a new view in `rango/views.py` called `show_category()`. The `show_category()` view will take a second parameter, `category_name_slug`, which will store the encoded category name.
 - We will need helper functions to encode and decode `category_name_slug`.
3. Create a new template, `templates/rango/category.html`.
4. Update Rango's `urlpatterns` to map the new category view to a URL pattern in `rango/urls.py`.

We'll also need to update the `index()` view and `index.html` template to provide links to the category page view.

Category View

In `rango/views.py`, we first need to import the `Page` model. This means we must add the following import statement at the top of the file.

```
from rango.models import Page
```

Next, we can add our new view, `show_category()`.

```
def show_category(request, category_name_slug):
    # Create a context dictionary which we can pass
    # to the template rendering engine.
    context_dict = {}

    try:
        # Can we find a category name slug with the given name?
        # If we can't, the .get() method raises a DoesNotExist exception.
        # The .get() method returns one model instance or raises an exception.
        category = Category.objects.get(slug=category_name_slug)

        # Retrieve all of the associated pages.
        # The filter() will return a list of page objects or an empty list.
        pages = Page.objects.filter(category=category)

        # Adds our results list to the template context under name pages.
        context_dict['pages'] = pages
        # We also add the category object from
        # the database to the context dictionary.
        # We'll use this in the template to verify that the category exists.
        context_dict['category'] = category
    except Category.DoesNotExist:
        # We get here if we didn't find the specified category.
        # Don't do anything -
        # the template will display the "no category" message for us.
        context_dict['category'] = None
        context_dict['pages'] = None

    # Go render the response and return it to the client.
    return render(request, 'rango/category.html', context=context_dict)
```

Our new view follows the same basic steps as our `index()` view. We first define a context dictionary. Then, we attempt to extract the data from the models and add the relevant data to the context dictionary. We determine which category has been requested by using the value passed `category_name_slug` to the `show_category()` view function (in addition to the `request` parameter).

If the category slug is found in the `Category` model, we can then pull out the associated pages, and add this to the context dictionary, `context_dict`. If the category requested was not found, we set the associated context dictionary values to `None`. Finally, we `render()` everything together, using a new `category.html` template.

Category Template

In your <workspace>/tango_with_django_project/templates/rango/ directory, create a new template called category.html. In the new file, add the following code.

```
1  <!DOCTYPE html>
2
3  <html>
4
5      <head>
6          <title>Rango</title>
7      </head>
8
9      <body>
10         <div>
11             {% if category %}
12                 <h1>{{ category.name }}</h1>
13                 {% if pages %}
14                     <ul>
15                         {% for page in pages %}
16                             <li><a href="{{ page.url }}">{{ page.title }}</a></li>
17                         {% endfor %}
18                     </ul>
19                 {% else %}
20                     <strong>No pages currently in category.</strong>
21                 {% endif %}
22             {% else %}
23                 The specified category does not exist.
24             {% endif %}
25         </div>
26     </body>
27
28 </html>
```

The HTML code example again demonstrates how we utilise the data passed to the template via its context through the tags {{ }}. We access the category and pages objects and their fields – such as category.name and page.url, for example.

If the category exists, then we check to see if there are any pages in the category. If so, we iterate through the returned pages using the {% for page in pages %} template tags. For each page in the pages list, we present their title and url attributes as listed

hyperlink (e.g. within `` and `<a>` elements). These are displayed in an unordered HTML list (denoted by the `` tags). If you are not too familiar with HTML, then have a look at the [HTML Tutorial by W3Schools.com](#)⁶ to learn more about the different tags.



Note on Conditional Template Tags

The Django template conditional tag – represented with `{% if condition %}` – is a really neat way of determining the existence of an object within the template's context. Make sure you check the existence of an object to avoid errors when rendering your templates, especially if your associated view's logic doesn't populate the context dictionary in all possible scenarios.

Placing conditional checks in your templates – like `{% if category %}` in the example above – also makes sense semantically. The outcome of the conditional check directly affects how the rendered page is presented to the user. Remember, presentational aspects of your Django apps should be encapsulated within templates.

Parameterised URL Mapping

Now let's have a look at how we pass the value of the `category_name_url` parameter to our function `show_category()` function. To do so, we need to modify Rango's `urls.py` file (remember, the file located at `<workspace>/tango_with_django_project/rango/`), and update the `urlpatterns` tuple as follows.

```
urlpatterns = [
    path('', views.index, name='index'),
    path('about/', views.about, name='about'),
    path('category/<slug:category_name_slug>',
         views.show_category, name='show_category'),
]
```

A parameter, represented by `<slug:category_name_slug>`, is added to a new mapping. This indicates to Django that we want to match a string which is a slug, and to assign it to variable `category_name_slug`. You will notice that this variable

⁶<http://www.w3schools.com/html/>

name is what we pass through to the view `show_category()`. If these two names do not match exactly, Django will get confused and raise an error. Instead of slugs, you can also extract out other variables like strings and integers. Refer to the [Django documentation on URL paths⁷](#) for more details. If you need to parse more complicated expressions, you can use `re_path()` instead of `path()`. This will allow you to match all sorts of regular (and irregular) expressions. Luckily for us, Django provides matches for the most common patterns.

All view functions defined as part of a Django app *must* take at least one parameter. This is typically called `request` – and provides access to information related to the given HTTP request made by the user. When parameterising URLs, you supply additional named parameters to the signature for the given view. That's why `show_category()` was defined as `def show_category(request, category_name_slug)`.



Regex Hell

“Some people, when confronted with a problem, think ‘I know, I’ll use regular expressions.’ Now they have two problems.” [Jamie Zawinski⁸](#)

Django’s `path()` method means you can generally avoid Regex Hell – but if you need to use a regular expression (with the `re_path()` function, for instance), this [cheat sheet⁹](#) is really useful.

Modifying the Index Template

Our new view is set up and ready to go – but we need to do one more thing. Our index page template needs to be updated so that it links to the category pages that are listed. We can update the `index.html` template to now include a link to the category page via the slug.

Locate the template code `{% for category in categories %}`, and adjust the `` element to match the example below.

⁷<https://docs.djangoproject.com/en/2.1/ref/urls/>

⁸<http://blog.codinghorror.com/regular-expressions-now-you-have-two-problems/>

⁹<http://cheatography.com/davechild/cheat-sheets/regular-expressions/>

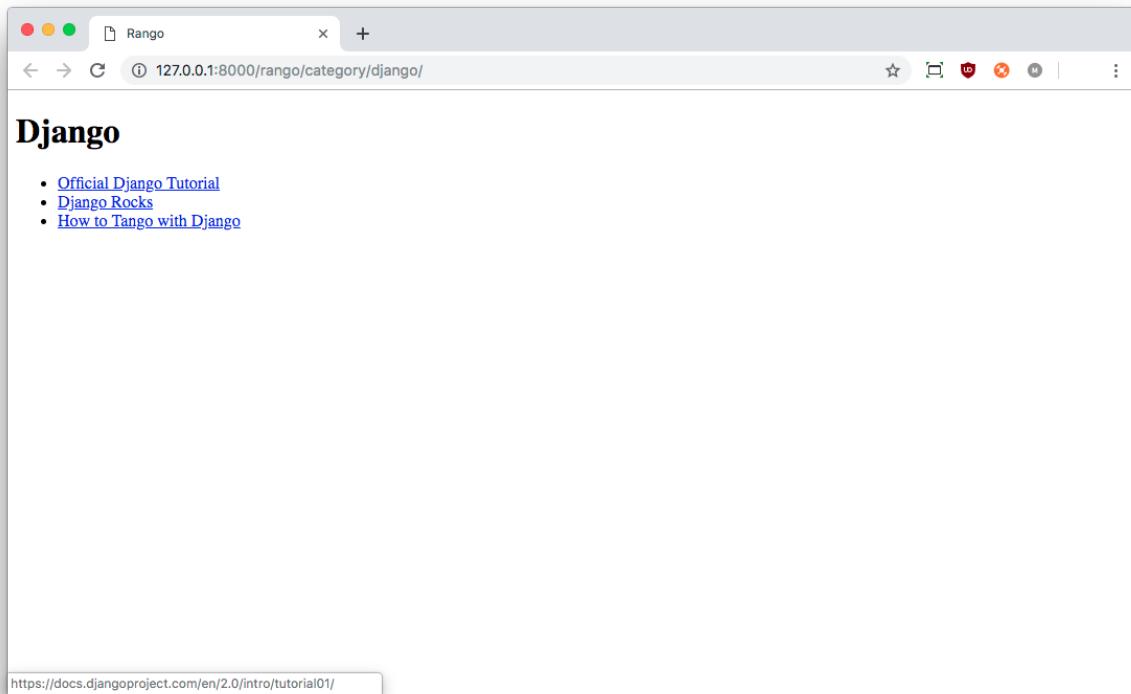
```
<ul>
    {% for category in categories %}
        <li>
            <!-- The following line is changed to add an HTML hyperlink --&gt;
            &lt;a href="/rango/category/{{ category.slug }}/"&gt;{{ category.name }}&lt;/a&gt;
        &lt;/li&gt;
    {% endfor %}
&lt;/ul&gt;</pre>
```

Again, we used the HTML tag `` to define an unordered list. Within the list, we create a series of list elements (``), each of which in turn contains an HTML hyperlink (`<a>`). The hyperlink has an `href` attribute, which we use to specify the target URL defined by `/rango/category/{{ category.slug }}/` which, for example, would turn into `/rango/category/other-frameworks/` for the category Other Frameworks.

Demo

Let's try everything out now by visiting the Rango homepage. You should see *up to* five categories on the index page (although the population script only creates three – you can test this by adding up to six categories in the admin interface). The categories should now be links. Clicking on Django should then take you to the Django category page, as shown in the [figure below](#). If you see a list of links like Official Django Tutorial, then you've successfully set up the new page.

What happens when you visit a category that does not exist? Try navigating a category which doesn't exist, like `/rango/category/computers/`. Do this by typing the address manually into your browser's address bar. You should see a message telling you that the specified category does not exist. Look at your template's logic and work through it to understand what is going on.



The links to Django pages. Note the mouse is hovering over the first link – you can see the corresponding URL for that link at the bottom left of the Google Chrome window.

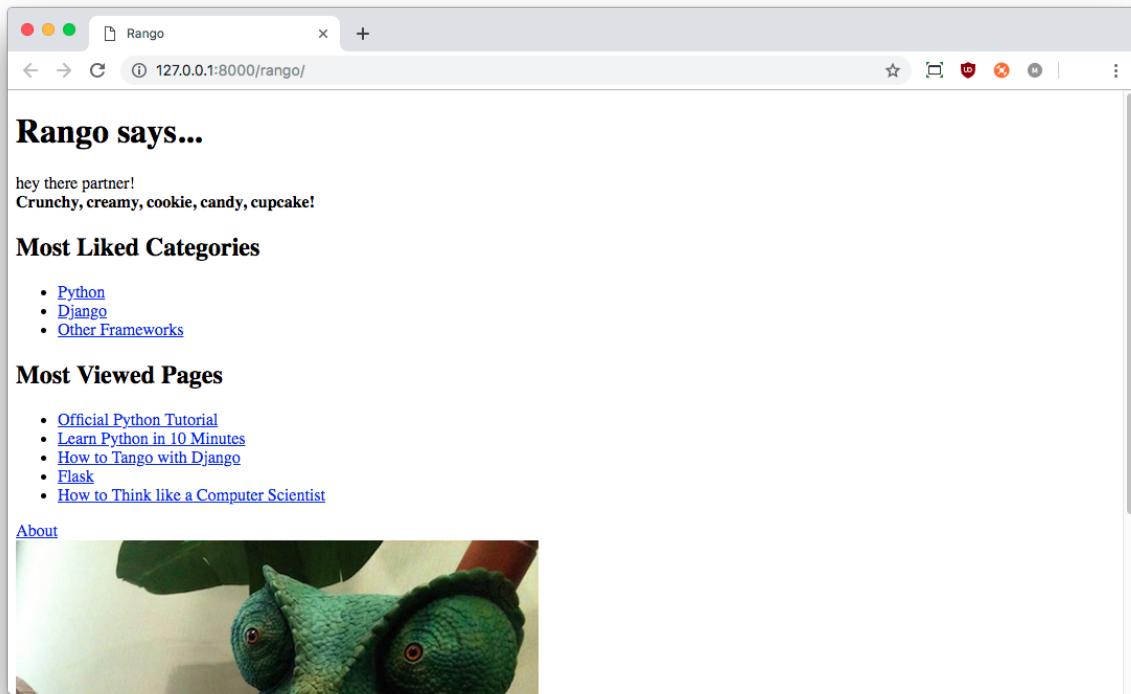


Exercises

Reinforce what you've learnt in this chapter!

- Update the population script to add some value to the `views` count for each **page**. Pick whatever integers you want – as long as each page receives a whole (integer) number greater than zero.
- Modify the index page to also include the top five most viewed pages. When no pages are present, you should include a friendly message in place of a list, stating: There are no pages present. This message should be bolded – wrap it around `...` tags. The entire top five pages block should then be wrapped in its own `<div>...</div>` tag.
- Leading on from the exercise above, include a heading for the Most Liked Categories and Most Viewed Pages. These must be placed as second-level headers, using the `<h2>` tag. Place each of the headers above their respective list.
- Include a link back to the index page from the category page.
- Undertake part three of official Django tutorial¹⁰ if you have not done so already to reinforce what you've learnt here.

¹⁰<https://docs.djangoproject.com/en/2.1/intro/tutorial03/>



The index page after you complete the exercises. Your output may vary slightly.

 **Hints**

- When updating the population script, you'll essentially follow the same process as you went through in the [previous chapter's](#) exercises. You will need to update the data structures for each page, and also update the code that makes use of them.
 - Update the `python_pages`, `django_pages` and `other_pages` data structures. Each page has a `title` and `url` – they all now need a count of how many `views` they see, too.
 - Look at how the `add_page()` function is defined in your population script. Does it allow for you to pass in a `views` count? Do you need to change anything in this function?
 - Finally, update the line where the `add_page()` function is *called*. If you called the `views` count in the data structures `views`, and the dictionary that represents a page is called `p` in the context of where `add_page()` is called, how can you pass the `views` count into the function?
 - Remember to re-run the population script so that the database is updated with your new counts.
- You will need to edit both the `index` view and the `index.html` template to put the most viewed (i.e. popular pages) on the index page.
 - Instead of accessing the `Category` model, you will have to ask the `Page` model for the most viewed pages.
 - Remember to pass the list of pages through to the context.
 - If you are not sure about the HTML template code to use, you can draw inspiration from the `category.html` template markup. The markup that you need to write is essentially the same. However, remember a `Page url` is just that – it's the entire URL!



Model Tips

For more tips on working with models you can take a look through the following blog posts:

1. [Best Practices when working with models¹¹](http://steelkiwi.com/blog/best-practices-working-django-models-python/) by Alexander Stepanov. In this post, you will find a series of tips and tricks when working with models.
2. [How to make your Django Models DRYer¹²](https://medium.com/@raiderrobert/make-your-django-models-dryer-4b8d0f3453dd#.ozrdt3rsm) by Robert Roskam. In this post, you can see how you can use the property method of a class to reduce the amount of code needed when accessing related models.



Test your Implementation

Like in the previous chapter, we've implemented a series of unit tests to allow you to check your implementation up until this point. [Follow the guide we provided earlier](#), using the test module `tests_chapter6.py`. How does your implementation stack up against our tests? Remember that your implementation should have fully completed the exercises listed above for the tests to pass.

Some of these tests may seem overly harsh – but remember, this book is a specification for the product we want you to develop. If you don't develop software *exactly* as specified, it can produce undesirable results when your bit of code is plugged into a larger framework. By following specifications to the letter, you'll be learning a valuable lesson that you can take forward in a future development career.

¹¹<http://steelkiwi.com/blog/best-practices-working-django-models-python/>

¹²<https://medium.com/@raiderrobert/make-your-django-models-dryer-4b8d0f3453dd#.ozrdt3rsm>

7. Forms

As part of the Rango application, we will want to capture new categories and new pages from users. In this chapter, we will run through how to capture data through web forms. Django comes with some excellent form handling functionality, making it a pretty straightforward process to collect information from users and save it to the database via the models. According to [Django's documentation on forms¹](#), the form handling functionality allows you to:

1. display an HTML form with automatically generated *form widgets* (like a text field or date picker);
2. check submitted data against a set of validation rules;
3. redisplay a form in case of validation errors; and
4. convert submitted form data to the relevant Python data types.

One of the major advantages of using Django's forms functionality is that it can save you a lot of time and hassle creating the HTML forms.

7.1 Basic Workflow

The basic steps involved in creating a form and handling user input is as follows.

1. If you haven't already got one, create a `forms.py` module within your Django app's directory (`rango`) to store form-related classes.
2. Create a `ModelForm` class for each model that you wish to represent as a form.
3. Customise the forms as you desire.
4. Create or update a view to handle the form...
 - including *displaying* the form,
 - *saving* the form data, and

¹<https://docs.djangoproject.com/en/2.1/topics/forms/>

- *flagging up errors* which may occur when the user enters incorrect data (or no data at all) in the form.
5. Create or update a template to display the form.
 6. Add a `urlpattern` to map to the new view (if you created a new one).

This workflow is a bit more complicated than those we have previously seen, and the views that we have to construct are lot more complex, too. However, once you undertake the process a few times, it will become clearer how everything pieces together. Trust us.

7.2 Page and Category Forms

Here, we will implement the necessary infrastructure that will allow users to add categories and pages to the database via forms.

First, create a file called `forms.py` within the `rango` application directory. While this step is not necessary (you could put the forms in the `models.py`), this makes your codebase tidier and easier to work with.

Creating ModelForm Classes

Within Rango's `forms.py` module, we will be creating several classes that inherit from Django's `ModelForm`. In essence, a `ModelForm`² is a *helper class* that allows you to create a Django `Form` from a pre-existing model. As we've already got two models defined for Rango (`Category` and `Page`), we'll create `ModelForms` for both.

In `rango/forms.py` add the following code.

²<https://docs.djangoproject.com/en/2.1/topics/forms/modelforms/#modelform>

```
1 from django import forms
2 from rango.models import Page, Category
3
4 class CategoryForm(forms.ModelForm):
5     name = forms.CharField(max_length=128,
6                            help_text="Please enter the category name.")
7     views = forms.IntegerField(widget=forms.HiddenInput(), initial=0)
8     likes = forms.IntegerField(widget=forms.HiddenInput(), initial=0)
9     slug = forms.CharField(widget=forms.HiddenInput(), required=False)
10
11 # An inline class to provide additional information on the form.
12 class Meta:
13     # Provide an association between the ModelForm and a model
14     model = Category
15     fields = ('name',)
16
17 class PageForm(forms.ModelForm):
18     title = forms.CharField(max_length=128,
19                            help_text="Please enter the title of the page.")
20     url = forms.URLField(max_length=200,
21                          help_text="Please enter the URL of the page.")
22     views = forms.IntegerField(widget=forms.HiddenInput(), initial=0)
23
24 class Meta:
25     # Provide an association between the ModelForm and a model
26     model = Page
27
28     # What fields do we want to include in our form?
29     # This way we don't need every field in the model present.
30     # Some fields may allow NULL values; we may not want to include them.
31     # Here, we are hiding the foreign key.
32     # we can either exclude the category field from the form,
33     exclude = ('category',)
34     # or specify the fields to include (don't include the category field).
35     fields = ('title', 'url', 'views')
```

We need to specify which fields are included on the form, via `fields`, or specify which fields are to be excluded, via `exclude`.

Django provides us with several ways to customise the forms that are created on our behalf. In the code sample above, we've specified the widgets that we wish to use for each field to be displayed. For example, in our `PageForm` class, we've

defined `forms.CharField` for the `title` field, and `forms.URLField` for `url` field. Both fields provide text entry for users. Note the `max_length` parameters we supply to our fields – the lengths that we specify are identical to the maximum length of each field we specified in the underlying data models. Go back to the [chapter on models](#) to check for yourself, or have a look at Rango's `models.py` module.

You will also notice that we have included several `IntegerField` entries for the `views` and `likes` fields in each form. Note that we have set the `widget` to be hidden with the parameter setting `widget=forms.HiddenInput()`, and then set the value to zero with `initial=0`. This is one way to set the field to zero by default. Since the fields will be hidden, the user won't be able to enter a value for these fields.

However, even though we have a `hidden` field in the `PageForm`, we still need to include the field in the form. If in `fields` we excluded `views`, then the form would not contain that field (despite it being specified). This would mean that the form would not return the value zero for that field. This may raise an error depending on how the model has been set up. If in the model we specified that the `default=0` for these fields, then we can rely on the model to automatically populate the field with the default value – and thus avoid a `not null` error. In this case, it would not be necessary to have these hidden fields. We have also included the field `slug` in the `CategoryForm`, and set it to use the `widget=forms.HiddenInput()`.

However, rather than specifying an initial or default value, we have said the field is not required by the form. This is because our model will be responsible for populating the field when the form is eventually saved. Essentially, you need to be careful when you define your models and forms to make sure that the form is going to contain and pass on all the data that is required to populate your model correctly.

Besides the `CharField` and `IntegerField` widgets, many more are available for use. As an example, Django provides `EmailField` (for e-mail address entry), `ChoiceField` (for radio input buttons), and `DateField` (for date/time entry). There are many other field types you can use, which perform error checking for you (e.g. *is the value provided a valid integer?*).

Perhaps the most important aspect of a class inheriting from `ModelForm` is the need to define *which model we're wanting to provide a form for*. We take care of this through

our nested `Meta` class. Set the `model` attribute of the nested `Meta` class to the model you wish to use. For example, our `CategoryForm` class has a reference to the `Category` model. This is a crucial step enabling Django to take care of creating a form in the image of the specified model. It will also help in handling the flagging up of any errors, along with saving and displaying the data in the form.

We also use the `Meta` class to specify which fields we wish to include in our form through the `fields` tuple. Use a tuple of field names to specify the fields you wish to include.



More about Forms

Check out the [official Django documentation on forms³](#) for further information about the different widgets and how to customise forms.

Creating an Add Category View

With our `CategoryForm` class now defined, we're now ready to create a new view to display the form and handle the posting of form data. To do this, add the following code to `rango/views.py`.

```
def add_category(request):
    form = CategoryForm()

    # A HTTP POST?
    if request.method == 'POST':
        form = CategoryForm(request.POST)

        # Have we been provided with a valid form?
        if form.is_valid():
            # Save the new category to the database.
            form.save(commit=True)
            # Now that the category is saved, we could confirm this.
            # For now, just redirect the user back to the index view.
            return redirect('/rango/')

    else:
        # The supplied form contained errors -
```

³<https://docs.djangoproject.com/en/2.1/ref/forms/>

```
# just print them to the terminal.  
print(form.errors)  
  
# Will handle the bad form, new form, or no form supplied cases.  
# Render the form with error messages (if any).  
return render(request, 'rango/add_category.html', {'form': form})
```

You'll need to add the following two `import` statements at the top of the module, too.

```
from rango.forms import CategoryForm  
from django.shortcuts import redirect
```

The new `add_category()` view adds several key pieces of functionality for handling forms. First, we create a `CategoryForm()`, then we check if the HTTP request was a `POST` (did the user submit data via the form?). We can then handle the `POST` request through the same URL. The `add_category()` view function can handle three different scenarios:

- showing a new, blank form for adding a category;
- saving form data provided by the user to the associated model, and *redirecting* to the Rango homepage; and
- if there are errors, redisplay the form with error messages.



GET and POST

What do we mean by `GET` and `POST`? They are *HTTP requests*.

- An HTTP `GET` is used to *request a representation of the specified resource*. In other words, we use a HTTP `GET` to retrieve a particular resource, whether it is a webpage, image or some other file.
- In contrast, an HTTP `POST` *sends data from the client's web browser to be processed*. This type of request is used for example when submitting the contents of a HTML form.
- Ultimately, an HTTP `POST` may end up being programmed to create a new resource (e.g. a new database entry) on the server. This could later be accessed through an HTTP `GET` request.
- Check out the [w3schools page on GET vs. POST⁴](#) for more details.

⁴http://www.w3schools.com/tags/ref_httpmethods.asp

Django's form handling machinery processes the data returned from a user's browser via an HTTP POST request. It not only handles the saving of form data into the chosen model but will also automatically generate any error messages for each form field (if any are required). This means that Django will not store any submitted forms with missing information that could potentially cause problems for your database's [referential integrity](#)⁵. For example, supplying no value in the category name field will return an error, as the field cannot be blank.

From the `render()` call, you'll see that we refer to `add_category.html` – a new template (we define this below!). This will contain the relevant Django template code and HTML for the form and page.

Creating the Add Category Template

Create the file `templates/rango/add_category.html`. Within the file, add the following HTML markup and Django template code.

```
1  <!DOCTYPE html>
2  <html>
3      <head>
4          <title>Rango</title>
5      </head>
6
7      <body>
8          <h1>Add a Category</h1>
9          <div>
10             <form id="category_form" method="post" action="/rango/add_category/">
11                 {% csrf_token %}
12                 {% for hidden in form.hidden_fields %}
13                     {{ hidden }}
14                 {% endfor %}
15                 {% for field in form.visible_fields %}
16                     {{ field.errors }}
17                     {{ field.help_text }}
18                     {{ field }}
19                 {% endfor %}
20                 <input type="submit" name="submit" value="Create Category" />
21             </form>
```

⁵https://en.wikipedia.org/wiki/Referential_integrity

```
22      </div>
23  </body>
24 </html>
```

You can see that within the `<body>` of the HTML page, we placed a `<form>` element. Looking at the attributes for the `<form>` element, you can see that all data captured within this form is sent to the URL `/rango/add_category/` as an HTTP POST request (the `method` attribute is case insensitive, so you can do `POST` or `post` – both provide the same functionality). Within the form, we have two for loops,

- with the first controlling *hidden* form fields, and
- the second controlling *visible* form fields.

The visible fields (those that will be displayed to the user) are controlled by the `fields` attribute within your `ModelForm Meta` class. These template loops produce the necessary HTML markup for each form element. For visible form fields, we also add in any errors that may be present with a particular field and help text that can be used to explain to the user what he or she needs to enter.



Hidden Fields

The need for hidden as well as visible form fields are necessitated by the fact that HTTP is a *stateless protocol*. You can't persist state between different HTTP requests that can make certain parts of web applications difficult to implement. To overcome this limitation, hidden HTML form fields were created which allow web applications to pass important information to a client (which cannot be seen on the rendered page) in an HTML form, only to be sent back to the originating server when the user submits the form.



Cross Site Request Forgery Tokens

You should also take note of the code snippet `{% csrf_token %}`. This is a *Cross-Site Request Forgery (CSRF) token*, which helps to protect and secure the HTTP POST request that is initiated on the subsequent submission of a form. *The Django framework requires the CSRF token to be present. If you forget to include a CSRF token in your forms, a user may encounter errors when he or she submits the form.* Check out the [official Django documentation on CSRF tokens](#)⁶ for more information about this.

Mapping the Add Category View

Now we need to map the `add_category()` view to a URL. In the template, we have used the URL `/rango/add_category/` in the form's action attribute. We now need to create a mapping from the URL to the view. In `rango/urls.py` modify the `urlpatterns` list to make it look like the following.

```
urlpatterns = [
    path('', views.index, name='index'),
    path('about/', views.about, name='about'),
    path('category/<slug:category_name_slug>/', views.show_category,
         name='show_category'),
    path('add_category/', views.add_category, name='add_category'),
]
```

Ordering doesn't necessarily matter in this instance. However, take a look at the [official Django documentation on how Django processes a request](#)⁷ for more information. The URL for adding a category is `/rango/add_category/`, with the name of `add_category`.

Modifying the Index Page View

As a final step, we can put a link on the index page so that users can then easily navigate to the page that allows them to add categories. Edit the template `rango/index.html`, and add the following HTML hyperlink in the `<div>` element with the *About* link.

⁶<https://docs.djangoproject.com/en/2.1/ref/csrf/>

⁷<https://docs.djangoproject.com/en/2.1/topics/http/urls/#how-django-processes-a-request>

```
<a href="/rango/add_category/">Add a New Category</a><br />
```

Demo

Now let's try it out! Start or restart your Django development server, and then point your web browser to Rango at `http://127.0.0.1:8000/rango/`. Use your new link to jump to the Add Category page, and try adding a category. The [figure below](#) shows screenshots of the *Add Category* and *Index* pages. In the screenshots, we add the category *Assembly*.

Please enter the category name. Create Category

Adding a new category called "Assembly"

Rango says...

hey there partner!
Crunchy, creamy, cookie, candy, cupcake!

Most Liked Categories

- [Python](#)
- [Django](#)
- [Other Frameworks](#)
- [Assembly](#)

Now it appears in the list!
Success!

Most Viewed Pages

- [Official Python Tutorial](#)
- [Learn Python in 10 Minutes](#)
- [How to Tango with Django](#)
- [Flask](#)
- [How to Think like a Computer Scientist](#)

[About](#)
[Add New Category](#)

Adding a new category to Rango with our new form.



Missing Categories?

If you play around with this new functionality and add several different categories, remember that they will not always appear on the index page. This is because we coded up our index view to only show the *top five categories* in terms of the number of likes they have received. If you log into the admin interface, you should be able to view all the categories that you have entered.

For confirmation that the category is being added, you can update the `add_category()` method in `rango/views.py` and change the line `form.save(commit=True)` to be `cat = form.save(commit=True)`. This will give you a reference to an instance of the created Category object. You can then print the category (e.g. `print(cat, cat.slug)`).

Cleaner Forms

Recall that our `Page` model has a `url` attribute set to an instance of the `URLField` type. In a corresponding HTML form, Django would reasonably expect any text entered into a `url` field to be a correctly formatted, complete URL. However, users can find entering something like `http://www.url.com` to be cumbersome – indeed, users [may not even know what forms a correct URL⁸!](#)



URL Checking

Most modern browsers will now check to make sure that the URL is well-formed for you, so this example will only work on older browsers. However, it does show you how to clean the data before you try to save it to the database. If you don't have an old browser to try this example (in case you don't believe it), try changing the `URLField` to a `CharField`. The rendered HTML will then not instruct the browser to perform the checks on your behalf, and the code you implemented will be executed.

In scenarios where user input may not be entirely correct, we can *override* the `clean()` method implemented in `ModelForm`. This method is called upon before saving form data to a new model instance, and thus provides us with a logical place

⁸<https://support.google.com/webmasters/answer/76329?hl=en>

to insert code which can verify – and even fix – any form data the user inputs. We can check if the value of `url` field entered by the user starts with `http://` – and if it doesn't, we can prepend `http://` to the user's input.

```
class PageForm(forms.ModelForm):  
    ...  
    def clean(self):  
        cleaned_data = self.cleaned_data  
        url = cleaned_data.get('url')  
  
        # If url is not empty and doesn't start with 'http://',  
        # then prepend 'http://'.  
        if url and not url.startswith('http://'):  
            url = f'http://{url}'  
            cleaned_data['url'] = url  
  
    return cleaned_data
```

Within the `clean()` method, a simple pattern is observed which you can replicate in your own Django form handling code.

1. Form data is obtained from the `ModelForm` dictionary attribute `cleaned_data`.
2. Form fields that you wish to check can then be taken from the `cleaned_data` dictionary. Use the `.get()` method provided by the dictionary object to obtain the form's values. If a user does not enter a value into a form field, its entry will not exist in the `cleaned_data` dictionary. In this instance, `.get()` would return `None` rather than raise a `KeyError` exception. This helps your code look that little bit cleaner!
3. For each form field that you wish to process, check that a value was retrieved. If something was entered, check what the value was. If it isn't what you expect, you can then add some logic to fix this issue before *reassigning* the value in the `cleaned_data` dictionary.
4. You *must* always end the `clean()` method by returning the reference to the `cleaned_data` dictionary. Otherwise, the changes won't be applied.

This trivial example shows how we can clean the data being passed through the form before being stored. This is pretty handy, especially when particular fields

need to have default values – or data within the form is missing, and we need to handle such data entry problems.



What about https?

The overridden `clean()` method we provide above only considers `http://` as a valid protocol/schema. As using secure HTTP is now commonplace in today's world, you should also really consider URLs starting with `https://`, too. The above however only serves as a simple example of how you can check and clean a form's fields before saving its data.



Clean Overrides

Overriding methods implemented as part of the Django framework can provide you with an elegant way to add that extra bit of functionality for your application. There are many methods which you can safely override for your benefit, just like the `clean()` method in `ModelForm` as shown above. Check out [the Official Django Documentation on Models](#)⁹ for more examples on how you can override default functionality to slot your own in.

⁹<https://docs.djangoproject.com/en/2.1/topics/db/models/#overriding-predefined-model-methods>



Exercises

Now that you've worked through the chapter, consider the following questions, and how you could solve them.

- What would happen if you don't enter in a category name on the add category form?
- What happens when you try to add a category that already exists?
- In the [section above where we implemented our ModelForm classes](#), we repeated the `max_length` values for fields that we had previously defined in [the models chapter](#). This is bad practice as we are *repeating ourselves!* How can you refactor your code so that you are *not* repeating the same `max_length` values?
- If you have not done so already undertake [part four of the official Django Tutorial¹⁰](#) to reinforce what you have learnt here.
- Now, implement functionality to let users add pages to each category. See below for the instructions (and hints!).

Creating an Add Pages View, Template and URL Mapping

A next logical step would be to allow users to add pages to a given category. To do this, repeat the same workflow above, but for adding pages.

- Create a new view, `add_page()`.
- Create a new template, `rango/add_page.html`.
- Create a mapping between `/rango/category/<category_name_slug>/add_page/` and the new view.
- Update the category page/view to provide a link from the category add page functionality.

To get you started, here is the code for the `add_page()` view function.

¹⁰<https://docs.djangoproject.com/en/2.1/intro/tutorial04/>

```
from rango.forms import PageForm

def add_page(request, category_name_slug):
    try:
        category = Category.objects.get(slug=category_name_slug)
    except Category.DoesNotExist:
        category = None

    # You cannot add a page to a Category that does not exist...
    if category is None:
        return redirect('/rango/')

    form = PageForm()

    if request.method == 'POST':
        form = PageForm(request.POST)

        if form.is_valid():
            if category:
                page = form.save(commit=False)
                page.category = category
                page.views = 0
                page.save()

            return redirect(reverse('rango:show_category',
                                  kwargs={'category_name_slug':
                                          category_name_slug}))

    else:
        print(form.errors)

    context_dict = {'form': form, 'category': category}
    return render(request, 'rango/add_page.html', context=context_dict)
```

Note that in the example above, we need to *redirect* the user to the `show_category()` view once the page has been created. This involves the use of the `redirect()` and `reverse()` helper functions to redirect the user and to lookup the appropriate URL, respectively. The following imports at the top of Rango's `views.py` module will therefore be required for this code to work – but you should have `redirect` from earlier.

```
from django.shortcuts import redirect
from django.urls import reverse
```

Here, the `redirect()` function is called which in turn calls the `reverse()` function. `reverse()` looks up URL names in your `urls.py` modules – in this instance, `rango:show_category`. If a match is found against the name provided, the complete URL is returned. The added complication here is that the `show_category()` view takes an additional parameter `category_name_slug`. By providing this value in a dictionary as `kwargs` to the `reverse()` function, it has all of the information it needs to formulate a complete URL. This completed URL is then used as the parameter to the `redirect()` method, and the response is complete!

Note also from the new view that when attempting to add a new page to a category that does not exist, we simply redirect the user back to the index page. This removes the need for you to implement something to cover this scenario in your template. Your template needs to only focus on displaying a form.



More on `reverse()`

This is our first exposure to the `reverse()` helper function. It's an incredibly useful and powerful function, and we'll be using it extensively in the [next chapter](#) of the book to make our views more maintainable.



Hints

To help you with the exercises above, the following hints may be of use.

- In the `add_page.html` template, you can access the slug with `{{ category.slug }}`. This is because the view passes the `category` object through to the template via the context dictionary.
- Update Rango's `category.html` template with a new hyperlink, complete with a line break immediately following it: `Add Page
`. Ensure that the link only appears when *the requested category exists* – with or without pages. In terms of code, we mean that your template should have the following conditional: `{% if category %} {% else %}` The specified category does not exist. `{% endif %}`.
- **Make sure that in your `add_page.html`, the `<form>` posts to `/rango/category/{{ category.slug }}/add_page/`.** When submitting a form requesting the creation of a new page, it is important to tell Django (via the URL) what category the new page should belong to! A potential solution to this can be found [here¹¹](#).
- Update `rango/urls.py` with a URL mapping `(/rango/category/<slug:category_name_slug>/add_page/)` to handle the above link. Provide a name of `add_page` to the new mapping.
- You can avoid the repetition of `max_length` parameters through the use of an additional attribute in your `Category` model. This attribute could be used to store the value for `max_length`, and then be referenced where required. For example, you can then include them in your forms by using the notation `Category.NAME_MAX_LENGTH`. [Check out our model solution online¹²](#) to see how we did it.

If all else fails and you still find yourself stuck, you can check out our [model solution on GitHub¹³](#).

¹¹https://github.com/maxwelld90/tango_with_django_2_code/blob/b90fc0b52abda784e09fd8ee99c9f7d356e54470/tango_with_django_project/templates/rango/add_page.html

¹²https://github.com/maxwelld90/tango_with_django_2_code/commit/a8c71c01d2fb34f900ac13433c7d7a3c3f17082b

¹³https://github.com/maxwelld90/tango_with_django_2_code/tree/b90fc0b52abda784e09fd8ee99c9f7d356e54470



Test your Implementation

We've implemented a series of unit tests to allow you to check your implementation up until this point. [Follow the guide we provided earlier](#), using the test module `tests_chapter7.py`. How does your implementation stack up against our tests? Remember that your implementation should have fully completed the exercises listed above for the tests to pass.

8. Working with Templates

So far, we've created several HTML templates for different pages within our Rango application. As you created each additional template, you may have noticed that a lot of the HTML code is repeated. Does that feel a bit strange, or wasteful?

It should! Anytime you repeat similar code, you are violating the [DRY Principle¹](#) – which states: **DO NOT REPEAT YOURSELF!** You might have noticed that in the templates we have written so far, we have referred to different pages with *hard-coded* URL paths. This is bad practice, as paths will invariably change over time.

Taken together, repetition and hard coding will result in a maintenance nightmare. If we want to make a change to the general site structure or change a URL path, you will have to modify *all* the templates which contain either the structure we want to change – or the URL. This might be tractable if your site has a few pages. However, what if your site had hundreds of pages? How long would it take to update everything? How likely would it be that you will make a mistake somewhere?

Luckily for us, the Django developers have already thought about how to solve such problems and provided solutions. In this chapter, we will use *template inheritance* to overcome the first problem, and the *URL template tag* to solve the second problem. We will start by addressing the latter problem first.

8.1 Using Relative URLs in Templates

So far, we have been directly coding the URL of the page or view we want to show within the template, i.e. `About`. This kind of hard coding of URLs means that if we change our URL mappings in `urls.py`, then we will have to also change all of the associated URL references in our templates. The preferred way is to use the template tag `url` to look up the *URL mapping name* in the `urls.py` modules, and dynamically insert the matching URL path.

¹[https://en.wikipedia.org/wiki/Don't_repeat_yourself](https://en.wikipedia.org/wiki/Don%27t_repeat_yourself)

It's pretty simple to include relative URLs in your templates. To refer to the *About* page, we would insert the following line into our templates:

```
<a href="{% url 'rango:about' %}">About</a>
```

The Django template engine will look up any `urls.py` module for a URL pattern with the attribute `name` set to `about` (`name='about'`) for the app with `app_name` set to `rango`, and then perform a reverse lookup on the URL. This means if we change the URL mappings in `urls.py`, we don't need to go through each template that refer to them.

One can also reference a URL pattern without a specified name, by referencing the view directly as shown below.

```
<a href="{% url 'rango.views.about' %}">About</a>
```

In this example, we must ensure that the app `rango` has the view `about`, contained within its `views.py` module. In your app's `index.html` template, you will notice that you have a parameterised URL pattern (the `show_category()` URL/view takes the `category.slug` as a parameter). To handle this, you can pass the `url` template tag the name of the URL/view and the slug within the template, as follows:

```
{% for category in categories %}
    <li>
        <a href="{% url 'rango:show_category' category.slug %}">
            {{ category.name }}
        </a>
    </li>
{% endfor %}
```

Before you run off to update all the URLs in all your templates with relative URLs, we need to restructure and refactor our templates by using inheritance to remove excessive repetition.



URLs and Multiple Django Apps

This book focuses on the development of a single Django app, Rango. However, you may find yourself working on a Django project with multiple apps being used at once. This means that you could literally have hundreds of potential URLs with which you may need to reference. This scenario begs the question: *how can we organise these URLs?* Two apps may have a view of the same name, meaning a potential conflict would exist.

Django provides the ability to *namespace* URL configuration modules² for each individual app that you employ in your project. **As we did earlier**, adding an `app_name` variable to your app's `urls.py` module is enough. The example below specifies the namespace for the Rango app to be `rango`.

```
from django.conf.urls import url
from rango import views

app_name = 'rango'

urlpatterns = [
    path('', views.index, name='index'),
    path('about/', views.about, name='about'),
    path('category/<slug:category_name_slug>/add_page/',
         views.add_page, name='add_page'),
    ...
]
```

Adding an `app_name` variable means that any URL you reference from the `rango` app is done so like:

```
<a href="{% url 'rango:about' %}">About</a>
```

where the colon in the `url` command separates the namespace from the actual URL name.

If an `app_name` is not specified, you would have to reference a URL with just the name of the mapping you want to refer to, such as in the example shown below.

```
<a href="{% url 'about' %}">About</a>
```

Of course, as we discuss above, this approach opens up the door to potential conflicts. If in doubt, it is good practice to specify an `app_name`!

²<https://docs.djangoproject.com/en/2.1/topics/http/urls/#url-namespaces>

8.2 Dealing with Repetition

While pretty much every professionally made website that you use will have a series of repeated components (such as page headers, sidebars, and footers, for example), repeating the HTML for each of these repeating components is not a particularly wise way to handle this. What if you wanted to change part of your website's header? You'd need to go through *every* page and change each copy of the header to suit.

Instead of spending (or wasting!) large amounts of time copying and pasting your HTML markup, we can minimise repetition in Rango's codebase by employing *template inheritance* provided by Django's template language.

The basic approach to using inheritance in templates is as follows.

1. Identify the reoccurring parts of each page that are repeated across your application (i.e. header bar, sidebar, footer, content pane). Sometimes, it can help to draw, on paper, the basic structure of your different pages to help you spot what components are used in common.
2. In a *base template*, provide the skeleton structure of a basic page, along with any common content (i.e. the copyright notice that goes in the footer, the logo and title that appears in the section). Then define several *blocks* which are subject to change depending on which page the user is viewing.
3. Create specific templates for your app's pages – all of which inherit from the base template – and specify the contents of each block.

Reoccurring HTML and The Base Template

Given the templates that we have created so far, it should be pretty obvious that we have been repeating a fair bit of HTML code. Below, we have abstracted away any page-specific details to show the skeleton structure that we have been repeating within each template.

```
1 <!DOCTYPE html>
2 {% load staticfiles %}
3
4 <html>
5     <head lang="en">
6         <meta charset="UTF-8" />
7         <title>Rango</title>
8     </head>
9     <body>
10        <!-- Page specific content goes here -->
11    </body>
12 </html>
```

For the time being, let's make this simple HTML page our app's base template. Save this markup in Rango's `base.html` template.



DOCTYPE goes First!

Remember that the `<!DOCTYPE html>` declaration **always needs to be placed on the first line** of your template. Not having a [document type declaration³](#) on the first line may mean that the resultant page generated from your template will not comply with [W3C HTML guidelines⁴](#).

Template Blocks

Now that we've created our base template, we can add template tags to denote what parts of the template can be overridden by templates that inherit from it. To do this, we will be using the `block` tag. For example, we can add a `body_block` to the base template in `base.html` as follows:

³https://en.wikipedia.org/wiki/Document_type_declaration

⁴<https://www.w3.org/standards/webdesign/htmlcss>

```
1 <!DOCTYPE html>
2 {% load staticfiles %}
3
4 <html>
5     <head lang="en">
6         <meta charset="UTF-8" />
7         <title>Rango</title>
8     </head>
9     <body>
10        {% block body_block %}
11        {% endblock %}
12    </body>
13 </html>
```

Recall that standard Django template commands are denoted by `{%` and `%}` tags. To start a block, the command is `{% block <NAME> %}`, where `<NAME>` is the name of the block you wish to create. You must also ensure that you close the block with the correct `{% endblock %}` command, again enclosed within Django template tags.

You can also specify *default content* for your blocks, which will be used if no inheriting template defines the given block (see [further down](#)). Specify default content by adding markup between `{% block %}` and `{% endblock %}` template commands, just like in the example below.

```
{% block body_block %}
    This is body_block's default content.
{% endblock %}
```

When we create templates for each page, we will inherit from `rango/base.html` and override the contents of `body_block`. However, you can place as many blocks in your templates as you so desire. For example, you could create a block for the page title, a block for the footer, a block for the sidebar, and more. Blocks are a really powerful feature of Django's templating system, and you can learn more about them check on [Django's official documentation on templates](#)⁵.

⁵<https://docs.djangoproject.com/en/2.1/topics/templates/>



Extract Common Structures

You should always aim to extract as much reoccurring content for your base templates as possible. While it may be a hassle to do, the time you will save in maintenance will far outweigh the initial overhead of doing it upfront.

Thinking hurts, but it is better than doing lots of grunt work!

Abstracting Further

Now that you have an understanding of blocks within Django templates, let's take the opportunity to abstract our base template a little bit further. Reopen the `rango/base.html` template and modify it to look like the following.

```
1  <!DOCTYPE html>
2  {% load staticfiles %}
3
4  <html>
5      <head>
6          <title>
7              Rango -
8                  {% block title_block %}
9                      How to Tango with Django!
10                     {% endblock %}
11             </title>
12         </head>
13         <body>
14             <div>
15                 {% block body_block %}
16                 {% endblock %}
17             </div>
18             <hr />
19             <div>
20                 <ul>
21                     <li><a href="{% url 'rango:add_category' %}">Add a New Category</a></li>
22 i>
23                     <li><a href="{% url 'rango:about' %}">About</a></li>
24                     <li><a href="{% url 'rango:index' %}">Index</a></li>
25                 </ul>
26             </div>
```

```
27     </body>
28 </html>
```

From the example above, we have introduced two new features into the base template.

- The first is a template block called `title_block`. This will allow us to specify a custom page title for each page inheriting from our base template. If an inheriting page does not override the block, then the `title_block` defaults to `How to Tango with Django!`, resulting in a complete title of `Rango - How to Tango with Django!`. Look at the contents of the `<title>` tag in the above template to see how this works.
- We have also included the list of links from our current `index.html` template – along with a new link to the index page – and placed them into an HTML `<div>` tag underneath our `body_block` block. This will ensure the links are present across all pages inheriting from the base template. The links are preceded by a *horizontal rule* (`<hr />`) which provides visual separation for the user between the content of the `body_block` block and the links.

8.3 Template Inheritance

Now that we've created a base template with blocks, we can now update all the templates we have created so that they inherit from the base template. Let's start by refactoring the template `rango/category.html`.

You can achieve this by first removing all the repeated HTML code, leaving only the HTML and template tags/commands specific to the page. Then at the beginning of the template add the following line of code:

```
{% extends 'rango/base.html' %}
```

The `extends` command takes one parameter – the template that is to be extended/inherited from (i.e. `rango/base.html`). The parameter you supply to the `extends` command should be relative from your project's templates directory. For example, all templates we use for Rango should extend from `rango/base.html`, not `base.html`.

We can then further modify the category.html template so it looks like the following complete example.

```
1  {% extends 'rango/base.html' %}            
2  {% load staticfiles %}                    
3    
4  {% block title_block %}                  
5      {% if category %}                   
6          {{ category.name }}             
7      {% else %}                        
8          Unknown Category             
9      {% endif %}                      
10     {% endblock %}                     
11   
12    {% block body_block %}              
13        {% if category %}             
14            <h1>{{ category.name }}</h1>   
15        {% if pages %}               
16            <ul>                    
17                {% for page in pages %}   
18                  <li><a href="{{ page.url }}">{{ page.title }}</a></li>   
19                {% endfor %}            
20            </ul>                    
21        {% else %}                   
22            <strong>No pages currently in category.</strong>   
23        {% endif %}                 
24   
25        <a href="{% url 'rango:add_page' category.slug %}">Add Page</a> <br />   
26        {% else %}                   
27            The specified category does not exist.   
28        {% endif %}                 
29     {% endblock %}
```



Loading staticfiles and URLs

You'll need to make sure you add `{% load staticfiles %}` to the top of **each template** that makes use of static media. If you don't, you'll get an error! Template inheritance *does not imply what template tags are available*. If you've programmed before, this works somewhat differently from object-orientated programming languages such as Java, where imports cascade down inheriting classes. Notice how we used the `url` template tag to refer to `rango/<category-name>/add_page/` URL pattern. The `category.slug` is passed through as a parameter to the `url` template tag and Django's template engine will produce the correct URL for us.

Now that we inherit from `rango/base.html`, the `category.html` template is much cleaner extending the `title_block` and `body_block` blocks. `category.html` does not need to be a complete, valid HTML document because `base.html` fills in the blanks for you. Remember: all you are doing here is plugging in additional content to the base template to create the complete, rendered HTML document that is sent to the client's browser. This rendered HTML document will then conform to the necessary standards, such as providing the [DTD⁶](#) on the first line.



More about Templates

Here we have shown how we can minimise the repetition of structure HTML in our templates. However, the Django templating language is very powerful, and even lets you create your own template tags.

Templates can also be used to minimise code within your application's views. For example, if you wanted to include the same database driven content on each page of your application, you could construct a template that calls a specific view to handle the repeating portion of your app's pages. This then saves you from having to call the Django ORM functions that gather the required data for the template in every view that renders it.

If you haven't already done so, it now would be a good time to read through the official [Django documentation on templates⁷](#).

⁶https://en.wikipedia.org/wiki/Document_type_declaration

⁷<https://docs.djangoproject.com/en/2.1/topics/templates/>



Exercises

Now that you've worked through the first part of this chapter, there are several exercises that you can work through to reinforce what you've learnt regarding Django and templating.

- Update all other previously defined templates in the Rango app to extend from the new `base.html` template. Follow the same process as we demonstrated above. Once completed, your templates should all inherit from `base.html`.
- While you're at it, make sure you remove the links from our `index.html` template. We don't need them anymore! This is because we moved them into `base.html` earlier. You can also remove the link to Rango's homepage within the `about.html` template.
- When you refactor `index.html` and `about.html`, keep the images that are served up from the static files and media server.
- Update all references to Rango URLs by using the `url` template tag. You can also do this in your `views.py` module too – check out the [reverse\(\) helper function⁸](#). Don't forget to update the URLs provided in the `action` attributes of your `<form>` elements!
- Make sure all of your new templates employ a `{% block title_block %}` within the HTML `<title>` tag. Remember to use the 'Rango - ' in `base.html`, which will mean all titles are prepended with 'Rango - '. For inheriting templates, use the following titles. If you use our tests at the end of each chapter, these changes will be important.
 - In `about.html`, use the title About Rango.
 - In `add_category.html`, use the title Add a Category.
 - In `add_page.html`, use the title Add a Page.
 - In `category.html`, use the category's name – and if a non-existent category is supplied, use Unknown Category. You'll need to write a basic conditional check (i.e. `{% if ... %}`) here.
 - Finally, in `index.html`, use the title Homepage.

⁸<https://docs.djangoproject.com/en/2.1/ref/urlresolvers/#reverse>



Hints

- Start refactoring the `about.html` template first.
- Update the `title_block` then the `body_block` in each template.
- To check if a category exists in your templates, remember you can simply use the conditional statement `{% if category %}` – so long as the `category` object is passed by that name in your template context!
- Have the development server running and check the page as you work on it. Don't change the whole page to find it doesn't work. Changing things incrementally and testing those changes as you go is a much safer solution.
- To reference the links to category pages, you can use the following template code, paying particular attention to the Django template `{% url %}` command.

```
<a href="{% url 'rango:show_category' category.slug %}">  
    {{ category.name }}  
</a>
```

8.4 The `render()` Method and the `request` Context

When writing views we have used some different methods, the preferred way being the Django shortcut method `render()`. The `render()` method requires that you pass through the `request` as the first argument. The `request` context houses a lot of information regarding the session (including information about the user, for example). Refer to the [official Django Documentation on Request objects](#)⁹.

By passing the `request` through to the template, this means that you will also have access to such information when creating templates. In the next chapter, we will access information about the user – but for now, check through all of your views and make sure that they have been implemented using the `render()` method. Otherwise, your templates won't have the information we need later on.

⁹<https://docs.djangoproject.com/en/2.1/ref/request-response/#httprequest-objects>



render() and Context

As a quick example of the checks you must carry out, have a look at the `about()` view. Initially, this was implemented with a hard-coded string response, as shown below. Note that we only send the string – we don't make use of the `request` passed as the `request` parameter.

```
def about(request):
    return HttpResponse('Rango says: Here is the about page.
                           <a href="/rango/">Index</a>')
```

To employ the use of a template, we call the `render()` function and pass through the `request` object. This will allow the template engine to access information such as the request type (e.g. GET/POST), and information relating to the user's status (have a look at [the user authentication chapter](#)).

```
def about(request):
    # prints out whether the method is a GET or a POST
    print(request.method)
    # prints out the user name, if no one is logged in it prints 'AnonymousUser'
    print(request.user)
    return render(request, 'rango/about.html', {})
```

Remember, the last parameter of `render()` is the context dictionary with which you can use to pass additional data to the Django template engine. As we have no additional data to give to the template, we pass through an empty dictionary, `{}`. Alternatively, we can simply take the third parameter off completely.

8.5 Custom Template Tags

It would be nice to show the different categories that users can browse through in the sidebar on each page. Given what we have learnt so far, we could do the following:

- in the `base.html` template, we could add some code to display an item list of categories; and

- within each view, we could access the Category object, get all the categories, and return that in the context dictionary.

However, this is a pretty nasty solution because we will need to include the same code in all views. A **DRYer**¹⁰ solution would be to create custom template tags that are included in the template, and which can request *their* data.

Using Template Tags

Create a directory <Workspace>/tango_with_django_project/rango/templatetags, and create two new modules within it. One must be called `__init__.py`, and this will be kept blank. Name the second module `rango_template_tags.py`. Add the following code to `rango_template_tags.py`.

```
1 from django import template
2 from rango.models import Category
3
4 register = template.Library()
5
6 @register.inclusion_tag('rango/categories.html')
7 def get_category_list():
8     return {'categories': Category.objects.all()}
```

There is a new method called `get_category_list()` in this snippet. This method returns a dictionary with one key/value pairing. `categories` represents a list of all the `Category` objects present in the database. From the `register.inclusion_tag()` decorator above, you will see that `rango/categories.html` is referred to – another new template. This new template is used by the Django template engine to render the list of categories you provide in the dictionary that is returned in the function. This rendered list can then be injected into the response of the view that initially called the template tag!

Now would be a good time to create the new template, `categories.html`, and add the following HTML markup. Note the similarity in the name of the template with the existing `category.html` template. This is done by design to make sure you are kept on your toes!

¹⁰[https://en.wikipedia.org/wiki/Don't_repeat_yourself](https://en.wikipedia.org/wiki/Don%27t_repeat_yourself)

```
1 <ul>
2 {% if categories %}
3     {% for c in categories %}
4         <li><a href="{% url 'rango:show_category' c.slug %}">{{ c.name }}</a></li>
5     {% endfor %}
6 {% else %}
7     <li><strong>There are no categories present.</strong></li>
8 {% endif %}
9 </ul>
```

To use the template tag in your `base.html` template, first load the custom template tag by including the command `{% load rango_template_tags %}` at the top of the `base.html` template. Put it underneath the `{% load staticfiles %}` line – but remember, **don't put it at the very top!** You can then create a new block to represent the sidebar – and we can call our new template tag with the following code. Put this code between the `<div>` containing the `body_block` and the `<hr />` tag that separates the main content of your page from your app's navigation links.

```
<div>
    {% block sidebar_block %}
        {% get_category_list %}
    {% endblock %}
</div>
```

Try it out. Now all pages that inherit from `base.html` will also include the list of categories (which we will move to the side later on).



Restart the Server!

You'll need to restart the Django development server (or ensure it restarted itself) every time you modify template tags. If the server doesn't restart, Django won't register the tags.

Parameterised Template Tags

We can also *parameterise* the template tags we create, allowing for greater flexibility. As an example, we'll use parameterisation to highlight which category we are looking at when visiting its page. Adding in a parameter is easy – we can update the `get_category_list()` method as follows.

```
def get_category_list(current_category=None):
    return {'categories': Category.objects.all(),
            'current_category': current_category}
```

Note the `current_category` parameter for `get_category_list()`. If a parameter is not passed along, `None` is used instead (implying there is no currently selected category).

We can then update our `base.html` template which makes use of the custom template tag to pass in the current category – but only if it exists.

```
<div>
    {% block sidebar_block %}
        {% get_category_list category %}
    {% endblock %}
</div>
```

We can now also update the `categories.html` template, too. This block of code lives inside the existing `{% if categories %}` conditional check; it doesn't replace the entire template.

```
{% for c in categories %}
    {% if c == current_category %}
        <li>
            <strong>
                <a href="{% url 'rango:show_category' c.slug %}">{{ c.name }}</a>
            </strong>
        </li>
    {% else %}
        <li>
            <a href="{% url 'rango:show_category' c.slug %}">{{ c.name }}</a>
        </li>
    {% endif %}
{% endfor %}
```

In the template, we check to see if the category being displayed is the same as the category being passed through during the `for` loop (i.e. `c == current_category`). If so, we highlight the category name by making it **bold** through use of the `` tag.

Be aware that this solution works because in every step previously that deals with categories, we have called the category variable that is placed in the template dictionary category. If you don't understand what we mean, look at your code for the views `show_category()` and `add_page()`, for instance. Check: *what keys are placed in the context dictionary in those views?* Consistency makes for an easier implementation.



Test your Implementation

We've implemented a series of unit tests to allow you to check your implementation up until this point. [Follow the guide we provided earlier](#), using the test module `tests_chapter8.py`. How does your implementation stack up against our tests? Remember that your implementation should have fully completed the exercises listed above for the tests to pass.

8.6 Summary

In this chapter, we showed how we can:

- reduce *coupling* between URLs and templates by using the `url` template tag to point to relative URLs;
- reduced the amount of boilerplate code by using template inheritance; and
- avoid repetitive code appearing in views by creating custom templates tags.

Taken together, your template code should be much cleaner and easier to maintain. Of course, Django templates offer a lot more functionality. Find out more by visiting the [Django documentation on templates¹¹](#).

¹¹<https://docs.djangoproject.com/en/2.1/ref/templates/>

9. User Authentication

Most web applications ask users to sign up and register, so that they can manage their account and have access to special features. For Rango, we want to control what actions are available to different users. Therefore, this next part of the tutorial aims to get you familiar with the basic user authentication mechanisms provided by Django. We'll be using the `auth` app provided as part of a standard Django installation, located in package `django.contrib.auth`. According to the [Django documentation on authentication¹](#), the app provides the following concepts and functionality.

- The concept of a *User* and the *User Model*.
- *Permissions*, a series of binary flags (e.g. yes/no) that determine what a user may or may not do.
- *Groups*, a method of applying permissions to more than one user.
- A configurable *password hashing system*, a must for ensuring data security.
- *Forms and view tools for logging in users*, or restricting content.

There's lots that Django can do for you regarding user authentication. In this chapter, we'll be covering the basics to get you started. This will help you build your confidence with the available tools and their underlying concepts. Note that we'll be showing you how to set up the user authentication manually, from first principles using Django, but in a later chapter we will show you how we can use a pre-made application that handles the registration process for us.

9.1 Setting up Authentication

Before you can begin to play around with Django's authentication offering, you'll need to make sure that the relevant settings are present in your Rango project's `settings.py` file.

¹<https://docs.djangoproject.com/en/2.1/topics/auth/>

Within the `settings.py` file find the `INSTALLED_APPS` list. Once you've found it, check that `django.contrib.auth` and `django.contrib.contenttypes` are listed, so that the list then looks similar to the example below.

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'rango',
]
```

While `django.contrib.auth` provides Django with access to the provided authentication system, the package `django.contrib.contenttypes` is used by the authentication app to track models installed in your database.



Migrate, if necessary!

If you had to add `django.contrib.auth` and `django.contrib.contenttypes` applications to your `INSTALLED_APPS` tuple, you will need to update your database with the `$ python manage.py migrate` command. This will add the underlying tables to your database e.g. a table for the `User` model.

It's generally good practice to run the `migrate` command whenever you add a new app to your Django project – the app could contain models that'll need to be synchronised to your underlying database.

9.2 Password Hashing

Storing passwords as plaintext within a database is something that should never be done under any circumstances.² If the wrong person acquired a database full of user accounts to your app, they could wreak havoc. Fortunately, Django's `auth` app by default

²<http://stackoverflow.com/questions/1197417/why-are-plain-text-passwords-bad-and-how-do-i-convince-my-boss-that-his-treasur>

stores a [hash of user passwords³](#) using the [PBKDF2 algorithm⁴](#), providing a good level of security for your user's data. However, if you want more control over how the passwords are hashed, you can change the approach used by Django in your project's `settings.py` module, by adding in a tuple to specify the `PASSWORD_HASHERS`. An example of this is shown below.

```
PASSWORD_HASHERS = (
    'django.contrib.auth.hashers.PBKDF2PasswordHasher',
    'django.contrib.auth.hashers.PBKDF2SHA1PasswordHasher',
)
```

Django considers the order of hashers specified as important, and will pick and use the first password hasher in `PASSWORD_HASHERS` (e.g. `settings.PASSWORD_HASHERS[0]`). If other password hashers are specified in the tuple, Django will also use these if the first hasher doesn't work.

If you want to use a more secure hasher, you can install [Bcrypt⁵](#) using `pip install bcrypt`, and then set the `PASSWORD_HASHERS` to be:

```
PASSWORD_HASHERS = [
    'django.contrib.auth.hashers.BCryptSHA256PasswordHasher',
    'django.contrib.auth.hashers.BCryptPasswordHasher',
    'django.contrib.auth.hashers.PBKDF2PasswordHasher',
    'django.contrib.auth.hashers.PBKDF2SHA1PasswordHasher',
]
```

As previously mentioned, Django by default uses the PBKDF2 algorithm to hash passwords. If you do not specify a `PASSWORD_HASHERS` tuple in `settings.py`, Django will use the `PBKDF2PasswordHasher` password hasher, by default. You can read more about password hashing in the [Django documentation on how Django stores passwords⁶](#).

³https://en.wikipedia.org/wiki/Cryptographic_hash_function

⁴<http://en.wikipedia.org/wiki/PBKDF2>

⁵<https://pypi.python.org/pypi/bcrypt/>

⁶<https://docs.djangoproject.com/en/2.1/topics/auth/passwords/#how-django-stores-passwords>

9.3 Password Validators

As people may be tempted to enter a password that is comparatively easy to guess, Django provides several [password validation⁷](#) methods. In your Django project's `settings.py` module, you will notice a list of nested dictionaries with the name `AUTH_PASSWORD_VALIDATORS`. From the nested dictionaries, you can see that Django 2.x comes with several pre-built password validators for common password checks, such as length. An `OPTIONS` dictionary can be specified for each validator, allowing for easy customisation. If, for example, you wanted to ensure accepted passwords are at least six characters long, you can set `min_length` of the `MinimumLengthValidator` password validator to 6. This can be seen in the example shown below.

```
AUTH_PASSWORD_VALIDATORS = [
    ...
    {
        'NAME': 'django.contrib.auth.password_validation.MinimumLengthValidator',
        'OPTIONS': { 'min_length': 6 },
    },
    ...
]
```

It is also possible to create your own password validators. Although we don't cover the creation of custom password validators in this tutorial, refer to the [official Django documentation on password validators⁸](#) for more information.

9.4 The User Model

The `User` object (located at `django.contrib.auth.models.User`) is considered to be the core of Django's authentication system. A `User` object represents each of the individuals interacting with a Django application. The [Django documentation on User objects⁹](#) states that they are used to allow aspects of the authentication system like access restriction, registration of new user profiles, and the association of creators with site content.

⁷<https://docs.djangoproject.com/en/2.1/ref/settings/#auth-passwordValidators>

⁸<https://docs.djangoproject.com/en/2.1/topics/auth/passwords/#password-validation>

⁹<https://docs.djangoproject.com/en/2.1/topics/auth/default/#user-objects>

The `User` model has five key attributes. They are:

- the `username` for the user account;
- the account's `password`;
- the user's `email address`;
- the user's `first name`; and
- the user's `surname`.

The `User` model also comes with other attributes such as `is_active`, `is_staff` and `is_superuser`. These are boolean fields used to denote whether the account is active, owned by a staff member, or has superuser privileges respectively. Check out the [Django documentation on the user model¹⁰](#) for a full list of attributes provided by the base `User` model.

9.5 Additional User Attributes

If you would like to include other user related attributes than what is provided by the `User` model, you will need to create a model that is *associated* with the `User` model. For our Rango app, we want to include two more additional attributes for each user account. Specifically, we wish to include:

- a `URLField`, allowing a user of Rango to specify their personal website's URL; and
- a `ImageField`, which allows users to specify a picture for their user profile.

This can be achieved by creating an additional model in Rango's `models.py` file. Let's add a new model called `UserProfile`:

¹⁰<https://docs.djangoproject.com/en/2.1/ref/contrib/auth/#django.contrib.auth.models.User>

```
class UserProfile(models.Model):
    # This line is required. Links UserProfile to a User model instance.
    user = models.OneToOneField(User, on_delete=models.CASCADE)

    # The additional attributes we wish to include.
    website = models.URLField(blank=True)
    picture = models.ImageField(upload_to='profile_images', blank=True)

    def __str__(self):
        return self.user.username
```

Note that we reference the `User` model using a one-to-one relationship. Since we reference the default `User` model, we need to import it within the `models.py` file:

```
from django.contrib.auth.models import User
```

For Rango, we've added two fields to complete our user profile. We have also provided a `__str__()` method to return a meaningful value when a string representation of a `UserProfile` model instance is requested.

For the two fields `website` and `picture`, we have set `blank=True` for both. This allows each of the fields to be blank if necessary, meaning that users do not have to supply values for the attributes.

Furthermore, it should be noted that the `ImageField` field has an `upload_to` attribute. The value of this attribute is conjoined with the project's `MEDIA_ROOT` setting to provide a path with which uploaded profile images will be stored. For example, a `MEDIA_ROOT` of `<workspace>/tango_with_django_project/media/` and `upload_to` attribute of `profile_images` will result in all profile images being stored in the directory `<workspace>/tango_with_django_project/media/profile_images/`. Recall that in the [chapter on templates and media files](#) we set up the media root there.



What about Inheriting to Extend?

It may have been tempting to add the additional fields defined above by inheriting from the `User` model directly. However, because other applications may also want access to the `User` model, it is not recommended to use inheritance – but rather to instead use a one-to-one relationship within your database.



Pillow

The Django `ImageField` field makes use of [Pillow¹¹](#), a fork of the *Python Imaging Library (PIL)*. If you have not done so already, install Pillow via Pip with the command `pip install pillow==5.4.1`. If you don't have jpeg support enabled, you can also install Pillow with the command `pip install pillow==5.4.1 --global-option="build_ext" --global-option="--disable-jpeg"`.

You can check what packages are installed in your (virtual) environment by issuing the command `pip list`.

To make the `UserProfile` model data accessible via the Django admin web interface, import the new `UserProfile` model to Rango's `admin.py` module.

```
from rango.models import UserProfile
```

Now you can register the new model with the admin interface, with the following line.

```
admin.site.register(UserProfile)
```



Once again, Migrate!

Remember that your database must be updated with the creation of a new model. Run: `$ python manage.py makemigrations rango` from your terminal or Command Prompt to create the migration scripts for the new `UserProfile` model. Then run: `$ python manage.py migrate` to execute the migration which creates the associated tables within the underlying database.

9.6 Creating a User Registration View and Template

With our authentication infrastructure laid out, we can now begin to build on it by providing users of our application with the opportunity to create user accounts. We can achieve this by creating a new view, template and URL mapping to handle new users registering with Rango.

¹¹<https://pillow.readthedocs.io/en/stable/>



Django User Registration Applications

It is important to note that there are several off the shelf user registration applications available which reduce a lot of the hassle of building your own registration and login forms.

However, it's a good idea to get a feeling for the underlying mechanics before using such applications. This will ensure that you have some sense of what is going on under the hood. *No pain, no gain.* It will also reinforce your understanding of working with forms, how to extend upon the `User` model, and how to upload media files.

To provide user registration functionality, we will now work through the following four steps:

- creating a `UserForm` and `UserProfileForm`;
- adding a view to handle the creation of a new user;
- creating a template that displays the `UserForm` and `UserProfileForm`; and
- mapping a URL to the view created.

As a final step to integrate our new registration functionality, we will also:

- link the index page to the register page.

Creating the `UserForm` and `UserProfileForm`

In `rango/forms.py`, we need to create two classes inheriting from `forms.ModelForm`. We'll be creating one for the base `User` class, as well as one for the new `UserProfile` model that we just created. The two `ModelForm`-inheriting classes allow us to display a HTML form displaying the necessary form fields for a particular model, taking away a significant amount of work for us.

In `rango/forms.py`, let's create our two classes which inherit from `forms.ModelForm`. Add the following code to the module.

```
class UserForm(forms.ModelForm):
    password = forms.CharField(widget=forms.PasswordInput())

    class Meta:
        model = User
        fields = ('username', 'email', 'password')

class UserProfileForm(forms.ModelForm):
    class Meta:
        model = UserProfile
        fields = ('website', 'picture')
```

You'll notice that within both classes, we added a nested `Meta` class. As [the name of the nested class suggests¹²](#), anything within a nested `Meta` class describes additional properties about the particular class to which it belongs. Each `Meta` class must supply a `model` field. In the case of the `UserForm` class the associated model is the `User` model. You also need to specify the `fields` or the `fields to exclude` to indicate which fields associated with the model should be present (or not) on the completed, rendered form.

Here, we only want to show the fields `username`, `email` and `password` associated with the `User` model, and the `website` and `picture` fields associated with the `UserProfile` model. For the `user` field within `UserProfile` model, we will need to make this association when we register the user. This is because when we create a `UserProfile` instance, we won't yet have the `User` instance to refer to.

You'll also notice that `UserForm` includes a definition of the `password` attribute. While a `User` model instance contains a `password` attribute by default, the rendered HTML form element will not hide the password. By overriding the `password` attribute, we can specify that the `CharField` instance should hide a user's input from prying eyes through use of the `PasswordInput()` widget.

Finally, remember to include the now required classes at the top of the `forms.py` module! We've listed them below for your convenience. Integrate these with your existing imports.

¹²<https://www.lexico.com/en/definition/meta>

```
from django.contrib.auth.models import User
from rango.models import UserProfile
```

Creating the register() View

Next, we need to handle both the rendering of the form and the processing of form input data. Within Rango's `views.py`, add `import` statements for the new `UserForm` and `UserProfileForm` classes.

```
from rango.forms import UserForm, UserProfileForm
```

Once you've done that, add the following new view, `register()`.

```
1 def register(request):
2     # A boolean value for telling the template
3     # whether the registration was successful.
4     # Set to False initially. Code changes value to
5     # True when registration succeeds.
6     registered = False
7
8     # If it's a HTTP POST, we're interested in processing form data.
9     if request.method == 'POST':
10         # Attempt to grab information from the raw form information.
11         # Note that we make use of both UserForm and UserProfileForm.
12         user_form = UserForm(request.POST)
13         profile_form = UserProfileForm(request.POST)
14
15         # If the two forms are valid...
16         if user_form.is_valid() and profile_form.is_valid():
17             # Save the user's form data to the database.
18             user = user_form.save()
19
20             # Now we hash the password with the set_password method.
21             # Once hashed, we can update the user object.
22             user.set_password(user.password)
23             user.save()
24
25             # Now sort out the UserProfile instance.
26             # Since we need to set the user attribute ourselves,
27             # we set commit=False. This delays saving the model
```

```
28     # until we're ready to avoid integrity problems.
29     profile = profile_form.save(commit=False)
30     profile.user = user
31
32     # Did the user provide a profile picture?
33     # If so, we need to get it from the input form and
34     # put it in the UserProfile model.
35     if 'picture' in request.FILES:
36         profile.picture = request.FILES['picture']
37
38     # Now we save the UserProfile model instance.
39     profile.save()
40
41     # Update our variable to indicate that the template
42     # registration was successful.
43     registered = True
44 else:
45     # Invalid form or forms - mistakes or something else?
46     # Print problems to the terminal.
47     print(user_form.errors, profile_form.errors)
48 else:
49     # Not a HTTP POST, so we render our form using two ModelForm instances.
50     # These forms will be blank, ready for user input.
51     user_form = UserForm()
52     profile_form = UserProfileForm()
53
54     # Render the template depending on the context.
55     return render(request,
56                   'rango/register.html',
57                   context = {'user_form': user_form,
58                               'profile_form': profile_form,
59                               'registered': registered})
```

While the view looks pretty complicated, it's actually very similar to how we implemented the `add category` and `add page` views. However, the key difference here is that we have to handle two distinct `ModelForm` instances – one for the `User` model, and one for the `UserProfile` model. We also need to handle a user's profile image, if he or she chooses to upload one.

One trick we use here that has caught many people out in the past is the use of `commit=False` when saving the `UserProfile` form. This stops Django from saving the

data to the database in the first instance. Why do this? Remember that information from the `UserProfileForm` form is passed onto a new instance of the `UserProfile` model. The `UserProfile` contains a foreign key reference to the standard Django `User` model – but the `UserProfile` does not provide this information! Attempting to save the new instance in an incomplete state would raise a referential integrity error. *The link between the two models is required.*

To counter this problem, we instruct the `UserProfileForm` to *not* save straight away, as seen on line 29 above. This then allows us to add the `User` reference in with the line `profile.user=user`, as shown on line 30. After this is done, we can then call the `profile.save()` method on line 39 to manually save the new instance to the database. This time, referential integrity is guaranteed, and everything works as it should.

Creating the Registration Template

Now we need to make the template that will be used by the new `register()` view. Create a new template file, `rango/register.html`, and add the following code.

```
1  {% extends 'rango/base.html' %}  
2  {% load staticfiles %}  
3  
4  {% block title_block %}  
5      Register  
6  {% endblock %}  
7  
8  {% block body_block %}  
9    <h1>Register for Rango</h1>  
10   {% if registered %}  
11     Rango says: <strong>thank you for registering!</strong>  
12     <a href="{% url 'rango:index' %}>Return to the homepage.</a><br />  
13   {% else %}  
14     Rango says: <strong>register here!</strong><br />  
15     <form id="user_form" method="post" action="{% url 'rango:register' %}"  
16         enctype="multipart/form-data">  
17  
18       {% csrf_token %}  
19  
20       <!-- Display each form -->
```

```
21     {{ user_form.as_p }}
```

```
22     {{ profile_form.as_p }}
```

```
23
```

```
24     <!-- Provide a button to click to submit the form. -->
```

```
25     <input type="submit" name="submit" value="Register" />
```

```
26 </form>
```

```
27 {% endif %}
```

```
28 {% endblock %}
```



Using the url Template Tag

Note that we are using the `url` template tag in the above template code e.g. `{% url 'rango:register' %}`. This means we will have to ensure that when we map the URL, we name it `register`.

The first thing to note here is that this template makes use of the `registered` variable we used in our view indicating whether registration was successful or not. Note that `registered` must be `False` for the template to display the registration form – otherwise the success message is displayed.

Next, we have used the `as_p` template method on the `user_form` and `profile_form`. This wraps each element in the form in a paragraph (denoted by the `<p>` HTML tag). This ensures that each element appears on a new line.

Finally, in the `<form>` element, we have included the attribute `enctype`. This is because if the user tries to upload a picture, the response from the form may contain binary data. This is different from text data, derived from the user's inputs into the form fields. The response therefore will have to be broken into multiple parts to be transmitted back to the server. As such, we need to denote this with `enctype="multipart/form-data"`. This tells the HTTP client (the web browser) to package and send the data accordingly. Otherwise, the server won't receive all the data submitted by the user.



Multipart Messages and Binary Files

You should be aware of the `enctype` attribute for the `<form>` element. When you want users to upload files from a form – whether it be an image or some other document – it's an absolute *must* to set `enctype` to `multipart/form-data`. This attribute and value combination instructs your browser to send form data in a special way back to the server. Essentially, the data representing your file is split into a series of chunks and sent. For more information, check out this great Stack Overflow answer¹³.

Furthermore, remember to include the CSRF token, i.e. `{% csrf_token %}` within your `<form>` element! If you don't do this, Django's [cross-site forgery](#)¹⁴ protection middleware layer will refuse to accept the form's contents, returning an error. This provides a layer of security, preventing form submissions to succeed from a spoof website. This is often called *phishing*.

The `register()` URL Mapping

With our new view and associated template created, we can now add in the URL mapping. In Rango's URLs module `rango/urls.py`, modify the `urlpatterns` tuple as shown below.

```
urlpatterns = [
    path('', views.index, name='index'),
    path('about/', views.about, name='about'),
    path('category/<slug:category_name_slug>/add_page/', views.add_page,
         name='add_page'),
    path('category/<slug:category_name_slug>', views.show_category,
         name='show_category'),
    path('add_category/', views.add_category, name='add_category'),
    path('register/', views.register, name='register'), # New mapping!
]

]
```

The newly added pattern (at the bottom of the list) points the URL `/rango/register/` to the `register()` view. Also note the inclusion of a `name` for our new URL, `register`,

¹³<http://stackoverflow.com/a/4526286>

¹⁴https://en.wikipedia.org/wiki/Cross-site_request_forgery

which we used in the template when we used the `url` template tag in our new `register.html` template (`{% url 'rango:register' %}`).

Linking Everything Together

Finally, we can add a link pointing to our new registration URL by modifying the `base.html` template. Update `base.html` so that the unordered list of links that will appear on each page contains a link allowing users to register for Rango.

```
<ul>
    <li><a href="{% url 'rango:add_category' %}">Add a New Category</a></li>
    <li><a href="{% url 'rango:about' %}">About</a></li>
    <li><a href="{% url 'rango:index' %}">Index</a></li>
    <li><a href="{% url 'rango:register' %}">Sign Up</a></li>
</ul>
```

Demo

Now everything is plugged together, try it out. Start your Django development server and try to register as a new user. Upload a profile image if you wish. Your registration form should look like the one illustrated in the [figure below](#).

Rango says: [register here!](#)

Username: Required. 150 characters or fewer. Letters, digits and @/./+/-/_ only.

Email address:

Password:

Website:

Picture: No file chosen

[Register](#)

• [Python](#)
• [Django](#)
• [Other Frameworks](#)

• [Add New Category](#)
• [About](#)
• [Index](#)
• [Sign Up](#)

A screenshot illustrating the basic registration form you create as part of this tutorial.

Upon seeing the message indicating your details were successfully registered, the database should have a new entry in the `User` and `UserProfile` models. Check that this is the case by going into the Django Admin interface.

9.7 Implementing Login Functionality

With the ability to register accounts completed, we now need to provide users of Rango with the ability to login. To achieve this, we'll need to undertake the workflow outlined below.

- Create a login view to handle the processing of user credentials.
- Create a login template to display the login form.
- Map the login view to a URL.
- Provide a link to login from the index page.

Creating the login() View

First, open up Rango's views module at `rango/views.py` and create a new view called `user_login()`. This view will handle the processing of data from our subsequent login form, and attempt to log a user in with the given details.

```
def user_login(request):
    # If the request is a HTTP POST, try to pull out the relevant information.
    if request.method == 'POST':
        # Gather the username and password provided by the user.
        # This information is obtained from the login form.
        # We use request.POST.get('<variable>') as opposed
        # to request.POST['<variable>'], because the
        # request.POST.get('<variable>') returns None if the
        # value does not exist, while request.POST['<variable>']
        # will raise a KeyError exception.
        username = request.POST.get('username')
        password = request.POST.get('password')

        # Use Django's machinery to attempt to see if the username/password
        # combination is valid - a User object is returned if it is.
        user = authenticate(username=username, password=password)

        # If we have a User object, the details are correct.
        # If None (Python's way of representing the absence of a value), no user
        # with matching credentials was found.
        if user:
            # Is the account active? It could have been disabled.
            if user.is_active:
                # If the account is valid and active, we can log the user in.
                # We'll send the user back to the homepage.
                login(request, user)
                return redirect(reverse('rango:index'))
            else:
                # An inactive account was used - no logging in!
                return HttpResponse("Your Rango account is disabled.")
        else:
            # Bad login details were provided. So we can't log the user in.
            print(f"Invalid login details: {username}, {password}")
            return HttpResponse("Invalid login details supplied.")

    # The request is not a HTTP POST, so display the login form.
```

```
# This scenario would most likely be a HTTP GET.  
else:  
    # No context variables to pass to the template system, hence the  
    # blank dictionary object...  
    return render(request, 'rango/login.html')
```

As before, this view may seem rather complex as it has to handle a variety of scenarios. As shown in previous examples, the `user_login()` view handles form rendering and processing – where the form this time contains `username` and `password` fields.

First, if the view is accessed via the HTTP GET method, then the login form is displayed. However, if the form has been posted via the HTTP POST method, then we can handle processing the form.

If a valid form is sent via a POST request, the `username` and `password` are extracted from the form. These details are then used to attempt to authenticate the user. The Django function `authenticate()` checks whether the `username` and `password` provided matches to a valid user account. If a valid user exists with the specified password, then a `User` object is returned, otherwise `None` is returned.

If we retrieve a `User` object, we can then check if the account is active or inactive – if active, then we can issue the Django function `login()`, which officially signifies to Django that the user is to be logged in.

However, if an invalid form is sent (since the user did not add both a `username` and `password`) the login form is presented back to the user with error messages (i.e. an invalid `username/password` combination was provided).

You'll also notice that we make use of the helper function `redirect()`. We used this back in the [chapter on creating forms](#), and it simply tells the client's browser to redirect to the URL that you provide as an argument. Note that this will return a HTTP status code of 302, which denotes a redirect, as opposed to an status code of 200 (success). See the [Django documentation on redirection](#)¹⁵ to learn more.

Finally, we use `reverse()` again to obtain the URL of the Rango application. This looks up the URL patterns in Rango's `urls.py` module to find a URL called `rango:index`, and substitutes in the corresponding pattern. This means that if we subsequently change the URL mapping, our new view won't break.

¹⁵<https://docs.djangoproject.com/en/2.1/topics/http/shortcuts/#redirect>

Django provides all of these functions and classes. As such, you'll need to import them. The following `import` statements must now be added to the top of `rango/views.py`.

```
from django.contrib.auth import authenticate, login
from django.http import HttpResponseRedirect
from django.urls import reverse
from django.shortcuts import redirect
```

You will find that some of these `import` statements are already present in Rango's `views.py` module from earlier on – check to see that you are not repeating yourself! For example, `HttpResponse` should be present from earlier.

Creating a Login Template

With our new view created, we'll need to create a new template, `login.html`. It will allow users to enter their credentials. While we know that the template will live in the `templates/rango/` directory, we'll leave you to figure out the name of the file. Look at the code example above to work out the name based upon the code for the new `user_login()` view. In your new template file, add the following code.

```
{% extends 'rango/base.html' %}
{% load staticfiles %}

{% block title_block %}
    Login
{% endblock %}

{% block body_block %}
<h1>Login to Rango</h1>
<form id="login_form" method="post" action="{% url 'rango:login' %}>
    {% csrf_token %}
    Username: <input type="text" name="username" value="" size="50" />
    <br />
    Password: <input type="password" name="password" value="" size="50" />
    <br />
    <input type="submit" value="submit" />
</form>
{% endblock %}
```

Ensure that you match up the `input name` attributes to those that you specified in the `user_login()` view. For example, `username` matches to the `username`, and `password` matches to the user's password. Don't forget the `{% csrf_token %}`, either!

Mapping the Login View to a URL

With your login template created, we can now match up the `user_login()` view to a URL. Modify Rango's `urls.py` module so that the `urlpatterns` list contains the following mapping.

```
path('login/', views.user_login, name='login'),
```

Linking Together

Our final step is to provide users of Rango with a handy link to access the login page. To do this, we'll edit the `base.html` template inside of the `templates/rango/` directory. Add the following link to your list.

```
<ul>
  ...
  <li><a href="{% url 'rango:login' %}">Login</a></li>
</ul>
```

If you like, you can also modify the header of the index page to provide a personalised message if a user is logged in, and a more generic message if the user isn't. Within the `index.html` template, find the message, as shown in the code snippet below.

```
hey there partner!
```

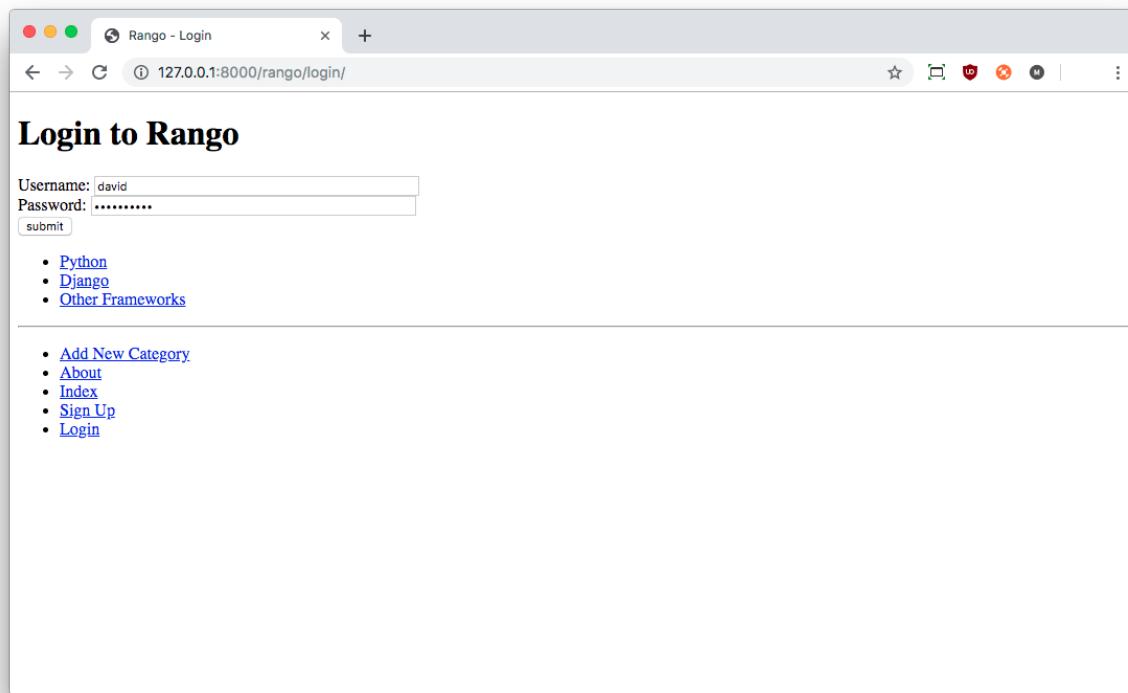
This line can then be replaced with the following code.

```
{% if user.is_authenticated %}
    howdy {{ user.username }}!
{% else %}
    hey there partner!
{% endif %}
```

As you can see, we have used Django's template language to check if the user is authenticated with `{% if user.is_authenticated %}`. If a user is logged in, then Django gives us access to the `user` object. We can tell from this object if the user is logged in (`authenticated`). If he or she is logged in, we can also obtain details about him or her. In the example above, the user's username will be presented to them if logged in – otherwise the generic `hey there partner!` message will be shown.

Demo

Start the Django development server and attempt to login to the application. The figure below shows a screenshot of the login page as it should look.



Screenshot of the login page, complete with username and password fields.

With this completed, user logins should now be working. To test everything out, try starting Django's development server and attempt to register a new account. After successful registration, you should then be able to login with the details you just provided. Check the `index()` view – when you log in, do you see it greeting you with your username?

9.8 Restricting Access

Now that users can login to Rango, we can now go about restricting access to particular parts of the application as per the specification. This means that only registered users will be able to create categories and add new pages to existing categories. With Django, there are several ways in which we can achieve this goal.

- In the template, we could use the `{% if user.is_authenticated %}` template tag to modify how the page is rendered (shown already).
- In the view, we could directly examine the `request` object and check if the user is authenticated.
- Or, we could use a *decorator* function `@login_required` provided by Django that checks if the user is authenticated.

The direct approach checks to see whether a user is logged in via the built-in Django `user.is_authenticated()` method. The `user` object is available via the `request` object passed into a view. The following example demonstrates this approach.

```
def some_view(request):
    if not request.user.is_authenticated():
        return HttpResponse("You are logged in.")
    else:
        return HttpResponse("You are not logged in.")
```

The third approach uses [Python decorators¹⁶](#). Decorators are named after a [software design pattern by the same name¹⁷](#). They can dynamically alter the functionality of a function, method or class without having to directly edit the source code of the given function, method or class.

¹⁶<http://wiki.python.org/moin/PythonDecorators>

¹⁷http://en.wikipedia.org/wiki/Decorator_pattern

Django provides a decorator called `login_required()`, which we can attach to any view where we require the user to be logged in. If a user is not logged in and attempts to access a view decorated with `login_required()`, they are then redirected to another page (that you can set¹⁸) – typically the login page.

Restricting Access with a Decorator

To try this out, create a view in Rango's `views.py` module called `restricted()`, and add the following code.

```
@login_required  
def restricted(request):  
    return HttpResponse("Since you're logged in, you can see this text!")
```

Note that to use a decorator, you place it *directly above* the function signature, and put a @ before naming the decorator. Python will execute the decorator before executing the code of your function/method. As a decorator is still a function, you'll still have to import it if it resides within an external module. As `login_required()` exists elsewhere, the following `import` statement is required at the top of `views.py`.

```
from django.contrib.auth.decorators import login_required
```

We'll also need to add in another URL mapping to Rango's `urlpatterns` list in the `urls.py` file. Add the following line of code.

```
path('restricted/', views.restricted, name='restricted'),
```

To make the new restricted page accessible, you can also add another link to Rango's `base.html` template.

¹⁸<https://docs.djangoproject.com/en/2.1/ref/settings/#login-url>

```
<ul>
  ...
  <li><a href="{% url 'rango:restricted' %}">Restricted Page</a></li>
</ul>
```

We will also need to robustly handle scenarios where users attempt to access the `restricted()` view, but are not logged in. What do we do with the user? The simplest approach is to redirect them to a page they can access, e.g. the registration page. Django allows us to specify this in our project's `settings.py` module, located in the project configuration directory. In `settings.py`, define the variable `LOGIN_URL` with the URL you'd like to redirect users to that aren't logged in. In Rango's case, we can redirect to the login page located at `/rango/login/`:

```
LOGIN_URL = '/rango/login/'
```

However, hardcoding URLs like this, regardless of where they are placed, is bad practice. We want to avoid this at all costs! Thankfully, you can rewrite this using the URL mapping that you defined in Rango's `urls.py` module, like so.

```
LOGIN_URL = 'rango:login'
```

Much better! Django supports URL names here, too. By entering a URL name here, Django will direct users to the right place, even if the URL for logging in is changed. Regardless of what you enter as the value of `LOGIN_URL`, providing a value here ensures that the `login_required()` decorator will redirect any user not logged in to the URL you specify.

9.9 Logging Out

To enable users to log out gracefully, it would be nice to provide a logout option to users. Django comes with a handy `logout()` function that takes care of ensuring that the users can properly and securely log out. The `logout()` function will ensure that their session is ended, and that if they subsequently try to access a view that requires authentication then they will not be able to access it, unless they then decide to log back in.

To provide logout functionality in `rango/views.py`, add the view called `user_logout()` with the following code.

```
# Use the login_required() decorator to ensure only those logged in can
# access the view.
@login_required
def user_logout(request):
    # Since we know the user is logged in, we can now just log them out.
    logout(request)
    # Take the user back to the homepage.
    return redirect(reverse('rango:index'))
```

You'll also need to import the `logout` function at the top of `views.py`.

```
from django.contrib.auth import logout
```

If you like, you can also append the `logout` import to the end of your existing import from the same module, like in the following example.

```
from django.contrib.auth import authenticate, login, logout
```

With the view created, map the URL `/rango/logout/` to the `user_logout()` view by modifying the `urlpatterns` list in Rango's `urls.py`.

```
path('logout/', views.user_logout, name='logout'),
```

Finally, we can add a further link to our `base.html` template.

```
<ul>
  ...
  <li><a href="{% url 'rango:logout' %}">Logout</a></li>
  ...
</ul>
```

9.10 Tidying up the `base.html` Hyperlinks

Now that all the machinery for logging a user out has been completed, we can add some finishing touches. Providing a link to log out as we did above is good, but is it smart? If a user is not logged in, what would the point be of providing a link to logout? Conversely, if a user is already logged in, would it be sensible to provide a link to login? Perhaps not – you can make the argument that the links presented to the user in `base.html` should change *depending on their circumstances*. We need to do some more work to make sure our Rango app satisfies the requirements we set off with.

Let's modify the links in the `base.html` template. We will employ the `user` object in the template's context to determine what links we want to show to a particular user. Find your growing list of links at the bottom of the page, and replace it with the following code. Feel free to copy and paste things around here, because all you are doing here is adding in some conditional statements, and moving existing links around.

```
<ul>
{% if user.is_authenticated %}
    <!-- Show these links when the user is logged in -->
    <li><a href="{% url 'rango:restricted' %}">Restricted Page</a></li>
    <li><a href="{% url 'rango:logout' %}">Logout</a></li>
{% else %}
    <!-- Show these links when the user is NOT logged in -->
    <li><a href="{% url 'rango:register' %}">Sign Up</a></li>
    <li><a href="{% url 'rango:login' %}">Login</a></li>
{% endif %}
    <!-- Outside the conditional statements, ALWAYS show -->
    <li><a href="{% url 'rango:add_category' %}">Add New Category</a></li>
    <li><a href="{% url 'rango:about' %}">About</a></li>
    <li><a href="{% url 'rango:index' %}">Index</a></li>
</ul>
```

This markup and template code states that when a user is authenticated and logged in, he or she can see the `Restricted Page` and `Logout` links. If he or she isn't logged in, `Login` and `Sign Up` are presented. As `Add New Category`, `About` and `Index` are not within

the template conditional blocks, these links are available to both anonymous and logged in users.

9.11 Taking it Further

In this chapter, we've covered several important aspects of managing user authentication within Django. We've covered the basics of including the built-in Django `django.contrib.auth` application into our project. Additionally, we have also shown how to implement a user profile model that can provide additional fields to the base `django.contrib.auth.models.User` model. We have also detailed how to setup the functionality to allow user registrations, login, logout, and to control access. For more information about user authentication and registration consult the [Django documentation on authentication¹⁹](#).

Many web applications however take the concepts of user authentication further. For example, you may require different levels of security when registering users, by ensuring a valid e-mail address is supplied. While we could implement this functionality, why reinvent the wheel when such functionality already exists? The `django-registration-redux` app has been developed to greatly simplify the process of adding extra functionality related to user authentication. We cover how you can use this package in a [following chapter](#).

¹⁹<https://docs.djangoproject.com/en/2.1/topics/auth/>



Exercises

For now, work on the following two exercises to reinforce what you've learnt in this chapter.

- Customise Rango so that only registered users can add categories and pages, while those unregistered can only view or use the categories and pages. Remember, you'll need to add some code to `views.py`, and you'll also need to update your template code around the links in `base.html` – as well as the `Add Page` link in `category.html`!
- Keep your templating knowledge fresh by converting the restricted page view to use a template. Call the template `restricted.html`, and ensure that it too extends from Rango's `base.html` template. Make sure you give this page a title block value of `Restricted Page`.



Test your Implementation

We've implemented a series of unit tests to allow you to check your implementation up until this point. [Follow the guide we provided earlier](#), using the test module `tests_chapter9.py`. How does your implementation stack up against our tests? The tests in this chapter are rather extensive! Remember that your implementation should have fully completed the exercises listed above for the tests to pass.

10. Cookies and Sessions

In this chapter, we will be touching on the basics of handling *sessions* and storing *cookies*. Both go hand in hand with each other and provide the basis for persisting the current state of the application. In the previous chapter, the Django framework used sessions and cookies to handle the login and logout functionality. However, all this was done behind the scenes. Here we will explore exactly what is going on under the hood, and how we can use cookies ourselves for other purposes.

10.1 Cookies, Cookies Everywhere!

Whenever a request to a website is made, the webserver returns the content of the requested page. In addition, one or more cookies may also be sent as part of the request. Consider a cookie as a small piece of information sent from the server to the client. When a request is about to be sent, the client checks to see if any cookies that match the address of the server exist on the client. If so, they are included in the request. The server can then interpret the cookies as part of the request's context and generate an appropriate response.

As an example, you may login to a site with a particular username and password. When you have been authenticated, a cookie may be returned to your browser containing your username, indicating that you are now logged into the site. At every request, this information is passed back to the server where your login information is used to render the appropriate page – perhaps including your username in particular places on the page. Your session cannot last forever, however – cookies *have* to expire at some point in time – they cannot be of infinite length. A web application containing sensitive information may expire after only a few minutes of inactivity. A different web application with trivial information may expire half an hour after the last interaction – or even weeks into the future.



Cookie Origins

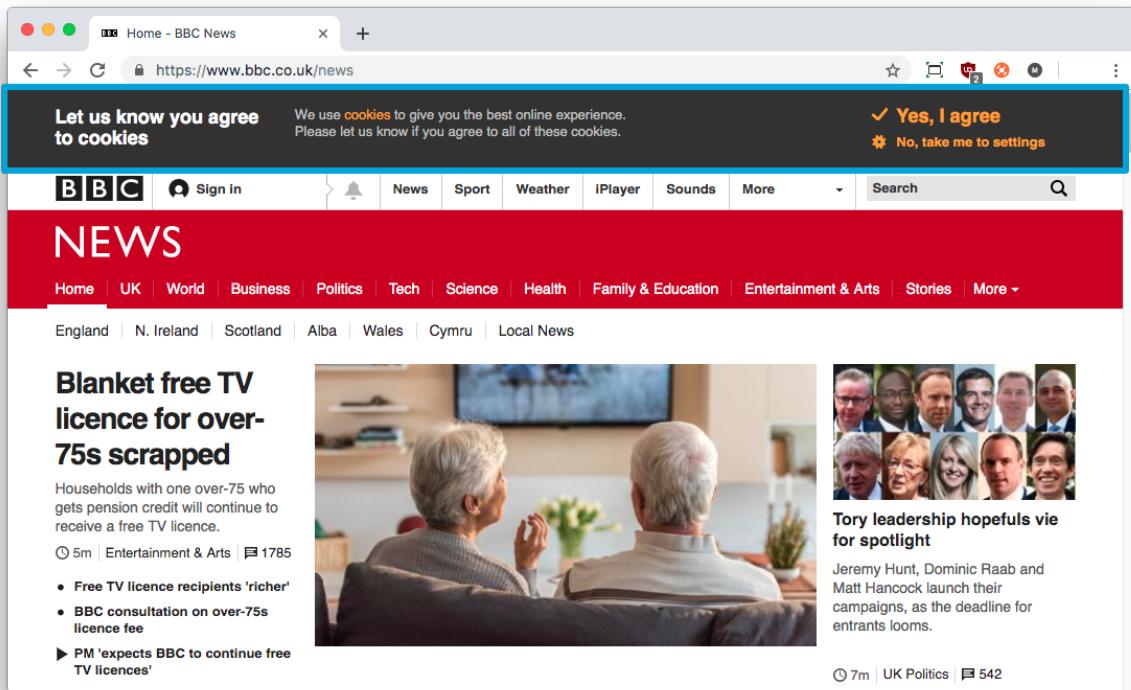
The term *cookie* wasn't derived from the food that you eat, but from the term *magic cookie*, a packet of data a program receives and sends again unchanged. In 1994, MCI sent a request to *Netscape Communications* to implement a way of implementing persistence across HTTP requests. This was in response to their need to reliably store the contents of a user's virtual shopping basket for an e-commerce solution they were developing. Netscape programmer Lou Montulli took the concept of a magic cookie and applied it to web communications.

You can find out more about [cookies and their history on Wikipedia¹](#). Of course, with such a great idea came a software patent – and you can read [US patent 5774670²](#) that was submitted by Montulli himself.

The passing of information in the form of cookies can open up potential security holes in your web application's design. This is why developers of web applications need to be extremely careful when using cookies. When using cookies, a designer must always ask himself or herself: *does the information you want to store as a cookie really need to be sent and stored on a client's machine?* In many cases, there are more secure solutions to the problem. Passing a user's credit card number on an e-commerce site as a cookie, for example, would be highly insecure. What if the user's computer is compromised? A malicious program could take the cookie. From there, hackers would have his or her credit card number – all because your web application's design is fundamentally flawed. This chapter examines the fundamental basics of client-side cookies – and server-side session storage for web applications.

¹http://en.wikipedia.org/wiki/HTTP_cookie#History

²<https://patents.google.com/patent/US5774670A/en>



A screenshot of the BBC News website (hosted in the United Kingdom) with the cookie warning message presented at the top of the page.



Cookies in the EU

In 2011, the European Union (EU) introduced an EU-wide '*cookie law*', where all hosted sites within the EU should present a cookie warning message when a user visits the site for the first time. The [figure above](#) demonstrates such a warning on the BBC News website. You can read about [the law here³](#).

If you are developing a site, you'll need to be aware of this law, and other laws especially regarding accessibility and more recently explainability.

³<https://ico.org.uk/for-organisations/guide-to-pecr/cookies-and-similar-technologies/>

10.2 Sessions and the Stateless Protocol

All communication between web browsers (clients) and servers is achieved through the [HTTP protocol⁴](#). As previously mentioned, HTTP is a [stateless protocol⁵](#). This means that a client computer running a web browser must establish a new network connection (a [TCP⁶](#) connection) to the server each time a resource is requested (HTTP GET) or sent (HTTP POST)⁷.

Without a persistent connection between the client and server, the software on both ends cannot simply rely on connections alone to *hold session state*. For example, the client would need to tell the server each time who is logged on to the web application on a particular computer. This is known as a form of *dialogue* between the client and server and is the basis of a *session* – a [semi-permanent exchange of information⁸](#). Being a stateless protocol, HTTP makes holding session state challenging, but there are luckily several techniques we can use to circumnavigate this problem.

The most commonly used way of holding state is through the use of a *session ID* stored as a cookie on a client's computer. A session ID can be considered as a token (a sequence of characters, or a *string*) to identify a unique session within a particular web application. Instead of storing all kinds of information as cookies on the client (such as usernames, names, or passwords), only the session ID is stored, which can then be mapped to a data structure on the webserver. Within that data structure, you can store all of the information you require. This approach is a **much more secure** way to store information about users. This way, the information cannot be compromised by an insecure client or a connection which is being snooped during transmission.

If you're using a modern browser that's properly configured, it'll support cookies. Most websites that you visit will create a new *session* for you when you visit. You can

⁴http://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol

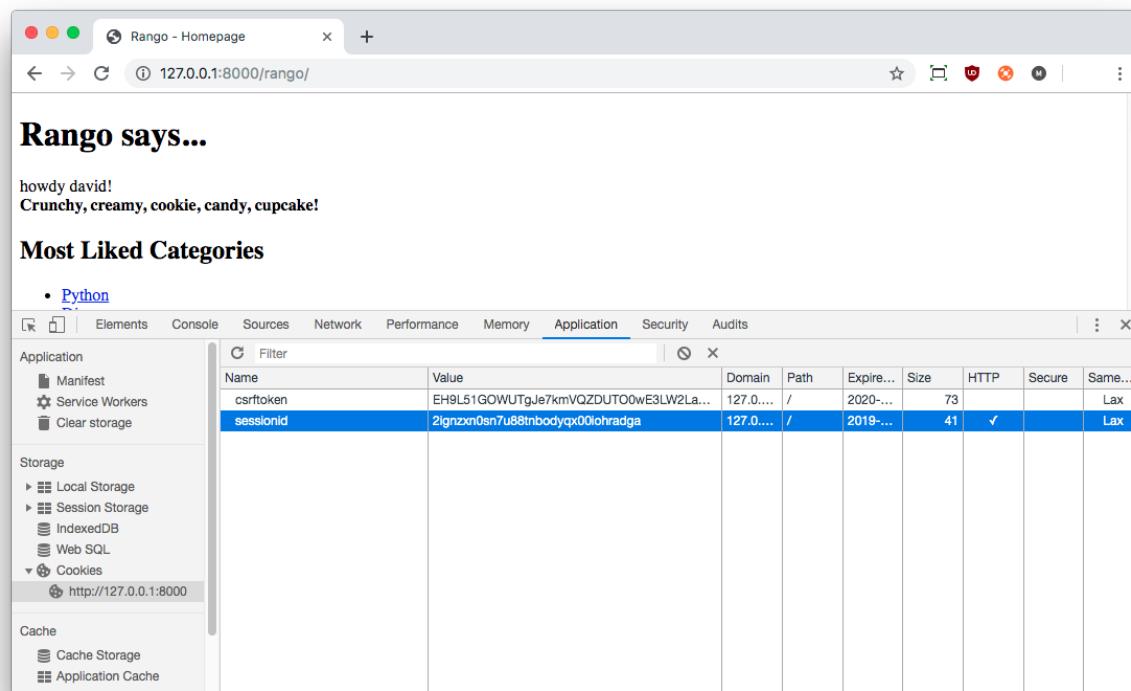
⁵http://en.wikipedia.org/wiki/Stateless_protocol

⁶https://en.wikipedia.org/wiki/Transmission_Control_Protocol

⁷The latest version of the HTTP standard HTTP 1.1 supports the ability for multiple requests to be sent in one TCP network connection. This provides huge improvements in performance, especially over high-latency network connections (such as via a traditional dial-up modem and satellite). This is referred to as *HTTP pipelining*, and you can read more about this technique on [Wikipedia](#).

⁸[http://en.wikipedia.org/wiki/Session_\(computer_science\)](http://en.wikipedia.org/wiki/Session_(computer_science))

see this for yourself now – check out the [screenshot below](#). For Google Chrome, you can view cookies for the currently open website by accessing Chrome’s Developer Tools, by accessing *Chrome Settings > More Tools > Developer Tools*. When the Developer Tools pane opens, click the *Application* tab, and look for *Cookies* down the left-hand side, within the *Storage* subcategory. If you’re running this with a Rango page open [on your computer⁹](#), you will then see a cookie called `sessionid`. The `sessionid` cookie contains a series of letters and numbers that Django uses to uniquely identify your computer with a given session. With this session ID, all your session details can be accessed – but they are only stored on the *server-side*.



A screenshot of Google Chrome’s Developer Tools with the `sessionid` cookie highlighted.

⁹<http://127.0.0.1:8000/rango/>



Without Cookies

An alternative way of persisting state information *without cookies* is to encode the Session ID within the URL. For example, you may have seen PHP pages with URLs like this one: <http://www.site.com/index.php?sessid=someseeminglyrandomandlongstring>. This means you don't need to store cookies on the client machine, but the URLs become rather ugly. These URLs go against the principles of Django, which is to provide clean, simple and human-friendly URLs.

10.3 Setting up Sessions in Django

Although this should already be setup and working correctly, it's nevertheless good practice to learn which Django modules provide which functionality. In the case of sessions, Django provides [middleware¹⁰](#) that implements session functionality.

To check that everything is in order, open your Django project's `settings.py` file. Within the file, locate the `MIDDLEWARE` list. You should find within this list a module represented by the string `django.contrib.sessions.middleware.SessionMiddleware`. If you can't see it, add it to the list now. It is the `SessionMiddleware` middleware that enables the creation of unique `sessionid` cookies.

The `SessionMiddleware` is designed to work flexibly with different ways to store session information. Many approaches can be taken – you could store everything in a file, in a database, or even in an in-memory cache. The most straightforward approach is to use the `django.contrib.sessions` application to store session information in a Django model/database. Specifically, we are referring to the Django model `django.contrib.sessions.models.Session`. To use this approach, you'll also need to make sure that `django.contrib.sessions` is in the `INSTALLED_APPS` tuple of your Django project's `settings.py` file. Remember, if you add the application now, you'll need to update your database with the usual migration commands.

¹⁰<https://docs.djangoproject.com/en/2.1/topics/http/middleware/>



Caching Sessions

If you want faster performance, you may want to consider a cached approach for storing session information. You can check out the [Django documentation for advice on cached sessions¹¹](#).

10.4 A Cookie Tasting Session

While all modern web browsers support cookies, certain cookies may get blocked depending on your browser's security level. Check that you've enabled support for cookies before continuing. However, it's likely you're ready to go.

Testing Cookie Functionality

To test out cookies, you can make use of some convenience methods provided by Django's `request` object. The three of particular interest to us are `set_test_cookie()`, `test_cookie_worked()` and `delete_test_cookie()`. In one view, you will need to set the test cookie. In another, you'll need to test that the cookie exists. Two different views are required for testing cookies because you need to wait to see if the client has accepted the cookie from the server.

We'll use two pre-existing views for this simple exercise, `index()` and `about()`. Instead of displaying anything on the pages themselves, we'll be making use of the terminal output from the Django development server to verify whether cookies are working correctly.

In Rango's `views.py` file, locate your `index()` view. Add the following line to the view. To ensure the line is executed, make sure you put it in before the `return` line.

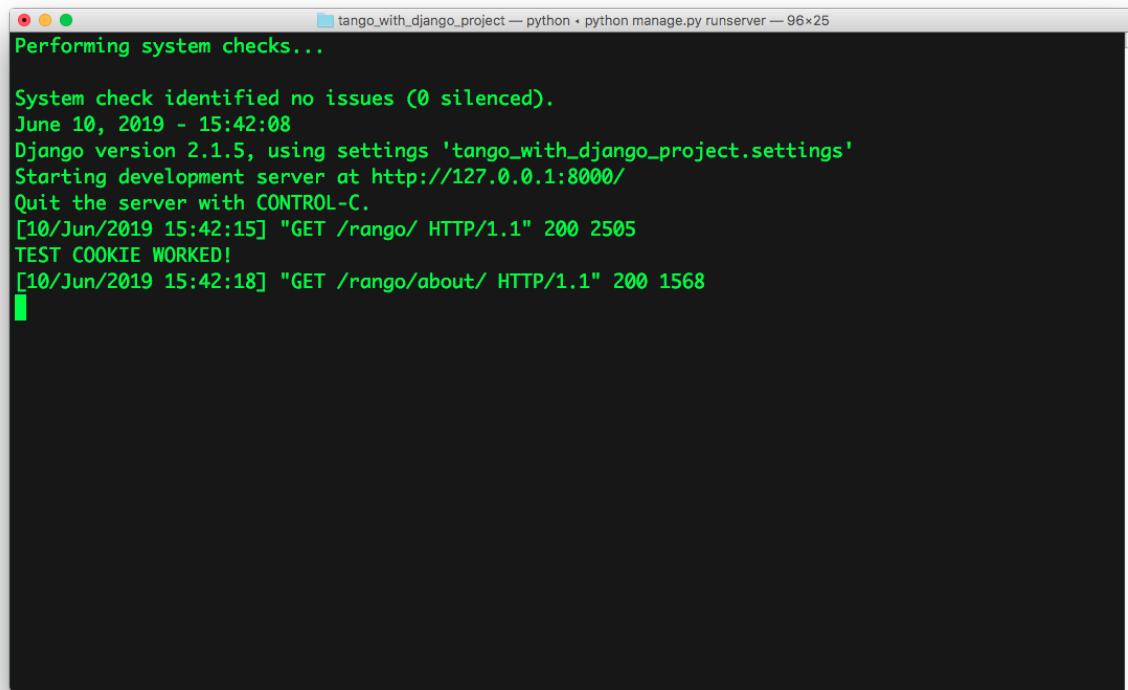
```
request.session.set_test_cookie()
```

In the `about()` view, add the following three lines. Again, make sure these go *before* the `return` statement to ensure they are executed. A `return` statement curtails execution at that point.

¹¹<https://docs.djangoproject.com/en/2.1/topics/http/sessions/#using-cached-sessions>

```
if request.session.test_cookie_worked():
    print("TEST COOKIE WORKED!")
    request.session.delete_test_cookie()
```

With these small changes saved, run the Django development server and navigate to Rango's homepage, `http://127.0.0.1:8000/rango/`. Now navigate to the about page, you should see `TEST COOKIE WORKED!` appear in your Django development server's console, like in the [figure below](#).

A screenshot of a terminal window titled "tango_with_django_project — python · python manage.py runserver — 96x25". The window displays the following text:

```
Performing system checks...

System check identified no issues (0 silenced).
June 10, 2019 - 15:42:08
Django version 2.1.5, using settings 'tango_with_django_project.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
[10/Jun/2019 15:42:15] "GET /rango/ HTTP/1.1" 200 2505
TEST COOKIE WORKED!
[10/Jun/2019 15:42:18] "GET /rango/about/ HTTP/1.1" 200 1568
```

A screenshot of the Django development server's console output with the `TEST COOKIE WORKED!` message when the about view is requested.

If the message isn't displayed, you'll want to check your browser's security settings. The settings may be preventing the browser from accepting the cookie.

10.5 Client-Side Cookies: A Site Counter Example

Now we know how cookies work, let's implement a very simple site visit counter. To achieve this, we're going to be creating two cookies: one to track the number of

times the user has visited the Rango app, and the other to track the last time they accessed the site. Keeping track of the date and time of the last access will allow us to only increment the site counter once per day (for example) and thus avoid people spamming the site to increment the counter.

The sensible place to assume where a user enters the Rango site is at the index page. Open Rango's `views.py` file. Let's first make a function – given a handle to both the `request` and `response` objects – to handle cookies (`visitor_cookie_handler()`). We can then make use of this function in Rango's `index()` view. In `views.py`, add in the following function. Note that it is not technically a view, because it does not return a `response` object – it is just a *helper function*¹².

```
def visitor_cookie_handler(request, response):
    # Get the number of visits to the site.
    # We use the COOKIES.get() function to obtain the visits cookie.
    # If the cookie exists, the value returned is casted to an integer.
    # If the cookie doesn't exist, then the default value of 1 is used.
    visits = int(request.COOKIES.get('visits', '1'))

    last_visit_cookie = request.COOKIES.get('last_visit', str(datetime.now()))
    last_visit_time = datetime.strptime(last_visit_cookie[:-7],
                                         '%Y-%m-%d %H:%M:%S')

    # If it's been more than a day since the last visit...
    if (datetime.now() - last_visit_time).days > 0:
        visits = visits + 1
        # Update the last visit cookie now that we have updated the count
        response.set_cookie('last_visit', str(datetime.now()))
    else:
        # Set the last visit cookie
        response.set_cookie('last_visit', last_visit_cookie)

    # Update/set the visits cookie
    response.set_cookie('visits', visits)
```

This helper function takes the `request` and `response` objects – because we want to be able to access the incoming cookies from the `request`, and add or update cookies in the `response`. In the function, you can see that we call the `request.COOKIES.get()` function, which is a further helper function provided by Django. If the cookie exists,

¹²<https://web.cs.wpi.edu/~cs1101/a05/Docs/creating-helpers.html>

it returns the value. If it does not exist, we can provide a default value. Once we have the values for each cookie, we can calculate whether a day (.days) has elapsed between the last visit.

If you want to test this code out without having to wait a day, change days to seconds. That way the visit counter can be updated every second, as opposed to every day.

Note that all cookie values are returned as strings; *do not assume that a cookie storing whole numbers will return an integer.* You have to manually cast this to the correct type yourself because there's no place in which to store additional information within a cookie telling us of the value's type.

If a cookie does not exist, you can create a cookie with the `set_cookie()` method of the `response` object you create. The method takes in two values, the name of the cookie you wish to create (as a string), and the value of the cookie. In this case, it doesn't matter what type you pass as the value – Django will automatically cast the value to a string.

Since we are using the `datetime` object, we need to `import` this into `views.py` at the top of the module. The `import` statement is shown below.

```
from datetime import datetime
```

Next, update the `index()` view to call the `cookie_handler_function()` helper function. To do this, we need to extract the `response` first. Note that we also remove the `set_test_cookie()` method call. You should also remove the test cookie code in the `about()` view at this point, too.

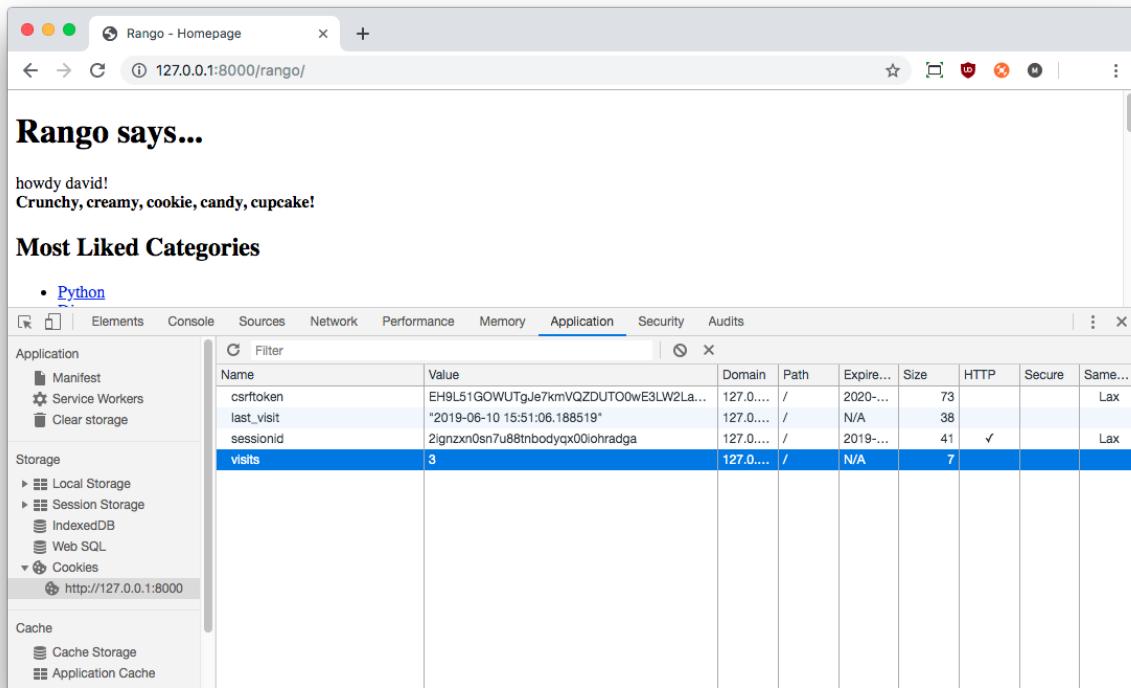
```
def index(request):
    category_list = Category.objects.order_by('-likes')[:5]
    page_list = Page.objects.order_by('-views')[:5]

    context_dict = {}
    context_dict['boldmessage'] = 'Crunchy, creamy, cookie, candy, cupcake!'
    context_dict['categories'] = category_list
    context_dict['pages'] = page_list

    # Obtain our Response object early so we can add cookie information.
    response = render(request, 'rango/index.html', context=context_dict)
```

```
# Call the helper function to handle the cookies
visitor_cookie_handler(request, response)

# Return response back to the user, updating any cookies that need changed.
return response
```



A screenshot of Google Chrome with the Developer Tools open showing the cookies for Rango, using the Django development server at 127.0.0.1. Note the visits cookie – the user here has visited a total of three times, with each visit at least one day apart. Note also the last_visit cookie, containing a string representation of the date and time this user accessed the index page.

Now if you visit the Rango homepage and open the cookie inspector provided by your browser (e.g. Google Chrome's Developer Tools), you should be able to see the cookies visits and last_visit. The figure above demonstrates the cookies in action. Instead of using the developer tools, update the index.html template and add `<p>Visits: {{ visits }}</p>` to show the number of visits. For this to work, you will also need to update the context dictionary to include the visits value (by setting the visits value in the context_dict for the index() view to `int(request.COOKIES.get('visits', '1'))`).

10.6 Session Data

The previous example shows how we can store and manipulate client-side cookies – or the data stored on the client. However, a more secure way to save session information is to store any such data on the server-side. We can then use the session ID cookie that is stored on the client-side (but is effectively anonymous) as the key to access the data.

To use session-based cookies you need to perform the following steps.

1. Make sure that the `MIDDLEWARE_CLASSES` list found in the `settings.py` module contains `django.contrib.sessions.middleware.SessionMiddleware`.
2. Configure your session backend. Make sure that `django.contrib.sessions` is in your `INSTALLED_APPS` in `settings.py`. If not, add it, and run the database migration command `python manage.py migrate`.
3. By default, a database backend is assumed, but you might want to a different setup (i.e. a cache). See the [Django documentation on sessions](#) for other back-end configurations¹³.

Instead of storing the cookies directly in the request (and thus on the client's machine), you can access server-side data via the method `request.session.get()` and store them with `request.session[]`. Note that a session ID cookie is still used to remember the client's machine (so technically a browser-side cookie exists). However, all the user/session data is stored server-side. Django's session middleware handles the client-side cookie and the storing of the user/session data.

To use the server-side data, we need to refactor `views.py`. First, we need to update the `visitor_cookie_handler()` function so that it accesses the cookies on the server-side. We can do this by calling `request.session.get()`, and store them by placing them in the dictionary `request.session[]`. To help us along, we have made a helper function called `get_server_side_cookie()` that asks the request for a cookie. If the cookie is in the session data, then its value is returned. Otherwise, the default value is returned.

¹³<https://docs.djangoproject.com/en/2.1/topics/http/sessions/>

Since all the cookies are stored server-side, we won't be changing the response directly. Because of this, we can remove `response` from the `visitor_cookie_handler()` function definition.

```
# A helper method
def get_server_side_cookie(request, cookie, default_val=None):
    val = request.session.get(cookie)
    if not val:
        val = default_val
    return val

# Updated the function definition
def visitor_cookie_handler(request):
    visits = int(get_server_side_cookie(request, 'visits', '1'))
    last_visit_cookie = get_server_side_cookie(request,
                                                'last_visit',
                                                str(datetime.now()))
    last_visit_time = datetime.strptime(last_visit_cookie[:-7],
                                        '%Y-%m-%d %H:%M:%S')

    # If it's been more than a day since the last visit...
    if (datetime.now() - last_visit_time).days > 0:
        visits = visits + 1
        # Update the last visit cookie now that we have updated the count
        request.session['last_visit'] = str(datetime.now())
    else:
        # Set the last visit cookie
        request.session['last_visit'] = last_visit_cookie

    # Update/set the visits cookie
    request.session['visits'] = visits
```

Now that we have updated the handler function, we can update the `index()` view. You first should remove the `response` parameter from the `visitor_cookie_handler()` function definition. After editing, without `response`, the definition should look like `visitor_cookie_handler(request)`. You should then update your `context_dict` assignment for `visits` to `request.session['visits']`, ensuring that this is executed *after* the call to the `visitor_cookie_handler()` function.

You should also ensure that all of these lines are executed *before* `render()` is called, or your changes won't be executed. The `index()` view should look like the code

below. Notice that the order in which the methods are called is different because we no longer need to manipulate the cookies in the response.

```
def index(request):
    category_list = Category.objects.order_by('-likes')[:5]
    page_list = Page.objects.order_by('-views')[:5]

    context_dict = {}
    context_dict['boldmessage'] = 'Crunchy, creamy, cookie, candy, cupcake!'
    context_dict['categories'] = category_list
    context_dict['pages'] = page_list

    visitor_cookie_handler(request)
    context_dict['visits'] = request.session['visits']

    response = render(request, 'rango/index.html', context=context_dict)
    return response
```

Before you restart the Django development server, delete the existing client-side cookies to start afresh. **Mixing up cookies from a previous implementation can result in strange, frustrating behaviours.**



Avoiding Cookie Confusion

It's highly recommended that you delete any client-side cookies for Rango *before* you start using session-based data. You can do this in your browser's cookie inspector by deleting each cookie individually, or simply clear your browser's cache entirely – ensuring that cookies are deleted in the process.



Data Types and Cookies

An added advantage of storing session data server-side is its ability to cast data from strings to the desired type. This only works however for **built-in types**¹⁴, such as `int`, `float`, `long`, `complex` and `boolean`. If you wish to store a dictionary or other complex type, don't expect this to work. In this scenario, you might want to consider **pickling your objects**¹⁵.

¹⁴<http://docs.python.org/3/library/stdtypes.html>

¹⁵<https://wiki.python.org/moin/UsingPickle>

10.7 Browser-Length and Persistent Sessions

When using cookies, you can use Django's session framework to set cookies as either *browser-length sessions* or *persistent sessions*. As the names of the two types of sessions may suggest to you,

- **browser-length** sessions expire when the user closes his or her browser; and
- **persistent sessions** can last over several browser instances – expiring at a time of your choice. This could be half an hour, or even as far as a month in the future.

By default, browser-length sessions are disabled. You can enable them by modifying your Django project's `settings.py` module. Add the new, boolean variable `SESSION_EXPIRE_AT_BROWSER_CLOSE`, setting it to `True`.

Alternatively, persistent sessions are enabled by default, with the settings variable `SESSION_EXPIRE_AT_BROWSER_CLOSE` either set to `False`, or not being present in your project's `settings.py` file (defaulting to `False`). Persistent sessions have an additional setting, `SESSION_COOKIE_AGE`, which allows you to specify how long the cookie can live for. This value should be an integer, representing the number of seconds the cookie can live for. For example, specifying a value of `1209600` will mean your website's cookies expire after a two week (14 day) period.

Check out the available settings you can use on the [official Django documentation on cookies¹⁶](#) for more details. You can also check out [Eli Bendersky's blog¹⁷](#) for an excellent tutorial on cookies and Django.

10.8 Clearing the Sessions Database

Sessions accumulate easily, and the data store that contains session information does too. If you are using the database backend for Django sessions, you will have to periodically clear the database that stores the cookies. This can be done using `$ python manage.py clearsessions`. The [Django documentation¹⁸](#) suggests running

¹⁶<https://docs.djangoproject.com/en/2.1/ref/settings/#session-cookie-age>

¹⁷<http://eli.thegreenplace.net/2011/06/24/django-sessions-part-i-cookies/>

¹⁸<https://docs.djangoproject.com/en/2.1/topics/http/sessions/#clearing-the-session-store>

this daily as a [Cron job¹⁹](#). If you don't, you could find your app's performance begin to degrade when it begins to experience more and more users.

10.9 Basic Considerations and Workflow

When using cookies within your Django application, there are a few things you should consider.

- First, consider what type of cookies your web application requires. Does the information you wish to store need to persist over a series of user browser sessions, or can it be safely disregarded upon the end of one session?
- Think carefully about the information you wish to store using cookies. Remember, storing information in cookies by their definition means that the information will be stored on client's computers, too. This is a potentially huge security risk: you simply don't know how compromised a user's computer will be. Consider server-side alternatives if potentially sensitive information is involved. Storing this information on a client is always a risk.
- As a follow-up to the previous bullet point, remember that users may set their browser's security settings to a high level that could potentially block your cookies. As your cookies could be blocked, your site may function incorrectly. You *must* cater for this scenario – *you have no control over the client browser's setup*.

If client-side cookies are the right approach, then work through the following steps.

1. You must first perform a check to see if the cookie you want exists within the client's cookie store. Checking the `request` parameter will allow you to do this. The `request.COOKIES.has_key('<cookie_name>')` function returns a boolean value indicating whether a cookie `<cookie_name>` exists on the client's computer or not.
2. If the cookie exists, you can then retrieve its value – again via the `request` parameter – with `request.COOKIES[]`. The `COOKIES` attribute is exposed as a dictionary, so pass the name of the cookie you wish to retrieve as a string

¹⁹<https://en.wikipedia.org/wiki/Cron>

between the square brackets. Remember, cookies are all returned as strings, regardless of what they contain. You must, therefore, be prepared to cast to the correct type (with `int()` or `float()`, for example).

3. If the cookie doesn't exist, or you wish to update the cookie, pass the value you wish to save to the response you generate. The function you would call to create or update the cookie is `response.set_cookie('<cookie_name>', value)`. Two parameters are supplied: the name of the cookie, and the value you wish to set it to.

If you need more secure cookies, then use session-based cookies.

1. Firstly, ensure that the `MIDDLEWARE_CLASSES` list in your project's `settings.py` module contains `django.contrib.sessions.middleware.SessionMiddleware`. If it doesn't, add it to the list.
2. Configure your session backend `SESSION_ENGINE`. See the [Django Documentation on Sessions²⁰](#) for the various backend configurations.
3. Check to see if the cookie exists via `request.sessions.get()`.
4. Update or set the cookie via the session dictionary. You can access the cookie using the following syntax within your view: `request.session['<cookie_name>']`. Replace `<cookie_name>` with the name of the cookie you wish to access.

²⁰<https://docs.djangoproject.com/en/2.1/topics/http/sessions/>



Exercises

Now you've read through this chapter and tried out the code, give these exercises a go.

- Check that your cookies are server-side. Clear the browser's cache and cookies, then check to make sure you can't see the `last_visit` and `visits` variables in the browser. Note you will still see the `sessionid` cookie. Django uses this cookie to look up the session in the database where it stores all the server-side cookies about that session.
- Update the *About* page view and template telling the visitors how many times they have visited the site. Remember to call our helper function `visitor_cookie_handler()` before you attempt to get the `visits` cookie from the `request.session` dictionary, otherwise if the cookie is not set it will raise an error. Ensure that you have `visits:` preceding the count.
- Remove the visitor presentation logic from the `index()` view, and the associated template, `index.html`. You *do* however want to keep the call to the `visitor_cookie_handler()` function to increment the counter (or initialise it) if required!



Test your Implementation

We've implemented a series of unit tests to allow you to check your implementation up until this point. [Follow the guide we provided earlier](#), using the test module `tests_chapter10.py`. How does your implementation stack up against our tests? Remember to complete the exercises above to ensure that all of the tests will pass.

11. User Authentication with Django-Registration-Redux

In a [previous chapter](#), we added in login and registration functionality by manually coding up the URLs, views and templates. However, such functionality is common to many web applications. Because of this, developers have created numerous add-on apps that can be included in your Django project to reduce the amount of code required to provide user-related functionality, such as logging in, registration, one-step and two-step authentication, password changing, password recovery, and so forth. In this chapter, we will change Rango's login and register functionality so that it uses a package called `django-registration-redux`.

This will mean we will need to refactor our code to remove the login and registration functionality we previously created. From there, we will then work through setting up and configuring Rango and our wider `tango_with_django_project` code-base to make use of the `django-registration-redux` application. This chapter also will provide you with some experience of using external Django apps and will show you how easy it is to plug them into your project.

11.1 Setting up Django Registration Redux

First, we need to install `django-registration-redux` version 2.2 into your development environment using `pip`. Issue the following command at your terminal/Command Prompt. Ensure your virtual environment is activated!

```
$ pip install -U django-registration-redux==2.2
```

With the package installed, we next need to tell the Django framework that we will be using the `registration` app that comes within the `django-registration-redux` Python package. Open up your project's `settings.py` module, and scroll until you

find the `INSTALLED_APPS` list. Update it to include the `registration` package, like in the `INSTALLED_APPS` list shown below.

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'rango',
    'registration', # Add in the registration package
]
```

While you are in the `settings.py` module, you can also add the following variables that are part of the `registration` package's configuration. We provide shorthand comments to explain what each configuration variable does.

```
# If True, users can register.
REGISTRATION_OPEN = True

# If True, the user will be automatically logged in after registering.
REGISTRATION_AUTO_LOGIN = True

# The URL that Django redirects users to after logging in.
LOGIN_REDIRECT_URL = 'rango:index'

# The page users are directed to if they are not logged in.
# This was set in a previous chapter. The registration package uses this, too.
LOGIN_URL = 'auth_login'
```

Remember, we can specify URL mapping names instead of absolute URLs to make our configuration more versatile. Have a look at the section below to see where the value for `LOGIN_URL` comes from. The value for `LOGIN_REDIRECT_URL` simply points to the Rango homepage; we redirect users here after successfully logging in.

In your *project's* `urls.py` module (`<Workspace>/tango_with_django_project/urls.py`, **NOT** the one in `rango!`), you can now update the `urlpatterns` list so that it includes a reference to the `registration` package:

```
path('accounts/', include('registration.backends.simple.urls')),
```

The django-registration-redux package provides different registration backends that you can use, depending on your needs. For example, you may want a two-step process where the user is sent a confirmation email and a verification link. Here we will just be using the simple one-step registration process where a user sets up their account by entering in a username, email, and password – and from there is automatically logged in.

11.2 Functionality and URL mapping

The Django Registration Redux package provides the machinery for numerous functions. In the `registration.backends.simple.urls`, it provides key mappings, as shown in the table below. For each URL, start with `/accounts/`. For example, for the Login URL, your resultant URL will be `/accounts/login/`.

Activity	URL	Mapping Name
	/accounts...	
Login	.../login/	auth_login
Logout	.../logout/	auth_logout
Registration	.../register/	registration_register
Registration Closed	.../register/closed/	registration_disallowed
Password Change	.../password/change/	auth_password_change
Change Complete	.../password/change/done/	auth_password_change_done

All too good to be true! While all of this different functionality is provided for you, the django-registration-redux package unfortunately does not provide templates for each of the required pages. This makes sense, as templates tend to be application specific. As such, we'll need to create templates for each view.

11.3 Setting up the Templates

In the [Django Registration Redux Quick Start Guide](#)¹, an overview of what templates are required is provided. However, it is not immediately clear what goes within each template. Rather than try and work it out from the code, we can take a look at a set of [templates written by Anders Hofstee](#)² to quickly get the gist of what we need to code up.

As we'll be working on templates that the `registration` app uses, we will need to create a new directory to keep these new templates separate from our `rango` templates. In the `templates` directory, create a new directory called `registration`. This should be at the same level as the `rango` templates directory – **not within it**. We will be working on creating the basic templates from within there, as the `registration` package will look in that directory for its templates.



Inheriting Templates across Apps

You can inherit from base templates that belong to other apps. We'll be doing this in the code snippets below. Although we create templates in the `registration` directory for the `registration` app, we extend these templates from the `base.html` template for `Rango`. This will allow us to ensure a consistent look is provided across each rendered page, which is key to a professional website design.



What does `form action=".">` do?

In the templates we add below, you will notice that those with a `<form>` element have an `action` set to `..`. What does the period mean? When referencing directories and files, the period `(.)` denotes the *current file or directory*. Translated to a form submission, this means *submit the contents of the form using a POST request to the same URL as we are currently on*.

This behaviour is the same as forms we created in prior chapters, where corresponding views had a conditional switch for processing data from a `POST` request, or rendering a blank form for a `GET` request.

¹<https://django-registration-redux.readthedocs.org/en/latest/quickstart.html>

²<https://github.com/macduuibh/django-registration-templates>

Login Template

In the `templates/registration` directory, create the file `login.html`. This will house the template used by the `registration auth_login` view. Add the following markup and template code to the template.

```
{% extends 'rango/base.html' %}

{% block title_block %}
    Login
{% endblock %}

{% block body_block %}
<h1>Login</h1>

<form method="post" action=".">
    {% csrf_token %}

    {{ form.as_p }}

    <input type="submit" value="Log In" />
    <input type="hidden" name="next" value="{{ next }}" />
</form>

<p>
    Not registered?
    <a href="{% url 'registration_register' %}">Register here!</a>
</p>
{% endblock %}
```

Notice that whenever a URL is referenced, the `url` template tag is once again used to reference it. If you visit `http://127.0.0.1:8000/accounts/`, you will see the list of URL mappings and the names associated with each URL (assuming that `DEBUG=True` in `settings.py`). Alternatively, refer to the shorthand table we provided above.

Logout Template

In the `templates/registration` directory, create the file `logout.html`. This template will be rendered whenever a user logs out from Rango. Add the following code to the new template.

```
{% extends 'rango/base.html' %}

{% block title_block %}
    Logged Out
{% endblock %}

{% block body_block %}
<h1>Logged Out</h1>

<p>
    You have been successfully logged out. Thanks for spending time on Rango!
</p>
{% endblock %}
```

Registration Template

In `templates/registration` directory, create the file `registration_form.html`. This will house the template used to render a registration form. Add the following code to the new template.

```
{% extends 'rango/base.html' %}

{% block title_block %}
    Register
{% endblock %}

{% block body_block %}
<h1>Register Here</h1>

<form method="post" action=".">
    {% csrf_token %}
    {{ form.as_p }}
    <input type="submit" value="Register" />
</form>
{% endblock %}
```

Registration Closed Template

In `templates/registration` directory, create the file `registration_closed.html`. This template will be rendered if a potential user attempts to register when registration is closed. Add the following code to the new file.

```
{% extends 'rango/base.html' %}

{% block title_block %}
    Registration Closed
{% endblock %}

{% block body_block %}
<h1>Registration Closed</h1>

<p>Unfortunately, registration is currently closed. Please try again later!</p>
{% endblock %}
```

Try out the Registration Process

With the basic forms created, we can now attempt to register a new user account using the `django-registration-redux` package.

- First, we will need to migrate our database using the `$ python manage.py migrate` command.
- Then you can issue the command `$ python manage.py makemigrations`.
- After these steps, visit the registration page at <http://127.0.0.1:8000/accounts/register/>.

The screenshot shows a web browser window titled "Rango - Register". The URL in the address bar is "127.0.0.1:8000/accounts/register/". The page content is titled "Register Here". It contains the following form fields:

- Username: Required. 150 characters or fewer. Letters, digits and @/./+/-/_ only.
- E-mail:
- Password:
- Password confirmation: Enter the same password as before, for verification.

Below the form is a "Log In" button and a sidebar with the following links:

- [Python](#)
- [Django](#)
- [Other Frameworks](#)

- [Login](#)
- [Sign Up](#)
- [About](#)
- [Index](#)

A screenshot of the updated registration page using django-registration-redux.

As seen in the [figure above](#), note how the registration form contains two fields for the new user's password. This is standard in pretty much every website nowadays and provides a means for validating the password entered by the user. Try registering, but enter different passwords.

Notice how the registration app is handling all of the logic for you. The app is also able to redirect you to Rango's homepage on a successful registration, as it is instructed to by the `LOGIN_REDIRECT_URL` setting specified in your project's `settings.py` module. Are you logged in or out at that point?

Refactoring your Existing Project

Now that you are happy with the registration code, we will need to update our existing `rango` codebase to point to the new django-registration-redux URLs.

In the `rango/base.html` template, update the following URLs.

- Update the Logout URL to point to `{% url 'auth_logout' %}?next=% url 'rango:index '%}`.
- Update the Login URL to point to `{% url 'auth_login' %}`.
- Update the Sign Up URL to point to `{% url 'registration_register' %}`.

Next, open your project's `settings.py` module. You should have updated this earlier, but it's good to double-check.

- Verify that the value of `LOGIN_URL` points to `auth_login`. This will translate to the `registration` URL for the login action.

Notice that for the logout hyperlink we have included an additional component to the URL, namely `?next=% url 'rango:index '%`. This is provided so when the user logs out, it will redirect them straight to the Rango index page. If we exclude it, then they will be directed to the logout page that we created earlier, telling them that they have been successfully logged out. We include this to demonstrate some of the advanced functionality of the `django-registration-redux` package.

Finally, you need to decommission existing user authentication code that you wrote in previous chapters. This will entail removing the `register()`, `user_login()` and `user_logout()` views from Rango's `views.py` module, the corresponding URL mappings from Rango's `urls.py` module, and templates from the `templates/rango/` directory. If you do not want to delete this code, simply comment them out. Remember, too, that there will now be some redundant `import` statements at the top of Rango's `views.py` module!



Exercise

- Using the `django-registration-redux` package, provide users of your Rango app with the ability to change their password.
- Add a link to Rango's `base.html` template that directs users to the new password changing functionality. Make sure only those who are logged in can view the link.



Hints

To help you with the exercises above, the following hints may be of some use.

- Have a look at [Anders Hofstee's Templates³](#) to get yourself started. In particular, looking at this repository will be very helpful in figuring out what to call the two new templates you require for this exercise.
- Refer to the [table we provided earlier in this chapter](#) to figure out what URLs and name mappings are required for this exercise.

³<https://github.com/macduibh/django-registration-templates/tree/master/registration>

12. Bootstrapping Rango

In this chapter, we will be styling Rango using the *Twitter Bootstrap 4* toolkit. Bootstrap is one of the most popular toolkits for styling HTML pages, utilising both CSS and JavaScript (JavaScript is used to provide functionality for user interface components such as menus). The toolkit lets you design and style *responsive web applications*¹, and is pretty easy to use once you familiarise yourself with it.



Cascading Style Sheets

If you are not familiar with CSS, have a look at the [CSS crash course](#). We provide a quick guide on the basics of using Cascading Style Sheets.

Now take a look at the [Bootstrap 4.0 website](#)². The website provides you with sample code and examples of the different components that the toolkit provides, and provides you with examples and documentation on how to style them by adding in the appropriate markup and/or code. Besides this, the website also provides several [complete example layouts](#)³ from which we can base our styled design for Rango on.

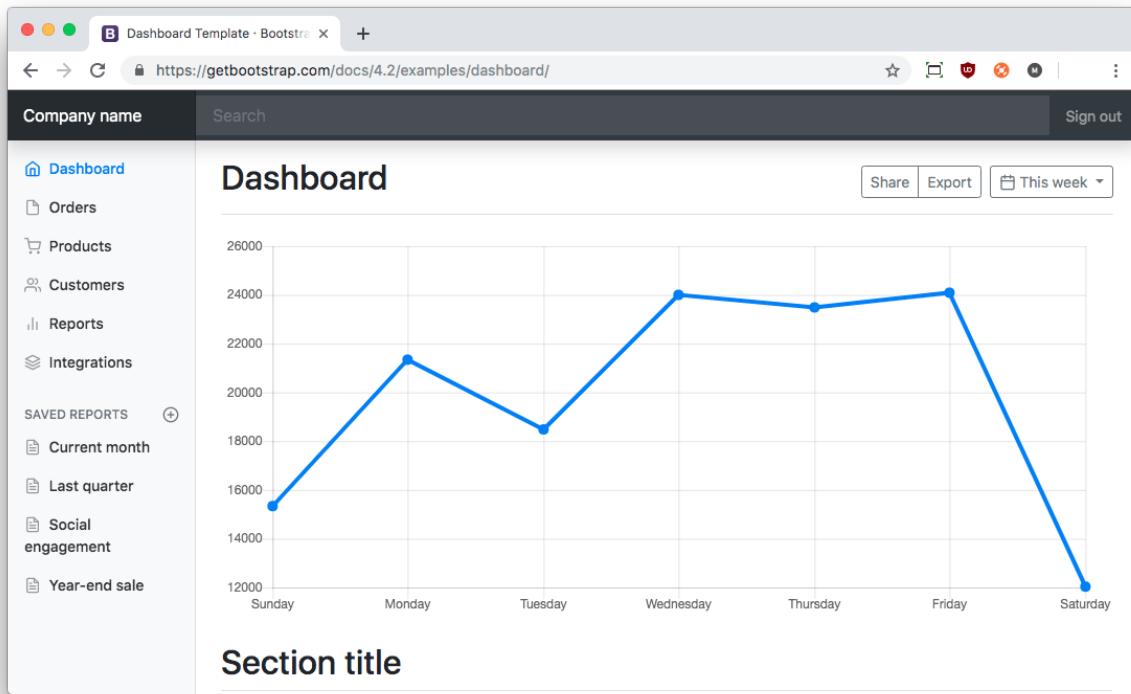
To style Rango, we have identified that the [dashboard layout](#)⁴ more or less meets our needs. The layout provides us with a menu bar at the top, a sidebar (which we will use to show categories), and a main content pane in which we can display page-specific information. You can see the example layout in [the figure below](#).

¹https://en.wikipedia.org/wiki/Responsive_web_design

²<https://getbootstrap.com/>

³<https://getbootstrap.com/examples/>

⁴<https://getbootstrap.com/docs/4.2/examples/dashboard/>



Version 4 of the Bootstrap dashboard layout, as taken from the Twitter Bootstrap examples website. This is the layout we will be working with during this chapter.

To tailor the dashboard HTML source to our Rango app, we need to make several modifications. Rather than going through and doing this yourself, the modified HTML source is available from our [GitHub Repository](#)⁵, but for completeness is also shown below.

Download the HTML source for our modified dashboard layout to a file called `bootstrap-base.html`. Like all other templates specific to Rango, this should be placed within your `<Workspace>/tango_with_django_project/templates/rango/` templates directory.

Below is a list of all the different modifications we made to the original dashboard HTML source.

- We replaced all references of `../../../../` to be `https://getbootstrap.com/docs/4.2/`. The two sets of dots indicates that we want to look back two directory levels back where we currently are in the filesystem (see our [UNIX Crash Course](#)).

⁵<http://bit.ly/twd2-bootstrap-base-template>

- This ensures that we use external files (such as stylesheets and JavaScript files) that are part of Bootstrap 4.2.
- Then, we changed `dashboard.css` to the absolute URL to access this stylesheet and removes any doubt as to what version of the file we are referring to.
- We took out the search form from the *navigation bar* at the top of the page.
- We stripped out all the non-essential demo content from the original HTML page, and replaced it with the Rango `body_block` code – most importantly including the start and end statements for the `body_block` (`{% block body_block %}{% endblock %}`).
- We set the `<title>` element of the page to fit with what we worked on earlier:
`<title>{% block title_block %}How to Tango with Django!{% endblock %}</title>`
- We changed project name to be Rango.
- We added the links to the index page, login page, register page and so forth to the navigation bar at the top.
- We added in a side block from earlier in the tutorial, by adding in a block once more – `{% block sidebar_block %}{% endblock %}`
- Finally, we added in two Django templating language load statements (`{% load staticfiles %}` and `{% load rango_template_tags %}`) after the DOCTYPE declaration tag to ensure all the necessary imports are present and correct.

We appreciate that this all seems rather hacky – and to some extent it is – but the main point of this exercise is to provide a consistent style to the application and learn by experimenting with the CSS. Taking an off-the-shelf example gets us up and running quickly so we can focus on adapting it for use with Rango.

12.1 The Template



Copying and Pasting

In the introductory chapter, we said not to copy and paste – but this is an exception. However if you directly cut and paste you will end up bringing additional text you do not want. To get started quickly, go to our GitHub page and get the [base template⁶](#) shown below.

If you don't understand what the specific Bootstrap classes do, check out the [Bootstrap documentation⁷](#) to improve your understanding.

```
1 <!DOCTYPE html>
2
3 {% load staticfiles %}
4 {% load rango_template_tags %}
5
6 <html lang="en">
7 <head>
8     <meta charset="utf-8">
9     <meta name="viewport"
10        content="width=device-width, initial-scale=1, shrink-to-fit=no">
11     <meta name="description" content="">
12     <meta name="author"
13        content="Mark Otto, Jacob Thornton, and Bootstrap contributors">
14     <meta name="generator"
15        content="Jekyll v3.8.5">
16     <link rel="icon" href="{% static 'images/favicon.ico' %}">
17     <title>
18         Rango - {% block title_block %}How to Tango with Django!{% endblock %}
19     </title>
20     <!-- Bootstrap core CSS -->
21     <link href="https://getbootstrap.com/docs/4.2/dist/css/bootstrap.min.css"
22           rel="stylesheet">
23     <!-- Custom styles for this template -->
24     <link href="https://getbootstrap.com/docs/4.2/examples/dashboard/dashboard.css"
25           rel="stylesheet">
```

⁶<http://bit.ly/twd2-bootstrap-base-template>

⁷<https://getbootstrap.com/docs/4.3/getting-started/introduction/>

```
26 </head>
27 <body>
28 <header>
29     <nav class="navbar navbar-expand-md navbar-dark fixed-top bg-dark p-0">
30         <a class="navbar-brand p-2" href="{% url 'rango:index' %}">Rango</a>
31
32         <button class="navbar-toggler" type="button" data-toggle="collapse"
33             data-target="#navbarCollapse" aria-controls="navbarCollapse"
34             aria-expanded="false" aria-label="Toggle navigation">
35             <span class="navbar-toggler-icon"></span>
36         </button>
37         <div class="collapse navbar-collapse" id="navbarCollapse">
38             <ul class="navbar-nav mr-auto">
39                 <li class="nav-item">
40                     <a class="nav-link" href="{% url 'rango:index' %}">Home</a></li>
41                 <li class="nav-item ">
42                     <a class="nav-link" href="{% url 'rango:about' %}">About</a></li>
43                 {% if user.is_authenticated %}
44                 <li class="nav-item ">
45                     <a class="nav-link"
46                         href="{% url 'rango:restricted' %}">Restricted</a></li>
47                 <li class="nav-item">
48                     <a class="nav-link"
49                         href="{% url 'auth_logout' %}?next=/rango/">Logout</a></li>
50                 <li class="nav-item">
51                     <a class="nav-link"
52                         href="{% url 'rango:add_category' %}">Add Category</a></li>
53                 <li class="nav-item">
54                     <a class="nav-link"
55                         href="{% url 'auth_password_change' %}">Change Password</a></li>
56                 {% else %}
57                 <li class="nav-item">
58                     <a class="nav-link"
59                         href="{% url 'registration_register' %}">Register Here</a></li>
60                 <li class="nav-item ">
61                     <a class="nav-link"
62                         href="{% url 'auth_login' %}">Login</a></li>
63                 {% endif %}
64             </ul>
65         </div>
66     </nav>
67 </header>
68
```

```
69 <div class="container-fluid">
70   <div class="row">
71     <nav class="col-md-2 d-none d-md-block bg-light sidebar">
72       <div class="sidebar-sticky">
73         {% block sidebar_block %}
74           {% get_category_list category %}
75         {% endblock %}
76       </div>
77     </nav>
78     <main role="main" class="col-md-9 ml-sm-auto col-lg-10 px-4">
79       {% block body_block %}
80     {% endblock %}
81     <footer>
82       <p class="float-right"><a href="#">Back to top</a></p>
83       <p>&copy; 2020 Tango With Django 2 &middot; <a href="#">Privacy</a> &middot;
84         <a href="#">Terms</a></p>
85     </footer>
86   </main>
87 </div>
88 </div>
89 <!-- Bootstrap core JavaScript -->
90 <!-- Placed at the end of the document so the pages load faster -->
91 <script src="https://code.jquery.com/jquery-3.3.1.slim.min.js"></script>
92 <script>
93   window.jQuery || document.write('<script src="https://getbootstrap.com/
94     docs/4.2/assets/js/vendor/jquery-slim.min.js"></script>')
95 </script>
96 <script src="https://getbootstrap.com/docs/4.2/dist/js/bootstrap.bundle.min.js">
97   </script>
98 <script src="https://cdnjs.cloudflare.com/ajax/libs/feather-icons/4.9.0/
99   feather.min.js">
100 </script>
101 <script src="https://getbootstrap.com/docs/4.2/examples/dashboard/dashboard.js">
102 </script>
103 </body>
104 </html>
```

Once you have prepared the new bootstrap-base.html template, download the [Rango Favicon⁸](#). This is the small icon that appears next to the URL in your browser! Save this file to <Workspace>/tango_with_django_project/static/images/.

⁸https://raw.githubusercontent.com/maxwelld90/tango_with_django_2_code/master/tango_with_django_project/static/images/favicon.ico

If you take a close look at the modified Bootstrap dashboard HTML source, you'll notice it has a lot of structure in it created by a series of `<div>` tags. Essentially the page is broken into two parts – the top navigation bar which is contained by `<header>` tags, and the main content pane denoted by the `<main ...>` tag. Within the main content pane, there is a `<div>` which houses two other `<div>`s for the `sidebar_block` and the `body_block`.

The code above assumes that you have completed the chapters on user authentication and used `django-registration-redux`, as outlined in the previous chapter. If you haven't done both of these activities, you will need to update the template and remove/modify the references to those links in the navigation bar in the header.

Also of note is that the HTML template makes references to external websites to request the required `css` and `js` files. For everything to work, you will need to be connected to the Internet for the styles and JavaScript files to be loaded when you run Rango.



Working Offline?

Rather than including external references to the `css` and `js` files, you could download all the associated files and store them in your project's static directory. We recommend storing CSS files in `static/css/`, with JavaScript files in `static/js/`. If you do this, you will need to update the `bootstrap-base.html` to point to the correct files locally using the `{% static '...' %}` template function.

12.2 Quick Style Change

To give Rango a much-needed facelift, we need to update our base template to make use of the new `bootstrap-base.html` template. It's now ready to go! There are many ways to do this, with one option being to rename `base.html` to `bootstrap-base.html` in all your other templates. However, a quicker solution would be to do the following.

1. Rename the `base.html` template in `templates/rango/` to `base-old.html`.
2. Rename the `bootstrap-base.html` template in `templates/rango/` to `base.html`.

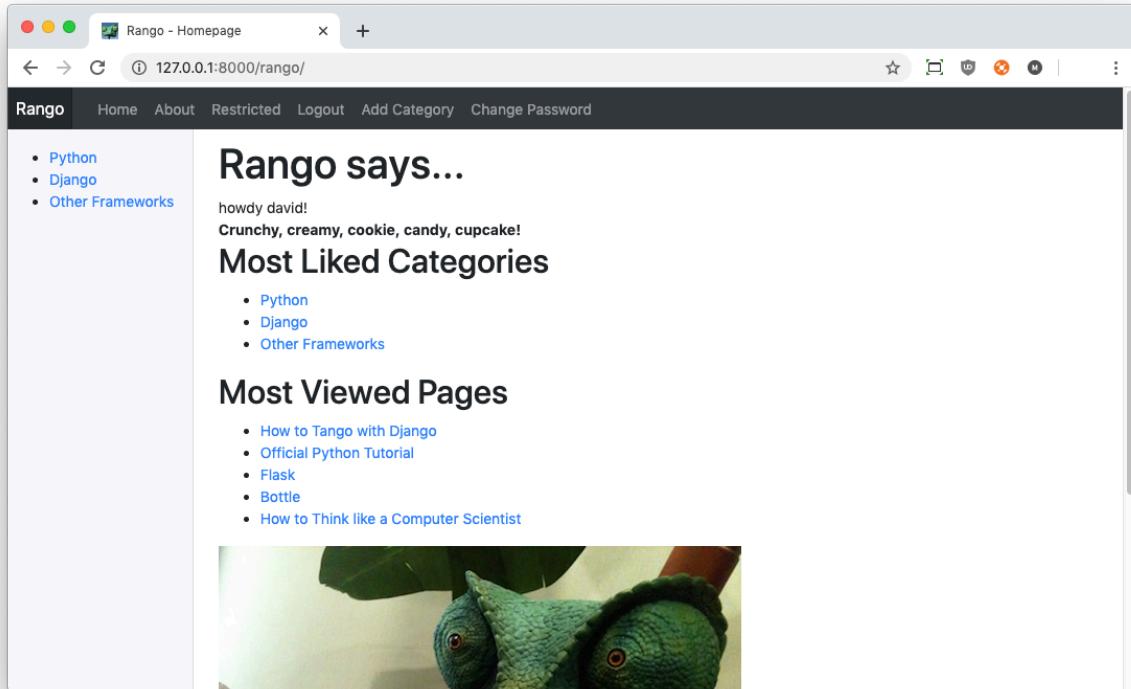
This ensures you keep a copy of your old `base.html` template and switch the active template that all other Rango templates inherit from to the Bootstrapped version.



Using git?

If you are using Git, it would be prudent to use the `git mv` commands in your terminal or Command Prompt to ensure that Git can keep track of these filename changes. If you are using a GUI to manage your Git repository, use that to perform the renaming instead.

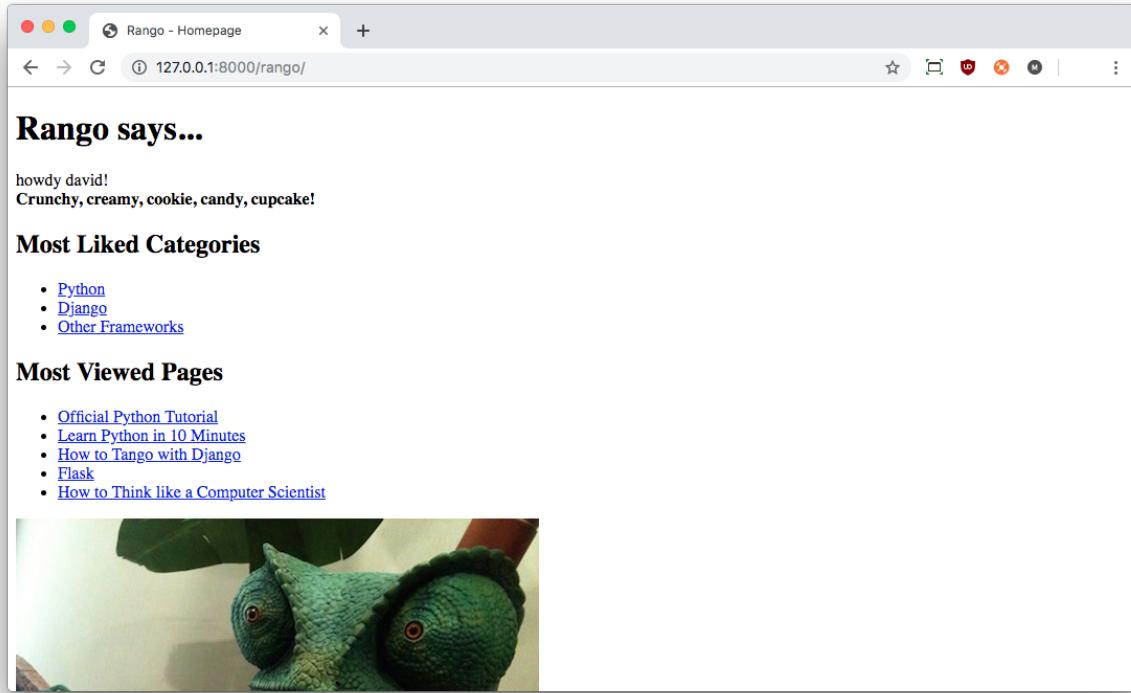
Now reload Rango in your browser. You should see a screen that looks similar [to the one below](#).



Rango, after switching `bootstrap-base.html` to be the template from which other templates inherit. Looks nice! Note that here, we are logged in with a username of `david`.

Have a look through the different pages of your app. Since they all inherit from `base.html`, they will all now employ the Bootstrap dashboard layout. However, they aren't perfect yet! In the remainder of this chapter, we will go through several changes to the templates, and use various Bootstrap components to improve the

look and feel of Rango further. But to remind yourself of what Rango looked like before, compare and contrast to [the figure below!](#)



The Rango app, before Bootstrap styling was applied. How much better does your app now look in comparison to this?

Sidebar Categories

One thing that we could improve is the way that the categories on the sidebar to the left appear. They look pretty basic at the moment, so let's make them look nicer! If we first take a look at the HTML source code for the example [Bootstrap dashboard page](#)⁹, we notice that a few *classes*¹⁰ have been added to the `` (*unordered list*) and `` (*list item*) tags to denote that they are navigation items (`nav-item`) and navigation links (`nav-link`) respectively.

Let's apply these classes to our `rango/categories.html` template. Refactor the file to look like the example below. Note that the logic and basic structure stays the same – we just add classes and some supplementary tags to make things look nicer.

⁹<https://getbootstrap.com/docs/4.2/examples/dashboard/>

¹⁰https://www.w3schools.com/cssref/sel_class.asp

```
1 <ul class="nav flex-column">
2 {% if categories %}
3     {% for c in categories %}
4         {% if c == current_category %}
5             <li class="nav-item">
6                 <a class="nav-link active" href="{% url 'rango:show_category' c.slug %}">
7                     <span data-feather="archive"></span>{{ c.name }}</a>
8             </li>
9         {% else %}
10            <li class="nav-item">
11                <a class="nav-link" href="{% url 'rango:show_category' c.slug %}">
12                    <span data-feather="archive"></span>{{ c.name }}</a>
13            </li>
14        {% endif %}
15    {% endfor %}
16    {% else %}
17        <li class="nav-item">There are no categories present.</li>
18    {% endif %}
19 </ul>
```

Rather than using `` to show what category page has been selected, we have added the `active` class to the currently shown category. We can also add in a feather-icon using the `` tag. Here, we chose the archive icon, but there are loads of icons you can choose from instead. Have a look at the [Feather Icons website¹¹](#) for a list.

The Index Page

For the index page, it would be nice to show the top categories and top pages in two separate columns, with the title kept at the top. This would be a much better use of your screen's real estate!

If we go back to the Bootstrap examples, we can see that [Jumbotron¹²](#) example provides a neat header element that we can put our title message in. To use that, we update our `index.html` template to incorporate the following, replacing the existing

¹¹<https://feathericons.com/>

¹²<https://getbootstrap.com/docs/4.2/examples/jumbotron/>

header message markup (including the existing `<h1>` tag and corresponding `<div>` immediately underneath).

```
<div class="jumbotron p-4">
  <div class="container">
    <h1 class="jumbotron-heading">Rango says...</h1>
    <div>
      <h2 class="h2">
        {% if user.is_authenticated %}
          howdy {{ user.username }}!
        {% else %}
          hey there partner!
        {% endif %}
      </h2>
      <strong>{{ boldmessage }}</strong>
    </div>
  </div>
</div>
```

For the `<div>` container, we applied classes `jumbotron` and `p-4`. The class `p-4` controls the [spacing¹³](#) around the jumbotron. Try changing the padding to be `p-6` or `p-1` to see what happens! You can also control the space of the top, bottom, left and right by specifically setting `pt`, `pb`, `pr` and `pl` instead of just `p`.



Site Styling Exercise

Update all other templates so that the page heading is encapsulated within a jumbotron. This will make the whole application have a consistent look, something crucial for a professionally-designed website.

Don't forget to update the templates used for the `registration` package!

After you have successfully added the jumbotron, we can move on to the two-column layout. Here, we draw upon the [album¹⁴](#) layout. While it has three columns, called cards, we only need two. Most – if not all – CSS frameworks use a [grid layout¹⁵](#) consisting of a total of 12 columns. If you inspect the HTML source for the album layout, you will see that within a row there is a `<div>` which sets the size of the cards.

¹³<https://getbootstrap.com/docs/4.2/utilities/spacing/>

¹⁴<https://getbootstrap.com/docs/4.2/examples/album/>

¹⁵<https://getbootstrap.com/docs/4.2/layout/grid/>

The `<div>` is `<div class="col-md-4">`, followed by `<div class="card mb-4 shadow-sm">`. This sets each card to be 4 units in length (out of 12) relative to the width (and 4 by 3 is 12). Since we want two cards (one for the most popular pages and most popular categories) then we can change the 4 to a 6 (i.e. 50%, with 2 by 6 is 12). To implement this, update the `index.html` template once again. This time, we are replacing the existing `<div>` elements that housed the most liked categories and most viewed pages.

```
<div class="container">
    <div class="row">
        <div class="col-md-6">
            <div class="card mb-6">
                <div class="card-body">
                    <h2>Most Liked Categories</h2>
                    <p class="card-text">
                        {% if categories %}
                            <ul>
                                {% for category in categories %}
                                    <li>
                                        <a href="{% url 'rango:show_category' category.slug %}">
                                            {{ category.name }}</a>
                                        </li>
                                {% endfor %}
                            </ul>
                        {% else %}
                            <strong>There are no categories present.</strong>
                        {% endif %}
                    </p>
                </div>
            </div>
        </div>

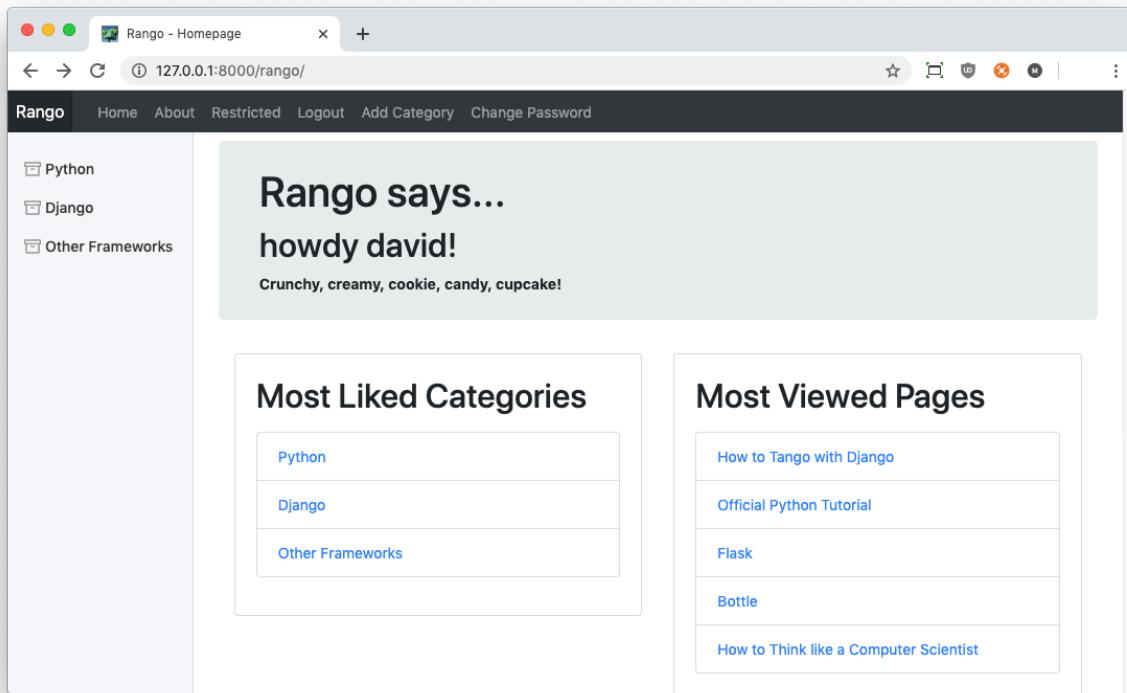
        <div class="col-md-6">
            <div class="card mb-6">
                <div class="card-body">
                    <h2>Most Viewed Pages</h2>
                    <p class="card-text">
                        {% if pages %}
                            <ul>
                                {% for page in pages %}
                                    <li>
```

```
    <a href="#">{% page.url %}{% page.title %}</a>
    </li>
    {% endfor %}
    </ul>
    {% else %}
        <strong>There are no pages present.</strong>
        {% endif %}
    </p>
</div>
</div>
</div>
</div>
</div>
```

Once you have updated the template, reload the page – it should look a lot better now, but the way the list items are presented is not the best. Once again, it looks pretty basic. Surely we can make it look even better!

Let's use the [list group styles provided by Bootstrap¹⁶](#) to improve how the hyperlinks are presented. We can do this quite easily by changing the two `` elements to `<ul class="list-group">` and each of the `` elements that we just added to include a class, `<li class="list-group-item">`. Once you have completed these steps, reload the index page. How does it look now? It should look similar to [the figure below](#).

¹⁶<https://getbootstrap.com/docs/4.2/components/list-group/>



The updated Rango index page, after applying both the jumbotron and two-column layout. How much better does it look now?

The Login Page

Now that the index page has been styled, let's turn our attention to Rango's login page. On the Bootstrap website, there is a demonstration [login form¹⁷](#). If you take a look at the source, you'll notice that there are several classes that we need to include to get a basic login form to work using Bootstrap. To do this, we can start by replacing the markup in `body_block` after your jumbotron header in the `registration/login.html` template with the following.

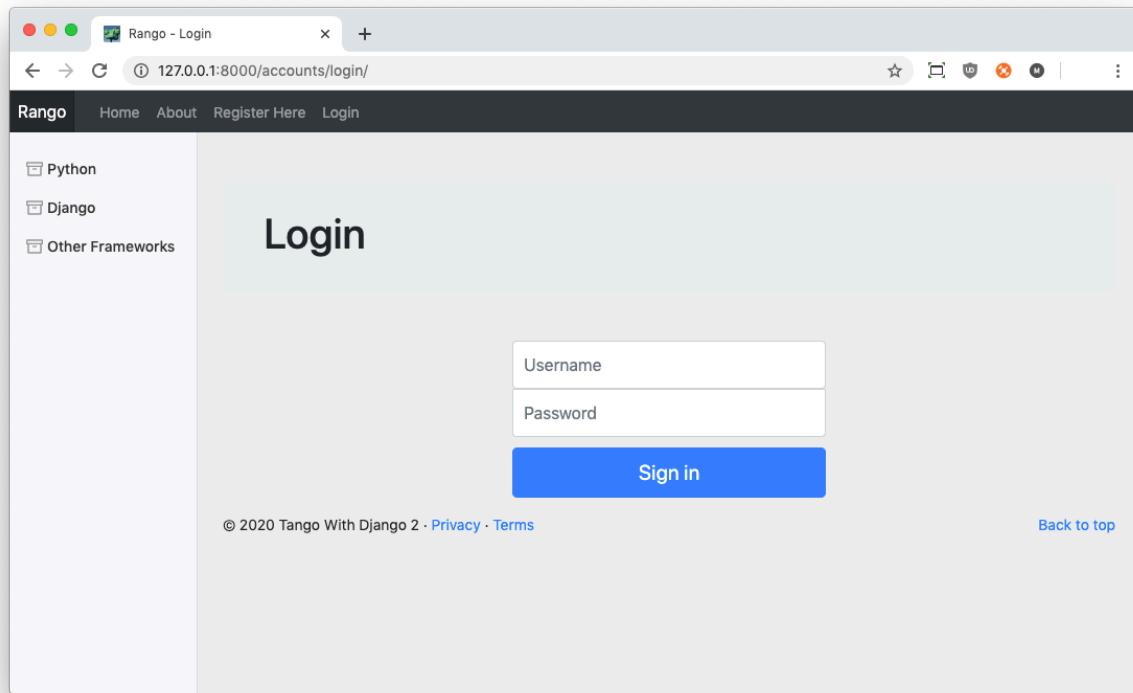
¹⁷<https://getbootstrap.com/docs/4.2/examples/sign-in/>

```
<div class="container">
<form class="form-signin" role="form" method="post" action=".">
    {% csrf_token %}
    <label for="inputEmail" class="sr-only">Username</label>
    <input type="text" name="username" id="id_username" class="form-control"
           placeholder="Username" required autofocus>
    <label for="inputPassword" class="sr-only">Password</label>
    <input type="password" name="password" id="id_password" class="form-control"
           placeholder="Password" required>
    <button class="btn btn-lg btn-primary btn-block" type="submit"
            value="Submit">Sign in</button>
</form>
</div>
```

Besides adding in a link to the bootstrap `signin.css` and a series of changes to the classes associated with elements, we have removed the code that automatically generates the login form (`{{ form.as_p }}`). Instead, we took the elements from the generated `<form>`, and importantly the `name` of the form elements. We then associated the names with the new elements we added above. To find out what the `names` were, we ran Rango, navigated to the login page, and then inspected the source to see what HTML was produced by the `{{ form.as_p }}` call.

In the button, we have set the classes to `btn` and `btn-primary`. If you check out the [Bootstrap section on buttons¹⁸](#), you can see there are lots of different colours, sizes and styles that can be assigned to buttons. The resultant output can be seen in [the figure below](#).

¹⁸<https://getbootstrap.com/docs/4.2/components/buttons/>



A screenshot of the login page with customised Bootstrap Styling.

Other Form-based Templates

You can apply similar changes to `add_category.html` and `add_page.html` templates. For the `add_page.html` template, we can set it up as follows.

```
{% extends "rango/base.html" %}  
{% block title %}Add Page{% endblock %}  
{% block body_block %}  


# Add a Page to {{category.name}}


```

```
{% for hidden in form.hidden_fields %}
    {{ hidden }}
{% endfor %}
{% for field in form.visible_fields %}
    {{ field.errors }}
    {{ field.help_text }}<br />
    {{ field }}<br />
    <div class="p-2"></div>
{% endfor %}
<br />
<button class="btn btn-primary" type="submit" name="submit">
    Add Page
</button>
<div class="p-5"></div>
</form>
</div>
</div>
{% endblock %}
```



Category Form Style Exercise

Create a similar template for the *Add Category* template, located at `rango/add_category.html`.

The Registration Template

Finally, let's tweak the registration template. Open the registration form template, located at `templates/registration/registration_form.html`. Once you have the file open, we can update the markup inside the `body_block` as follows. Make sure you keep the existing jumbotron you added earlier as part of an exercise!

```
<div class="container">
    <div class="row">
        <div class="form-group" >
            <form role="form" method="post" action=". ">
                {% csrf_token %}
                <div class="form-group">
                    <p class="required"><label class="required" for="id_username">
                        Username:</label>
                    <input class="form-control" id="id_username" maxlength="30"
                           name="username" type="text" />
                    <span class="helptext">
                        Required. 30 characters or fewer.
                        Letters, digits and @/./+/-/_ only.
                    </span>
                </p>
                <p class="required"><label class="required" for="id_email">
                    E-mail:</label>
                    <input class="form-control" id="id_email" name="email"
                           type="email" />
                </p>
                <p class="required"><label class="required" for="id_password1">
                    Password:</label>
                    <input class="form-control" id="id_password1" name="password1"
                           type="password" />
                </p>
                <p class="required">
                    <label class="required" for="id_password2">
                        Password confirmation:</label>
                    <input class="form-control" id="id_password2" name="password2"
                           type="password" />
                    <span class="helptext">
                        Enter the same password as before, for verification.
                    </span>
                </p>
            </div>
            <button type="submit" class="btn btn-primary">Submit</button>
        </form>
    </div>
</div>
</div>
```

Once again, we have transformed the form created by the `{} form.as_p {}` method

call, and added the various bootstrap classes to the manual form.



Bootstrap, HTML and Django Kludge

This is not the best solution – we have kind of mashed things together. It would be much nicer and cleaner if we could instruct Django when building the HTML for the form to insert the appropriate classes. But we will leave that to you to figure out! Toolkits do exist, so perhaps there is a solution that automatically handles everything for you! Nevertheless, by manually bringing everything together, you can obtain a better appreciation and understanding of how the different components fit together.

Next Steps

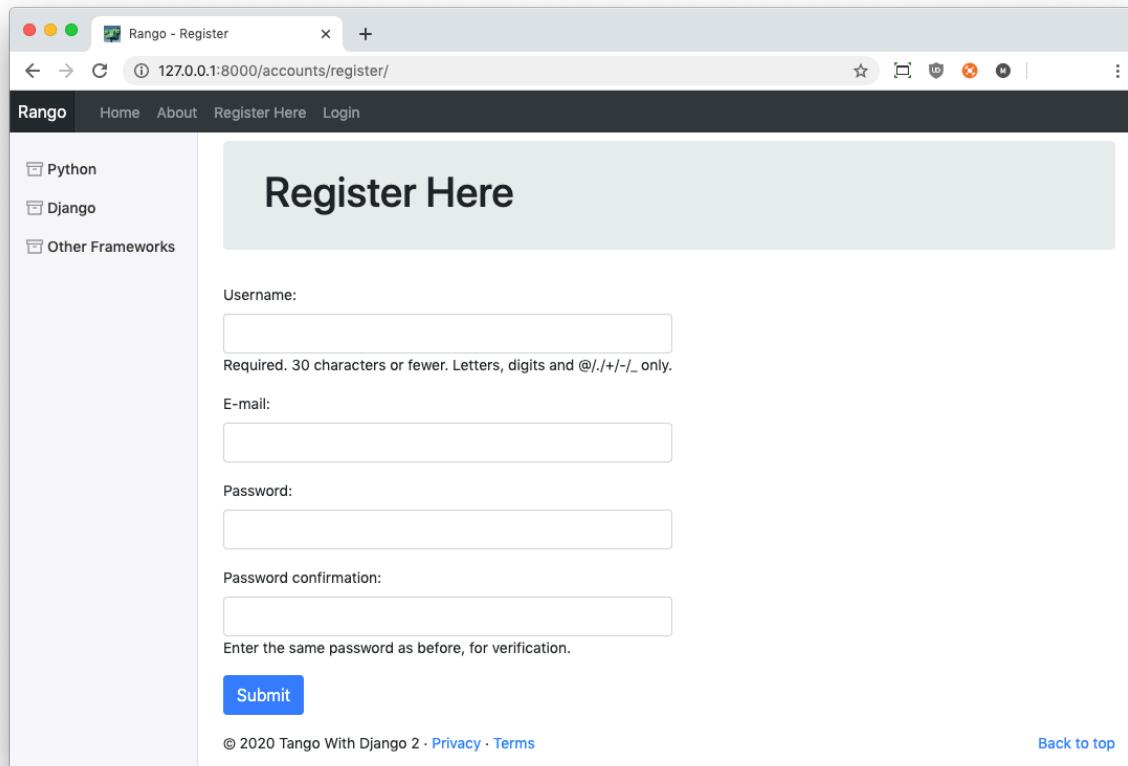
This chapter has described how to quickly style your Django application using the Bootstrap toolkit. Bootstrap is highly extensible and it is relatively easy to change themes – check out the [StartBootstrap website¹⁹](#) for a whole series of free themes. Alternatively, you might want to use a different CSS toolkit like: [Zurb²⁰](#), [Pure²¹](#) or [GroundWorkd²²](#). Now that you have an idea of how to hack the templates and set them up to use a responsive CSS toolkit, we can now go back and focus on finishing off the extra functionality that will pull the application together.

¹⁹<http://startbootstrap.com/>

²⁰<http://zurb.com>

²¹<https://purecss.io>

²²<https://groundworkcss.github.io/groundwork/>



A screenshot of the registration form page with customised Bootstrap styling.



Another Style Exercise

While this tutorial uses Bootstrap, an additional, and optional exercise, would be to style Rango using one of the other responsive CSS toolkits. If you do create your style, let us know and we can link to it to show others how you have improved Rango's styling!

13. Adding Search to Rango

Now that most of the core functionality of Rango has been implemented (and it looks good, too!), we can move to address some more advanced functionality. In this chapter, we will connect Rango up to a *search API* so that users can also *search* for pages, rather than simply *browse* categories. Therefore, the main point of this chapter is to show you how you can connect and use other web services – and how to integrate them within your own Django app.

The search API that we will be using will be *Microsoft’s Bing Search API*. However, you could just as readily use any available search API, such as those provided by [Webhose¹](#) or [Yandex²](#).

To use the Bing Search API, we will need to write a [wrapper³](#), which enables us to send a query and obtain the results from Bing’s API – all the while returning results to us in a convenient format that we can readily use in our code. However, before we can do so, we first need to set up a Microsoft Azure account to be able to make use of the Bing Search API.

13.1 The Bing Search API

The [Bing Search API⁴](#) provides you with the ability to embed search results from the Bing search engine within your own applications. Through a straightforward interface, you can request results from Bing’s servers to be returned in either XML or JSON. The data returned can then be interpreted by an XML or JSON parser, with the results then, for example, rendered as part of a template within your wider application.

Although the Bing API can handle requests for different kinds of content, we’ll be focusing on web search only for this tutorial, with JSON-formatted responses. To

¹<https://webhose.io/>

²<https://yandex.com/support/search/robots/search-api.html>

³https://en.wikipedia.org/wiki/Adapter_pattern

⁴<https://docs.microsoft.com/en-gb/rest/api/cognitiveservices/bing-web-api-v7-reference>

use the Bing Search API, you will need to sign up for an *API key*. The key currently provides subscribers with access to 1000 queries per month, which should be more than enough for our experimental purposes.



Application Programming Interface (API)

An [Application Programming Interface](#)⁵ specifies how software components should interact with one another. In the context of web applications, an API is considered as a set of HTTP requests along with a definition of the structures of response messages that each request can return. Any meaningful service that can be offered over the Internet can have its API. We aren't limited to web search! For more information on web APIs, [Luis Rei provides an excellent tutorial on APIs](#)⁶.

Registering for a Bing API Key

To obtain a Bing API key, you must first register for a Microsoft Azure account. The account provides you with access to a wide range of Microsoft services. If you already have a Microsoft account, you do not need to register – simply enter your details to login with your existing account details. Otherwise, you can go online and create a free account with Microsoft at <https://account.windowsazure.com>⁷.

When you have logged in, go to the portal. The link is at the top right of the page.

Once the portal has loaded, you should see a list of options at the top of the viewport (or down the side, depending on your screen size). Find the top option called Create a resource and click it. The right-hand side of the page will then be populated with more lists. From there, find the AI + Machine Learning option and select that. Scroll through the options on the subsequent menu that appears, and select the Bing Search option.

⁵http://en.wikipedia.org/wiki/Application_programming_interface

⁶<http://blog.luisrei.com/articles/rest.html>

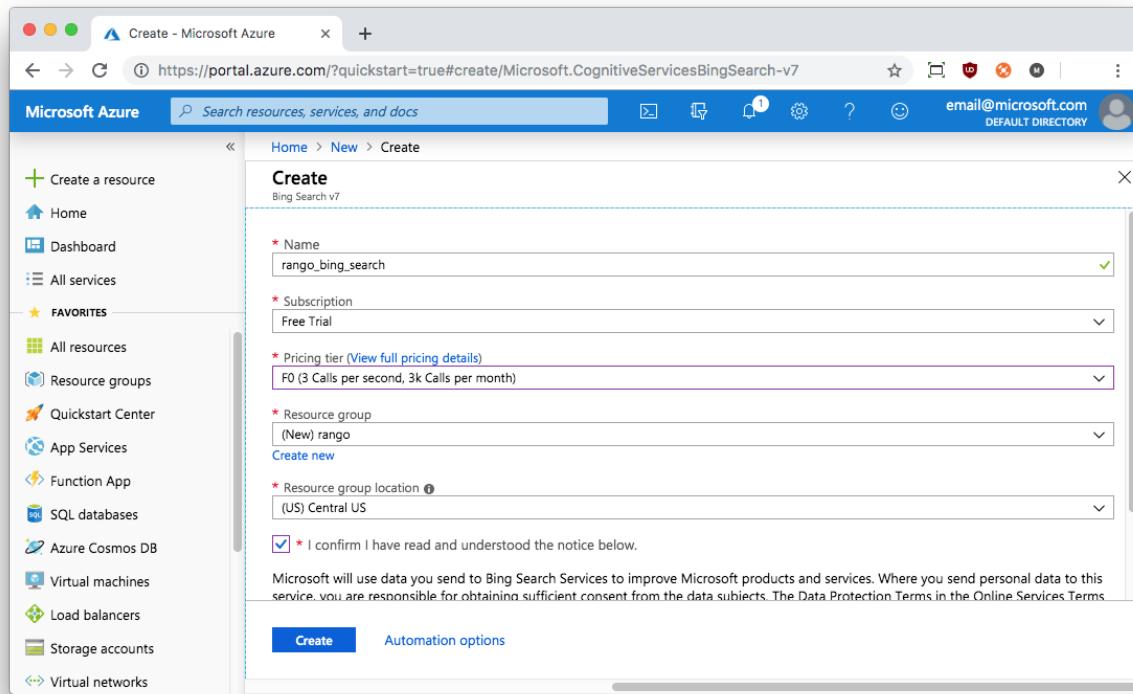
⁷<https://account.windowsazure.com>



Entering Personal Information

At this stage, you may be redirected to a page where you have to supply details such as your address and payment card. Microsoft says that this information is required to ensure that spammers and bots do not infiltrate their services – and rest assured, if you need to provide payment details, no money will be taken from your bank account unless you specifically authorise it. We will be using the free Bing Search API allowance, so no money will need to be transferred. If you do need to provide this information, you'll need to head back to the portal and look for the `Bing Search v7` option once more.

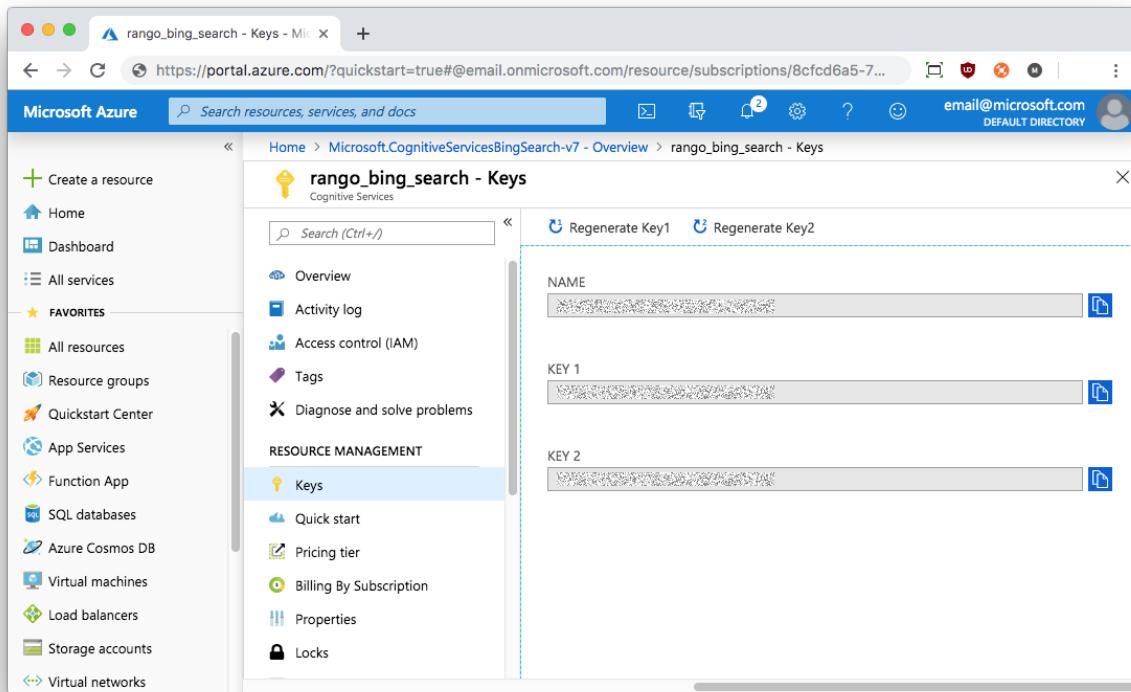
You'll then be greeted with a page similar to [the one below](#). Here, you need to provide a name for your Bing Search service – something like `rango_bing_search` will do the job nicely. Ensure that you select `Free Trial` for the subscription, and pricing tier `F0` or `F1` (allowing 1000 free requests per month). Selecting these options will ensure that you will not be charged for access to the API. You'll also need to make a new group – we made one called `rango`. The resource location doesn't matter as we won't be worrying about things like response times and the like. Once you are happy, click `Create` at the bottom of the page.



Creating a free Bing Search API resource in the Microsoft Azure web application.

You'll then need to wait for the resource to be created – this will take a few minutes. After it has been created, you should see a screen confirming that your new resource has been created. On that screen, click the Go to resource button.

A new page will then load. Here, you should look for the Keys option under the Resource Management header. You should see the screen [like the one below](#). The keys in this figure are deliberately obscured. Take note of each of the three items on this page – name, key 1 and key 2. Copy the value of key1 into a text file, as we will need this later! The key you save here will be required when we attempt to access the Bing Search API later on.



The keys screen for the Bing Search API resource. Make sure you take a note of the key 1 value! You will need them later on.

13.2 Adding Search Functionality

With your keys in hand, it's now time to get coding. Below, we have provided the necessary code so that we can programmatically issue search queries to the Bing Search API. Create a file called `bing_search.py` inside your `rango` app directory – the same directory with modules like `views.py` and `models.py`. Add the code to the module as shown below.



The `requests` Package

For this code to work, you'll need to add the `requests` package to Rango's environment. You can do this by running the command `$ pip install requests` in your terminal or Command Prompt. If you don't do this, the code won't work.

```
1 import json
2 import requests
3
4
5 # Add your Microsoft Account Key to a file called bing.key
6 def read_bing_key():
7     """
8         reads the BING API key from a file called 'bing.key'
9         returns: a string which is either None, i.e. no key found, or with a key
10            remember to put bing.key in your .gitignore file to avoid committing it.
11
12        See Python Anti-Patterns - it is an awesome resource to improve your python code
13        Here we using "with" when opening documents
14        http://bit.ly/twd-antipattern-open-files
15        """
16
17     bing_api_key = None
18     try:
19         with open('bing.key','r') as f:
20             bing_api_key = f.readline().strip()
21     except:
22         try:
23             with open('../bing.key') as f:
24                 bing_api_key = f.readline().strip()
25         except:
26             raise IOError('bing.key file not found')
27
28     if not bing_api_key:
29         raise KeyError('Bing key not found')
30
31     return bing_api_key
32
33 def run_query(search_terms):
34     """
35         See the Microsoft's documentation on other parameters that you can set.
36         http://bit.ly/twd-bing-api
37     """
38
39     bing_key = read_bing_key()
40     search_url = 'https://api.cognitive.microsoft.com/bing/v7.0/search'
41     headers = {'Ocp-Apim-Subscription-Key': bing_key}
42     params = {'q': search_terms, 'textDecorations': True, 'textFormat':'HTML'}
43
44     # Issue the request, given the details above.
45     response = requests.get(search_url, headers=headers, params=params)
```

```
44     response.raise_for_status()
45     search_results = response.json()
46
47     # With the response now in play, build up a Python list.
48     results = []
49     for result in search_results['webPages']['value']:
50         results.append({
51             'title': result['name'],
52             'link': result['url'],
53             'summary': result['snippet']})
54
55     return results
```



Python Anti-Patterns

In the wrapper we avoided problems with opening files (and not remembering to close them) by using the `with` command. Opening the file directly is a common anti-pattern that should be avoided. To find out more about anti-patterns in Python, then check out the [Little Book of Python Anti-Patterns](#)⁸. It is an awesome resource that will help you to improve your python code.



Reading `bing.key`

In function `read_bing_key()`, you may be a bit unsure as to why we attempt to read the `bing.key` file from a different location if the first attempt fails. This block of code is present to make future exercises that little bit easier for you.

In the module(s) above, we have implemented two functions: one to retrieve your Bing API key from a local file, and another to issue a query to the Bing search engine. Below, we discuss how both of the functions work.

`read_bing_key()` – Reading the Bing Key

The `read_bing_key()` function reads in your key from a file called `bing.key`, located in your Django project's root directory. Create a new text file now – as named above, this will be called `bing.key`, and will live in the same directory as the one containing your existing `manage.py` and `populate_rango.py` modules. Putting your

⁸<http://bit.ly/twd-python-anti-patterns>

API key in a different file from the logic of handling API requests *separates your concerns*. We create this additional file because if you are putting your code into a public repository (on GitHub for example), you should take some precautions to avoid sharing your API key publicly.

Take the value of key 1 that you took from the Azure portal earlier, and paste it into your new `bing.key` file. **The key value itself should be the only contents of the file** – nothing else should exist within it. **This file should NOT be committed to your GitHub repository.** To make sure that you do not accidentally commit it, update your [repository's `.gitignore` file](#) to exclude any files with a `.key` extension, by adding the line `*.key`. This way, your key file will only be stored locally and will reduce the risk of individuals from acquiring and using your key without your knowledge. If you haven't created a `.gitignore` file yet, now would be a good time to [refer to the Git Crash Course chapter](#) to see what this file does, and how you can go about creating one!



Keep your Keys Safe!

Keys to access APIs are exclusively yours to use. Keep them secret, keep them safe! Do not publish them online. If your app requires the use of an API key, then provide clear instructions describing how people can acquire their key, and what they should do with it.

`run_query()` – Executing the Query

The `run_query()` function takes a query as a string (via `search_terms`), and by default returns the top ten results from Bing in a list that contains a dictionary of the result items (including the title, a link, and a summary).

To summarise though, the logic of the `run_query()` function can be broadly split into six main tasks.

- First, the function prepares for connecting to Bing by preparing the URL that we'll be requesting.

- The function then prepares authentication, making use of your Bing API key. This is obtained by calling `read_bing_key()`, which in turn pulls your Account key from the `bing.key` file that you created earlier.
- The response is then parsed into a Python dictionary object using the `json` Python package.
- We loop through each of the returned results, populating a `results` dictionary. For each result, we take the `title` of the page, the `link` (or URL), and a short summary of each returned result.
- The list of dictionaries is then returned by the function.



Bing it on!

There are many different parameters that the Bing Search API can handle which we don't cover here. If you want to know more about the API, and maybe even try out some different settings, check out the [Bing Search API Documentation](#)⁹.



Exercises

Extend your `bing_search.py` module so that it can be run independently, i.e. running `python bing_search.py` from your terminal or Command Prompt. Specifically, you should implement functionality that:

- prompts the user to enter a query, i.e. use `input()`; and
- issues the query via `run_query()`, and prints the results.

Update the `run_query()` method so that it handles network errors gracefully.

⁹<https://docs.microsoft.com/en-gb/rest/api/cognitiveservices/bing-web-api-v7-reference>



Hint

Add the following code, so that when you run `python bing_search.py` it calls the `main()` function:

```
def main():
    # Insert your code here. What will you write?

if __name__ == '__main__':
    main()
```

When you run the module via `$ python bing_search.py`, the `bing_search` module is treated as the `__main__` module, and thus triggers `main()`. However, when the module is imported by another, then `__name__` will not equal `__main__`, and thus the `main()` function not be called. This way you can import it with your application without having to call `main()`.

You'll also most likely want to make use of the built-in `input()` function to complete this exercise.

13.3 Putting Search into Rango

Now that we have successfully implemented (and tried out!) the search functionality module, we need to integrate it into our Rango app. There are two main steps that we need to complete for this to work.

- We must first create a `search.html` template that extends from our `base.html` template. The `search.html` template will include an HTML `<form>` to capture the user's query as well as the necessary template code to present any results.
- We then create a view to handle the rendering of the `search.html` template. This new view will also call the `run_query()` function we provided you with earlier.

Adding a Search Template

Let's first create a template called `rango/search.html`. Add the following HTML markup and Django template code to the template.

```
1  {% extends 'rango/base.html' %}            
2  {% load staticfiles %}                    
3  
4  {% block title_block %}                  
5      Search  
6  {% endblock %}  
7  
8  {% block body_block %}                  
9  <div class="jumbotron p-4">  
10     <div class="container">  
11         <h1 class="jumbotron-heading">Search with Rango</h1>  
12     </div>  
13 </div>  
14 <div>  
15     <form class="form-inline"             
16         id="user-form"                   
17         method="post"                   
18         action="{% url 'rango:search' %}">  
19         {% csrf_token %}  
20  
21         <div class="form-group">  
22             <input class="form-control"      
23                 type="text"                
24                 size="50"                 
25                 name="query"              
26                 id="query" />  
27     </div>  
28  
29         <button class="btn btn-primary"     
30             type="submit"                
31             name="submit">Search</button>  
32     </form>  
33 </div>  
34 <div>  
35     {% if result_list %}  
36         <h2>Results</h2>  
37  
38         <div class="list-group">  
39             {% for result in result_list %}  
40                 <div class="list-group-item">  
41                     <h3 class="list-group-item-heading">  
42                         <a href="{{ result.link }}">  
43                             {{ result.title|safe|escape }}</a>
```

```
44          </a>
45      </h3>
46      <p class="list-group-item-text">
47          {{ result.summary|safe|escape }} 
48      </p>
49      </div>
50      {% endfor %}
51  </div>
52  {% endif %}
53 </div>
54 {% endblock %}
```

The template code above performs two key tasks.

- In all scenarios, the template provides a search box and a search button within an HTML `<form>`. The `<form>` allows users to enter (into a field) and submit (with a button) their search queries.
- If a `result_list` object is passed to the template's context when being rendered, the template assumes that the user has issued a query and is looking for results. As such, the template takes the `result_list` object and iterates through it, displaying each result contained within.

To style the HTML, we have made use of the Bootstrap [jumbotron¹⁰](#), [list groups¹¹](#), and [forms¹²](#) components.

To render the title and summary correctly, we have used the built-in `safe` and `escape` template tags to inform the template that `result.title` and `result.summary` should both be rendered as-is (i.e. as HTML). This is because some results can be returned from Bing complete with HTML tags (i.e. `` or `` to bold the surrounded terms).

Adding the View

With our template defined, we can now code up the associated view. We'll be calling this view `search()`, and as usual this will be located within Rango's `views.py`.

¹⁰<https://getbootstrap.com/docs/4.2/components/jumbotron/>

¹¹<https://getbootstrap.com/docs/4.2/components/list-group/>

¹²<https://getbootstrap.com/docs/4.2/components/forms/>

module. Matching up with the assumptions made in our template defined above, the `search()` view assumes that with a `POST` request, users will be expecting to see results – with a basic form only being displayed with a `GET` request.

```
def search(request):
    result_list = []

    if request.method == 'POST':
        query = request.POST['query'].strip()
        if query:
            # Run our Bing function to get the results list!
            result_list = run_query(query)

    return render(request, 'rango/search.html', {'result_list': result_list})
```

The code should by now be pretty self-explanatory to you. If the request's method is `POST`, `query` is taken from the request and used as the query (the `strip()` method removes any leading or trailing whitespace). We then call the `run_query()` function we defined earlier to get the results, and then call the `render()` function, passing the results (referred to by `result_list`) to the context dictionary.

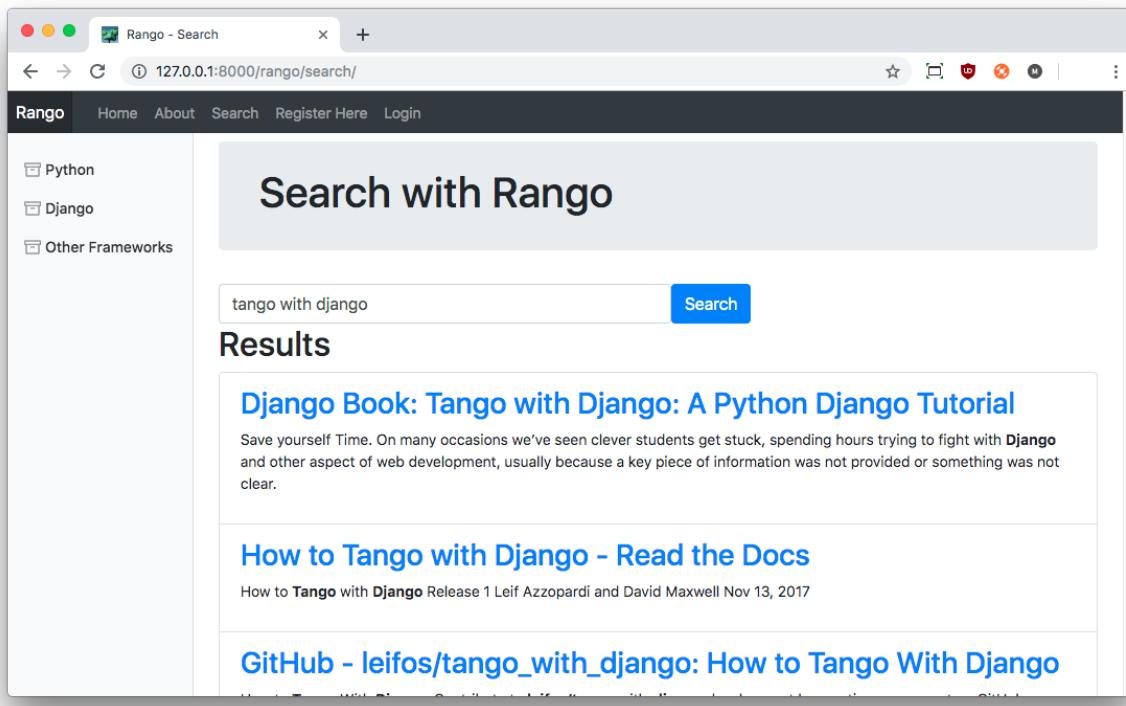
However, for this to work, we need to import our new `run_query()` function. To do that, make sure you include the `import` statement at the top of your `views.py` module.

```
from rango.bing_search import run_query
```

As is standard for adding new functionality to your app, you'll also need to do the following steps to get everything working as it should.

- Add a new URL mapping between your `search()` view and the `/rango/search/` URL. To make sure it works with everything above, give it a `name` of `search`. Remember, this should go into Rango's `urls.py` module.
- You must also update the `base.html` navigation bar (at the top of the template) to include a link to the search page. Remember to use the `url` template tag to reference the link (`{% url 'rango:search' %}`). This link should be available to anyone, regardless of whether they are logged into Rango or not.
- Again, check that your `bing.key` is in your project's root directory (i.e. in the directory that contains `manage.py`).

Once you have completed the preceding three steps, you should now be able to navigate to your Rango app in your browser and search! The result should look something like [what you see below](#).



The completed Bing search page, showing results for the query **Tango with Django**.

Well done! You have successfully added search to Rango and thus learnt how to integrate your app with an external API. Integrating other APIs is broadly the same – you may find that some provide their Python wrappers so you can literally import a module and query the API using Python commands.



Search Box Exercise

You may have noticed that when you issue a query, the query disappears when the results are shown. This is not very user-friendly, because research has shown that people like to *reformulate* their queries (or in different words, *tweak* their original query). Your exercise here is to figure out how to update the view and template so that the user's query is displayed within the search box when results are returned.



Hint

You need to think about how to provide an `input` text field with a predetermined, default `value`. You know the query – that's available to you in your `search()` view as the `query` variable. How can you make the query display in the `query` `input` field? What attribute does the `input` tag have that allows you to specify an initial value?

14. Making Rango Tango Exercises

So far, we have added in several key pieces of functionality to Rango. We've been building the app up in a manner that hopefully gets you familiar with using the Django framework and to learn how to construct the various parts of a modern web application.

However, Rango at the moment is not very cohesive or interactive. In this chapter, we challenge you to improve the app and its user experience further by bringing together some of the functionality we have implemented along with some new features. To make Rango more coherent, integrated and interactive, let's look at implementing the following functionality.

In Rango's Category and Page models, we've included a `views` field. However, we haven't used them yet, so why not implement the functionality to track views to both types of content? Specifically, we'll be looking at:

- counting the number of times that a category is viewed; and
- counting the number of times a page is clicked from a Rango category view.

We'll also be wanting to collect likes for different categories, as this is also something that we have not yet implemented. We'll be looking at implementing this with AJAX (see [AJAX in Django](#)). We'll also make use of AJAX to allow users to filter the categories that they see on the left-hand categories bar.

In addition to these new features, we'll be working to provide further services to registered users of Rango. Specifically, we'll be working to:

- allow users registering to the site to again specify a profile image and website (if you worked through the [chapter on django-registration-redux](#), this functionality will have been lost);
- let users view and edit their profile; and
- let users who are logged in view a list of other users and their profiles.



Note

As we have alluded to above, we won't be working through all of these tasks in this chapter. The AJAX-related tasks (as well as a definition of what AJAX is!) will be left to the [AJAX in Django](#) later on, while others will be left to you to complete as additional exercises.

Before we start to add this additional functionality, we will make a *todo* list to plan our workflow for each task. Breaking tasks down into smaller sub-tasks will greatly simplify the implementation process. We'll then be attacking each main activity with a clear plan. In this chapter, we'll be providing you with the workflow for a number of the above tasks. From what you have learnt so far, you should now be able to fill in the gaps and implement most of it on your own (except for those requiring AJAX, where we'll talk about that in more detail in a [later chapter](#)).

We've included hints, tips and code snippets to help you along. Of course, if you get stuck, you can always check out [our implementation for the exercises on GitHub](#)¹. Let's get started with tracking page clicks!

14.1 Tracking Page Clickthroughs

Currently, Rango provides direct links to external pages. While simple, this approach does not allow us to record how many clicks each page receives – a click takes you away from Rango without any ability to record information on what has just happened. To address this, we can implement a simple view to record a click – and *then* redirect the user to the request page.

This is what happens in many contemporary social media platforms, as an example – if someone sends you a link in Facebook Messenger, for instance, Facebook inserts a simple redirect device to be able to track that you clicked the link.

To implement functionality to track page links, have a go at the following steps.

1. First, create a new view called `goto_url()`. This should be mapped to the URL `/rango/goto/`, with a URL mapping name of `goto`.

¹https://github.com/maxwelld90/tango_with_django_2_code

- The `goto_url()` view will examine the HTTP GET request parameters, and pull out the `page_id` that is passed along with the request. The HTTP GET request will take the form of something like `/rango/goto/?page_id=1`. This means that we want to redirect the user to a `Page` model instance with an `id` of 1.
 - In the view, `get()` the `Page` with an `id` of `page_id` (from the GET request).
 - For that particular `Page` instance, increment the `views` field count by one, and then `save()` the changes you have made.
 - After incrementing the `views` field, redirect the user to the `page` instance's `url` value using the `redirect()` function, available at `django.shortcuts`.
 - If the `page_id` parameter is not supplied in the GET request, or an unknown `page_id` is supplied, then you should simply redirect the user to Rango's homepage. Again, you'll need to make use of the `redirect()` helper function, and the `reverse()` function to look up the URL for the Rango homepage. Error handling should mean that you structure your code with `try/except` blocks.
 - See [Django shortcut functions²](#) for more on `redirect` and [Django reverse function³](#) for more on `reverse`.
2. Update Rango's `urls.py` URL mappings to incorporate the new view to use the URL `/rango/goto/`. The mapping should also be given the name `goto`.
 3. Then you can update Rango's `category.html` template so that instead of providing a direct link to each page, you link to the new `goto_url()` view.
 - Remember to use the `url` template tag instead of hard-coding the URL into the template. `{% url 'rango:goto' %}?page_id={{page.id}}` should help.
 - Update the `category.html` template to also report the number of views that individual pages receive. Remember, watch out for grammar – use the singular for one view, and plural for more than one!
 4. Our final step involves updating the `show_category()` view. As we are now incrementing the `views` counter for each page when the corresponding link is clicked, how can you update the ORM query to return a list of pages, ordered by the number of clicks they have received? This should be in *descending* order, with the page boasting the largest number of clicks being ranked first.

²<https://docs.djangoproject.com/en/2.1/topics/http/shortcuts/>

³<https://docs.djangoproject.com/en/2.1/ref/urlresolvers/#django.urls.reverse>



GET Parameters Hint

If you're unsure of how to retrieve the `page_id` *querystring* from the HTTP GET request, the following code sample should help you.

```
page_id = None
if request.method == 'GET':
    page_id = request.GET.get('page_id')
    ...
```

Always check the request method is of type `GET` first. Once you have done that, you can then access the dictionary `request.GET` which contains values passed as part of the request. If `page_id` exists within the dictionary, you can pull the required value out with `request.GET.get('page_id')`.

You could also do this without using a *querystring*, but through the URL instead. This would mean that instead of having a URL that looks like `/rango/goto/?page_id=x`, you could create a URL pattern that matches to `/rango/goto/<int:page_id>/`. Either solution would work, but if you do this, you would obtain `page_id` in `goto_url()` from a parameter to the function call, rather than from the `request.GET` dictionary.

14.2 Searching Within a Category Page

Rango aims to provide users with a helpful dictionary of useful web pages. At the moment, the search page functionality is essentially independent of the categories. It would be nicer to have search integrated within the categories.

To implement this functionality, we will assume that a user will first browse through the category of interest. If they can't find a relevant page, they can then search. If they find a relevant page, then they can add it to the category.

For this section, let us focus on the first problem that entails putting search functionality on the category page. To do this, perform the following steps.

1. Remove the generic *Search* link from the navigation bar. This implies that we are decommissioning the global search functionality that we implemented

earlier. You can also comment out the URL mapping and view for search if you like, too. **Comment out, don't delete** – you'll be using the code to form the basis of your updated search functionality in subsequent steps!

2. Take the search form and results template markup from `search.html`, and place it into `category.html` *underneath* the list of pages. You'll only want to show the search functionality if the category exists in the database.
3. Update the search form definition such that the action refers back to the category URL, rather than the search URL.
4. Update the `show_category()` view to handle a HTTP POST request. The view must then include any search results in the context dictionary for the template to render. Remember that the search query will be provided as part of the POST request in field `query`.
5. Finally, update Rango so that only users who are logged in can view and use the updated search functionality. To restrict access within the `category.html` template, we can use something along the lines of the following code snippet.

```
{% if user.is_authenticated %}  
    <!-- Your search code goes here -->  
{% endif %}
```

The markup and template code for the third step can be seen below.

```
<form class="form-inline"  
      id="user-form"  
      method="post"  
      action="{% url 'rango:show_category' category.slug %}">
```



Commenting out Code

Commenting out code stops the interpreter or compiler from using the selected code. Although it cannot see it and execute it, you can still read it and use it. In Python, the simplest way to comment out a line of code is to prepend a `#` to the line – put it before any text.

Other languages will have different commenting syntax. HTML, for example, uses `<!--` to denote the start of a commented-out block of markup, and `-->` to denote the end.

14.3 Create and View User Profiles

If you have swapped over to the `django-registration-redux` package [as we worked on in an earlier chapter](#), then users who are registering won't be asked for a website or profile image. Essentially the `UserProfile` information is not being collected. To fix this issue, we'll need to change the steps that users go through when registering. Instead of simply redirecting users to the index page once they have successfully filled out the initial registration form, we'll redirect them to a new form to collect the additional `UserProfile` information.

To add the `UserProfile` registration functionality, you need to undertake the following steps.

1. First, create a `profile_registration.html` template, which will display the elements comprised from the `UserProfileForm`. This will need to be placed within the `rango` templates directory. Although it makes sense to place it in the `registration` directory, it is Rango specific; as such, it should live in the `rango` directory.
2. Create a new `register_profile()` view in Rango's `views.py` to capture details required by a `UserProfile` instance.
3. Finally, change where you redirect newly-registered users. We'll have to provide you with some code to show you how to do this. You can find that in the [next chapter](#).

It would also be useful to let users inspect and edit their profiles once they have been created. To do this, undertake the following steps.

1. Create a new template called `profile.html`. This should live in the `rango` templates directory. Within this template, add in the fields associated with the `UserProfile` and the `User` instances, such as the username, website and user profile image.
2. Create a new view in Rango's `views.py`. This view will obtain the necessary information required to render the `profile.html` template.
3. Map the URL `/rango/profile/` to your new view. The mapping should have a name of `profile`.

4. Finally, you should add a link in the base.html navigation bar called *Profile*. This link should only be available to users who are logged in (those that pass the check `{% if user.is_authenticated %}`).

To allow users to browse through other user profiles, you can also create a *users page* that lists all users registered on Rango. If you click on a username, you will then be redirected to their *profile page*. However, **you must ensure that a user is only able to edit their profile**, and that logged in users can see the users page!

To allow users to view different profiles, you'll need to tweak the URL mapping to allow for the provision of a username, too. Instead of simply using /rango/profile/, you should allow for a username using the mapping /rango/profile/<username>/. For this to work, you will, of course, need to update the view you create, too.



Referencing Uploaded Content in Templates

If you have completed all of the [Templates and Media Files chapter](#), your Django setup should be ready to deal with the uploading and serving of user-defined media files (in this case, profile images). You should be able to reference the `MEDIA_URL` URL (defined in `settings.py`) in your templates through use of the `{{ MEDIA_URL }}` tag, provided by the [media template context processor⁴](#). We asked you to try this out as an exercise much earlier in the book. Displaying an image `cat.jpg`, the associated HTML markup would be ``.

The next chapter provides you with complete solutions to each of the exercises we outlined here. Try and do as much of them as you can by yourself without turning over to the next chapter!

⁴<https://docs.djangoproject.com/en/2.1/howto/static-files/>

15. Making Rango Tango Hints

Hopefully, you will have been able to complete the exercises in the previous chapter using only the workflows we provided. If not, or if you need a little push in the right direction, this chapter is for you. We provide model solutions to each of the exercises we set, and you can incorporate them within your version of Rango if you need some help.



Got a better solution?

The solutions provided in this chapter are only one way to solve each problem. There are many ways you could approach each problem, and use solutions that exclusively use techniques that we have learnt so far. However, if you successfully implement them differently, please feel free to share your solution(s) with us – and tweet links to [@tangowithdjango¹](#) for others to see!

15.1 Track Page Clickthroughs

As we said when we introduced [this problem earlier](#), Rango provides only a direct link to the pages of external pages saved to each category. This approach limits our ability to track (or simply count) the number of times a particular link is clicked. To be able to track clicks, we'll need to work on the following steps. The subsections we provide here correspond to the four main steps we [outlined earlier](#) in our workflow.

Creating a URL Tracking View

First, create a new view called `goto_url()` in Rango's `views.py` module. The view which takes a parameterised HTTP GET request (i.e. `rango/goto/?page_id=1`) and updates the number of views for the page. The view should then redirect to the actual URL.

¹<https://www.twitter.com/tangowithdjango>

```
def goto_url(request):
    if request.method == 'GET':
        page_id = request.GET.get('page_id')

        try:
            selected_page = Page.objects.get(id=page_id)
        except Page.DoesNotExist:
            return redirect(reverse('rango:index'))

        selected_page.views = selected_page.views + 1
        selected_page.save()

    return redirect(selected_page.url)

return redirect(reverse('rango:index'))
```

Be sure that you import the `redirect()` function to `views.py` if it isn't included already! As the function defined above also makes use of `reverse()` to perform URL lookups, you'll want to make sure that is included, too – it should be present from prior efforts, however.

```
from django.shortcuts import redirect
from django.urls import reverse
```

Mapping the View to a URL

The second major step involves mapping the new `goto_url()` view to the URL `/rango/goto/`. To do this, update Rango's `urls.py` module. Add the following code to the `urlpatterns` list.

```
path('goto/', views.goto_url, name='goto'),
```

Note that we are complying with the specification in the previous chapter and using a mapping name of `goto`.

Updating the category.html Template

The third step involves updating the category.html template. We were tasked to implement two changes, the first of which changed page links to use the new goto_view() view, rather than providing a direct URL link. Secondly, we were tasked to report back to users how many clicks each page had received.

Find the block of code that handles looping through the pages context variable. Update it accordingly.

```
{% for page in pages %}
    <li>
        <a href="{% url 'rango:goto' %}?page_id={{ page.id }}">{{ page.title }}</a>
        {% if page.views > 1 %}
            ({{ page.views }} views)
        {% elif page.views == 1 %}
            ({{ page.views }} view)
        {% endif %}
    </li>
{% endfor %}
```

Notice the change to the URL's href attribute, and the inclusion of some new template code to control what is displayed immediately after the hyperlink – a count on the number of views for the given page. As we also check how many clicks each page has received (one or more clicks?), we can also control our grammar, too!

Updating the Category View

The fourth and final step for this particular exercise was to update the existing show_category() view to reflect the change in the way we present our list of pages for each category. The specification now requires us to order the pages for each category by the number of clicks each page has received. This has to be descending, meaning the page with the largest number of clicks will appear first.

This involves the simple addition of chaining on an order_by() call to our ORM request. Find the line dealing with the Page model in show_category() and update it to look like the line shown below.

```
pages = Page.objects.filter(category=category).order_by('-views')
```

Now that this is all done, confirm it all works by clicking on categories and then pages. Go back to the category and refresh the page to see if the number of clicks has increased. Remember to refresh your browser; the updated count may not show up straight away!

15.2 Searching Within a Category Page

The main aim of Rango is to provide users of the app with a helpful directory of page links. At the moment, the search functionality is essentially independent of the categories. It would be better to integrate the search functionality within an actual category page.

Let us assume that a user will first navigate to and browse their category of interest first. If they cannot find the page that they want, they will then be able to search for it. Once they examine their search results and find the page they are looking for, they will be able to add the page to the category they are browsing.

We'll tackle the first part of the description here – adding search functionality to the category page. To accomplish this, we first need to remove the [search functionality that we added in a previous chapter](#). This will essentially mean decommissioning the current search page and associated infrastructure (including the view and URL mapping). After this, there are several tasks we will need to undertake. The subsections listed here again correspond to the five main steps we outlined in the [previous chapter](#).



Don't Delete your Code!

When decommissioning your existing search functionality, don't delete it. Simply comment things out where appropriate (such as in the `urls.py` module). You'll be copying some of the code across to a new home later on, anyway.

Decommissioning Generic Search

The first step for this exercise is to decommission the existing search functionality that you implemented in a [previous chapter](#).

1. First, open Rango's `base.html` template and find the navigation bar markup (found at the top of the page). Locate the `` element for the `Search` link that you added earlier on and delete it. This needs to be deleted as you'll be commenting out the URL mapping shortly, meaning that a reverse URL lookup for `rango:search` will no longer work. Wrapping this with an HTML comment tag (`<!-- ... -->`) won't work either, as the Django template engine will still process what's inside of it!
2. Second, open Rango's `urls.py` module and locate the URL mapping for the `/rango/search/` URL. Comment this line out by adding a `#` to the start of the line. This will prevent users from reaching the search URL.
3. Finally, open Rango's `views.py` module and locate the `search()` view you implemented previously. Again, comment out this function by prepending a `#` to the start of each line.

You'll still have the `search.html` template in Rango's templates directory; don't remove this as we will be using the contents of this template as the basis for integrating search functionality within the category template.

Migrating `search.html` to `category.html`

As we will be wanting to provide search functionality to users who are browsing a category, we need to add in the search presentation functionality (e.g. displaying the search form and results area) to the `category.html` template. This is essentially a simple copy and paste job!

Let's split this into two main steps. First, open your decommissioned `search.html` template and locate the `<div>` element containing the entire search form. This will be the `<div>` whose child is a `<form>` element. Select and copy the `<div>` and its contents, then open Rango's `category.html` file.

We now need to paste the code from `search.html` into `category.html`. As the brief for this problem was to add the search functionality *underneath* the list of pages in the category, find and locate the link inside the `{% block body_block %}` to add a page. Paste the contents from `search.html` underneath that link.

You can then go back to `search.html` and repeat the process, this time selecting the `<div>` for displaying the results list. You can identify this by looking for the `<div>` with Django template code that iterates through the `result_list` list. Copy that and then move back over to `category.html`. Now paste that in underneath the `<div>` containing the form that you previously pasted in.

Updating the `category.html` Search Form

Now that the markup required has been added to `category.html`, we need to make one simple change. Instead of submitting the contents of the search form to the `/rango/search/` URL, we instead will simply direct submitted responses to the category URL (`/rango/category/<slug:category_name_slug>`). This is as simple as finding the `<form>` definition in `category.html` and changing the `action` attribute to Rango's `show_category()` URL mapping.

```
<form class="form-inline"
      id="user-form"
      method="post"
      action="{% url 'rango:show_category' category.slug %}">
```

Updating the `show_category()` View

You should have identified that since we are now redirecting search requests to the `show_category()` view, we'll need to make some changes within that view so that it can handle the processing of the search request, as well as being able to handle the generation of a list of pages for a given category.

This is again a relatively straightforward process in which we update the view based upon code from our decommissioned `search()` view. We provide the complete listing of our model `show_category()` view below. Notice the comments denoting the start of the code we have taken from our existing `search()` view.

```
1 def show_category(request, category_name_slug):
2     context_dict = {}
3
4     try:
5         category = Category.objects.get(slug=category_name_slug)
6         pages = Page.objects.filter(category=category).order_by('-views')
7
8         context_dict['pages'] = pages
9         context_dict['category'] = category
10    except Category.DoesNotExist:
11        context_dict['category'] = None
12        context_dict['pages'] = None
13
14    # Start new search functionality code.
15    if request.method == 'POST':
16        if request.method == 'POST':
17            query = request.POST['query'].strip()
18
19        if query:
20            context_dict['result_list'] = run_query(query)
21    # End new search functionality code.
22
23    return render(request, 'rango/category.html', context_dict)
```

We keep the `show_category()` view the same at the top, and add in an additional block of code towards the end to handle the processing of a search request. If a POST request is made, we then attempt to take the `query` from the request and issue the query to the `run_query()` function we defined in the [Bing Search chapter](#). This then returns a list of results which we put into the `context_dict` with a key of `result_list` – exactly the variable name that is expected in the updated `category.html` template.

Search functionality should then all work as expected. Try it out! Navigate to a category page, and you should see a search box. Enter a search query, submit it, and see what happens.

Once you are satisfied, you can safely delete the existing `search.html` template from your codebase, along with the decommissioned `search()` view.

Restricting Access to Search Functionality

Our final requirement was to restrict the migrated search functionality only to those who are logged into Rango. This step is straightforward – one can simply wrap the search-handling template code added to `category.html` with a conditional template check to determine if the user is authenticated.

```
{% if user.is_authenticated %}  
<div>  
    ...  
</div>  
{% endif %}
```

Too easy! You could also go further and add checks within the `show_category()` view to ensure the search functionality part is not executed when the current user is not logged in. Be wary, though – don't be inclined to add the `login_required()` decorator to the view. Doing so will restrict all category-viewing functionality to logged in users only – you only want to restrict the *search* functionality!

Query Box Exercise

At the end of the Bing Search API chapter, we set an exercise. We noted that in its current state, users would issue a query and then be presented with the results. However, the query box would then be blanked again – thus making *query reformulation* more taxing.

In our code examples above, we've deliberately kept our model solution to this particular exercise out. How could you allow the results page to '*remember*' the query that was entered, and place it back in the search box?

The easiest solution to this problem would be to simply place the `query` variable into the `context_dict` of `show_category()`, and then make use of the variable within the `category.html` template by specifying its value as the `value` attribute for the search box `<input>` element.

In Rango's `show_category()` view, locate the code block that deals with the search functionality, and add the `query` to the `context_dict`, like so.

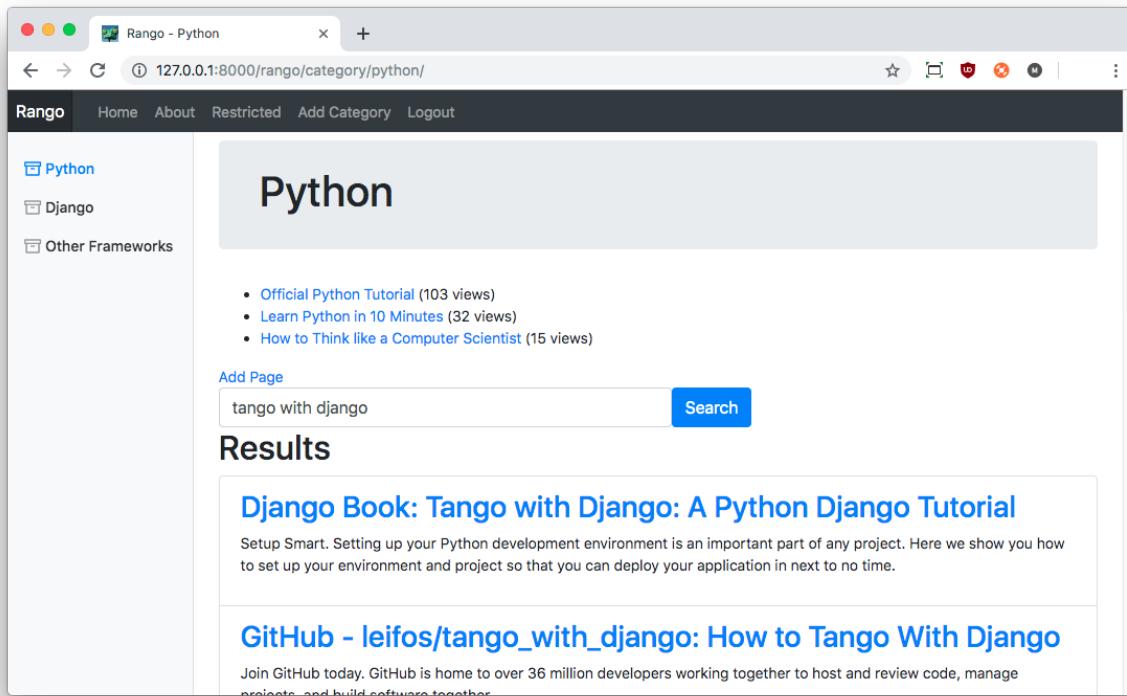
```
# Start new search functionality code.  
if request.method == 'POST':  
    if request.method == 'POST':  
        query = request.POST['query'].strip()  
  
        if query:  
            context_dict['result_list'] = run_query(query)  
            context_dict['query'] = query  
# End new search functionality code.
```

Once this has been completed, open Rango's category.html template and modify the query <input> field like so.

```
<input class="form-control"  
      type="text"  
      size="50"  
      name="query"  
      value="{{ query }}"  
      id="query" />
```

Template variable {{ query }} will be replaced with the user's query, thus setting it to be the default value for the <input> field when the page loads.

Once everything has been completed, you should have a category page that looks similar to the example below. Well done!



Rango's updated category view, complete with Bing Search API search functionality. Note also the inclusion of the search terms in the query box – they are still retained!

15.3 Creating a UserProfile Instance

This section provides one solution for creating Rango UserProfile accounts in conjunction with the `django-registration-redux` package. Recall that the standard `django.contrib.auth.models.User` model contains several standard fields related to user accounts, such as a `username` and `password`. However, we chose to implement an additional `UserProfile` model within Rango to store additional information such as a user's website URL, and a profile image. Here, we'll go through the steps required to implement this functionality. The steps that we'll work on are listed below, and each corresponds to a subsection below.

1. We first will create a new `profile_registration.html` template that will be our pre-existing `UserProfileForm` Django form.

2. We'll then work on creating a new `register_profile()` view to capture the details for a new `UserProfile` instance.
3. After that, the standard procedure applies. We'll first map the new view to a new URL – in this instance, the URL will be `/rango/register_profile/`. Even though this details with user accounts, it is *Rango specific*, so it makes sense to place it within the `/rango/` URL pattern.
4. We'll then need to write some code for the `django-registration-redux` package to tell it to redirect to a different place when a `User` object has been created – or in other words, redirect the user to the new `/rango/register_profile/` URL.

The fourth step requires some additional code that requires some specialist knowledge of the `django-registration-redux` package. We've taken care of that for you, but we'll point to the relevant parts of the associated documentation to show you how we figured everything out.

Once complete, the basic flow/experience for a user registering with Rango will be as follows.

1. The user will jump to the Rango website.
2. They will then click the `Register Here` link on the navigation bar.
3. They will then be redirected to the `django-registration-redux` registration form. This form is located at the URL `/accounts/register/`.
4. Once this form has been completed, the form will be submitted and processed. If successful, they will then be redirected to the new `/rango/register_profile/` page, allowing them to create a `UserProfile` instance.
5. Once this has been completed and submitted, the user will be redirected to the homepage. Registration will then be complete.

Order here is important as the way we implement the new `register_profile()` view will assume that a user has created a standard `User` instance beforehand. This will be required to link to their new `UserProfile` instance and ensure referential integrity is maintained!

Creating a Profile Registration Template

First, let's focus on creating a simple template that will provide the necessary HTML markup for displaying the `UserProfileForm` form fields. As we mentioned previously, we'll be keeping the `django-registration-redux` forms separate from the new profile registration form – this new template will belong in the `rango` templates directory. Remember, we're not dealing with built-in Django models here – we are dealing with the creation of an instance of a custom-made model.

Create a new template in `templates/rango/` called `profile_registration.html`. Within this new template, add in the following HTML markup and Django template code.

```
1  {% extends 'rango/base.html' %}            
2  {% load staticfiles %}                    
3  
4  {% block title_block %}                  
5      Register - Step 2                 
6  {% endblock %}                        
7  
8  {% block body_block %}                  
9  <div class="jumbotron p-4">             
10 <div class="container">                   
11     <h1 class="jumbotron-heading">Register Here - Step 2</h1>      
12 </div>                                  
13 </div>  
14  
15 <div class="container">                  
16   <div class="row">                    
17     <form method="post"               
18       action="{% url 'rango:register_profile' %}"          
19       enctype="multipart/form-data">                      
20       {% csrf_token %}                  
21       {{ form.as_p }}                
22       <input type="submit" value="Create Profile" />      
23     </form>                              
24   </div>                              
25 </div>                              
26 {% endblock %}
```

Like the `django-registration-redux` forms that we created previously (with an example [here](#)), this template inherits from Rango's `base.html` template. Recall that

this template incorporates the basic layout for our Rango app. We also create an HTML `<form>` within the `body_block` block. This will be populated with fields from the `form` objects that we'll be passing to the template from the corresponding view – we'll work on this next. You should also be aware of the new URL mapping that we use, `register_profile`. We'll be defining this in the third step.



Don't Forget `multipart/form-data`!

When creating your form, don't forget to include the `enctype="multipart/form-data"` attribute in the `<form>` tag. We need to set this to **instruct the web browser and server that no character encoding should be used²**. Why? Here, we could be performing *file uploads* (a user profile image). If you don't include this attribute, uploading images with this form will not work.

Creating a Profile Registration View

The second step in this process is to create the corresponding view for our new `profile_registration.html` template. This new view will handle the processing of the `UserProfileForm` form we created way back in the [User Authentication chapter](#), and instructing Django to render any response with our new template.

By now, this sequence of actions should be pretty straightforward for you to implement. Handling a form means being able to handle a request to render the form (via an HTTP GET request), and being able to process any entered information (via an HTTP POST request). A possible implementation for this view is shown below. You can add it to Rango's `views.py` module.

²<https://stackoverflow.com/questions/4526273/what-does-enctype-multipart-form-data-mean>

```
@login_required
def register_profile(request):
    form = UserProfileForm()

    if request.method == 'POST':
        form = UserProfileForm(request.POST, request.FILES)

        if form.is_valid():
            user_profile = form.save(commit=False)
            user_profile.user = request.user
            user_profile.save()

            return redirect(reverse('rango:index'))
    else:
        print(form.errors)

    context_dict = {'form': form}
    return render(request, 'rango/profile_registration.html', context_dict)
```

Remember to check that you also have the relevant import statements at the top of the views.py module. You should have them all by now from previous chapters, but nevertheless it's good to check!

```
from django.contrib.auth.decorators import login_required
from rango.forms import UserProfileForm
from django.shortcuts import render, redirect
```

Upon creating a new UserProfileForm instance, we then check our request object to determine if a GET or POST request was made. If the request was a POST, we then recreate the UserProfileForm, this time using data gathered from the POST request (request.POST). As we are also handling a file image upload (for the user's profile image), we also need to pull the uploaded file from request.FILES. We then check if the submitted form was valid – meaning that the form fields were filled out correctly. In this case, we only really need to check if the URL supplied is valid – since the URL and profile picture fields are optional (we specified the blank=True attribute for both the website and picture fields in the UserProfile model).

With a valid UserProfileForm, we can then create a new instance of the UserProfile model with the line user_profile = form.save(commit=False). Setting commit=False

gives us the time to manipulate the new `UserProfile` instance that is created before we commit it to the database. This is where we can then add in the necessary step to associate the new `UserProfile` instance with the associated `User` instance (refer to the user flow/experience list at the top of this section to refresh your memory). After then saving the new `UserProfile` instance, we then redirect the user with a new account to Rango's index view, using the `reverse()` URL lookup. If form validation failed for any reason, errors are simply printed to the console. You'll most likely want to deal with this in a better way to make error handling more robust, and thus more intuitive for the user.

If the request was instead sent as an HTTP GET, the user simply wants to request a blank form to fill out. In this scenario, we respond by simply rendering the `rango/profile_registration.html` template created above with a blank instance of the `UserProfileForm`, passed to the rendering context dictionary as `form`. Doing this satisfies the requirement that we created previously in our new template.

Therefore, this solution should handle all required scenarios for creating, parsing and saving data from a `UserProfileForm` form.



Why use `login_required`?

Remember, once a newly registered user hits this view, they will have had a new account created for them. This means that we can safely assume that he or she is now logged into Rango. This is why we are using the `@login_required` decorator at the top of our view to prevent individuals from accessing the view when they are unauthorised to do so.

See the [Django Documentation](#) for more details about Authentication³.

Mapping the `register_profile()` View to a URL

Now that our template and corresponding view have been implemented, a seasoned Djangoer should now be thinking: *map it!* We need to map our new view to a URL so that users can access the newly created content. Opening up Rango's `urls.py` module and adding the following line to the `urlpatterns` list will achieve this.

³<https://docs.djangoproject.com/en/2.1/topics/auth/default/>

```
path('register_profile/', views.register_profile, name='register_profile'),
```

This maps out new `register_profile()` view to the URL `/rango/register_profile/`. Remember, the `/rango/` part of the URL comes from your *project's* `urls.py` module. The remainder of the URL is then handled by Rango's `urls.py`.

Modifying the Registration Flow

Now that everything is (almost!) working, we need to tweak the process – or flow – that users undertake when registering. To do this, we need to do some *overriding* within the `django-registration-redux` package. Specifically, what we need to do is be able to tell the package where to redirect users who successfully create an account. We want to redirect them to the new `register_profile()` view!

To accomplish this, we need to use something called a *class-based view* to override a method provided by the `django-registration-redux` package. Given that we are working towards overriding a URL, it makes sense to add this to your *project's* `urls.py` – the one that lives in the `tango_with_django_project` directory. Add the following code *above* the definition of your `urlpatterns` list, but after the `import` statements.

```
class MyRegistrationView(RegistrationView):
    def get_success_url(self, user):
        return reverse('rango:register_profile')
```

As we extend this class from the `RegistrationView` class, we need to add this to our imports at the top of the `urls.py` module. As we are also using the `reverse()` helper function, we'll need to import that, too!

```
from registration.backends.simple.views import RegistrationView
from django.urls import reverse
```

With this complete, we now need to tell our project what to do with this extended class! It's just sitting there by itself at the moment, doing nothing. To tell the `django-registration-redux` package to use this class, we also need to update the `urlpatterns` list by adding a new entry. **Make sure you add this line directly above**

the existing entry for the accounts/ URL – not after! To ensure you get it in the right place, we have included a complete copy of what your `urlpatterns` list should look like.

```
urlpatterns = [
    path('', views.index, name='index'),
    path('rango/', include('rango.urls')),
    path('admin/', admin.site.urls),

    # New line below -- don't forget the slash after register!
    path('accounts/register/',
        MyRegistrationView.as_view(),
        name='registration_register'),

    path('accounts/', include('registration.backends.simple.urls')),
] + static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
```

This overrides the existing `registration_register` URL mapping, provided to us in the standard, out-of-the-box `django-registration-redux` `urls.py` module, and replaces the URL we jump to with a mapping to our new class-based view.

15.4 Class-Based Views

What is a class-based view, though? What *exactly* have we made implement up in the example solution above? Class-based views are a more sophisticated and elegant mechanism for handling requests. Rather than taking a functional approach as we have done so far in this tutorial – that is creating *functions* in Rango’s `views.py` module to serve requests – the class-based approach means creating classes that inherit and implement a series of methods to handle your app’s requests.

For example, rather than checking if a request was an HTTP GET or an HTTP POST request, class-based viewed allow you to implement a `get()` and `post()` method within the class handling your request for a given view. When your project and handlers become more complex, using a class-based approach is the more preferable solution. You can look at the [Django Documentation for more information about class-based views⁴](#).

⁴<https://docs.djangoproject.com/en/2.1/topics/class-based-views/>

To give you more of an idea about how class-based views work, let's try converting the `about()` view from a function-based to a class-based approach. First, in Rango's `views.py` module, define a class called `AboutView` which inherits from the `View` base class. This base class needs to be imported from `django.views`, as shown in the example code below.

```
from django.views import View

class AboutView(View):
    def get(self, request):
        context_dict = {}

        visitor_cookie_handler(request)
        context_dict['visits'] = request.session['visits']

        return render(request,
                      'rango/about.html',
                      context_dict)
```

Note that we have simply taken the code from the existing `about()` function-based view, and added it to the `get()` method of our new class-based `AboutView` approach.



Defining Methods in Classes

When defining a method within a class in Python, the method must always take at minimum one parameter, `self`. `self` denotes the instance of the class being used, and you can use it to obtain access to other methods within the instance, or instance variables defined using the `self.object_name`. This is the same technique as accessing instance-specific items in Java using the `this.` approach.

You *can* create static methods in Python classes, meaning that the `self` parameter need not be passed – but you should use the `staticmethod` decorator. This is more advanced, and we don't make use of static methods here. However, you can [read more about this online⁵](#).

Next, you need to update Rango's `urls.py` module. Find and update the entry in the `urlpatterns` list that deals with the `about` mapping to make use of the new `AboutView`.

⁵<https://docs.python.org/3/library/functions.html#staticmethod>

```
urlpatterns = [
    ...
    #Updated path that point to the new about class-based view.
    path('about/', views.AboutView.as_view(), name='about')
    ...
]
```

As our existing `import` statement simply imports Rango's `views.py`, we must specify `views.AboutView` to tell Python exactly where the new class is. You can always of course `import` the `AboutView` separately, like in the example below.

```
from rango.views import AboutView

urlpatterns = [
    ...
    #Updated path that point to the new about class-based view.
    path('about/', AboutView.as_view(), name='about')
    ...
]
```

Note that we also call the `as_view()` function. This is part of the base `View` class that provides the necessary code for Django to be able to process the view using your logic defined in the `get()` method.

For such a simple view, you may think that switching to a class-based approach doesn't save you much time or space. However, you should now get the idea – and you should now be able to begin refactoring more complex views to use the class-based approach.

A more complex view that considers both a HTTP GET and HTTP POST could be the `add_category()` view. To convert this from a functional- to class-based view, we can create a new `AddCategoryView` class in Rango's `views.py`. From there, we can begin the process of moving our code over from the function to the new class, as we show below. Look at the code in the `get()` and `post()` methods – where does it come from?

```

class AddCategoryView(View):
    @method_decorator(login_required)
    def get(self, request):
        form = CategoryForm()
        return render(request, 'rango/add_category.html', {'form': form})

    @method_decorator(login_required)
    def post(self, request):
        form = CategoryForm(request.POST)

        if form.is_valid():
            form.save(commit=True)
            return redirect(reverse('rango:index'))
        else:
            print(form.errors)

        return render(request, 'rango/add_category.html', {'form': form})

```

Compare the code you've got currently written in your `add_category()` function-based view against the code we have above – the logic for a HTTP `get()` request is mirrored in the `get()` method, with the same for a HTTP `POST` request in `post()`. You'll also notice that we are making use of a new decorator function called `method_decorator`. This needs to be imported, so add the following `import` statement to the top of your `views.py` module.

```
from django.utils.decorators import method_decorator
```

By associating the `@method_decorator(login_required)` line for both the `get()` and `post()` methods, we are ensuring that neither method can be accessed by users who are not logged into Rango. If we were to drop the decorator from the `get()` method, what do you think would happen? Users would see the form to add a new category, but given that the `post()` method is still protected, those who are not logged in wouldn't be able to create a new category!



Why do we need `@method_decorator`?

`@method_decorator` is required because function decorators [need to be transformed to method decorators](#)⁶ – `method_decorator()` achieves this for us.

⁶<https://docs.djangoproject.com/en/2.2/topics/class-based-views/intro/#decorating-the-class>

One last thing – we also need to update Rango’s `urls.py` module to reflect the fact we now want to use the class-based view. In the `urlpatterns` list, locate the `add_category` mapping, and change the entry to look like the example shown below.

```
path('add_category/', views.AddCategoryView.as_view(), name='add_category'),
```

Looking carefully at the example, we’re only changing the target view. Instead of pointing to the `add_category()` view, we instead point to our new `AddCategoryView` class – and call the `as_view()` method that Django provides which passes control to the relevant method for us depending on the request received (do I want to execute `get()` or `post()`?).



Class-based View Exercises

Now that you’ve seen the basics on how to implement class-based views in Django, it’s a good time for you to put this to the test.

- Have a look at the [Django documentation](#)⁷ and look for some more examples on how you can create class-based views.
- Update your Rango app to make sure that *all* views in Rango’s `views.py` module make use of class-based views!
 - As a time-saving tip, remember that *both* the `urls.py` modules you’ve been working on reference your current `index()` view! Remember that when you implement a class-based alternative.

This won’t take as long as you think. The hardest part here is for the more complex views with both HTTP GET and HTTP POST functionality. Being able to delineate between the flow for each type of request and put the necessary code in separate methods will test your understanding of the code!

There’s a couple of things we also want you to think about when you work through this exercise. We’ve listed them below as individual tips.

⁷<https://docs.djangoproject.com/en/2.1/topics/class-based-views/>



Naming Conventions

It's good practice to make sure your new class-based views conform to [Python naming conventions⁸](#). For classes, use CamelCase. Capitalise the start of a class name, then capitalise each subsequent term – without spaces between them. Hence the new add category view becomes `AddCategoryView`. Tack `View` on the end to make sure those who look at your code later will know they are dealing with a class-based view.

Methods within your classes should follow the same convention as naming functions. Use lower case names, and separate individual terms with underscores – like `some_method_name()`.



Mapping the IndexView

When you create a class-based view for your old `index()` view, be wary that you will have mapped it in both Rango's `urls.py` module *and your project's urls.py module, too!*



Passing URL parameters

Earlier on, you created a view `show_category()`. Aside from taking the obligatory `request` object, it also took a `category_name_slug` – it was a *parameterised view*, with the parameter coming from the URL. For all views implemented like this, you'll need to make sure that their class-based equivalent not only contain `self` and `request` as parameters, but the one or more URL parameters, too. In the example above, a `get()` method definition may look like the example below.

```
class ShowCategoryView(View):
    def get(self, request, category_name_slug):
        ...
```

If you're dealing with a view with both `get()` and `post()` methods, you'll need the parameter(s) for both!

⁸<https://www.python.org/dev/peps/pep-0008/>



Classes and Helper Methods

You may find that when you're implementing your class-based views that you are repeating yourself. For instance, in the `show_category()` function-based view, we have some reasonably extensive code that deals with populating a dictionary for use as the context dictionary when we pass it to `render()`. This code is executed for *both* HTTP GET and HTTP POST requests. In a class-based implementation, one solution would be to simply add the code to the `get()` and `post()` methods... but remember, **DRY**⁹! How could you engineer a solution that means you only need one instance of the context dictionary generating code?

A possible solution would be to create a *helper method* within your class-based implementation. Name it whatever – but to be able to successfully generate the basics for the context dictionary of the `show_category()` view, you'll also need the `category_name_slug`. How could you implement this? We give you a hint in the tip above. Once you have cracked it, you'll also be able to apply the same technique to other class-based views within your code.

15.5 Viewing your Profile

With the code now in place to reinstate the use of the `UserProfile` model, we can now implement functionality to allow a user to view his or her profile – and edit it. The process is again similar to what we have done many times before. We'll need to consider the following three steps.

1. First, we will need to create a new template, this time called `profile.html`. This will live inside the `rango/templates` directory.
2. We'll have to create a new view that renders the new template. We'll make this one a class-based view from the outset!
3. Finally, the new view must be mapped to a URL – in this case, `/rango/profile`.

Of course, to make the new page accessible, we'll also need to provide an additional hyperlink in Rango's base `.html` template to provide simple access. For this solution,

⁹[https://en.wikipedia.org/wiki/Don't_repeat_yourself](https://en.wikipedia.org/wiki/Don%27t_repeat_yourself)

we'll be creating a generalised view that allows you access the information of any user of Rango. The code will also allow users who are logged into the app to also edit their profile – *but only theirs* – thus satisfying the requirements of this exercise.

Creating the Template

First, let's work on creating a simple template for displaying a user's profile. The following HTML markup and Django template code should be placed within the new `profile.html` file within Rango's templates directory.

```
{% extends 'rango/base.html' %}  
{% load staticfiles %}  
  
{% block title_block %}  
    Profile for {{ selected_user.username }}  
{% endblock %}  
  
{% block body_block %}  
<div class="jumbotron p-4">  
    <div class="container">  
        <h1 class="jumbotron-heading">{{ selected_user.username }}'s Profile</h1>  
    </div>  
</div>  
  
<div class="container">  
    <div class="row">  
          
        <br />  
        <div>  
            {% if selected_user == user %}  
                <form method="post" action=". " enctype="multipart/form-data">  
                    {% csrf_token %}  
                    {{ form.as_p }}  
  
                    <input type="submit" value="Update" />  
                </form>  
            {% else %}  
                <p>
```

```
<strong>Website:</strong>
<a href="<% user_profile.website %}">% user_profile.website %</a>
</p>
{%
  endif %}
</div>
</div>
{%
  endblock %}
```

Note that in the template, we make use of a few variables (namely `selected_user`, `user_profile` and `form`) that we need to make sure are defined in the template's context. In this template, we assume that:

- `selected_user` represents a Django `User` instance of the user whose profile we want to display;
- `user_profile` represents a Rango `UserProfile` instance of the associated `User` that we want to display; and
- `form` represents a `UserProfileForm` instance that is used to display the necessary fields to allow a user to edit his or her profile (i.e. website and/or profile image), with the website field pre-populated where necessary.

We'll be making sure that this template is given the required variables to its context in the next section.

The template block of interest here is, of course, the `body_block` block. At the top, we display the selected user's profile image, set to dimensions of 300x300 pixels. We also provide alt text to display if the image cannot be located.

However, the interesting part of this template is underneath. We use a conditional statement to work out if the user who is currently looking at a given profile is the said user (`selected_user == user`) – and if he or she is, we display a form to allow the user to edit their profile. If the user does not match up, then we don't want to provide a form to edit – so instead, we simply display the profile.

You should also note that we once again use `enctype="multipart/form-data"` when defining our `<form>` because users can upload a new profile image – remember, whenever file uploads are involved, this attribute **must** be specified.

Creating the `profile()` View

Based on the template created above, we can then implement a simple view to handle the viewing of user profiles and submission of form data. In Rango's `views.py` module, create a new class-based view called `ProfileView`. Add the code as we show below to the new view.

```
class ProfileView(View):
    def get_user_details(self, username):
        try:
            user = User.objects.get(username=username)
        except User.DoesNotExist:
            return None

        user_profile = UserProfile.objects.get_or_create(user=user)[0]
        form = UserProfileForm({'website': user_profile.website,
                               'picture': user_profile.picture})

        return (user, user_profile, form)

    @method_decorator(login_required)
    def get(self, request, username):
        try:
            (user, user_profile, form) = self.get_user_details(username)
        except TypeError:
            return redirect(reverse('rango:index'))

        context_dict = {'user_profile': user_profile,
                       'selected_user': user,
                       'form': form}

        return render(request, 'rango/profile.html', context_dict)

    @method_decorator(login_required)
    def post(self, request, username):
        try:
            (user, user_profile, form) = self.get_user_details(username)
        except TypeError:
            return redirect(reverse('rango:index'))

        form = UserProfileForm(request.POST, request.FILES, instance=user_profile)
```

```

    if form.is_valid():
        form.save(commit=True)
        return redirect('rango:profile', user.username)
    else:
        print(form.errors)

    context_dict = {'user_profile': user_profile,
                   'selected_user': user,
                   'form': form}

    return render(request, 'rango/profile.html', context_dict)

```

We'll also need to import two classes that we haven't used before. One is the standard Django User model, and the other is Rango's UserProfile model. Note we did import the UserProfileForm earlier – that's different!

```

from django.contrib.auth.models import User
from rango.models import UserProfile

```

Our new class-based view requires that a user be logged in – hence the use of the `@method_decorator(login_required)` decorator for both the `get()` and `post()` methods.

Within the `ProfileView` class, we also created a `get_user_details()` helper method to ensure that we don't repeat ourselves. This method begins by selecting the `django.contrib.auth.User` instance from the database – if it exists. If it does not exist, the method simply returns `None`, which signals to both the `get()` and `post()` methods to do a simple redirect to Rango's homepage, rather than greet the user with a bespoke error message. We can't display information for a non-existent user! If the user does exist, the `get_user_details()` method selects the `UserProfile` instance (or creates a blank one, if one does not exist). We then take the selected user's details, and use those details to populate a `UserProfileForm` instance. The `user`, `user_profile` and populated `form` objects are then returned in a tuple and are `unpacked10` by the `get()` and `post()` methods that call the helper method.

Within the `post()` method, we handle the case where a user wants to update their profile's information. To do this, we extract information from the form into a

¹⁰<https://www.geeksforgeeks.org/packing-and-unpacking-arguments-in-python/>

`UserProfileForm` instance that can refer to the `UserProfile` instance that it is saving to – rather than creating a new `UserProfile` instance each time. Remember, we are *updating*, not creating! A valid form is then saved. An invalid form (or an HTTP GET request) triggers the rendering of the `profile.html` template, complete with the relevant variables that are passed through to the template via its context.



Authentication Exercise

How can we change the code above to prevent unauthorised users from changing the details of a user account that isn't theirs? What conditional statement do we need to add to enforce this additional check?

Mapping the View to a URL

We then need to map our new `ProfileView` to a URL. This involves editing Rango's `urls.py` module. We want to add a further line to the `urlpatterns` list, as shown below.

```
path('profile/<username>/' , views.ProfileView.as_view() , name='profile') ,
```

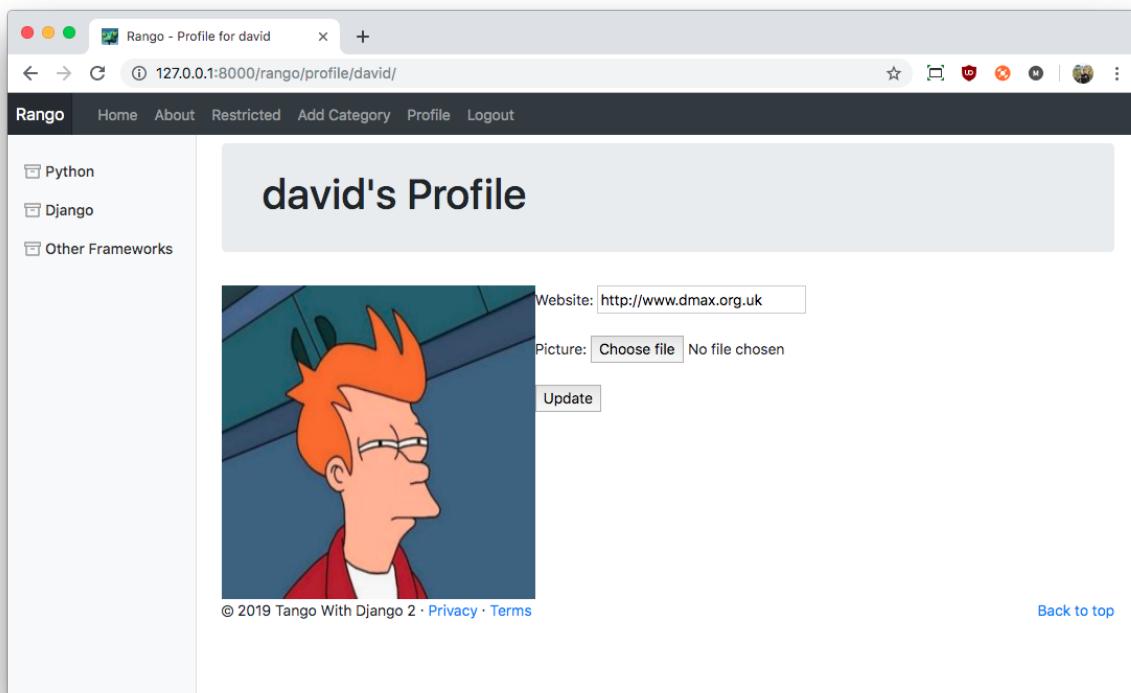
Note the inclusion of a `username` variable – this matches to anything after `/profile/`. This means that the URL `/rango/profile/maxwelld90/` would have a `username` of `maxwelld90`. This is in turn passed to the `get()` or `post()` methods in our class-based view as parameter `username`. This is how we are able to determine what user has been requested.

Tweaking the Base Template

Everything should now be working as expected – but how can users access the new functionality? We can provide a link by modifying Rango's `base.html` template. Find the series of `` elements that you created earlier in the book that comprise the navigation bar. Add a new link to allow users to jump to their `Profile` – ensuring that this link is *only visible when a user is logged in (authenticated)*. If that is the case, what template conditional statement does the following link need to be placed within?

```
<li class="nav-item">
  <a class="nav-link" href="{% url 'rango:profile' user.username %}">Profile</a>
</li>
```

All being well, you should see something like [the following example](#).



A completed Rango profile page, showing a profile for `david`. Note that this is being viewed from the same account – the ability to edit the website and profile image are both available.

15.6 Listing all Users

Our final challenge is to create a further page that allows one to view a list of all users registered on the Rango app. This one is relatively straightforward – implementing a further template, view and URL mapping are all once again required – but the code in this view is simplistic to the code we added above. We'll be creating a list of users registered to Rango – and providing a hyperlink to view their profile using the code we implemented in the previous section. *Note that this new view should only be accessible to those who are logged in.*

Creating a Template for Listing User Profiles

You know the drill by now. Create a new template! Once again, this belongs to Rango, but this time, we will call it `list_profiles.html`. Within the file, add the following HTML markup and Django template code.

```
{% extends 'rango/base.html' %}  
{% load staticfiles %}  
  
{% block title_block %}  
    User Profiles  
{% endblock %}  
  
{% block body_block %}  
<div class="jumbotron p-4">  
    <div class="container">  
        <h1 class="jumbotron-heading">User Profiles</h1>  
    </div>  
</div>  
  
<div class="container">  
    <div class="row">  
        {% if user_profile_list %}  
            <div class="panel-body">  
                <div class="list-group">  
                    {% for list_user in user_profile_list %}  
                        <div class="list-group-item">  
                            <h4 class="list-group-item-heading">  
                                <a href="{% url 'rango:profile'  
                                         list_user.user.username %}">  
                                    {{ list_user.user.username }}  
                                </a>  
                            </h4>  
                        </div>  
                    {% endfor %}  
                </div>  
            </div>  
        {% else %}  
            <p>There are no users present on Rango.</p>  
        {% endif %}  
</div>
```

```
</div>
{% endblock %}
```

As mentioned previously, this template is pretty straightforward compared to what we have been doing as of late! A series of `<div>` tags have been created using various Bootstrap classes to style the list. For each user, we display their username and provide a link to their profile page. Notice that since we pass through a list of `UserProfile` objects, we need to pass through the `user` attribute to lead to the username of the user!

Creating the View for Listing User Profiles

With our template created, we can now create the corresponding view that selects all users from the `UserProfile` model. We also assume that the current user must be logged in to view the other users of Rango, as stated above.

The following simple class-based view should satisfy these requirements nicely. Again, this is a straightforward view – selecting all of the `UserProfile` instances via an ORM query is perhaps the trickiest part here. Remember, the list must have a name of `userprofile_list` in the context dictionary to match up with our template defined above. This code would live in `views.py`.

```
class ListProfilesView(View):
    @method_decorator(login_required)
    def get(self, request):
        profiles = UserProfile.objects.all()

        return render(request,
                      'rango/list_profiles.html',
                      {'userprofile_list': profiles})
```

Mapping and Linking the View

Our final step is to map a URL to the new `ListProfilesView` class-based view. Add the following pattern to the `urlpatterns` list provided in Rango's `urls.py`.

```
path('profiles/', views.ListProfilesView.as_view(), name='list_profiles'),
```

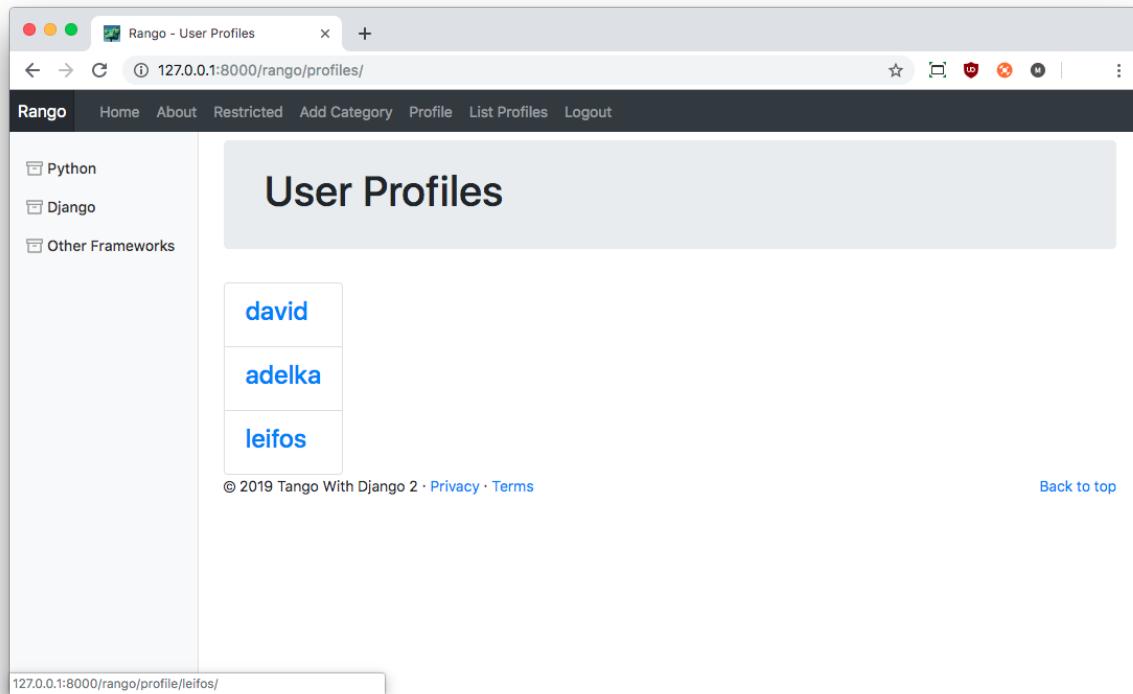
No parameters are required for this new mapping. The new page should now be accessible at <http://127.0.0.1:8000/rango/profiles/>¹¹ – but what about providing a link to the new feature, too?

Once again, this will involve editing Rango's base.html template. Locate the list of links that make up the navigation bar, and add a further link that allows users to List Profiles. *Make sure that this new link is only visible to those who are logged into Rango.*

```
<li class="nav-item">
    <a class="nav-link" href="{% url 'rango:list_profiles' %}">
        List Profiles
    </a>
</li>
```

With this link in place, logged in users should now be able to navigate to the new view and see a list of all users registered with Rango. [The screenshot below](#) shows the completed list with three registered users.

¹¹<http://127.0.0.1:8000/rango/profiles/>



The completed profile listing functionality. Three users are registered to use Rango in this example: david, adelka and leifos.



Profile Page Exercises

With the above now completed, have a look at the following additional exercise to test your knowledge and understanding further.

- Update the profile list to include a thumbnail of the user's profile picture.
- If a user does not have a profile picture, then insert a substitute picture by using the [service provide by LoremPixel¹²](#) that lets you automatically generate images.
- Update the user profile view (responsible for rendering `profile.html`) to also use a placeholder image if no profile image has been provided.

¹²<https://lorempixel.com/>



Hints

The following hints may help you complete the above exercises.

- For the first exercise, you will need to use the `` tag.
- Substitution images can be obtained from <https://lorempixel.com>¹³.
As an example, you can use ``¹⁴ to obtain a 64x64 placeholder image. Adjust the dimensions as you see fit. Beware that the image may take a few seconds to load!

¹³<https://lorempixel.com>

¹⁴<https://lorempixel.com/64/64/people/>

16. JQuery Crash Course

So far in this tutorial, we have focused primarily on *server-side* coding. In other words, a lot of our energy has been spent on figuring out the mechanics of writing Python code to make Django do things *on the server*. We've also been writing HTML markup (and used the Twitter Bootstrap toolkit) for use on the *client-side*, or your web browser.

However, did you know you can do so much more on the client-side? You can use [JavaScript¹](#) to add functionality to your web application on the client-side, from changing the way things are presented to obtaining additional information from the server *without* having to reload the entire webpage.

In this chapter, we'll be introducing you to the basics of JavaScript while using the [JQuery²](#) framework. JQuery makes writing JavaScript code easier – and much more enjoyable! A few lines of JQuery can encapsulate hundreds of lines of pure JavaScript. JQuery also provides a suite of APIs that are mainly focused on manipulating HTML elements. We'll be looking at:

- how to incorporate JQuery within Rango;
- explaining how to interpret basic JQuery code; and
- providing several small examples to show you how everything works.

16.1 Including JQuery

To include JQuery (or any other JavaScript file), you need to tell the *client* where to find JQuery. At the end of the day, the web browser will be the part of the chain that utilises the JavaScript code to provide additional functionality to the user. Thinking about Rango, how can we do this? We will need to modify our *templates*, as when

¹<https://developer.mozilla.org/en-US/docs/Web/JavaScript>

²<https://jquery.com/>

fully rendered, these are the files that are sent back to the client when something is requested.

In Rango's `base.html` template, let's make sure we have referenced JQuery. If you completed the [Bootstrap chapter](#) earlier on in the tutorial, you may have noticed that Bootstrap uses JQuery! To make sure nothing breaks, we'll stick with the version that Bootstrap uses – version 3.3.1. At the bottom of your `base.html` template (before the close of the `<body>` tag), look for the following line.

```
<script src="https://code.jquery.com/jquery-3.3.1.slim.min.js"
crossorigin="anonymous"></script>
```

It should already be there! The `<script>` tag can contain JavaScript code *inline* – or if you want to load an external JavaScript file (`.js`), then you can use the `src` attribute and specify the filename. In this instance, we load the JQuery file directly from the JQuery servers. You can alternatively load this file in your web browser, save it, and put it in your `static` folder, meaning that you are then not reliant on the JQuery servers for your app to work.

This is what we'll be doing. However, we aren't using the right JQuery file. Currently, Bootstrap is using the *slimline* version of JQuery. This means several key pieces of functionality that we want to use are missing. To fix this, open a new tab in your browser and enter the following URL. This is the non-slimline version that provides everything we need.

<https://code.jquery.com/jquery-3.3.1.min.js>

When it loads, you should be greeted with a load of JavaScript code that is seemingly mashed together. We then want you to create a new `js` directory within your project's `static` directory. It's good to separate your concerns, hence why it's good to create a new directory to store JavaScript (`js`) files. Save the `jquery-3.3.1.min.js` file that you have just accessed in the new directory, and update the `<script>` tag in `base.html` to the following.

```
<script src="{% static "js/jquery-3.3.1.min.js" %}" crossorigin="anonymous"></script>
```

This now points the browser to look for JQuery in your static/js directory, rather than directly from the JQuery servers. Once you have done this, you'll want to then create a new, blank file in the js directory called rango-jquery.js. We'll be adding in some JavaScript code that uses JQuery to add functionality to our Rango app. Of course, we'll also need to add a reference to this new file in our base.html template, too. Underneath the line you just tweaked, add the following.

```
<script src="{% static "js/rango-jquery.js" %}" crossorigin="anonymous"></script>
```

Make sure this new markup is *underneath* the line including the JQuery framework. As a sanity check, make sure your static files have been set up correctly (see [Templates and Media Files](#)), and that you have `{% load staticfiles %}` at the top of `base.html`.

16.2 Testing your Setup

With the above steps completed, you should now be ready to try things out! Open the new `rango-jquery.js` file, and add the following lines of code.

```
$(document).ready(function() {  
    alert('Hello, world!');  
});
```

This small piece of code utilises JQuery. But how does it work? We've broken it down into a series of different steps to show you what's going on.

1. We first select the document object via `$(document)`. The `$()` is the shortcut that you can use to access the JQuery framework – everything that uses JQuery will have commands that start with a `$`. The command `$(document)` tells the JQuery framework to *select* the [HTML document object](#)³ – the object that a complete HTML page becomes when a web browser loads it into memory. This object encapsulates the entirety of the page. When we select something like this, we call the mechanism a *selector*.

³https://www.w3schools.com/jsref/dom_obj_document.asp

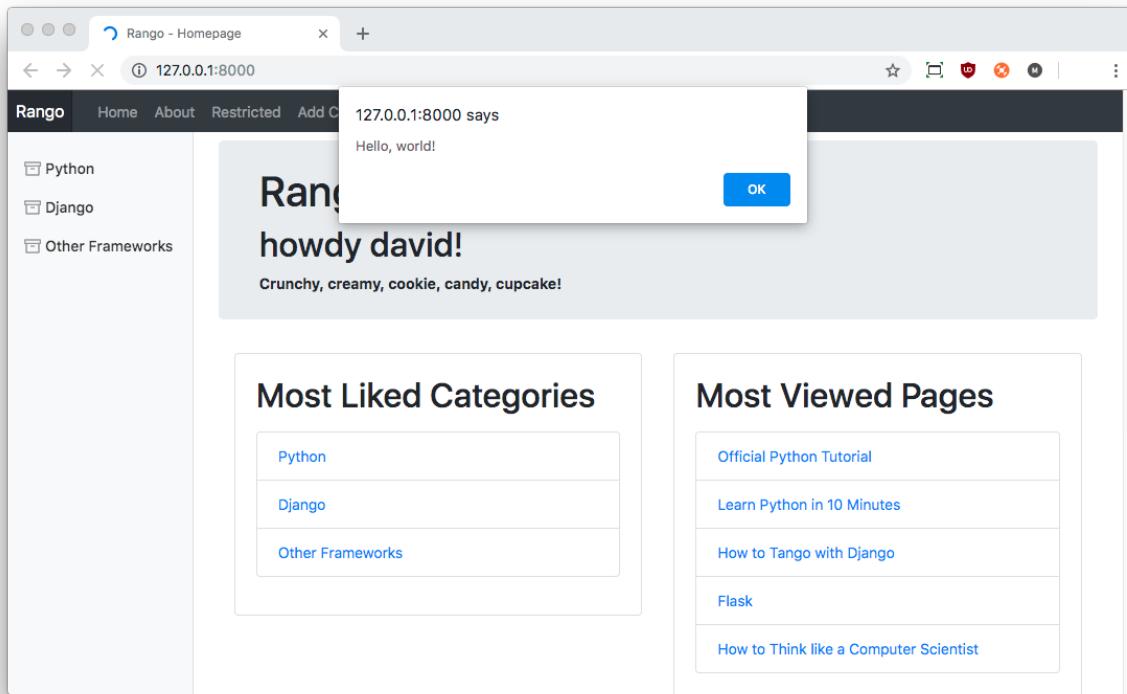
2. With the `document` selected, we then *chain* a function onto our selector. In this instance, we chain the `ready()` function. As stated in the [JQuery documentation⁴](#), this is a function that JQuery executes *when the DOM⁵/page is fully loaded in memory, and is ready to go*. It's pretty typical to want to wait until everything on the page has been loaded (including external resources), so you can get a guarantee that everything will be there when the code starts executing. If you don't wait, the code can begin executing before everything that is expected to be present is ready – and the user will get a broken website.
3. Inside `ready()`, we define an *anonymous function* to execute when the DOM is ready to go. Just like any other function in JavaScript, it can take parameters and is surrounded by braces (like `{` and `}`). This is different from Python in that Python relies solely on indentation to tell it what lines of code to execute, and when. With a C-style language such as JavaScript and Java, braces are used to this effect.
4. Inside the function, we have the line `alert("Hello, world!");`. Note that at the end of each line/command, place a semicolon `(;)`. The `alert()` function takes a string, and displays it as a popup box in the browser. In this instance, it will display the string `Hello, world!`.

To summarise, this code displays the message `Hello, world!`, just like in the [screenshot we show below](#). However, the key here is that the browser waits until everything is loaded, then displays the message. We strongly encourage you to keep the [JQuery API documentation⁶](#) open in a separate tab to help you understand the different concepts we demonstrate here better.

⁴<https://api.jquery.com/ready/>

⁵<https://www.w3.org/TR/WD-DOM/introduction.html>

⁶<https://api.jquery.com/>



The result of the code we demonstrate above – a small popup box that displays `Hello, world!` when the page is loaded. This should be visible on every page of Rango.

You should, of course, remove the `alert()` function call once you have seen it working in front of your very eyes – this is perhaps the simplest demonstration we can implement. Being greeted with this popup on every page is very annoying!



Select and Act Pattern

JQuery requires you to think in a more *functional* programming style, as opposed to typical JavaScript which is written in a more *procedural* way. For all JQuery commands, they follow a similar pattern: **Select and Act**. Select an element, and then perform some kind of action on/with the element.

Acting on Mouse Clicks

In the previous example, we executed the `alert()` function call when the page had been loaded, and everything was ready to go. This is based upon an *event* being

fired internally within the browser – but what about trying different events? In this section, we'll work on a more concrete example making use of user interactions.

The goal of this simple exercise will be to add a button to Rango's `about.html` template and show you how events are bound to different elements of a page. When the user clicks the button, a popup alert appears. We will, of course, need to work with the `about.html` template and the new `rango-jquery.js` static JavaScript file.

Within the `about.html` template, let's first add a button that achieves what we want using vanilla JavaScript. This demonstrates a very simple way of *binding code to an event*. Add the code underneath the closing `</div>`, but before the closure of the `body_block` template block. If you still have your images of Rango and the cat from earlier, place this markup *before* the images so the button is easy to see. You may want to also add a line break (`
`) to separate the button from the images.

```
<button class="btn btn-primary"
        onclick="alert('You clicked the button using inline JavaScript.');" >
    Click me - I run on vanilla JavaScript!
</button>
```

With this code in place, navigate to the `about` page in your browser. You should see a blue button underneath your image. Click it, and you will be greeted with a popup box with the message 'You clicked the button using inline JavaScript.'. What happens with this markup is we use the `onclick` attribute that is present for HTML elements – and add some simple JavaScript code within it. This binds the code you provide to the event for when a user clicks on the element within the DOM – in this case, the button. When the user clicks, the code is executed. [HTML has heaps of different attributes⁷](#) used for assigning code to different events.

However, is embedding code within a template a neat solution? We argue no. By incorporating code within your markup, you are adding code *inline*. This violates the principle of separation of concerns. Templates should contain presentational aspects only. Code should be placed in a separate location, such as the `rango-jquery.js` file.

To move towards a more elegant solution, add a further button to the `about.html` template.

⁷https://www.w3schools.com/tags/ref_eventattributes.asp

```
<button class="btn btn-primary" id="about-btn">  
    Click me - I'm using JQuery!  
</button>
```

This time, we have another button, but there is no code associated with it. Instead, we replaced the `onclick` attribute with a unique identifier – or `id` – that allows us to reference the button within our external JavaScript file. If we then add the following code to `rango-jquery.js`, we'll start to see what is going on.

```
$(document).ready(function() {  
  
    $('#about-btn').click(function() {  
        alert('You clicked the button using JQuery!');  
    });  
  
});
```

Reload the page, and try things out. Hopefully, you will see that when you click both buttons, you will be greeted with a popup message from both, albeit with different messages.

Like the JQuery code that we tried earlier on, our code sample above starts by selecting the `document` object and includes a function that is only executed when the page has been loaded (i.e. `ready()`). Within this anonymous function, we then *select* the `about-btn` element (by a unique identifier, as instructed to by prepending the ID with a `#`). A further anonymous function is bound to the event when the element with ID `about-btn` is *clicked* (`click()`) – and this function contains a further call to the `alert()` function that displays a popup box to the user.

For more complex functions, this approach is much more desirable. Why? The JavaScript/JQuery code is maintained in a separate file to the template. Here, we programmatically assign the event to a handler at *runtime*, rather than *statically* within the HTML markup. Therefore, this achieves separation of concerns between the JavaScript/JQuery code and the HTML markup.



Keep things Separated

Separation of concerns⁸ is a design principle that is good to keep in mind. In terms of web apps, the HTML is responsible for the page content. CSS is used to style the presentation of the content. Finally, JavaScript is responsible for how the user can interact with the content – as well as programmatically manipulating the page's content and style.

By keeping them separated, you will have cleaner code, and you will reduce maintenance woes in the future. You'll be a better software engineer by adhering to this simple principle!

Put another way, *never mix, never worry!*

Selectors

In the example above, we *selected* our second button by referencing the unique identifier that we assigned to it. This was achieved through the use of the # selector. We simply prepended the selector to the identifier, yielding a completed selector of #about-btn. Of course, you may find yourself wanting to manipulate a group, or class, of elements – rather than just one. In this instance, you can use the class selector instead, which is a period (.). Check out the simple example below.

```
$('.ouch').click(function() {
    alert('You clicked me! Ouch!');
});
```

This code would select all elements with class="ouch" – and when you click any of them, the popup You clicked me! Ouch! would be shown.

You can also select elements by their tag name – such as div, p or strong. The following example uses the hover event on all p (paragraph) elements.

⁸https://en.wikipedia.org/wiki/Separation_of_concerns

```
$( 'p' ).hover(  
    function() {  
        $(this).css('color', 'red');  
    },  
    function() {  
        $(this).css('color', 'black');  
    } );
```

Add this JavaScript to your `rango-jquery.js`. In the `about.html` template, you should then add a paragraph underneath your buttons – something like `<p>This is some text for the JQuery example.</p>` should suffice. Look at the code, and think what will happen... then try it out! Does it do what you expect to happen?

Here, we are selecting all the `p` HTML elements. We assign a `hover()` event to the elements. `hover()` requires two functions – one to be executed when you *hover over* something (the first function), and one to execute when you leave something (the second function). This means that when a user hovers over a `p` element, the text colour within the element will change to red. When the user's cursor leaves the element, it changes to black. Have a look at the [hover\(\) event's documentation](#)⁹ for more information.

Note that we can use the `this` variable to reference *the element that is being hovered over*, and the `css()` function (part of JQuery) to set the `color` attribute for the said element to whatever we so choose (as we specify in the second parameter).



Remember – `ready()`!

Remember, it's good practice to place your JQuery code within a `$(document).ready()` function call. Doing so will ensure that this code will only ever execute when the page has been completely loaded into memory, reducing the chances of your code breaking in very strange ways.

It takes time for content to download to a client's computer; make sure your code is patient!

⁹<https://api.jquery.com/hover/>



Play Around

Play around with the code you have just written. Have a look at the [JQuery documentation](#)¹⁰ to see what else you can do. And one little thing: what do you think will happen if you change the two instances of `$(this)` above to `$('p')`? Think about it, then once you have worked out a solution, experiment and see if you were correct!

16.3 Further DOM Manipulation Examples

In the above example, we used the `hover()` function to assign an event handler to the hover in and hover out events. For each of those events, we then used the `css()` function to change the text colour of the element being hovered over. The `css()` function is one example of *DOM manipulation* – or the changing of the page loaded dynamically. However, `css()` is not the only way you can manipulate a page. JQuery provides other ways to manipulate the page!

For example, we can remove a class from elements with the `removeClass()` function, and add classes to elements with the `addClass()` function.

```
$("#about-btn").removeClass('btn-primary').addClass('btn-success');
```

This is an example of *chaining* calls. This code selects the `about-btn` element (by its unique ID), and removes the `btn-primary` class from the element. It then adds a new class, `btn-success`. This has the effect of making the button turn from blue to green. Why? The Bootstrap toolkit implements these classes – `btn-success` just so happens to set the background colour of the element to green.

It is also possible to access the *inner HTML* of a given element – or, in other words, the markup nested within a given element. For example, let's put a new `<div>` element in Rango's `about.html` template, underneath our sample `<p>` element.

¹⁰<https://api.jquery.com/category/events/mouse-events/>

```
<div id="msg">Hello! I'm a further example for JQuery testing.</div>
```

Once you have added this new element, jump to the `rango-jquery.js` file and add the following code. Remember to add this *within* the `$(document).ready()` call!

```
$('#about-btn').click(function() {  
    msgStr = $('#msg').html();  
    msgStr = msgStr + ' ooo, fancy!';  
  
    $('#msg').html(msgStr);  
});
```

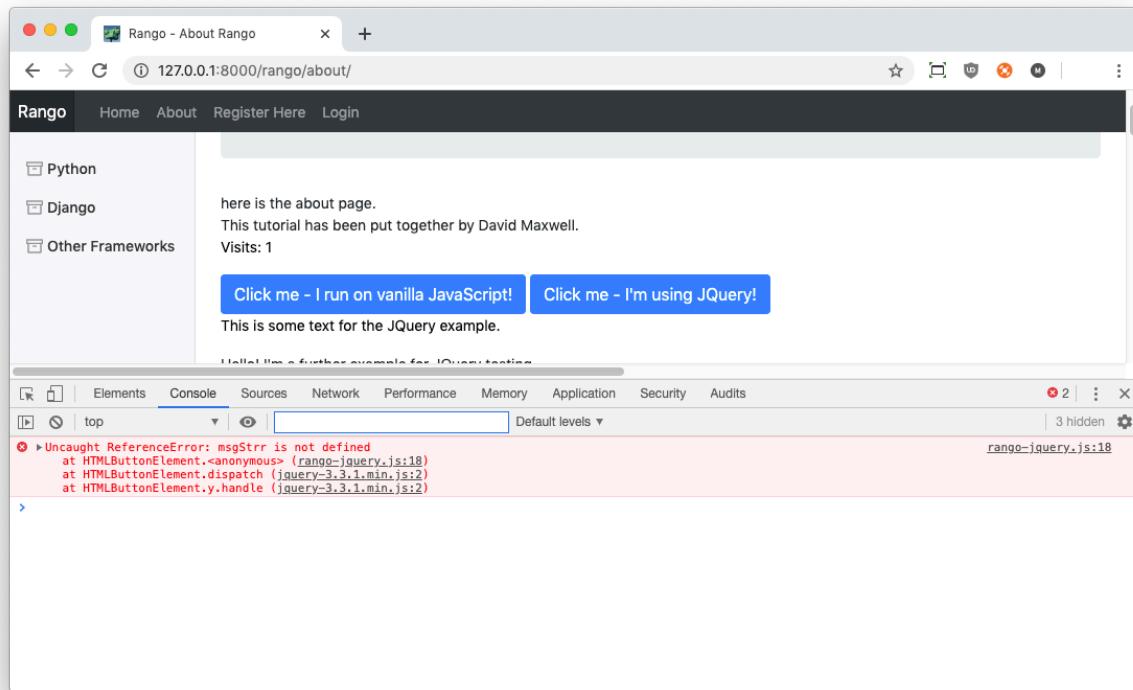
When the element that matches the selection of `#about-btn` (i.e. your second button) is clicked, we first get the HTML markup inside the element selected by `#msg` by using the `html()` function call. As this is then simply a string, we append '`ooo, fancy!`' to the end, and then set the `html()` of the `msg` element to the new value, stored within `msgStr`.

16.4 Debugging Hints

One thing that often throws beginners is how to figure out what is going wrong with your JavaScript code! You'll most likely encounter an error, but your browser is very good at hiding the error! This is by design – would a typical user want to be greeted with heaps of JavaScript errors? We highly doubt it!

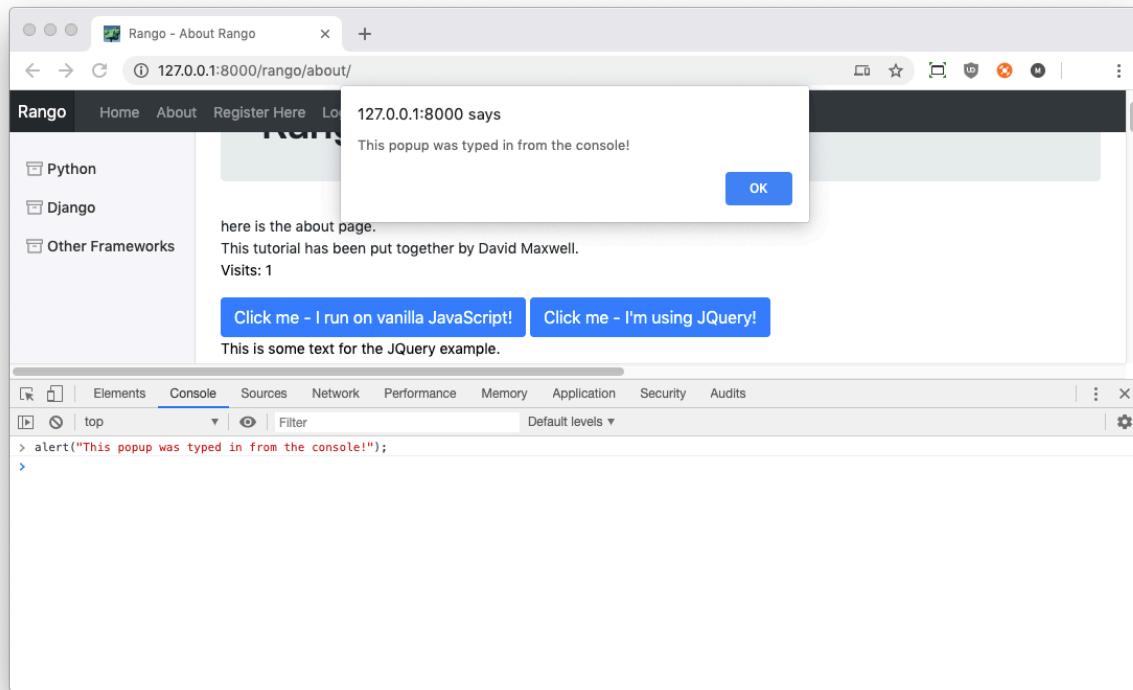
Most contemporary web browsers have a series of *developer tools* within them that allow you to see what's going on with your client-side JavaScript code. Accessing these tools depends on what browser you are using. On Chrome, you can access developer tools by clicking the options menu (three dots), selecting `More Tools`, and clicking `Developer Tools`.

When the tools load, you can click the `Console` tab to see the output from the JavaScript interpreter – including any error messages! In the [example screenshot below](#), you can see that an error is preventing the code from running. A cursory look at the error message shows that we included one too many `r`'s in our `msgStr` variable, thus providing us with a vital clue to figure out what is going on.



The developer tools in Google Chrome, showing the console output. Look at the console output to figure out what's going wrong with your code if it's not working as expected.

One further feature of the console is that you can *enter commands, too!* The [screenshot below](#) demonstrates this in action – the `alert()` function was called, and the output can be seen as a popup box. You can execute JQuery commands, examine the contents of arrays, hashmaps or whatever you need to do within this console. It's an essential bit of kit for debugging contemporary code-heavy apps.



The developer tools in Google Chrome, showing the console output. Look at the console output to figure out what's going wrong with your code if it's not working as expected.



Always use Developer Tools

Always have your browser's Developer Tools open when you're writing client-side JavaScript. It makes things so much easier to figure out when things don't work as expected!

This chapter has provided a very rudimentary guide to using JQuery, and how you can incorporate it within your web app. From here, you should be able to now understand how JQuery operates, and experiment with the different functions provided by the JQuery framework – and even those by third-party developers. In the next chapter, we'll be using JQuery to provide simple [AJAX¹¹](#) functionality within Rango.

¹¹[https://en.wikipedia.org/wiki/Ajax_\(programming\)](https://en.wikipedia.org/wiki/Ajax_(programming))



Performing a Hard Refresh

When working with static files – especially JavaScript (that changes often when you are developing) – you may find that your changes don't filter down to your browser, even though you saved the file!

Browsers are developed to try and reduce used bandwidth as much as possible. If you request a page multiple times, it is likely that after the initial request, the content you see will be stored in a *browser cache* on your computer. This means that when developing your project (and changing things rapidly), content may not be refreshed with the most up-to-date implementation. The solution to this problem is to either clear your browser's cache, or to perform a *hard refresh*¹². The command you issue depends on your browser and operating system. Windows systems should work with CTRL+F5, with Macs using CMD+SHIFT+R.

¹²<https://refreshyourcache.com/en/cache/>

17. AJAX in Django with JQuery

Asynchronous JavaScript and XML (AJAX) can be described as a series of different *client-side* technologies that work seamlessly together to allow for the development of asynchronous web applications. Recall that a standard request over HTTP is initiated by the client, with the server then sending back some form of response. Typically, a synchronous web application will return an entire page as the response to a request, which the browser then loads. Asynchronous web applications permit the request for data – or pieces of a webpage – that can then be used by the browser to update the currently loaded page *in situ*, without the need to refresh it in its entirety. AJAX is the basic technology that drives many of the web applications that are ubiquitous today, such as messenger apps, for example.



Need More Information?

If you're haven't used AJAX before, or require more information to grasp a better understanding of what AJAX achieves, check out the [AJAX resources page at the Mozilla website¹](#).

As the title may suggest, we'll be working towards incorporating some AJAX requests into Rango during this chapter. To aid in our development, we'll also be using the JQuery framework. If you haven't worked through the [JQuery chapter](#), we recommend you have a look at that now so everything is set up and ready to go.

17.1 AJAX and Rango

What exactly will we be implementing in this chapter? Well, there's still one more major requirement outlined in our [introductory chapter](#) that we still haven't satisfied! Users should be able to *like* a particular category. You may remember that we worked on this earlier in the book – but we only covered *displaying* the number of

¹<https://developer.mozilla.org/en-US/docs/AJAX>

likes that a category had received. We didn't implement functionality that allows a user to increase the count! This is what we'll be doing in this chapter. Specifically, we're going to:

- add a *like button* to allow registered users to “like” a particular category;
- add inline category suggestions, so that when a user types, they can quickly find a specific category; and
- add an *add button* to allow registered users to quickly and easily add a new page to a category when they perform a search.

All of these features will be implemented using AJAX technologies to make the user experience seamless. However, before we get stuck in, let's create a new JavaScript file called `rango-ajax.js`. This file will house all of our AJAX code for the features we implement in this chapter. Like existing JavaScript files, create this file in your project's `static/js/` directory. You'll also want to add a reference to this file in Rango's `base.html` template, underneath the existing references that we worked on in the previous chapter. As a reminder, these are located towards the bottom of the template.



JQuery Assumption

We assume that you have completed the [previous chapter on JQuery](#) – and are using version 3.3.1 of JQuery for this exercise. If you haven't gone through the setup steps of this chapter *at the very least*, we highly recommend that you go through those steps now.

Now with everything set up and ready to go, let's get to work.

17.2 Adding a “Like” Button

The functionality for ordering categories by likes was previously implemented in a previous chapter. Checking out Rango's homepage, categories are ordered by the number of likes they receive (in descending order). However, it would be nice to allow registered users of Rango to express their fondness of a particular category by providing them with the ability to *like* it, too!

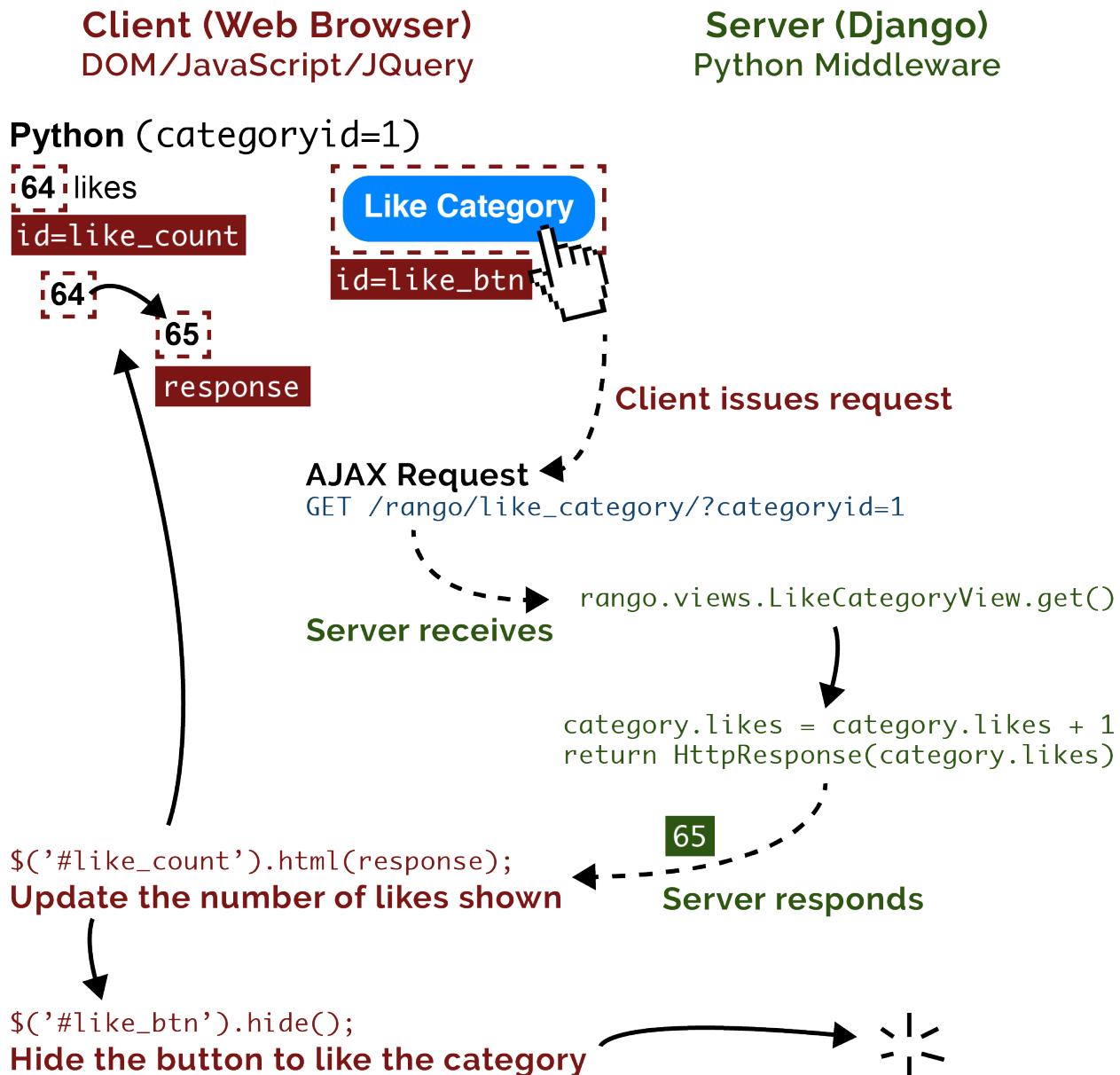
In the following workflow, we will allow registered users to *like* categories, but we won't be keeping track of what they have *liked*. A registered user could click the like button multiple times if they refresh the page. This is the simplest implementation – if we wanted to keep track of what categories they liked, too, we would have to add an additional model and other supporting infrastructure. We'll leave this advanced step for you to complete as an exercise.

Workflow

To permit registered users of Rango to *like* categories, we'll progress through the following workflow.

1. In Rango's `category.html` template, the following steps need to be undertaken.
 - Add in a *like* button, complete with an `id="like"`. There'll only be one of these buttons on the page, so using an `id` is fine.
 - We'll add in a template tag to display the number of likes the category has received. This count will be placed inside a tag with an `id` of `like_count`. This sets the template up to display likes for a given category.
2. We'll create a new view called `LikeCategoryView`. This view will examine the request, and pick out the `category_id` passed to it. The category being referred to will then have its `likes` field incremented by one.
 - Of course, we'll need to create a new URL mapping for this view, too. This should be named `/rango/like_category/` – with the `category_id` passed as a querystring. As an example, a complete URL would look something like `/rango/like_category/?category_id=1`, for liking the category with ID 1.
 - We'll make use of a `GET` request to implement this functionality.
 - A key point regarding this view will be that it *does not return a complete HTML page* – but simply the number of likes that the category now has. This means our response *does not* need to inherit from Rango's `base.html` template!
3. For the AJAX code, we'll add some JavaScript/JQuery code to `rango-ajax.js` to complete the link between the client (displaying likes and providing the option to increment the count) and the server (incrementing the count in the database).

To graphically demonstrate what we are looking to achieve here, check out the diagram below.



The sequence of events we are looking to implement using AJAX and JQuery. On the left are client-side events, triggered by the user clicking the Like Category button for the Python category. Following the arrows, you can then work out the events following this.

Study the diagram carefully to help your understanding of what will be implemented. On the left are events that take place client-side (within the browser), with events that take place server-side on the right (within the Django middleware). Everything starts from the user clicking the Like Category button. The JQuery code

we will implement then fires off an AJAX GET request to the server. The Django middleware receives the request, executes the relevant view, and sends a simple response. For this exercise, the response will simply be the new number of likes for the given category – nothing more. This response is then received by our JQuery code, and the value returned is then placed inside the element containing the count (using the `.html()` method). Finally, we hide the Like Category button from view!

Updating the Category Template

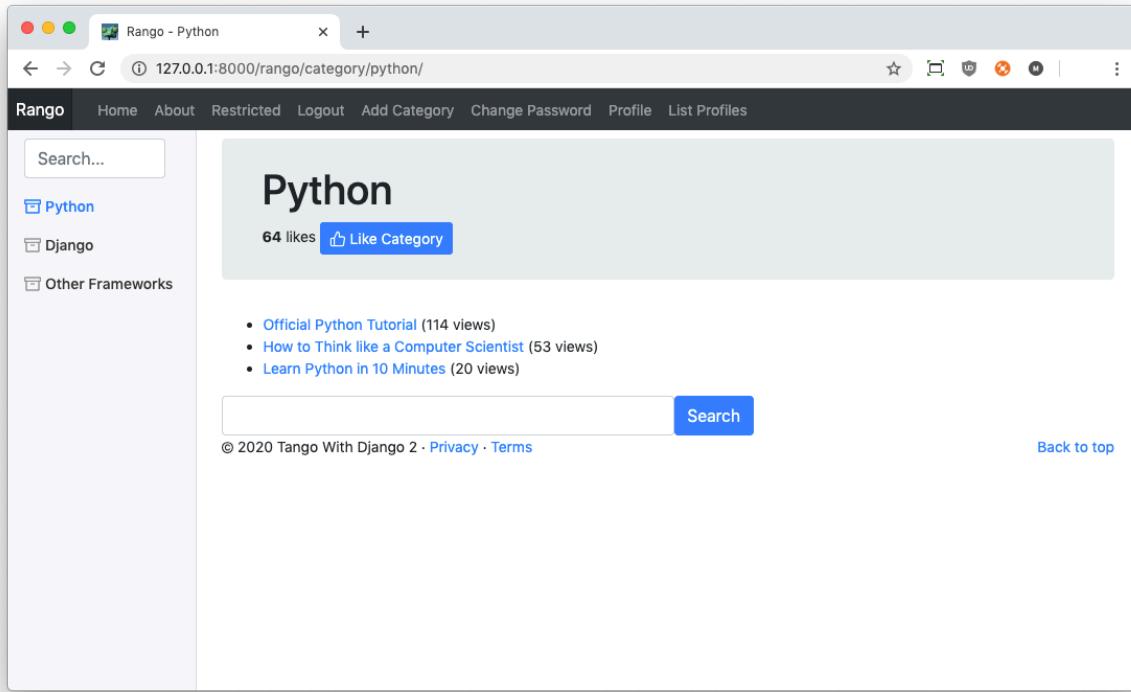
Our first step is to prepare Rango's `category.html` template for the new AJAX functionality. We'll need to add in the Like Category button, complete with a unique ID of `like_btn`, as well as adding a new element that will contain the number of likes a category has received.

To do this, open up the `category.html` template and locate the `<h1>` tag that displays the `{{ category.name }}`. After the `<h1>` tag has been closed, take a line break, and add in the following markup and template code.

```
<div>
  <strong id="like_count">{{ category.likes }}</strong> likes
  {% if user.is_authenticated %}
    <button id="like_btn"
      data-categoryid="{{ category.id }}"
      class="btn btn-primary btn-sm"
      type="button">
      <span data-feather="thumbs-up"></span>
      Like Category
    </button>
  {% endif %}
</div>
```

Once this has been added, a user who is logged into Rango should see a page similar to [the one shown below](#). This markup adds a `<div>` containing all of the infrastructure required – from a `` tag exclusively containing the number of likes the given category has received, to a new `<button>` element that will allow people to increment the number of likes. Note that within the button, we also add a small thumbs up icon (represented by the `` element). This icon is provided

by [Feather²](#), and is included as part of the Bootstrap framework. Note also the inclusion of the `id` attributes for both the `` and `<button>` elements. The `id` values we assign are important as our JQuery code will make use of them.



The result of the update to the `category.html` template. Note the `Like Category` button, with the number of likes next to it – all directly underneath the category title. If you view the same page when you are not logged in, the button will not be visible.

Creating the Like Category View

With the basics now laid out in Rango's `category.html`, let's turn our attention server-side and focus on implementing the view that will handle incoming requests for liking categories. Recall that in our overview, we stated that we'd make use of a GET request for liking categories. These requests would be serviced with URLs of the form `/rango/like_category/?category_id=1`. From these two requirements, our class-based view need only implement the `get()` method, with this method pulling the `category_id` querystring from the GET object.

Our implementation, shown below, would live inside Rango's `views.py` module.

²<https://feathericons.com/>

```
class LikeCategoryView(View):
    @method_decorator(login_required)
    def get(self, request):
        category_id = request.GET['category_id']

        try:
            category = Category.objects.get(id=int(category_id))
        except Category.DoesNotExist:
            return HttpResponse(-1)
        except ValueError:
            return HttpResponse(-1)

        category.likes = category.likes + 1
        category.save()

        return HttpResponse(category.likes)
```

Upon examination of the code above, you can see that we are only allowing users who are logged in to access this view – hence the `@method_decorator(login_required)` decorator. We also implement some rudimentary error handling. If the user provides a category ID that does not exist, or the category ID supplied cannot be converted to an integer, -1 is returned in the response. Otherwise, the `likes` attribute for the given category is incremented by one. The updated value is then returned by itself.

You should have all of the necessary `import` statements required for this to work – but double-check that the following is present. This one was used right back at the beginning of the book – you may have taken it out when cleaning up your code in previous chapters. You definitely need it now!

```
from django.http import HttpResponse
```

Of course, this view would be useless without a URL mapping to it. To comply with our requirements, let's add one. Add the following to the `urlpatterns` list in Rango's `urls.py` module.

```
path('like_category/', views.LikeCategoryView.as_view(), name='like_category'),
```

You should now be ready to proceed to the next step – making the AJAX request.

Making the AJAX request

To implement AJAX functionality, open up the blank `rango-ajax.js` file, located in your project's `static` directory. Add the following JavaScript.

```
$document).ready(function() {
    $('#like_btn').click(function() {
        var catecategoryIDVar;
        catecategoryIDVar = $(this).attr('data-categoryid');

        $.get('/rango/like_category/',
            {'category_id': catecategoryIDVar},
            function(data) {
                $('#like_count').html(data);
                $('#like_btn').hide();
            }
        );
    });
});
```

This JavaScript/JQuery code will be fired once the page has been loaded, and then binds code to an event handler on the Like Category button (identified by the unique identifier `like_btn`). When the user clicks this button, the `category_id` is extracted from the button `data-categoryid` attribute. If you look closely at the template code we defined earlier in this chapter, the `data-categoryid` attribute is populated by the Django templating engine with the unique ID of the category when the page is rendered server-side! Once the `category_id` has been obtained, we then call `$.get()`.

`$.get()` is a JQuery function that handles AJAX GET requests. We first specify the URL that we want to reach (hard-coded in this instance, which is undesirable!), with a dictionary-like object then passed as the second argument. From this dictionary-like object, a querystring is constructed from the key/value pairs supplied. This would mean the final request that JQuery constructs would look similar to `GET /rango/like_category/?category_id=<category_id_var>`. The actual ID value is of course dropped in at the end of the URL where appropriate.

The final argument we supply is a further anonymous function, this time taking a `data` parameter. This is called when the server responds from the request, with `data` containing the server's response. In our case, this will contain the number of

likes that the given category now has associated with it. Within the function, we simply replace the existing HTML of the `` tag (identified by `like_count`) with the updated value from the server – and hide the button (`like_btn`).



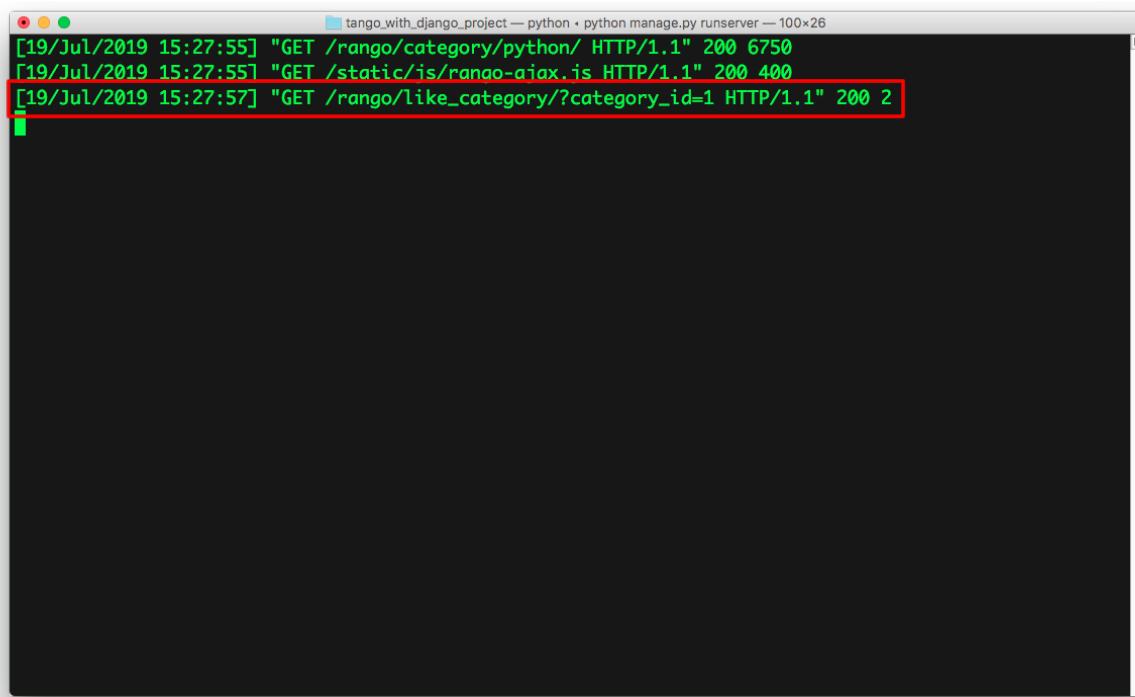
Remember, `$.get()` is Asynchronous!

This can take a while to get your head around, but this is worth repeating – the call to the server is *asynchronous*. JQuery fires off the request to the server, but it doesn't hang around waiting for the server to respond. It could take several seconds, or minutes if the request takes a while! In the meantime, the browser focuses on other inputs from the user.

Once the request comes back from the server, the browser and JQuery jump back into action, calling the anonymous function we provided in the code above. In short, this anonymous function is called when the response comes through, *not immediately when the user clicks the Like Category button*.

If the browser waited for the response to come back, it would appear to crash! It wouldn't be able to accept any further input from the user until the response comes back. This is highly undesirable. What if the server doesn't respond, or takes substantially longer to respond than you design for? This is why designing asynchronous systems is such a challenge and is one reason why using JQuery is such a bonus – the library can take care of almost all of the issues for you!

With this code implemented, try everything out! Make sure you are logged into Rango, and like the category. You should see the `Like Category` button disappear, and the count of the number of likes increase. To double-check, check the development server log. You'll see a request to `/rango/like_category/` – as shown in [the screenshot below](#). This is proof that the request was passed to the server.



```
[19/Jul/2019 15:27:55] "GET /rango/category/python/" HTTP/1.1" 200 6750
[19/Jul/2019 15:27:55] "GET /static/js/rango-ajax.js HTTP/1.1" 200 400
[19/Jul/2019 15:27:57] "GET /rango/like_category/?category_id=1 HTTP/1.1" 200 2
```

Output of the Django development server, showing the AJAX request going through (highlighted in red). The request shows that the category_id being passed was set to 1 – and from the log above (which shows the initial category page load), we can see that the request was for the Python category.

17.3 Adding Inline Category Suggestions

With a simple AJAX example implemented, let's try something more complex. Let's imagine a scenario where Rango has been populated with dozens of categories. If a user wishes to find one specific category, they would currently need to examine a long list of categories. That's annoying!

A faster way to allow users to find the category they are looking for would be to provide a suggestion component. Users can start typing in the name of the category they are searching for, and Rango would respond with a list of suggested categories that the user can then select from. In essence, this feature would provide a form of *filtering* to allow the user to narrow down the options available to him or her.

We can, of course, achieve this with AJAX. Whenever the user types in a letter from their keyboard, we can send a request off to the Django server, and ask it to return

a list of potential filtered down category matches. This list of categories can then be rendered within the loaded page. To ensure this functionality is present across the Rango app, we'll be looking to implement it on the left-hand category listing section.

Workflow

To implement this, we'll need to undertake the following steps.

1. First, we'll need to create a parameterised function called `get_category_list()`. This function will take two parameters, namely `max_results` (defaulting to 0), and `starts_with` (defaulting to an empty string, `' '`). The function will return a list of categories that match what the user is looking for.
 - `max_results` will limit how many results to return. If 0 is specified, no limit is placed on how many categories are returned.
 - `starts_with` will represent what the user has supplied so far. Imagine if the user is looking for `python`, a potential string being passed here could be `pyt` – meaning that the user has supplied the first three characters of what they are looking for.
2. A further view will need to be created. We'll call this `CategorySuggestionView`. This will again be class-based and will be accessed through the URL mapping `/rango/suggest/` (with a name of `suggest`). This view will take the user's suggestion, and will return a list of the top eight suggestions for what the user has provided.
 - We will assume that this is achieved through an HTTP GET request.
 - We will check if the querystring provided to the view is empty. If it contains something, we'll call `get_category_list()` to get the top eight results for the supplied string.
 - The results from `get_category_list()` will be munged together in an existing template, `categories.html`. This is what we used to display the category listing on the left-hand side of Rango's pages, through the custom template tag we implemented earlier.



Notice the Pattern

Like in the first example where we implemented the ability to like a category, we will once again be implementing a further view here. **AJAX requests need their views as a means to communicate with the server.** It's just the same as a standard, synchronous request!

With the URL mapping, view and template then in place, we will need to alter Rango's base.html template to provide a search box, which will allow users to enter their request. After that, we'll then implement some further JavaScript/JQuery to glue the client-side and server-side functionality together.

Preparing the Base Template

Let's tweak Rango's base.html template first. We'll modify the block of code responsible for rendering the sidebar. Find the definition of the sidebar_block block, and update the surrounding `<div>` element to include an id, like in the example below.

```
<div class="sidebar-sticky" id="categories-listing">
  {% block sidebar_block %}
    {% get_category_list category %}
  {% endblock %}
</div>
```

By adding an ID of categories-listing, we are providing a way for our future JavaScript/JQuery code to reference this `<div>` and update its contents, thus updating the list of categories presented to the user. But how do we allow the user to enter characters in the first place? By providing an `<input>` field! Add this in *above* the `<div>` you have just modified. We include a complete example of both components below.

```
<div class="w-75 ml-3">
    <input type="search"
        id="search-input"
        class="form-control ds-input"
        placeholder="Search..." />
</div>

<div class="sidebar-sticky" id="categories-listing">
    {% block sidebar_block %}
        {% get_category_list category %}
    {% endblock %}
</div>
```

Our new input element is of type search (which permits string entry), and is assigned an id of search-input. We also use a placeholder message of Search... to greet the user and prompt them that they can use this box to search for a category in the provided list. Note the inclusion of several Bootstrap classes to aid styling.

We'll come back to glueing these components to our server-side code later on.



Populate Exercise

Update Rango's population script. Add in the following categories: Pascal, Perl, PHP, Prolog, PostScript and Programming. Run the population script to add these new categories to your database. You don't need to worry about adding pages for each new category, although you can if you wish to do so. If you don't want to do this, just pass an empty list ([]) for the value of pages. By adding these additional categories, trying out the new inline category suggestion functionality will be a more impressive experience later on.

The `get_category_list()` Helper Function

Recall that this helper function is to return a list of categories whose names closely match the string provided by the user. We can use a filter operation in the Django ORM to find all of the categories whose names begin with the provided string. This filter is called `istartwith`, and, as the name suggests, filters to categories that *start with* the given string. This is case insensitive, so it doesn't matter if the database

has Python with a capital P, and the user starts typing pyt – this will still match with Python.

This function can be added to Rango's `views.py` module. If you want to, you could always create a new module – something like `helpers.py`, and place it in there. We'll just leave it in `views.py` for now. If you do however choose to create a new module, you'll need to `import` everything that is required!

```
def get_category_list(max_results=0, starts_with=''):
    category_list = []

    if starts_with:
        category_list = Category.objects.filter(name__istartswith=starts_with)

    if max_results > 0:
        if len(category_list) > max_results:
            category_list = category_list[:max_results]

    return category_list
```

Note that this function meets the requirements we defined above in that it takes two parameters – `max_results`, allowing one to specify how many items to return, and `starts_with`, the user's input string. Note how we used the `filter` method to perform the `__istartswith` filter, looking for matches to `starts_with`. If `max_results==0`, we assume that all results are to be returned; otherwise, we chop the list using a standard Python list sub indexing operation.

The CategorySuggestionView

With our helper function defined, we can then make a new class-based view to make use of it. Remember, we're only dealing with a GET request here – so, like in the example before, our view will only implement a `get()` method. The method will also be available to all, so there is no need to restrict it to users who are logged in.

```
class CategorySuggestionView(View):
    def get(self, request):
        if 'suggestion' in request.GET:
            suggestion = request.GET['suggestion']
        else:
            suggestion = ''

        category_list = get_category_list(max_results=8,
                                           starts_with=suggestion)

        if len(category_list) == 0:
            category_list = Category.objects.order_by('-likes')

    return render(request,
                  'rango/categories.html',
                  {'categories': category_list})
```

Note that we reuse the existing template tags template, categories.html – so we don't need to define a new template here. We do however need to make sure our template variable categories is specified correctly – hence the name for the key in the context dictionary.

The implemented get() view retrieves the user's input from the suggestion variable as part of the GET request (with a sanity check to see if it is supplied!), and then calls get_category_list(), passing this string to the starts_with parameter. We set max_results to 8 as per our specification. If no results are returned, we default to displaying all categories, sorted by the number of likes received in descending order.

Of course, no view is of any use without a URL mapping for accessibility! We'll add a further mapping to Rango's urls.py urlpatterns list.

```
path('suggest/', views.CategorySuggestionView.as_view(), name='suggest'),
```



Test the Filtering Functionality

With this all complete, you should then be able to test the filtering functionality. We'll leave this to you – if you know the URL is <http://127.0.0.1:8000/rango/suggest/>, and you need to provide a querystring of the form ?suggestion=pro, what would you do to do a simple test?



Hint

This is just a straightforward HTTP GET request, like any request *your browser issues when you type in a URL or click a link*. What happens when you point your browser to `http://127.0.0.1:8000/rango/suggest/?suggestion=pro?`

Adding AJAX to Request Suggestions

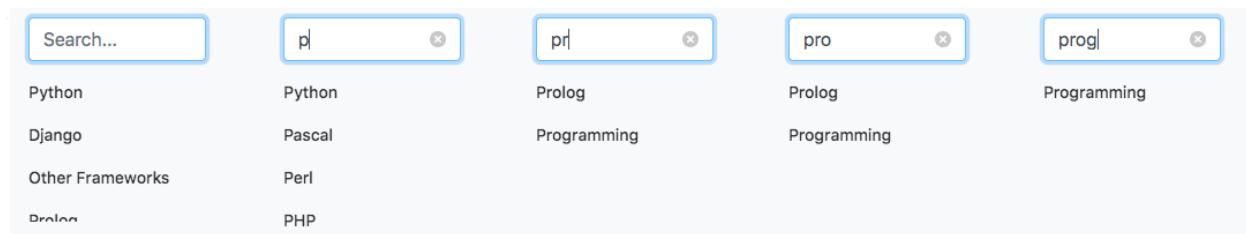
If you tested your server-side functionality and are happy with it, now it's time to glue the left-hand side component of Rango's pages to the server. We'll do this by adding some more code to the static `rango-ajax.js` file. Note that you should add this *within* the `$(document).ready(function() { ... })` block, to ensure that all parts of the page are loaded before the code can be executed.

```
$('#search-input').keyup(function() {
    var query;
    query = $(this).val();

    $.get('/rango/suggest/',
        {'suggestion': query},
        function(data) {
            $('#categories-listing').html(data);
        }
});
```

This code binds a keypress event (when a user presses a key on their keyboard-smartphone screen) when the `search-input` element is focused. In other words, when the user types a letter into the search box, the code is fired. The value of the input box (`$(this).val()`) is then extracted, and passed to an AJAX GET request. We wrap the string provided up by the user in a dictionary-like object, again to help construct the correct URL and querystring.

Once the server responds, the anonymous function is called, with `data` containing the response – or list of categories that match the user's query. This list is then simply placed inside the `categories-listing` `<div>` element we updated earlier – replacing any existing content within the `<div>`. Have a look at the screenshots below to see the new functionality in action.



An example of the inline category suggestions. Notice how the suggestions populate and change as the user types each character.

17.4 Further AJAX-ing

Having gone through two examples of how to incorporate AJAX within the Rango app, why not try one more? This time, *treat this as an exercise*. We'll lay out the specification for you and provide code snippets that will help you towards a solution. However, we implore you to solve it for yourself. If you need help, try going online and searching for a solution. Figuring out what query to type into a search engine can sometimes be half the battle in getting the help you require.

Specification

We added several new categories to test out our category suggestion feature earlier on. If you didn't add pages to those categories, they'll just be blank. With the Bing Search functionality integrated within a category page, one missing link that we have not yet covered is providing a user with the ability to *add a page from search results*.

That is when a registered user issues a query and retrieves results, can we provide an Add button next to a search result to let the user add this page to the category they are looking at? Furthermore, can this be done through the use of AJAX?

Here's a rough workflow to assist you in finding a solution to this problem.

1. First, update Rango's category.html template.
 - Add a small button next to each search result. Within the <button> element, you'll want to add data attributes that provide the page title and page URL. Doing this will make it easier for JQuery to pick out this information.

- Where pages are listed, you'll want to wrap that block of markup and template code with a `<div>` with a unique ID. Once again, this is so that JQuery can easily select this element and update the contents within it.
 - Remove the link to the existing Add Page view – this is now essentially redundant.
2. With the changes in place to the template, you then need to create a new view to handle the addition of a new page.
 - The view will accept a GET request, containing the new title and url for a page, as well as the category_id for the category the page is being added to. This view should then add the new page to the given category.
 - Map the new view to a URL – `/rango/search_add_page/`.
 - The new view should return an updated list of pages for the category. This may involve the addition of a new template to generate this list.
 3. Glue the new view together with the updated `category.html` infrastructure. When an Add button is clicked, JQuery code should then be executed, issuing an AJAX GET request to your new view. The data returned from the new view should then be placed within your `<div>` container for the list of pages.



Use a class for the Add Button!

Until now, we've selected buttons in JQuery using a unique id. This is because until now, only one such button has existed on a page.

However, having an Add button for *each page* returned from the Bing Search API presents a problem. How do you get JQuery to be able to bind code to all of them? Use the class selected `(.)` instead of the ID selector `(#)`! You can create a new class, apply it to each button, and then select items using that class with JQuery.

Code Snippets

We've included several code snippets to help you along with this exercise. Our sample markup and template code that provides a `<button>` to Add a new page from search results are shown below.

```
<button class="btn btn-info btn-sm rango-page-add"
        type="button"
        data-categoryid="{{ category.id }}"
        data-title="{{ result.title }}"
        data-url="{{ result.link }}">
    Add
</button>
```

Note the three attributes beginning with `data-` to provide the `categoryid` of the category to add a page to, and the `title` and `url` values of the page we are adding. You should also be aware of the extra class we assign to the button – `rango-page-add` – that will help us with our JQuery code to select and bind code to the button when it is clicked.

Remember to also update your page listing component by wrapping a `<div>` element around it. For the examples below, we assume you give this `<div>` an `id` of `page-listing`.

Given the button that we add for each page above, we also need some JQuery code to request the addition of the new page, along with dealing with the incoming, updated list of pages returned from the initial request. Here is a model solution.

```
$('.rango-page-add').click(function() {
    var categoryid = $(this).attr('data-categoryid');
    var title = $(this).attr('data-title');
    var url = $(this).attr('data-url');
    var clickedButton = $(this);

    $.get('/rango/search_add_page/',
        {'category_id': categoryid, 'title': title, 'url': url},
        function(data) {
            $('#page-listing').html(data);
            clickedButton.hide();
        }
    );
});
```

This code should by now be relatively self-explanatory, and you should also know by now to add this code within a `$(document).ready` block! We extract the necessary information we require from the clicked button's attributes (e.g. `title`, `url`,

categoryid), then initiate an AJAX GET request. We pass the required information over to the URL /rango/search_add_page/.

Given the appropriate URL mapping (that we will leave for you to implement), we then pass control over to the new view. In this sample solution, we call the view SearchAddPageView. This implements the one required get() method.

```
class SearchAddPageView(View):
    @method_decorator(login_required)
    def get(self, request):
        category_id = request.GET['category_id']
        title = request.GET['title']
        url = request.GET['url']

        try:
            category = Category.objects.get(id=int(category_id))
        except Category.DoesNotExist:
            return HttpResponse('Error - category not found.')
        except ValueError:
            return HttpResponse('Error - bad category ID.')

        p = Page.objects.get_or_create(category=category,
                                         title=title,
                                         url=url)

        pages = Page.objects.filter(category=category).order_by('-views')
        return render(request, 'rango/page_listing.html', {'pages': pages})
```

This sample class-based view complies with all requirements – it only allows those who are logged in to add a page, and it expects input from the client with key names as per the JQuery AJAX call we implemented previously. While error handling is very rudimentary (look at the except blocks), it does at least catch basic error cases. Passing all of these, the code then creates (or gets, if it already exists), a page with the details passed, obtains a list of all the pages associated with the given category and calls render() with a new page_listing.html template.

Within this new rango/page_listing.html template, we simply copy and paste the markup and template code within Rango's category.html template that dealt with listing pages.

```
{% if pages %}  
<ul>  
    {% for page in pages %}  
        <li>  
            <a href="{% url 'rango:goto' %}?page_id={{ page.id }}">{{ page.title }}</a>  
            {% if page.views > 1 %}  
                ({{ page.views }} views)  
            {% elif page.views == 1 %}  
                ({{ page.views }} view)  
            {% endif %}  
        </li>  
    {% endfor %}  
</ul>  
{% else %}  
<strong>No pages currently in category.</strong>  
{% endif %}
```

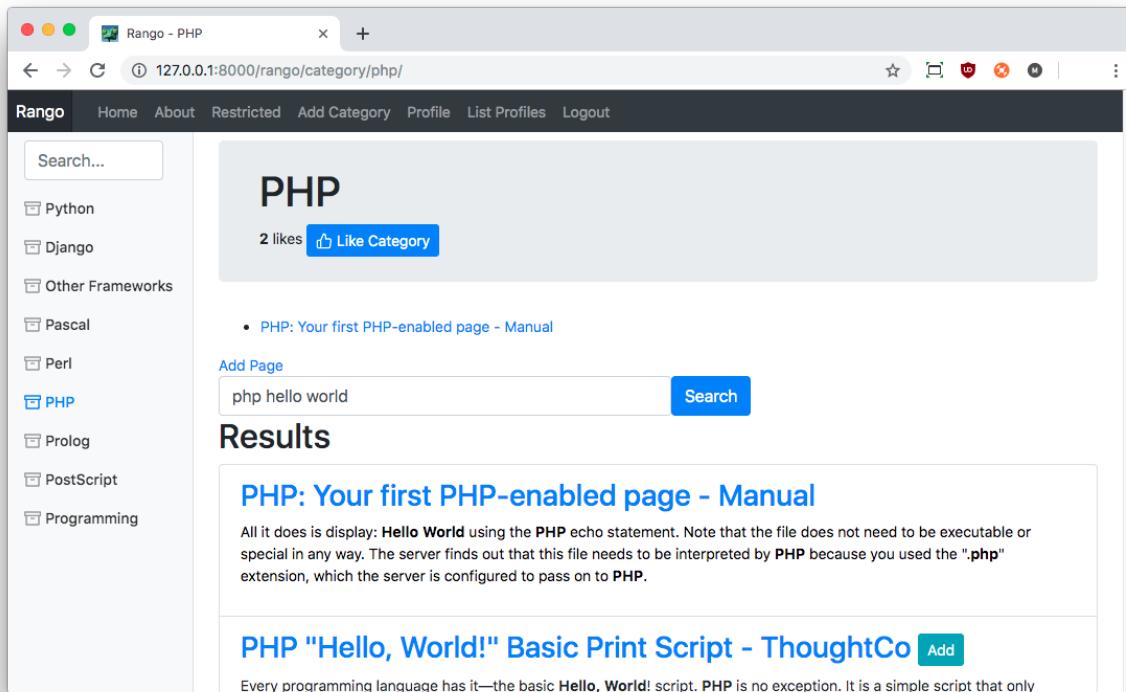
This is *identical* from that existing template. Do not cut it – you need the original markup and template code to do the initial rendering when the category view page is requested in the first place!

Once you have added in the URL mapping, you should then be able to see everything working. Issue a query in a category of your choice, and click the Add button next to one of the search results. The button should disappear, and the page listing at the top of the category view should then be updated to include the page you have added. Check out the two screenshots below – [the first screenshot](#) shows the PHP category, without any pages. We issued the search term `php hello world`, and want to add the first result to our PHP category. Clicking the Add button next to the result fires off the JQuery code, issues an AJAX GET request to the server, with the server adding the page and returning a list of pages for the category – just one in this instance! The page is then updated with this new list, with the Add button being hidden – as seen in [the second screenshot](#).

The screenshot shows a web browser window titled 'Rango - PHP'. The URL in the address bar is '127.0.0.1:8000/rango/category/php/'. The page content is as follows:

- Left sidebar:** A navigation menu with categories: Python, Django, Other Frameworks, Pascal, Perl, PHP, Prolog, PostScript, and Programming. 'PHP' is the selected category.
- Category Title:** 'PHP' with '2 likes' and a 'Like Category' button.
- Search Bar:** Contains the query 'php hello world' with a 'Search' button.
- No pages currently in category:** A message indicating there are no pages in this category.
- Add Page:** A link to add a new page.
- Results Section:** Displays two search results:
 - PHP: Your first PHP-enabled page - Manual** [Add] - A brief description about PHP echo statements.
 - PHP "Hello, World!" Basic Print Script - ThoughtCo** [Add] - A brief description about a basic PHP script.

The category page, showing the PHP category without any pages added. We issued the query `php hello world` to the search functionality, and want to add the first result to the category.



The updated PHP category page, once the AJAX functionality has added the first result. Note that there is now a list of pages (albeit with one entry), and the Add button has disappeared.

With this functionality implemented, you're almost there! Look forward to being able to deploy your project online so that people from all around the world can look at your work!



Completely Stuck?

We've implemented everything in the book on our model solution Git repository. There's a commit for this exercise which [you can view here](#)³. However, we again implore you **not** to use this resource unless you're absolutely, positively unsure about what to do.

³https://github.com/maxwelld90/tango_with_django_2_code/tree/073f876e461333b6f03100499758bc59260e574d

18. Automated Testing

If you're new to software development, it would be a good thing to get into the habit of writing and developing tests for the code you write. A lot of software engineering is about writing and developing tests and test suites to ensure the developed software is robust. Of course, most of the time, we are too busy trying to build things to bother about making sure that they work – or too arrogant to believe that what we create would fail!

However, trust us. **Through our experiences writing code in both industrial and academic settings, writing tests have saved us. On multiple occasions.** They are a vital part of a software engineer's toolbox, and provide you with confidence that the software you write does what you think it will!

According to the [Django Tutorial¹](#), there are numerous reasons why you should include tests. Several key reasons listed are repeated below.

- **Writing tests will save you time.** Even a slight change in a complex system can cause failures in places that you simply wouldn't think of.
- **Tests don't just identify problems, they prevent them.** Tests show where the code is not meeting expectations.
- **Test make your code more attractive.** “*Code without tests is broken by design*” – Jacob Kaplan-Moss, one of Django’s original developers.
- **Tests help teams work together.** Writing tests will make sure your team doesn’t inadvertently break your code, or you don’t break the code of your team members!

In addition to these reasons, the [Python Guide²](#) lists several general rules that you should try to follow when writing tests. Below are some of the main rules.

1. Tests should focus on *one small bit of functionality*.

¹<https://docs.djangoproject.com/en/2.1/intro/tutorial05/>

²<http://docs.python-guide.org/en/latest/writing/tests/>

2. Tests should have a *clear purpose*.
3. Tests should be *independent from your existing codebase*.
4. Run your tests before you code. Make sure everything works before you start.
5. Run your tests after you code, but before you `commit` and `push`. Make sure you haven't broken anything.
 - Why not write a hook that executes your tests every time you do a `commit`?
6. The longer and more descriptive the names you give for your tests, the better.



Testing in Django

This chapter provides the very basics of testing with Django, and follows a similar structure to the [Django Tutorial³](#) – with some additional notes. If there is a strong desire from readers for us to expand this chapter further, we will! Get in touch if you think this would be advantageous to you.

18.1 Running Tests

Django comes complete with a suite of tools to test the apps that you build. You can test your developed Rango app by using the terminal or Command Prompt. Simply issue the following command, and observe the output you see.

```
$ python manage.py test rango

Creating test database for alias 'default'...

-----
Ran 0 tests in 0.000s

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

This command will run through all tests that have been created for the Rango app. However, at the moment, nothing much happens. This is because the `tests.py` that lives inside the `rango` app directory is pretty much blank, except for a single import

³<https://docs.djangoproject.com/en/2.1/intro/tutorial05/>

statement. Every time you create a new Django app, this nearly-blank tests.py module is created for you to encourage you to write tests!

From the output shown above, you might also notice that a database called default is referred to. When you run tests, a temporary database is constructed, which your tests can populate and perform operations on. This way your testing is *performed independently* of your live database, satisfying one of the rules we listed above.

Testing Rango's Models

Given that there are presently no tests, let's create one! In the Category model, we want to ensure that the number of views received is zero or greater because you cannot have a negative number of views. To create a test for this, we can put the following code into Rango's tests.py module, being careful to keep the existing import statement.

```
class CategoryMethodTests(TestCase):
    def test_ensure_views_are_positive(self):
        """
        Ensures the number of views received for a Category are positive or zero.
        """
        category = Category(name='test', views=-1, likes=0)
        category.save()

        self.assertEqual((category.views >= 0), True)
```

You'll also want to make sure that you import the Category model.

```
from rango.models import Category
```

The first thing to notice about the test we have written is that we must place the tests within a class. This class must inherit from django.test.TestCase. Each method implemented within a class tests a particular piece of functionality. These methods should always have a name that starts with test_, and should always contain some form of assertion. In this example, we use assertEquals which checks whether two items are equal. However, there are lots of other assert checks you can use – as demonstrated in the [official Python documentation⁴](#). Django's testing machinery

⁴<https://docs.python.org/3/library/unittest.html#assert-methods>

is derived from the Python implementation, but also provides several asserts and specific test cases unique to Django and web development.

If we then run the test, we will see the following output.

```
$ python manage.py test rango

Creating test database for alias 'default'...
F
=====
FAIL: test_ensure_views_are_positive (rango.tests.CategoryMethodTests)
-----
Traceback (most recent call last):
  File "/Users/maxwelld90/Workspace/tango_with_django_project/rango/tests.py",
    line 12, in test_ensure_views_are_positive
      self.assertEqual((cat.views>=0), True)
AssertionError: False != True

-----
Ran 1 test in 0.001s

FAILED (failures=1)
```

We can see that Django picked up our solitary test, and it FAILED. This is because the model does not check whether the value for views is less than zero. Since we want to ensure that the values are non-zero for this particular field, we will need to update the model to ensure that this requirement is fulfilled. Update the model now by adding some code to the `save()` method of the `Category` model, located in Rango's `models.py` module. The code should check the value of the `views` attribute, and set the value to zero if the provided value is less than zero. A simple conditional check on `self.views` should suffice.

Once you have updated your model, re-run the test. See if your code now passes the test. If not, try again and work out a solution that passes the test.

Let's try adding a further test that ensures that an appropriate `slug` is created. This should mean a slug with dashes instead of spaces (-), and all in lowercase. Add the following test method to your new `CategoryMethodTests` class.

```
def test_slug_line_creation(self):
    """
    Checks to make sure that when a category is created, an
    appropriate slug is created.
    Example: "Random Category String" should be "random-category-string".
    """
    category = Category(name='Random Category String')
    category.save()

    self.assertEqual(category.slug, 'random-category-string')
```

Run the tests again. Does your code pass both tests? You should now be starting to see that if you have tests written up that you are confident satisfy the requirements of a particular component, you can write code that complies with these tests – ergo your code satisfies the requirements provided!

Testing Views

The two simple tests that we have written so far focus on ensuring the integrity of the data housed within Rango's Category model. Django also provides mechanisms to test views. It does this with a mock client (or browser) that internally makes calls to the Django development server via a URL. In these tests, you have access to the server's response (including the rendered HTML markup), as well as the context dictionary that was used.

To demonstrate this testing feature, we can create a test that checks when the index page loads. When the Category model is empty, it should present the user with a message that *exactly* says 'There are no categories present.'

```
class IndexViewTests(TestCase):
    def test_index_view_with_no_categories(self):
        """
        If no categories exist, the appropriate message should be displayed.
        """
        response = self.client.get(reverse('rango:index'))

        self.assertEqual(response.status_code, 200)
        self.assertContains(response, 'There are no categories present.')
        self.assertQuerysetEqual(response.context['categories'], [])
```

As we are using the Django `reverse()` function to perform a URL lookup, we'll need to make sure that the correct `import` statement is included at the top of the `tests.py` module, too.

```
from django.urls import reverse
```

Looking at the code above, the Django `TestCase` class has access to a `client` object which can make requests. Here, it uses the helper function `reverse()` to lookup the URL of Rango's `index` page. It then tries to issue an HTTP GET request on that page. The response is returned and stored in `response`. The test then checks several things: whether the page loaded successfully (with a 200 status code returned); whether the response's HTML contains the string '`There are no categories present.`'; and whether the context dictionary used to render the response contains an empty list for the `categories` supplied.

Recall that when you run tests, a new database is created, which by default is not populated. This is true for each test method – and explains why the categories you create in the two `CategoryMethodTests` tests are not visible to the test in `IndexViewTests`.

Now let's check the `index` view when categories *are* present. We can add a helper function for us to achieve this. This simple function doesn't live within a class – just place it in the `test.py` module.

```
def add_category(name, views=0, likes=0):
    category = Category.objects.get_or_create(name=name)[0]
    category.views = views
    category.likes = likes

    category.save()
    return category
```

`add_category()` takes a `name` string, `views` and `likes` integers, and adds the category to the `Category` model, returning a reference to the model instance it creates (or retrieves, if it already exists). Note that `views` and `likes` are set to optional parameters, defaulting to zero if they are not supplied.

Make use of this helper method by creating a further test method inside your `IndexViewTests` class.

```
def test_index_view_with_categories(self):
    """
    Checks whether categories are displayed correctly when present.
    """

    add_category('Python', 1, 1)
    add_category('C++', 1, 1)
    add_category('Erlang', 1, 1)

    response = self.client.get(reverse('rango:index'))
    self.assertEqual(response.status_code, 200)
    self.assertContains(response, "Python")
    self.assertContains(response, "C++")
    self.assertContains(response, "Erlang")

    num_categories = len(response.context['categories'])
    self.assertEqual(num_categories, 3)
```

In this test, we populate the database with three sample categories, Python, C++ and Erlang. We make use of our helper function `add_category()` here. We then again request the `index` page, check the response was successful (HTTP 200), and check whether all three categories are presented on the page. We also check the number of categories listed in the context dictionary is equal to three – the number of categories present in the database at the time.

Run the test. Does it pass?



Update your Existing Tests

Update the existing tests that you have created so far to make use of the helper `add_category()` function.

Testing the Rendered Page

Django's test suite also allows you to perform tests that load up your web app, and programmatically interact with the DOM elements on the rendered HTML pages. This is incredibly useful, as you can test your web app as a human would – by ‘clicking’ links, or entering information into form fields and submitting them. This is achieved with some further third-party “driver” software that controls interactions on a webpage for you.

We don't explicitly cover how to set these tests up here, but you should refer to the [official Django documentation](#)⁵ to learn more and see how this is achieved.

If you really want us to include examples of how to run tests this way, let us know!

18.2 Examining Testing Coverage

One further point of discussion about testing is *coverage*. Code coverage measures how much of your codebase has been tested. You can install an additional package called `coverage` (`$ pip install coverage`) that can automatically analyse how much of your codebase that has been covered by tests. Once `coverage` is installed, run the following command at your terminal or Command Prompt.

```
$ coverage run --source='.' manage.py test rango
```

This will run through all of the tests that you have implemented so far, and collect coverage data for the Rango app. To see the report, you then need to type the following command once the first command is completed.

⁵<https://docs.djangoproject.com/en/2.2/topics/testing/tools/#liveservertestcase>

```
$ coverage report
```

Name	Stmts	Miss	Cover
manage.py	9	2	78%
populate_rango.py	33	33	0%
rango/__init__.py	0	0	100%
rango/admin.py	10	0	100%
rango/apps.py	3	3	0%
rango/bing_search.py	38	32	16%
rango/forms.py	34	6	82%
rango/migrations/0001_initial.py	6	0	100%
rango/migrations/0002_auto_20200111_1313.py	4	0	100%
rango/migrations/0003_category_slug.py	4	0	100%
rango/migrations/0004_auto_20200111_1702.py	6	0	100%
rango/migrations/__init__.py	0	0	100%
rango/models.py	33	3	91%
rango/templatetags/__init__.py	0	0	100%
rango/templatetags/rango_template_tags.py	6	0	100%
rango/tests.py	34	0	100%
rango/urls.py	4	0	100%
rango/views.py	207	139	33%
tango_with_django_project/__init__.py	0	0	100%
tango_with_django_project/settings.py	28	0	100%
tango_with_django_project/urls.py	12	1	92%
tango_with_django_project/wsgi.py	4	4	0%
TOTAL	475	223	53%

We can see from the output of this command that critical parts of the code have not been tested. The `views.py` has a pretty low coverage percentage as an example of 34%. Therefore, this output can provide you with a measure-based approach to determine where to focus your efforts on writing tests.

The coverage package [has many more features⁶](#) that you can explore to make your tests even more comprehensive!

⁶<https://coverage.readthedocs.io/en/latest/>



Testing Exercises

Let's assume that we want to extend the `Page` model to include an additional field – `last_visit`. This field will be of the type `models.DateTimeField`, and represents the date and time when the page was last accessed. If the page has never been accessed, the value given will be the date and time at which the page was saved – whether this was the initial creation or an update. Given this requirement, complete the following tasks.

- Update the `Page` model to include this new field.
- Update the `Page` model to set this value on creation to the current date and time.
- Update the `GotoView` to update the field when the page is clicked.

Once you have completed these tasks, implement some tests.

- Add in a test to ensure that `last_visit` is not in the future.
- Add in a test to ensure that `last_visit` is updated when a page is requested.



Hints

You'll want to use the timezone-aware functionality of Django to get the current date and time. This requires you to import: `from django.utils import timezone`, with the current date and time accessible with `timezone.now()`.

In order to compare two dates, you will need to make use of the `assertTrue()` method to perform the assertion. This takes a boolean expression, where you can perform your evaluation, such as `page.last_visit < timezone.now()`.

When working on the second test, you'll need to make a call to the `rango:goto` view, passing the `page_id` as a parameter. You can do this with the `client.get()` method by providing a dictionary as the second argument with `page_id` as the key, and the ID of the page you are looking to access as the value. Depending upon how you implement this test, it may also require you to refresh a model instance from the database. This can be achieved using the `refresh_from_db()` method.

As a final hint, it may also be helpful to write a further helper function to add a page to a given category. If you are stuck, we have implemented a [model solution on GitHub](https://github.com/maxwelld90/tango_with_django_2_code/tree/ce0911025891970d6ee3ac5b8745ea970477151c/).



Other Test Resources

- Run through Part Five of the official Django Tutorial⁷ to learn more about testing.
- Check out the tutorial on test-driven development by Harry Percival⁸.

⁷<https://docs.djangoproject.com/en/2.1/intro/tutorial05/>

⁸<https://www.obeythetestinggoat.com/book/part1.harry.html>

19. Deploying Your Project

This chapter provides a step-by-step guide on how to deploy your Django application on [PythonAnywhere](#)¹. PythonAnywhere is an online IDE and web hosting service, geared towards hosting Python applications. The service provides in-browser access to the server-based Python and Bash command line interfaces, meaning you can interact with PythonAnywhere's servers just like you would with a regular terminal instance on your computer. Currently, PythonAnywhere is offering a free beginner account, which is perfect to get started. It provides you with an adequate amount of storage space and CPU time to get a single Django project up and running.



Are you using Git?

You can do this chapter independently as we have already implemented Rango – our implementation is [available from GitHub](#)². If you haven't used Git/GitHub before, you can check out our [chapter on using Git](#).

19.1 Creating a PythonAnywhere Account

First, [sign up for a Beginner PythonAnywhere account](#)³. We recommend that you take the seven-step tour to get familiar with the PythonAnywhere interface. If your application takes off and becomes popular, you'll always have the ability to upgrade your account at a later date to gain more storage space and CPU time along with several other benefits, such as hosting specific domains and the ability to [SSH](#)⁴ in, for example.

Once your account has been created and you have confirmed your email, you will have your little slice of the World Wide Web at <http://<username>.pythonanywhere.com>,

¹https://www.pythonanywhere.com/?affiliate_id=000116e3

²https://github.com/maxwelld90/tango_with_django_2_code

³https://www.pythonanywhere.com/?affiliate_id=000116e3

⁴https://en.wikipedia.org/wiki/Secure_Shell

where <username> is your PythonAnywhere username. It is from this URL that your hosted application will be available.



Your Username

Your username will vary from the one we have used to demonstrate the functionality of PythonAnywhere. We chose `rangodemo2020`, and you'll see this throughout the screenshots and code snippets in this chapter. Simply substitute `rangodemo2020` with your username when required.

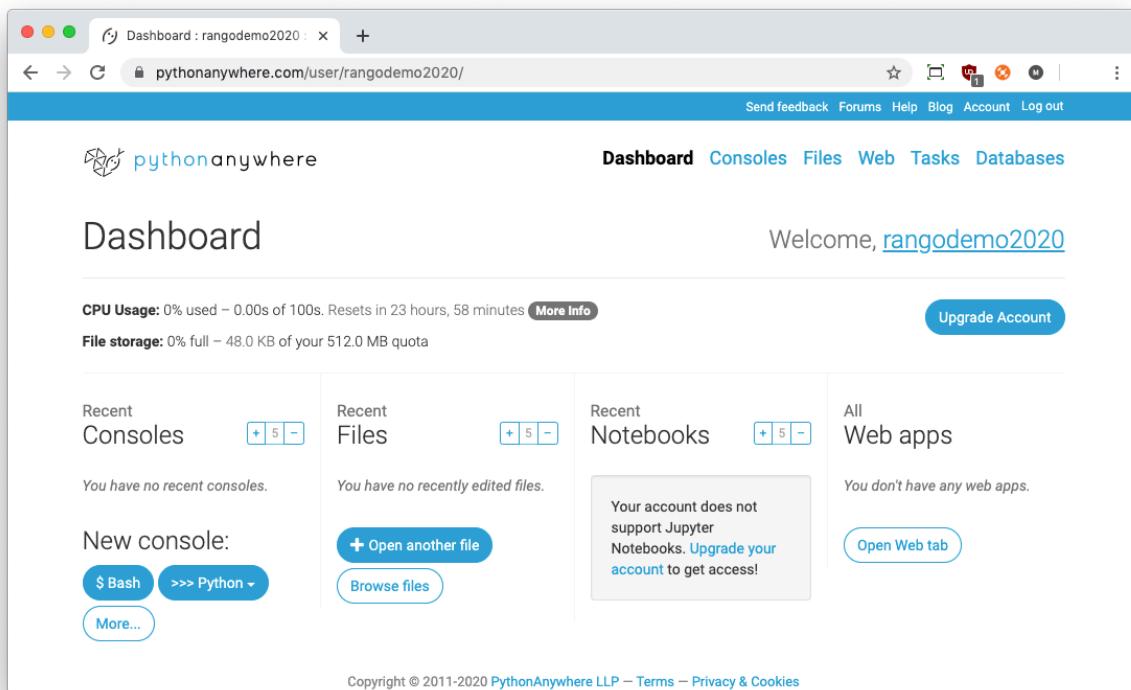
19.2 The PythonAnywhere Web Interface

The PythonAnywhere web interface contains a *dashboard* which in turn provides a series of different components allowing you to manage your application. The components as [illustrated in the figure below](#) include:

- *Consoles*, allowing you to create and interact with Python and Bash console instances;
- *Files*, which allows you to upload to and organise files within your disk quota; and
- *Web apps*, allowing you to configure settings for your hosted web application;

Other components exist, such as *Notebooks*, but we won't be using them here – we'll be working primarily with the *consoles* and *web app* components. The [PythonAnywhere Wiki](#)⁵ provides a series of detailed explanations on how to use the other components if you are interested in finding out more.

⁵<https://www.pythonanywhere.com/wiki/>



The PythonAnywhere dashboard, showing the main components you can use. We're logged in as `rangodemo2020` here. You can see links to Consoles, Files, Notebooks (that we don't use here), and web apps (or the settings for web apps).

19.3 Creating a Virtual Environment

As part of its standard default Bash environment, PythonAnywhere comes with Python 2.7+ and several pre-installed Python Packages (including *Django*). Since we are using a different setup, we need to set up the Python 3 version manually through the creation of a virtual environment.

First, open a Bash console. By clicking `$ Bash` from under the New console header. If you run `python --version` then you will see that the Python is 2.7+. To set up our Python 3.7+ environment, we'll need to issue the following command.

```
$ mkvirtualenv -p python3.7 rangoenv
Running virtualenv with interpreter /usr/bin/python3.7
Using base prefix '/usr'
New python executable in /home/rangodemo2020/.virtualenvs/rangoenv/bin/python3.7
Also creating executable in /home/rangodemo2020/.virtualenvs/rangoenv/bin/python
Installing setuptools, pip, wheel...done.
virtualenvwrapper.user_scripts creating
    /home/rangodemo2020/.virtualenvs/rangoenv/bin/predeactivate
virtualenvwrapper.user_scripts creating
    /home/rangodemo2020/.virtualenvs/rangoenv/bin/postdeactivate
virtualenvwrapper.user_scripts creating
    /home/rangodemo2020/.virtualenvs/rangoenv/bin/preactivate
virtualenvwrapper.user_scripts creating
    /home/rangodemo2020/.virtualenvs/rangoenv/bin/postactivate
virtualenvwrapper.user_scripts creating
    /home/rangodemo2020/.virtualenvs/rangoenv/bin/get_env_details
```

Above we have used Python 3.7, but you can check which version of Python you are using, and use that instead. Remember that `rangodemo2020` will be replaced with your username. The process of creating the virtual environment will take a little while to complete (as you are using a host shared with many others). After the environment has been created, you will be presented with a slightly different prompt. Exciting stuff!

(rangoenv) ~ \$

Note the inclusion of (rangoenv) compared to the previous prompt. This signifies that the `rangoenv` virtual environment has been activated, so any package installations will be done within that virtual environment, leaving the wider system setup alone. If you issue the command `ls -la`, you will see that a directory called `.virtualenvs` has been created. This is the directory in which all of your virtual environments and associated packages will be stored. To confirm the setup, issue the command `which pip`. This will print the location in which the active `pip` binary is located - hopefully within `.virtualenvs` and `rangoenv`, as shown in the example below.

```
/home/rangodemo2020/.virtualenvs/rangoenv/bin/pip
```

To see what packages are already installed, enter `pip list`. Now we can customise the virtual environment by installing the required packages for our Rango application. Install all the required packages by issuing the following commands.

```
$ pip install django==2.1.5  
$ pip install pillow==5.4.1  
$ pip install django-registration-redux==2.2  
$ pip install requests  
$ pip install coverage
```

If you decided to use the [optional Bcrypt password hasher](#), you'll also need to install the Bcrypt library.

```
$ pip install bcrypt
```

Ensure that you replace the version of Django specified above with the one you are using for development. Here we have used version 2.1.5. Remember that this book supports a range of versions, from 2.0 to 2.2. Not using the same version as your development machine could lead to some weird errors, which will be frustrating, to say the least. To ensure you have the same setup, we recommend using `pip freeze > requirements.txt` on your local development machine. This will note down all the packages in your current development environment. Then on PythonAnywhere, you can run `pip install -r requirements.txt` to install all the packages in one go.



Using requirements.txt?

If you went down the path of using `pip freeze` on your computer to create a `requirements.txt` file of all packages, this should have been added to your Git repository.

In this instance, [skip to cloning your repository first](#), then come back here. When you clone your repository, you gain access to your requirements files, which means you can then install all the packages to your `rangoenv` virtual environment on PythonAnywhere with the command `pip install -r requirements.txt`.



Waiting to Download...

Since you're on a shared host, downloading and installing these packages will take considerably longer than doing so on your own computer. Don't worry if you think it's crashed – give it time!

Once installed, check if Django has been installed correctly. You can do this with the command `which django-admin.py`. You should receive output similar to the following.

```
/home/rangodemo2020/.virtualenvs/rangoenv/bin/django-admin.py
```



Virtual Environments on PythonAnywhere

PythonAnywhere also provides instructions on how to set up virtual environments. [Check out their Wiki documentation for more information⁶.](#)

Virtual Environment Switching

Moving between virtual environments can be done pretty easily. PythonAnywhere should have this covered for you. Below, we provide you with a quick tutorial on how to switch between virtual environments.

You can launch into an existing virtual environment with the `workon` command. For example, load up the `rangoenv` environment with the following command.

```
~ $ workon rangoenv
```

Here, `rangoenv` can be replaced with the name of the virtual environment you wish to use. Your prompt should then change to indicate you are working within a virtual environment. This is shown by the addition of `(rangoenv)` to your prompt.

```
(rangoenv) ~ $
```

You can then leave the virtual environment using the `deactivate` command. Your prompt should then be missing the `(rangoenv)` prefix, with an example shown below. This confirms that the environment has been successfully deactivated.

⁶<https://help.pythonanywhere.com/pages/VirtualEnvForNewerDjango>

```
(rangoenv) ~ $ deactivate  
~ $
```

Cloning your Git Repository

Now that your virtual environment for Rango is all set up, you can now clone your Git repository to obtain a copy of your project's files. On PythonAnywhere, clone your repository by issuing the following command from your home directory.

```
$ git clone https://github.com/<OWNER>/<REPO_NAME>
```

You will of course be looking to replace <OWNER> with the username of the person who owns the repository and <REPO_NAME> with the name of your project's repository. For example, a repository `tango_with_django_project` hosted by `djangolearner` would have the following command.

```
$ git clone https://github.com/djangolearner/tango_with_django_project
```

You'll see the following output showing that the cloning of the repository was successful.

```
$ git clone https://github.com/djangolearner/tango_with_django_project/  
Cloning into 'tango_with_django_project'...  
remote: Enumerating objects: 335, done.  
remote: Counting objects: 100% (335/335), done.  
remote: Compressing objects: 100% (239/239), done.  
remote: Total 335 (delta 203), reused 213 (delta 81), pack-reused 0  
Receiving objects: 100% (335/335), 1.11 MiB | 484.00 KiB/s, done.  
Resolving deltas: 100% (203/203), done.  
Checking connectivity... done.  
Checking out files: 100% (54/54), done.
```

Setting Up the Database

With your repository cloned, you must then prepare your database. If you have the database committed to the repository, delete this copy. We'll start from scratch.

We'll also be making use of the `populate_rango.py` module that we created earlier in the book to populate the database with sample data. As we'll be running the module, you must ensure that you are using the `rangoenv` virtual environment (i.e. you see `(rangoenv)` as part of your prompt – if not, invoke `workon rangoenv`).

From your home directory (denoted by the `~` in your prompt), you want to navigate to the directory containing your `manage.py` and `populate_rango.py` scripts. Following the instructions we provided earlier, this would be at `~/tango_with_django_project/` – the directory that you just cloned. If you chose to follow a different structure, navigate to the location where these files are stored.



How do I navigate?

In a Bash terminal, you can change the directory with the `cd` command, followed by the path you wish to change to. In this instance, you would issue the command `cd ~/tango_with_django_project/`.

Note that when you change directory, your prompt will change to reflect your *present working directory*. `~ $` would change to `~/tango_with_django_project $`, for instance.

Once you have changed directory, issue the following commands. These should be second nature to you by now!

```
$ python manage.py makemigrations rango
$ python manage.py migrate
$ python populate_rango.py
$ python manage.py createsuperuser
```

You should know exactly what all of these commands do. Of course, the final command will require some interaction on your part to create an administrator's account for your Django project.

19.4 Setting up your Web Application

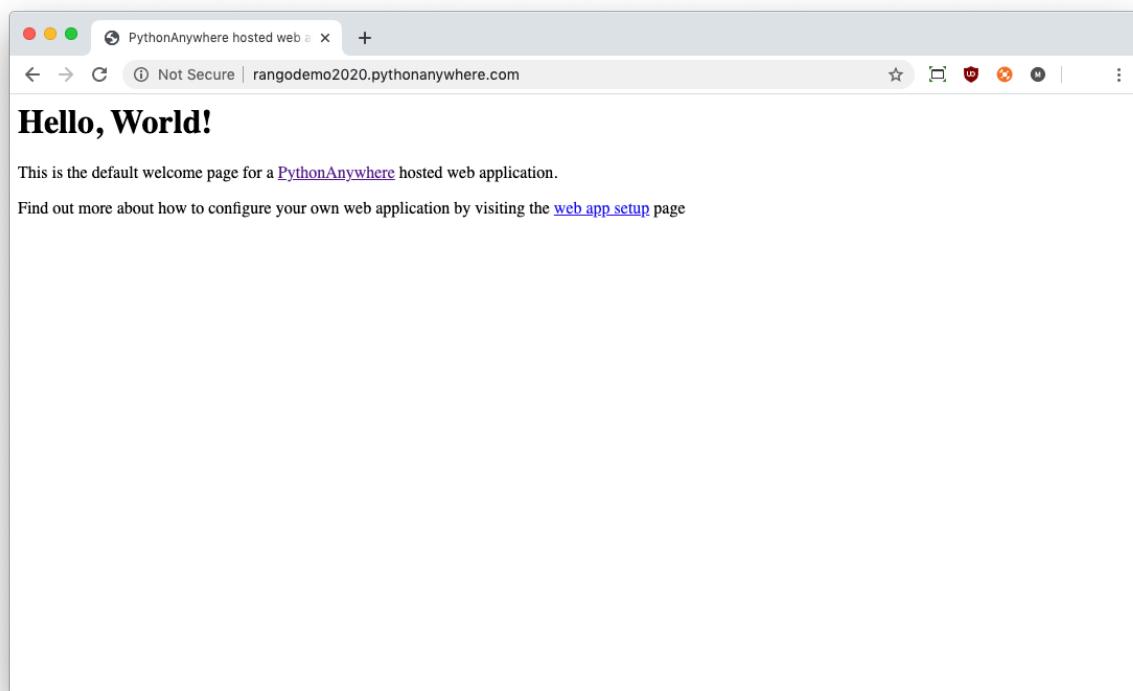
Now that the database is set up, we need to configure the PythonAnywhere *NGINX*⁷ webserver to serve up your application. To do this, navigate back to your Pytho-

⁷<https://www.nginx.com/resources/wiki/>

nAnywhere dashboard, find the *Web apps* section, and click the *Open Web tab* button. On the left of the page that appears, click *Add a new web app*.

A popup box will then appear. Follow the instructions on-screen, and when the time comes, select the *manual configuration* option and complete the wizard. Make sure you select the same Python version as the one you selected earlier. Click *Next*, and PythonAnywhere will then set up your web app, redirecting you to an updated *Web app* page, complete with a green button to reload the app.

In a new tab or window in your web browser, go visit your PythonAnywhere subdomain at the address `http://<username>.pythonanywhere.com`. You should be presented with the [default Hello, World! webpage, as shown below](#). This is because the WSGI script is currently serving up this simple page, and not your Django application. This is the next thing we need to work on – updating the WSGI to serve your Django app instead!



The default PythonAnywhere hello world webpage.

Configure the Virtual Environment

However, before we do this, we need to configure our new web app to use the virtual environment we set up previously. Navigate to the *Web* tab in PythonAnywhere's interface if you aren't already there. From there, scroll down the page until you see the heading *Virtualenv*.

Enter the path to your virtual environment. Click the red text, which is replaced with an input box. Assuming you created a virtual environment called `rangoenv`, the path would be:

```
/home/rangodemo2020/.virtualenvs/rangoenv
```

When you have entered the path, click the tick so submit it. Once again, we ask you to replace `rangodemo2020` with your PythonAnywhere username.

Now scroll to the *Code* section. From here, you can set the path to your Django project's source code. Click the red text next to *Source code*, which will again turn into an input field.

```
/home/rangodemo2020/tango_with_django_project/
```

Once again, replace `rangodemo2020` with your username, and make sure that the path after the username points to the directory that stores your `manage.py` script. This is imperative: you can open a new Bash *Console* to double-check if you aren't sure what the path is. If you do have to do this, you can navigate to the correct directory, type `ls` to confirm that `manage.py` is indeed present in that directory, and then issue the command `pwd` to retrieve the full path to that directory. This is the path you will need to supply.

Configuring the WSGI Script

The *Web Server Gateway Interface*⁸ (*WSGI*) provides a simple and universal interface between web servers and web applications. PythonAnywhere uses WSGI to bridge

⁸http://en.wikipedia.org/wiki/Web_Server_Gateway_Interface

the server-application link and map incoming requests to your subdomain to your web application.

To configure the WSGI script, navigate to the *Web* tab in PythonAnywhere's interface. Under the *Code* header, you can see a link to the WSGI configuration file in the Code section: e.g. `/var/www/rangodemo2020_pythonanywhere_com_wsgi.py`. Of course, the `rangodemo2020` part will have your username instead.

The people at PythonAnywhere have set up a sample WSGI file for us with several possible configurations. For your web application, you'll need to configure the Django section of the file by clicking on the link to open a simple editor. The example below can be used as the WSGI configuration for your application.

```
import os
import sys

# Add your project's directory to the PYTHONPATH
path = '/home/rangodemo2020/tango_with_django_project/'
if path not in sys.path:
    sys.path.append(path)

# Move to the project directory
os.chdir(path)

# Tell Django where the settings.py module is located
os.environ.setdefault('DJANGO_SETTINGS_MODULE',
                      'tango_with_django_project.settings')

# Set up Django -- let it instantiate everything!
import django
django.setup()

# Import the Django WSGI to handle requests
import django.core.handlers.wsgi
application = django.core.handlers.wsgi.WSGIHandler()
```

Once again, ensure that you replace `rangodemo2020` with your PythonAnywhere username, and update any other path settings to suit your application. The `path` variable should contain the absolute path to the directory containing your `manage.py` script. You should also remove all other code from the WSGI configuration script to ensure no conflicts take place, leaving only the code we provide above.

The script adds your project’s directory to the `PYTHONPATH` for the Python instance that runs your web application. This allows Python to access your project’s modules. If you have additional paths to add, you can easily insert them here. You can then specify the location of your project’s `settings.py` module. The final step is to include the Django WSGI handler and invoke it for your application by setting the `application` variable.

When you have completed the WSGI configuration, click the green *Save* button at the top of the webpage. Before we can (re)load the application, we need to set a few security parameters (see the [Django Documentation for a Security Deployment Checklist⁹](#)).

Allowing your Hostname

Django provides a useful feature which will only accept requests from *allowed* hosts. By only allowing specified domains to be served by your web server, this reduces the chance that your app could be part of a [HTTP Host Header attack¹⁰](#). If you were to load and view your application now, you would encounter a `DisallowedHost` exception stopping your app from loading.

This is a simple problem to fix and involves a change in your project’s `settings.py` module. First, work out your app’s URL on PythonAnywhere. For a basic account, this will be `rangodemo2020.pythonanywhere.com`, where `rangodemo2020` is replaced with your PythonAnywhere username. It’s a good idea to edit this file locally (on your computer), then `git add`, `git commit` and `git push` your changes to your Git repository, before downloading the changes to your PythonAnywhere account. Alternatively, you can edit the file directly on PythonAnywhere by editing the file in the web interface’s files component – or using a text editor in the terminal, like `nano` or `vi`.

With this information, open your project’s `settings.py` module and locate the `ALLOWED_HOSTS` list, which by default will be empty (and found near the top of the file). Add a string with your PythonAnywhere URL into that list – such that it now looks like the following example.

⁹<https://docs.djangoproject.com/en/2.1/howto/deployment/checklist/>

¹⁰<https://www.acunetix.com/blog/articles/automated-detection-of-host-header-attacks/>

```
ALLOWED_HOSTS = ['rangodemo2020.pythonanywhere.com']
```

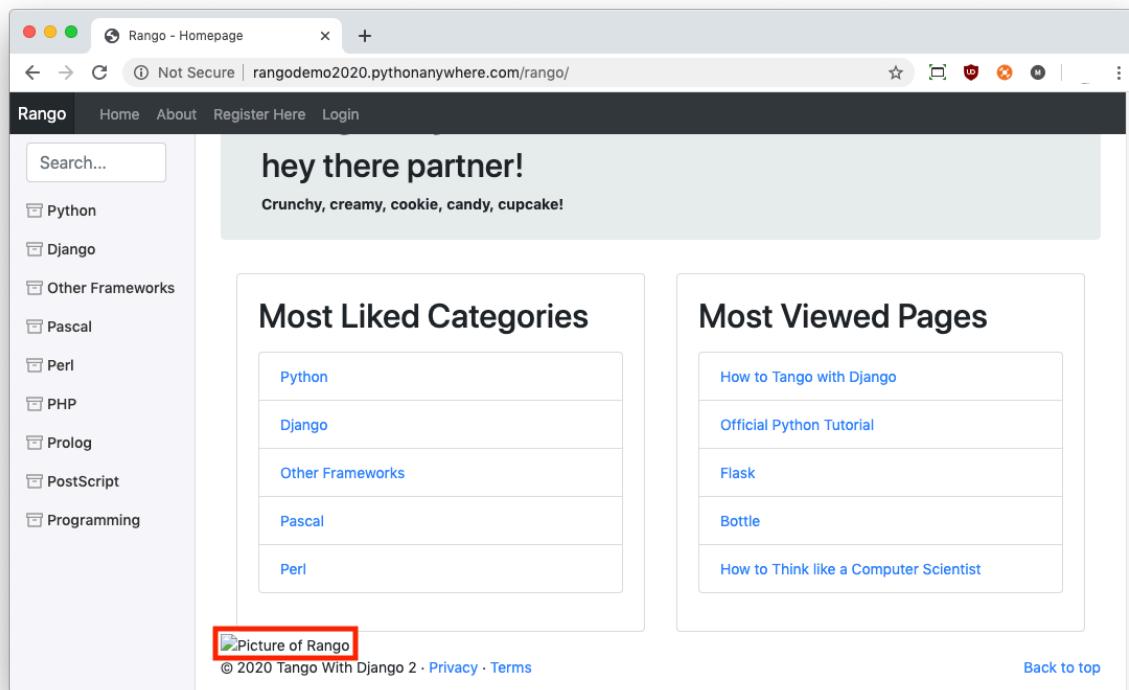
If you have edited the file on your computer, you can now go through the (hopefully) familiar process of running the `git add settings.py`, `git commit` and `git push` commands to make changes to your Git repository. Once done, you should then run the `git pull` command to retrieve the changes on your PythonAnywhere account through a console. If you have edited the file directly on PythonAnywhere, simply save the file.



No Place like 127.0.0.1

When debugging, you might also want to add the host `127.0.0.1` to your `ALLOWED_HOSTS` list so that you'll be able to continue working on your app locally. Of course, this should be removed in production.

All that then remains is for you to reload your PythonAnywhere app. This can be done by clicking the *Reload* button in the PythonAnywhere *Web* page. Once you've done this, access your app's URL, and you should [see your app working, but without static media](#). We'll address this issue shortly.



The index page of Rango, almost successfully deployed. Static files are not working, as highlighted with the red box. The picture of Rango does not appear!



Bad Gateway Errors

During deployment sometimes you might receive HTTP 502 - Bad Gateway errors instead of your application. Try reloading your application again, and then waiting a little longer. If the problem persists, try reloading again. If the problem persists, [check out your log files](#) to see if any errors are occurring before contacting the PythonAnywhere support.

Setting the Secret Key

It is wise not to commit your secret key to your git repository – and instead load the key in from a separate, uncommitted file. To do this, open your project's `settings.py` file and either change the secret key that you are using in production (a hacky solution), or load it from a file, as we demonstrate with the example below.

```
key = None
with open('secret.key') as f:
    key = f.read().strip()

SECRET_KEY = key
```

This sets SECRET_KEY to the contents of the secret.key file, located in your project's root directory (i.e. tango_with_django_project directory – the directory with manage.py). Create this file, and take the string representation of SECRET_KEY from settings.py, and paste it into that new file. This neatly separates your secret key from being placed in a repository! Of course, this is just to demonstrate – anyone can look at a previous commit from your repository and take the key. In future Django projects, keep this in mind!

Adding your Search API Key

While you are adding secret keys, also [add your search API key](#) to the file bing.key. This will ensure that the search functionality in Rango is enabled. You'll again want to keep this file uncommitted from your repository. **Never push this key to a public repository!**

Turning off DEBUG Mode

Next, you need to turn off debug mode. If you leave debug mode on, your deployment can provide malicious users with sensitive information about your web application when they get it to crash through the helpful error messages that will have undoubtedly helped you during development.

Therefore, we highly recommend that you instruct Django that it is now running on a production server. To do this, open your project's settings.py file and change DEBUG = True to DEBUG = False. This disables [Django's debug mode](#)¹¹, and removes explicit error messages. However, you can still view Python stack traces to debug any exceptions that are raised as people use your app. These are provided in the server logs, as we will [discuss later](#).

¹¹<https://docs.djangoproject.com/en/2.1/ref/settings/#debug>



Remember to `git add`, `git commit` and `git push` – and Reload!

If you edit your files locally, you'll need to push them up to your Git repository and `git pull` in a PythonAnywhere console. Furthermore, remember to reload your app on PythonAnywhere anytime you make changes to it! If you don't, your changes won't be applied. This can be done by pushing the green *Reload* button in the *Web* tab of the PythonAnywhere interface.

Assigning Static Paths

We're almost there. One issue that we still have to address is to sort out the static paths for our application. Doing so will allow PythonAnywhere's servers to serve your project's static content. Open the *Web* tab of the PythonAnywhere interface to fix this.

Once loaded, perform the following under the *Static files* header. Here, we need to add the correct URLs and filesystem paths to allow PythonAnywhere's web server to find and serve your static media files.

First, we should set the location of the Django admin interface's static media files. Click the *Enter URL* text, and type in `/static/admin/`. Click the tick to confirm your input, and then click the *Enter path* link to the right, entering the following long-winded filesystem path (all on a single line). Be careful! We need to split the line up here to make it fit...

```
/home/rangodemo2020/.virtualenvs/rangoenv/lib/python3.7/site-packages/django/contrib/admin/static/admin
```

As usual, replace `rangodemo2020` with your PythonAnywhere username. `python3.7` should also be replaced with the version of Python that you are using, be it `python3.6` or `python3.7`. This has to match the version you selected for your virtual environment. You may also need to change `rangoenv` if this is not the name of your application's virtual environment. Remember to click the tick to confirm your input.

We will also need to add a mapping to serve static files. Once again, click *Enter URL*, this time on the second line, and enter `/static/`. Confirm this entry. The

path should then look like `/home/rangodemo2020/tango_with_django_project/static`. In essence, you should point this path to the location of the `static` directory in your project. For our example, this path works – `rangodemo2020` is our PythonAnywhere username, and `tango_with_django_project` is the directory in which our Git repository was cloned.

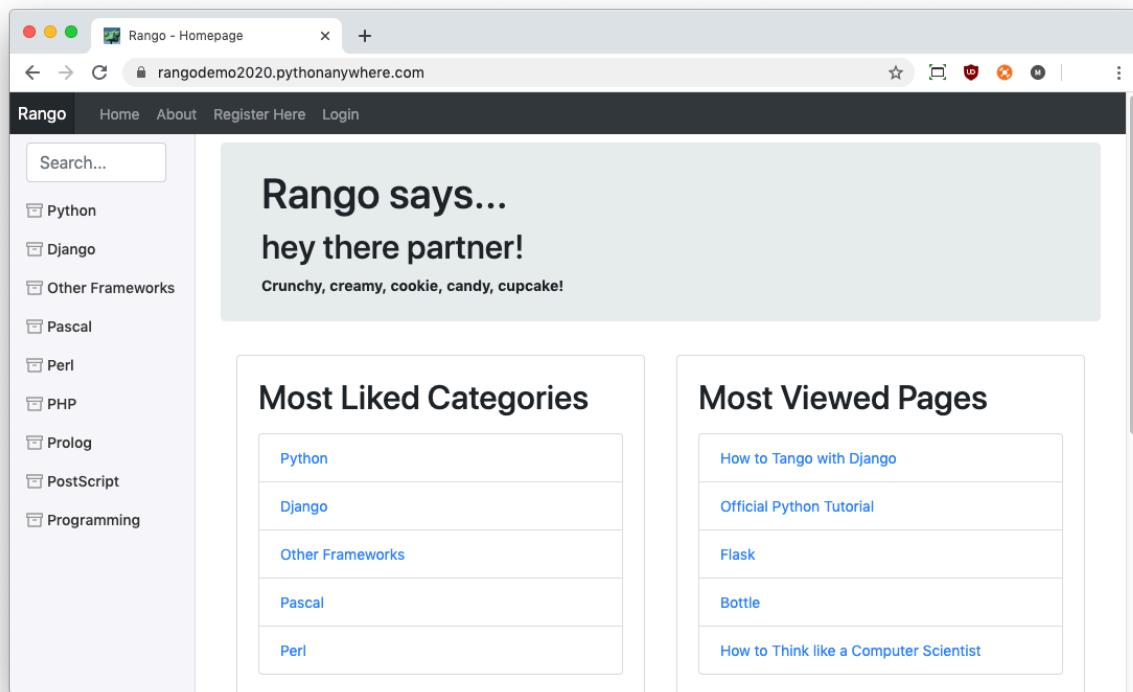
Once completed, your app should then have access to its stylesheets, images and JavaScript files. The same applies to the Django admin interface. Don't forget to *Reload* your app once these paths have been applied!

Enabling HTTPS

PythonAnywhere also provides really straightforward support for enabling [HTTPS¹²](#) functionality, adding a layer of encryption between the client and the server. PythonAnywhere can provide you with a security certificate to say your app is genuine. It's super easy, and definitely worth doing!

To enable HTTPS, go to the *Web* tab of the PythonAnywhere interface, and look for the *Security* section. Find the switch for *Force HTTPS*, and switch this feature on. Once you have done this, scroll up to the top of the page and *Reload* your app once more. If you then navigate to your app's URL, you should then notice that you are redirected to `https://rangodemo2020.pythonanywhere.com` (of course, replacing `rangodemo2020` with your username!) – denoting that you are using your app using HTTPS.

¹²<https://en.wikipedia.org/wiki/HTTPS>



Rango, deployed using HTTPS. Excellent!

19.5 Log Files

Deploying your app to an online environment introduces another layer of complexity to proceedings. You will likely encounter new and potentially confusing errors due to a whole host of additional problems you can face. When facing such errors, vital clues to help you solve them can be found in one of three log files that PythonAnywhere's web server creates for you.

If you find yourself confused with an issue, you can view the log files on the PythonAnywhere web interface. Clicking the *Web* tab, look for the *Logfiles* section. Alternatively, you can access logs from the console in directory `/var/log/`. The three log files are:

- `access.log`, which provides a log of requests made to your subdomain;

- `error.log`, which logs any error messages produced by your web application; and
- `server.log`, providing log details for the UNIX processes running your application.

Note that the names for each log file are prepended with your subdomain. For example...

`access.log`

would have the complete filename

`rangodemo2020.pythonanywhere.com.access.log`.

When debugging, you may find it useful to delete or move the log files so that you don't have to scroll through a huge list of previous attempts. If the files are moved or deleted, they will be recreated automatically when a new request or error arises. The most recent issue appears at the bottom of the file, and it will provide you with the clues you need to solve it. Remember, you can always paste the error message into your search engine to see what others have done to solve the problem. We are sure that you won't be the first person to encounter the problem you are facing!



Exercises

Congratulations, you've completed and deployed Rango!

- Go through the [Django's Security Deployment Checklist¹³](#) and configure your application to be more secure.
- Tweet a link of your application to [@tangowithdjango¹⁴](#).
- Tweet or e-mail us to let us know your thoughts on the tutorial!

¹³<https://docs.djangoproject.com/en/2.1/howto/deployment/checklist/>

¹⁴<https://twitter.com/tangowithdjango>

20. Final Thoughts

In this tutorial, we have gone through the process of web development from specification to deployment. Along the way, we have shown how to use the Django framework to construct the necessary models, views and templates associated with a sample web application, *Rango*. We have also demonstrated how toolkits and services like Bootstrap, JQuery and PythonAnywhere can be integrated within an application. However, the road doesn't stop here.

We have only painted the broad brush strokes of a web application in this tutorial. As you have probably noticed, there are lots of improvements that could be made to Rango – and these finer details often take a lot more time to complete as you polish the application. By developing your application with solid foundations, you will be able to construct up to 80% of your site very rapidly and get a working demo online.

In future versions of this book, we intend to provide some more details on various aspects of the framework – along with covering the basics of some of the other fundamental technologies associated with web development. If you have any suggestions or comments about how to improve the book please get in touch.

Please report any typos, bugs, or other issues online via [Twitter¹](#), or submit change requests via [GitHub²](#). Thank you!



One Last Exercise

There's one last little thing that we're sure you can now address. On Rango's index page, page links currently go straight to the URL stored in the database. We're not tracking these clicks! *What would you have to do to ensure when a link here is clicked, it's tracked?*

¹<https://twitter.com/tangowithdjango>

²https://github.com/leifos/tango_with_django_2

20.1 Acknowledgements

This book was written to help teach web application development to computing science students. In writing the book and the tutorial, we have had to rely upon the awesome Django community and the Django Documentation for the answers and solutions. This book is really the combination of that knowledge pieced together in the context of building Rango.

We would also like to thank all the people who have helped to improve this resource by sending us comments, suggestions, Git issues and pull requests. If you've sent in changes over the years, please do remind us if you are not on the list!

Adam Kikowski, **Adam Mertz³**, **Alessio Oxilia**, Ally Weir, **bernieyangmh⁴**, **Breakerfall⁵**, **Brian⁶**, Burak K.⁷, **Burak Karaboga**, **Can Ibanoglu⁸**, **Charlotte**, Claus Conrad⁹, **Codenius¹⁰**, **cspollar¹¹**, **Dan C**, **Darius¹²**, **David Manlove**, **David Skillman¹³**, **Deep Sukhwani¹⁴** Devin Fitzsimons¹⁵, **Dhiraj Thakur¹⁶**, Duncan Drizy, **Gerardo A-C¹⁷**, Giles T.¹⁸, **Grigoriy M¹⁹**, James Yeo, **Jan Felix Trettow²⁰**, Joe Maskell, **Jonathan Sundqvist²¹**, Karen Little, **Kartik Singhal²²**, koviusesGitHub²³, **Krace Kumar²⁴**, ma-152478²⁵, **Manoel Maria**, marshwiggle²⁶ **Martin de G.²⁷**, Matevz P.²⁸, **mHulb²⁹**,

³<https://github.com/Amertz08>

⁴<https://github.com/bernieyangmh>

⁵<https://github.com/breakerfall>

⁶<https://github.com/flycal6>

⁷<https://github.com/McMutton>

⁸<https://github.com/canibanoglu>

⁹<https://github.com/cconrad>

¹⁰<https://twitter.com/Codenius>

¹¹<https://github.com/cspollar>

¹²<https://github.com/dariushazimi>

¹³<https://github.com/reggaedit>

¹⁴<https://github.com/ProProgrammer>

¹⁵<https://github.com/aisflat439>

¹⁶<https://github.com/dhirajt>

¹⁷<https://github.com/gerac83>

¹⁸<https://github.com/gpjt>

¹⁹<https://github.com/GriMel>

²⁰<https://twitter.com/JanFelixTrettow>

²¹<https://github.com/jonathan-s>

²²<https://github.com/k4rtik>

²³<https://github.com/koviusesGitHub>

²⁴<https://github.com/kracekumar>

²⁵<https://github.com/ma-152478>

²⁶<https://github.com/marshwiggle>

²⁷<https://github.com/martindegroot>

²⁸<https://github.com/matonsjojc>

²⁹<https://github.com/mHulb>

Michael Herman³⁰, Michael Ho Chum³¹, Mickey P.³², Mike Gleen, Muhammad Radifar, nCrazed³³, Nitin Tulswani, nolan-m³⁴, Oleg Belausov, Olivia Foulds, pawonfire³⁵, pdehaye³⁶, Peter Mash, Pierre-Yves Mathieu³⁷, Piotr Synowiec³⁸, Praestgias, pzkpfwVI³⁹, Ramdog⁴⁰, Rezha Julio⁴¹, rnevius⁴², Sadegh Kh, SaeX⁴³, Saurabh Tandon⁴⁴, Serede Sixty Six, Svante Kvarnstrom, Tanmay Kansara, Thomas Murphy, Thomas Whyyou⁴⁵, William Vincent, and Zhou⁴⁶.

Thank you all very much!

³⁰<https://github.com/mjhea0>

³¹<https://github.com/michaelchum>

³²<https://github.com/mickeydash>

³³<https://github.com/nCrazed>

³⁴<https://github.com/nolan-m>

³⁵<https://github.com/pawonfire>

³⁶<https://github.com/pdehaye>

³⁷<https://github.com/pywebdesign>

³⁸<https://holysheep.co/>

³⁹<https://github.com/pzkpfwVI>

⁴⁰<https://github.com/ramdog>

⁴¹<https://github.com/kimiamania>

⁴²<https://github.com/rnevius>

⁴³<https://github.com/SaeX>

⁴⁴<https://twitter.com/saurabhtand>

⁴⁵<https://twitter.com/thomaswhyyou>

⁴⁶<https://github.com/AugustLONG>

Appendices

Setting up your System

This supplementary chapter provides additional setup guides that complement the [initial setup chapter](#). We provide setup guides for installing and configuring the various technologies that you will be using within Tango with Django tutorial. Refer to the section that is relevant to you; you do not need to work through all of this chapter if things are already working for you.



Common Guides

This chapter provides instructions on how to set up the various technologies that you'll be using throughout Tango with Django that we *believe* will work on the largest number of systems. However, every computer setup is different. Different versions of software exist, complete with subtle differences. These differences make providing universal setup guides a very difficult thing to do.

If you are using this book as part of a course, you may be provided with setup instructions unique to your lab computers. Follow these instructions instead – a majority of the setup work will likely be taken care of for you already.

However if you are working solo on your computer and you follow the instructions provided in this chapter without success, we recommend heading to your favourite search engine and entering the problem you're having. Typically, this will involve copying and pasting the error message you see at whatever step you're struggling at. By pasting in the message verbatim, chances are you'll find someone who suffered the same issue as you – and from that point, you'll hopefully find a solution to resolve your problem.

Installing Python 3 and pip

How do you go about installing Python 3.7 on your computer? This section answers that question. As we [discussed previously](#), you may find that you already

have Python installed on your computer. If you are using a Linux distribution or macOS, you will have it installed. Some of your operating system's functionality is implemented in Python⁴⁷, hence the need for an interpreter! Unfortunately, most modern operating systems that come preloaded with Python use a version that is much older than what we require. Exceptions include Ubuntu, coming with version 3.6.5 which should be sufficient for your needs. If you do need to install 3.7, we must install this version of Python *side-by-side* with the old one.

There are many different ways in which you can install Python. We demonstrate here the most common approaches that you can use on Apple's macOS, various Linux distributions and Windows 10. Pick the section associated with your operating system to proceed. Note that we favour the use of package managers⁴⁸ where appropriate to ensure that you have the means of maintaining and updating the software easily (when required).

Apple macOS

The simplest way to acquire Python 3 for macOS is to download a .dmg image file from the [official Python website⁴⁹](#). This will provide you with a step-by-step installation interface that makes setting everything up straightforward. If your development environment will be kept lightweight (i.e. Python only), this option makes sense. Simply download the installer, and Python 3 should then be available on your Mac's terminal!

However, package managers make life easier⁵⁰ when development steps up and involves a greater number of software tools. Installing a package manager makes it easy to maintain and update the software on your computer – and even to install new software, too. macOS does not come preinstalled with a package manager, so you need to download and install one yourself. If you want to go down this route, we'll introduce you to *MacPorts*, a superb package manager offering a large host of tools⁵¹ for you to download and use. We recommend that you follow this route. Although more complex, the result will be a complete development environment,

⁴⁷http://en.wikipedia.org/wiki/Yellowdog_Updater,_Modified

⁴⁸https://en.wikipedia.org/wiki/Package_manager

⁴⁹<https://www.python.org/downloads/mac-osx/>

⁵⁰<https://softwareengineering.stackexchange.com/questions/372444/why-prefer-a-package-manager-over-a-library-folder>

⁵¹<https://www.macports.org/ports.php>

ready for you to get coding and working on whatever project you (or your future self!) will be entrusted with.

A prerequisite for using MacPorts is that you have Apple's *Xcode* environment installed. This download is several gigabytes in size. The easiest way to acquire this is through the App Store on your macOS installation. You'll need your Apple account to download that software. Once XCode has been installed, follow the following steps to setup MacPorts.

1. Verify that XCode is installed by launching it. You should see a welcome screen.
If you see this, everything is ready, so you can then quit the app.
2. Open a Terminal window. Install the XCode command line tools by entering the command

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

This will download additional software tools that will be required by XCode and additional development software that you later install.

3. Agree to the XCode license, if you have not already. You can do this by entering the command

```
$ xcode-build license
```

Read the terms to the bottom of the page, and type Y to complete – but only if you agree to the terms!

4. From [the MacPorts installation page⁵²](#), download the MacPorts installer for your correct macOS version.
5. On your Mac's Finder, open the directory where you downloaded the installer to, and double-click the file to launch the installation process.
6. Follow the steps, and provide your password to install the software.
7. Once complete, delete the installer file – you no longer require it. Close down any Terminal windows that are still open.

Once the MacPorts installation has been completed, installing Python is straightforward.

⁵²<https://www.macports.org/install.php>

1. Open a new Terminal window. It is important that you launch a new window after MacPorts installation to ensure that all the changes the installer made come into effect!
2. Enter the following command.

```
$ sudo port install python37
```

After entering your password, this will take a few minutes. Several dependencies will be required – agree to these being installed by responding with `y`.

3. Once installation completes, activate your new Python installation. Enter the command

```
$ sudo port select --set python python37
```

4. Test that the command succeeds by issuing the command `$ python`. You should then see the interpreter for Python 3.7.2 (or whatever version you just installed – as long as it starts with 3.7, this will be fine).

Once this has been completed, Python has been successfully installed and is ready to use. However, we still need to set up virtual environments to work with your installation.

1. Enter the following commands to install `virtualenv` and helper functions provided in the user-friendly wrapper.

```
$ sudo port install py37-virtualenv  
$ sudo port install py37-virtualenvwrapper
```

2. Activate `py37-virtualenv` with the following command.

```
$ sudo port select --set virtualenv virtualenv37
```

3. Edit your `~/.profile` file. This can be done with the *TextEdit* app in macOS by issuing the command `$ open ~/ .profile`. Add the following four lines at the end of the file.

```
export VIRTUALENVWRAPPER_PYTHON='/opt/local/bin/python3.7'  
export VIRTUALENVWRAPPER_VIRTUALENV='/opt/local/bin/virtualenv-3.7'  
export VIRTUALENVWRAPPER_VIRTUALENV_CLONE='/opt/local/bin/virtualenv-clone-3.7'  
source /opt/local/bin/virtualenvwrapper.sh-3.7
```

4. Save the file and close all open Terminals. After doing this, open a new Terminal. Everything should now be working and ready for you to use. Test with the following command. You should *not* see the command `not found` error.

```
$ mkvirtualenv
```



Installing Additional Software with MacPorts

MacPorts provides an extensive, preconfigured library of open-source software suited specifically for development environments. When you need something new, it's a cinch to install. **For example**, you want to install the *LaTeX* typesetting system, search the [MacPorts ports list⁵³](#) – the resultant package name being `texlive-latex`. This could then be installed with the command `$ sudo port install texlive-latex`. All software that *LaTeX* is dependent upon is also installed. This saves you significant amounts of time trying to find all the right bits of software to make things work.

To view the packages MacPorts has already installed on your system, issue the command `$ port list installed`. You will see `python37` listed!

Linux Distributions

There are many different ways in which you can download, install and run an updated version of Python on your Linux distribution. Unfortunately, methodologies vary from distribution to distribution. To compound this, almost all distributions of Linux don't have a precompiled version of Python 3.7 ready for you to download and start using (at the time of writing), although the latest release of Ubuntu uses Python 3.6 (which is sufficient).

If you do choose to install a new version of Python, we've put together a series of steps that you can follow. These will allow you to install Python 3.7.2 from scratch.

⁵³<https://www.macports.org/ports.php>

The steps have been tested thoroughly in Ubuntu 18.04 LTS; other distributions should also work with minor tweaks, especially concerning the package manager being used in step 1. A cursory search on your favourite search engine should reveal the correct command to enter. For example, on a *Red Hat Enterprise Linux* installation, the system package manager is `yum` instead of `apt`.



Assumption of Knowledge

To complete these steps, we assume you know the basics for Bash interaction, including what the tilde (~) means, for example. Be careful with the `sudo` command, and do not execute it except for the steps we list requiring it below.

1. Install the required packages for Python to be built successfully. These are listed below, line by line. These can be entered into an Ubuntu Terminal as-is; slight modifications will be required for other Linux distributions.

```
$ apt install wget  
$ apt install build-essential  
$ apt install libssl-dev  
$ apt install libffi-dev  
$ apt install python-dev  
$ apt install zlib1g-dev  
$ apt install libbz2-dev  
$ apt install libreadline-dev  
$ apt install libsqlite3-dev
```

2. Once the above packages are installed, download the source for Python 3.7.2. We create a `pytemp` directory to download the file to. We'll delete this once everything has completed.

```
$ mkdir ~/pytemp  
$ cd ~/pytemp  
$ wget https://www.python.org/ftp/python/3.7.2/Python-3.7.2.tgz
```

3. Extract the `.tgz` file.

```
$ tar zxf Python-3.7.2.tgz  
$ cd Python-3.7.2
```

4. Configure the Python source code for your computer, and build it. `altinstall` tells the installer to install the new version of Python to a different directory from the pre-existing version of Python on your computer. You'll need to enter your password for the system to make the necessary changes. This process will take a few minutes, and you'll see a lot of output. Don't panic. This is normal. If the build fails, you haven't installed all of the necessary prerequisites. Check you have installed everything correctly from step 1, and try again.

```
$ sudo ./configure --enable-optimizations  
$ sudo make altinstall
```

5. Once complete, delete the source files. You don't need them anymore.

```
$ cd ~  
$ rm -rf ~/pytemp
```

6. Attempt to run the new installation of Python. You should see the interpreter prompt for version 3.7.2, as expected.

```
$ python3.7  
Python 3.7.2 (default, Jan. 8 2019, 20:05:08)  
[GCC 7.3.0] on linux  
Type "help", "copyright", "credits" or "license" for more information  
>>> quit()  
$
```

7. By default, Python executables install to `/usr/local/bin`. Check this is correct for you. If not, run the `which` command to find out where it is installed. You'll need this path later.

```
$ which python3.7  
/usr/local/bin/python3.7
```

8. Install `virtualenv` and `virtualenvwrapper` for your new Python installation.

```
$ pip3.7 install virtualenv  
$ pip3.7 install virtualenvwrapper
```

9. Modify your `~/.bashrc` file, and include the following lines at the very bottom of the file. Note that if you are not using Ubuntu, you might need to edit `~/.profile` instead. Check the documentation of your distribution for more information. A simple text editor will allow you to do this, like `nano`.

```
EXPORT VIRTUALENVWRAPPER_PYTHON=/usr/local/bin/python3.7
source /usr/local/bin/virtualenvwrapper.sh
```

10. Restart your Terminal. Python 3.7.2 will now be set up and ready for you to use, along with the pip and virtualenv tools.

Windows

By default, Microsoft Windows comes with no installation of Python. This means that you do not have to worry about leaving existing installations alone; installing from scratch should work just fine. You can download a 64-bit or 32-bit version of Python from [the official Python website⁵⁴](#). If you aren't sure what one to download, you can determine if your computer is 32-bit or 64-bit by looking at the instructions provided on [the Microsoft website⁵⁵](#).

1. Download the appropriate installer from the [official Python website⁵⁶](#). At the time of writing, the latest release was version 3.7.2.
2. Run the installer. You'll want to make sure that you check the box saying that Python 3.7 is added to PATH. You'll want to install for all users, too. Choose the Customize option.
3. Proceed with the currently selected checkboxes, and choose Next.
4. Make sure that the checkbox for installing Python for all users is checked. The installation location will change. Refer to [the figure below](#) for an example.
5. Click Next to install Python. You will need to give the installer elevated privileges to install the software.
6. Close the installer when completed, and delete the file you downloaded.

Once the installer is complete, you should have a working version of Python 3.7 installed and ready to go. Following the instructions above, Python 3.7 is installed to the directory C:\Program Files\Python37. If you checked all of the options correctly, the PATH environment variable used by Windows should also have been updated to incorporate the new installation of Python. To test this, launch a Command Prompt window and type \$ python. Execute the command. You should see the

⁵⁴<http://www.python.org/download/>

⁵⁵<https://support.microsoft.com/en-gb/help/13443/windows-which-operating-system>

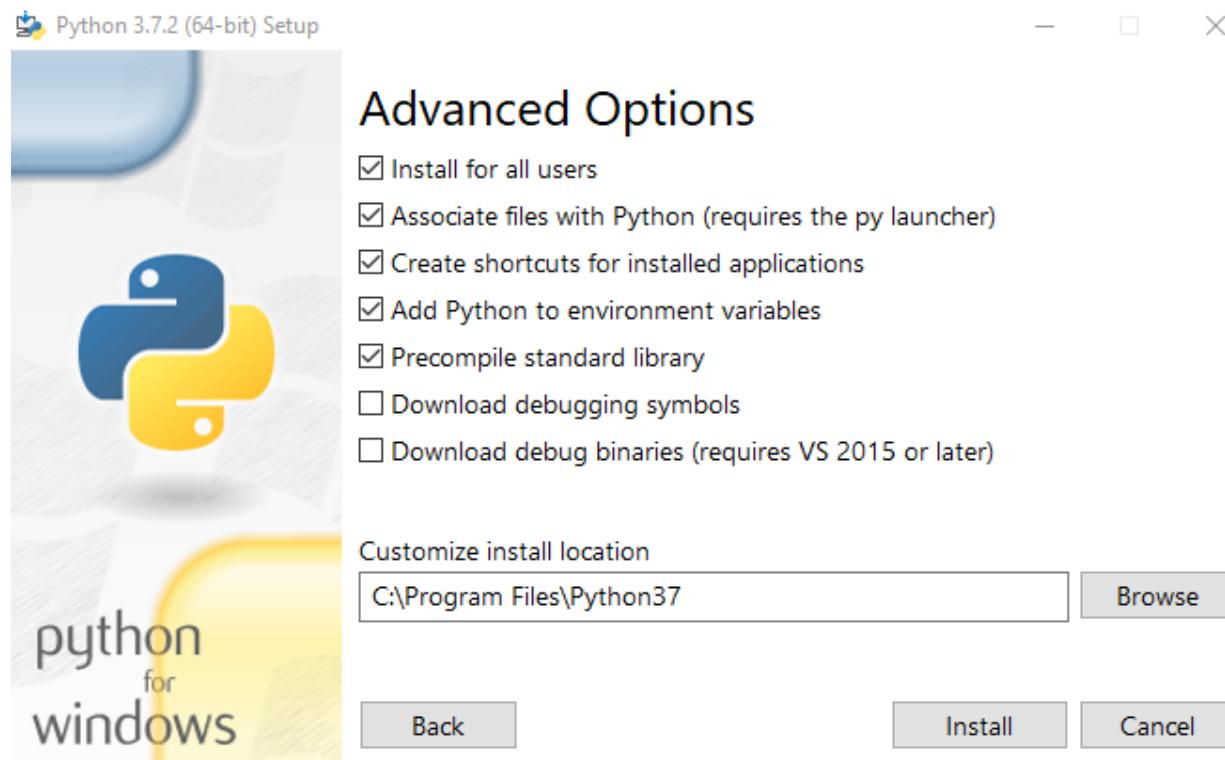
⁵⁶<http://www.python.org/download/>

Python interpreter launch. If this fails, check your PATH environment variable is set correctly by following [an online guide⁵⁷](#).

Once you are sure that Python is installed correctly, you need to install virtual environment support. Issue the following two commands, and then restart all open Command Prompt windows.

```
$ pip install virtualenv  
$ pip install virtualenvwrapper-win
```

Once completed, everything should be set up and ready for you to use.



Configuring Python 3.7.2 on Windows 10 x64 – allowing the installation to be run by all users.

Virtual Environments

By default, when you install software for Python, it is installed *system-wide*. All Python applications can see the new software and make use of it. However, issues

⁵⁷<https://www.architectryan.com/2018/03/17/add-to-the-path-on-windows-10/>

can occur with this setup. [Earlier in the book](#), we discussed a scenario of two pieces of software requiring two different versions of the *same dependency*. This presents a headache; you cannot typically install two different versions of the same software into your Python environment!

The solution to this is to use a *virtual environment*. Using a virtual environment, each different piece of software that you wish to run can be given its environment, and by definition, its set of installed dependencies. If you have ProjectA requiring Django 1.11 and ProjectB requiring Django 2.1, you could create a virtual environment for each with their packages installed.

The five basic commands one would use to manipulate virtual environments are listed below.

- `mkvirtualenv <name>` creates and activates a new virtual environment of name `<name>`.
- `workon <name>` switches on a virtual environment of name `<name>`.
- `deactivate` switches off a virtual environment you are currently using.
- `rmvirtualenv <name>` deletes a virtual environment of name `<name>`.
- `lsvirtualenv` lists all user-created virtual environments.

Following the examples above, we can then create an environment for each, installing the required software in the relevant environment. For ProjectA, the environment is called `projAENV`, with `projBENV` used for ProjectB. Note that to install software to the respective environments we use `pip`. The commands used for `pip` are [discussed below](#).

```
$ mkvirtualenv projAENV
(projAENV) $ pip install Django==1.11
(projAENV) $ pip freeze
Django==1.11

(projAENV) $ deactivate
$ pip
<Command not found>

$ workon projAENV
(projAENV) $ pip freeze
Django=1.11

(projAENV) $ cd ProjectA/
(projAENV) $ python manage.py runserver
...

(projAENV) $ deactivate
$
```

The code blocks above create the new virtual environment, projAENV. We then install Django 1.11 to that virtual environment, before issuing `pip freeze` to list the installed packages – confirming that Django was installed. We then deactivate the virtual environment. `pip` then cannot be found as we are no longer in the virtual environment! By switching the virtual environment back on with `workon`, our Django package can once again be found. The final two commands launch the Django development server for ProjectA.

We can then create a secondary virtual environment, projBENV.

```
$ mkvirtualenv projBENV
(projBENV) $ pip install Django==2.1
(projBENV) $ pip freeze
Django==2.1

(projBENV) $ cd ProjectA/
(projBENV) $ python manage.py runserver
<INCORRECT PYTHON VERSION!>

(projBENV) $ workon projAENV
(projAENV) $ python manage.py runserver

(projAENV) $ workon projBENV
(projBENV) $ cd ProjectB/
$ python manage.py runserver
...
```

We create our new environment with the `mkvirtualenv` command. This creates and activates `projBENV`. However, when trying to launch the code for `ProjectA`, we get an error! We are using the wrong virtual environment. By switching to `projAENV` with `workon projAENV`, we can then launch the software correctly. This demonstrates the power of virtual environments, and the advantages that they can bring. [Further tutorials can be found online.](#)⁵⁸



workon and deactivate

Start your session by switching on your virtual environment with the `workon` command. Finish your session by closing it with `deactivate`.

You can tell if a virtual environment is active by the brackets before your prompt, like `(envname) $`. This means that virtual environment `envname` is currently switched on, with its settings loaded. Turn it off with `deactivate` and the brackets will disappear.

⁵⁸<https://realpython.com/python-virtual-environments-a-primer/>



Multiple Python Versions

If you have multiple versions of Python installed, you can choose what version of Python to use when you create a virtual environment. If you have installations for `python` (which launches 2.7.15) and `python3` (which launches 3.7.2), you can issue the following command to create a Python 3 virtual environment.

```
$ mkvirtualenv -p python3 someenv  
$ python  
$ Python 3.7.2  
>>>
```

Note that when you enable your virtual environment, the command you enter to start Python is simply `python`. The same is applied for `pip` – if you launch `pip3` outside a virtual environment, `pip` will be the command you use inside the virtual environment.

Using pip

The Python package manager is very straightforward to use and allows you to keep track of the various Python packages (software) that you have installed. We highly recommend that you use `pip` alongside a virtual environment, as packages installed using `pip` appear only within the said virtual environment.

When you find a package that you want to install, the command required is `$ pip install <packagename>==<version>`. Note that the version component is optional; omitting the version of a particular package will mean that the latest available version is installed.

You can find the name of a package by examining the *PyPi package index*⁵⁹, from which `pip` downloads software. Simply search or browse the index to find what you are looking for. Once you know the package's name, you can issue the installation command in your Terminal or Command Prompt.

`pip` is also super useful for listing packages that are installed in your environment.

⁵⁹<https://pypi.org/>

This can be achieved through the `pip freeze` command. Sample output is issued below.

```
(rangoenv) $ pip freeze
Django==2.1.5
Pillow==5.4.1
pytz==2018.9
```

This shows that three packages are installed in a given environment: `Django`, `Pillow` and `pytz`. The output from this command is typically saved to a file called `requirements.txt`, stored in the root directory of a Python project. If somebody wants to use your software, they can then download your software – complete with the `requirements.txt` file. Using this file, they can then create a new virtual environment set up with the required packages to make the software work.

The recommended way to create your own `requirements.txt` file is to pipe the output of the `pip freeze` command to the file. Navigate to the directory where you want to create the file, and issue the following command.

```
$ cd awesome_python_project
$ pip freeze > requirements.txt
```

This then creates your `requirements.txt` file. You can then add this to your project's version control. A Python developer who sees a `requirements.txt` file should know exactly what to do with it!

If you find yourself in a situation like this, you run `pip` with the `-r` switch. Given a `requirements.txt` in a directory `downloaded_project` with only `pytz==2018.9` listed, an example CLI session would involve something like the following.

```
$ cd downloaded_project  
$ mkvirtualenv someenv  
(someenv) $ pip install -r requirements.txt  
...  
(someenv) $ pip freeze  
pytz==2018.9
```

pip install installs packages from requirements.txt, and pip freeze, once everything has been installed, demonstrates that the packages have been installed correctly.

Version Control System

When developing code, it's highly recommended that you house your codebase within a version-controlled repository such as [SVN⁶⁰](#) or [Git⁶¹](#). We have provided a [chapter on how to use Git](#) if you haven't used Git and GitHub before. We highly recommend that you set up a Git repository for your projects. Doing so could save you from disaster.

To use Git, we recommend that you use the command-line tool to interact with your repositories. This is done through the git command. On Windows, you'll need to [download Git from the Git website⁶²](#). If using macOS or Linux, the [Git website also has downloadable installers for you to use⁶³](#). However, why not get into the habit of using a package manager to install the software? This is generally the recommended way for downloading and using software developed on the UNIX design principles (including macOS).

For example, installing Git is as simple as typing `$ sudo apt install git`. Let the software download, and the apt package manager takes care of the rest. If you installed MacPorts on your macOS installation as described above, Git will already be present for you as it is part of the Apple XCode Command Line Developer Tools.

Once installed, typing git will show the commands you can use, as shown in the example below.

⁶⁰<http://subversion.tigris.org/>

⁶¹<http://git-scm.com/>

⁶²<https://git-scm.com/download/win>

⁶³<https://git-scm.com/downloads>

```
$ git
usage: git [--version] [--help] [-C <path>] [-c name=value]
[--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]
[-p | --paginate | --no-pager] [--no-replace-objects] [--bare]
[--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]
<command> [<args>]
```

A Crash Course in UNIX-based Commands

Depending on your computing background, you may or may not have encountered a UNIX-based system – or a derivative of. This small crash course focuses on getting you up to speed with the *terminal*, an application in which you issue commands for the computer to execute. This differs from a point-and-click *Graphical User Interface (GUI)*, the kind of interface that has made computing so much more accessible. A terminal-based interface may be more complex to use, but the benefits of using such an interface include getting things done quicker – and more accurately, too.



Not for Windows!

Note that we're focusing on the Bash shell, a shell for UNIX-based operating systems and their derivatives, including macOS and Linux distributions. If you're a Windows user, you can use the [Windows Command Prompt⁶⁴](#) or [Windows PowerShell⁶⁵](#). Users of Windows 10 can now install the necessary infrastructure [straight from Microsoft⁶⁶](#) to use Bash on your Windows installation! You could also experiment by [installing Cygwin⁶⁷](#) to bring Bash commands to Windows.

Using the Terminal

UNIX based operating systems and derivatives – such as macOS and Linux distributions – all use a similar-looking terminal application, typically using the [Bash shell⁶⁸](#). All possess a core set of commands that allow you to navigate through your computer's filesystem and launch programs – all without the need for any graphical interface.

⁶⁴<http://www.ai.uga.edu/mc/winforunix.html>

⁶⁵<https://msdn.microsoft.com/en-us/powershell/mt173057.aspx>

⁶⁶<https://docs.microsoft.com/en-us/windows/wsl/install-win10>

⁶⁷<https://www.cygwin.com/>

⁶⁸[https://en.wikipedia.org/wiki/Bash_\(Unix_shell\)](https://en.wikipedia.org/wiki/Bash_(Unix_shell))

Upon launching a new terminal instance, you'll be typically presented with something resembling the following.

```
sibu:~ david$
```

What you see is the *prompt*, and indicates when the system is waiting to execute your every command. The prompt you see varies depending on the operating system you are using, but all look generally very similar. In the example above, there are three key pieces of information to observe:

- your username and computer name (username of `david` and computer name of `sibu`);
- your *present working directory* (the tilde, or `~`); and
- the privilege of your user account (the dollar sign, or `$`).



What is a Directory?

In the text above, we refer to your present working directory. But what exactly is a *directory*? If you have used a Windows computer up until now, you'll probably know a directory as a *folder*. The concept of a folder is analogous to a directory – it is a cataloguing structure that contains references to other files and directories.

The dollar sign (`$`) typically indicates that the user is a standard user account. Conversely, a hash symbol (`#`) may be used to signify the user logged in has **root privileges**⁶⁹. Whatever symbol is present is used to signify that the computer is awaiting your input.



Prompts can Differ

The information presented by the prompt on your computer may differ from the example shown above. For example, some prompts may display the current date and time, or any other information. It all depends on how your computer is set up.

⁶⁹<http://en.wikipedia.org/wiki/Superuser>

When you are using the terminal, it is important to know where you are in the file system. To find out where you are, you can issue the command `pwd`. This will display your *Present Working Directory* (hence `pwd`). For example, check the example terminal interactions below.

```
Last login: Sun Jul 21 15:47:24 2019
sibu:~ david$ pwd
/users/grad/david
sibu:~ david$
```

You can see that the present working directory in this example is `/users/grad/david`.

You'll also note that the prompt indicates that the present working directory is a tilde `~`. The tilde is used as a special symbol which represents your *home directory*. The base directory in any UNIX based file system is the *root directory*. The path of the root directory is denoted by a single forward-slash (`/`). As folders (or directories) are separated in UNIX paths with a `/`, a single `/` denotes the root!

If you are not in your home directory, you can *Change Directory* (`cd`) by issuing the following command:

```
sibu:/ david$ cd ~
sibu:~ david$
```

Note how the present working directory switches from `/` to `~` upon issuing the `cd ~` command.



Path Shortcuts

UNIX shells have several different shorthand ways for you to move around your computer's filesystem. You've already seen that a forward slash (/) represents the [root directory](#)⁷⁰, and the tilde (~) represents your home directory in which you store all your files. However, there are a few more special characters you can use to move around your filesystem in conjunction with the cd command.

- Issuing cd ~ will always return you to your home directory. On some UNIX or UNIX derivatives, simply issuing cd will return you to your home directory, too.
- Issuing cd .. will move your present working directory **up one level** of the filesystem hierarchy. For example, if you are currently in /users/grad/david/code/, issuing cd .. will move you to /users/grad/david/.
- Issuing cd - will move you to the **previous directory you were working in**. Your shell remembers where you were, so if you were in /var/tmp/ and moved to /users/grad/david/, issuing cd - will move you straight back to /var/tmp/. This command only works if you've moved around at least once in a given terminal session.

Now, let's create a directory within the home directory called code. To do this, you can use the *Make Directory* command, called mkdir.

```
sibu:~ david$ mkdir code  
sibu:~ david$
```

There's no confirmation that the command succeeded. We can change the present working directory with the cd command to change to code. If this succeeds, we will know the directory has been successfully created.

⁷⁰https://en.wikipedia.org/wiki/Root_directory

```
sibu:~ david$ cd code  
sibu:code david$
```

Issuing another `pwd` command to confirm our present working directory returns the output `/users/grad/david/code`. This is our home directory with `code` appended to the end. You can also see from the prompt in the example above that the present working directory changes from `~` to `code`.



Change Back

Now issue the command to change back to your home directory. What command do you enter?

From your home directory, let's now try out another command to see what files and directories exist. This new command is called `ls`, shorthand for *list*. Issuing `ls` in your home directory will yield something similar to the following.

```
sibu:~ david$ ls  
code
```

This shows us that there is something present in our home directory called `code`, as we would expect. We can obtain more detailed information by adding a `-l` switch to the end of the `ls` command – with `l` standing for *list*.

```
sibu:~ david$ ls -l  
drwxr-xr-x  2 david  grad  68 21 Jul 15:00 code
```

This provides us with additional information, such as the modification date (2 Apr 11:07), whom the file belongs to (user `david` of group `grad`), the size of the entry (68 bytes), and the file permissions (`drwxr-xr-x`). While we don't go into file permissions here, the key thing to note is the `d` at the start of the string that denotes the entry is a directory. If we then add some files to our home directory and reissue the `ls -l` command, we then can observe differences in the way files are displayed as opposed to directories.

```
sibu:~ david$ ls -l
drwxr-xr-x  2 david  grad      68 21 Jul 15:00 code
-rw-r--r--@ 1 david  grad  303844  2 Apr 16:16 document.pdf
-rw-r--r--  1 david  grad      14  2 Apr 11:14 readme.md
```

One final useful switch to the `ls` command is the `a` switch, which displays *all* files and directories. This is useful because some directories and files can be *hidden* by the operating system to keep things looking tidy. Issuing the command yields more files and directories!

```
sibu:~ david$ ls -la
-rw-r--r--  1 david  grad     463 20 Feb 19:58 .profile
drwxr-xr-x  16 david  grad    544 25 Mar 11:39 .virtualenvs
drwxr-xr-x  2 david  grad      68 21 Jul 15:00 code
-rw-r--r--@ 1 david  grad  303844  1 Apr 16:16 document.pdf
-rw-r--r--  1 david  grad      14  2 Apr 11:14 readme.md
```

This command shows a hidden directory `.virtualenvs` and a hidden file `.profile`. Note that hidden files on a UNIX based computer (or derivative) start with a period (.). There's no special `hidden` file attribute you can apply, unlike on Windows computers.



Combining `ls` Switches

You may have noticed that we combined the `l` and `a` switches in the above `ls` example to force the command to output a list displaying all hidden files. This is a valid command – and there are [even more switches you can use⁷¹](#) to customise the output of `ls`.

Creating files is also easy to do, straight from the terminal. The `touch` command creates a new, blank file. If we wish to create a file called `new.txt`, issue `touch new.txt`. If we then list our directory, we then see the file added.

⁷¹<http://man7.org/linux/man-pages/man1/ls.1.html>

```
sibu:~ david$ ls -l
drwxr-xr-x  2 david  grad      68 21 Jul 15:00 code
-rw-r--r--@ 1 david  grad  303844  1 Apr 16:16 document.pdf
-rw-r--r--  1 david  grad       0  2 Apr 11:35 new.txt
-rw-r--r--  1 david  grad      14  2 Apr 11:14 readme.md
```

Note the filesize of `new.txt` – it is zero bytes, indicating an empty file. We can start editing the file using one of the many available text editors that are available for use directly from a terminal, such as `nano`⁷² or `vi`⁷³. While we don't cover how to use these editors here, you can [have a look online for a simple how-to tutorial](#)⁷⁴. We suggest starting with `nano` – while there are not as many features available compared to other editors, using `nano` is much simpler.

Core Commands

In the short tutorial above, you've covered a few of the core commands such as `pwd`, `ls` and `cd`. There are however a few more standard UNIX commands that you should familiarise yourself with before you start working for real. These are listed below for your reference, with most of them focusing upon file management. The list comes with an explanation of each and an example of how to use them.

- `pwd`: As explained previously, this command displays your *present working directory* to the terminal. The full path of where you are presently is displayed.
- `ls`: Displays a list of files in the present working directory to the terminal.
- `cd`: In conjunction with a path, `cd` allows you to change your present working directory. For example, the command `cd /users/grad/david/` changes the present working directory to `/users/grad/david/`. You can also move up a directory level without having to provide the [absolute path](#)⁷⁵ by using two dots, e.g. `cd ..`
- `cp`: Copies files and/or directories. You must provide the *source* and the *target*. For example, to make a copy of the file `input.py` in the same directory, you could issue the command `cp input.py input_backup.py`.

⁷²<http://www.nano-editor.org/>

⁷³<http://en.wikipedia.org/wiki/Vi>

⁷⁴<http://www.howtogeek.com/howto/42980/the-beginners-guide-to-nano-the-linux-command-line-text-editor/>

⁷⁵<http://www.coffeecup.com/help/articles/absolute-vs-relative-pathslinks/>

- `mv`: Moves files/directories. Like `cp`, you must provide the *source* and *target*. This command is also used to rename files. For example, to rename `numbers.txt` to `letters.txt`, issue the command `mv numbers.txt letters.txt`. To move a file to a different directory, you would supply either an absolute or relative path as part of the target – like `mv numbers.txt /home/david/numbers.txt`.
- `mkdir`: Creates a directory in your present working directory. You need to supply a name for the new directory after the `mkdir` command. For example, if your present working directory was `/home/david/` and you ran `mkdir music`, you would then have a directory `/home/david/music/`. You will need to then `cd` into the newly created directory to access it.
- `rm`: Shorthand for *remove*, this command removes or deletes files from your filesystem. You must supply the filename(s) you wish to remove. Upon issuing a `rm` command, you will be prompted if you wish to delete the file(s) selected. You can also remove directories [using the recursive switch⁷⁶](#). Be careful with this command – recovering deleted files is very difficult, if not impossible!
- `rmdir`: An alternative command to remove directories from your filesystem. Provide a directory that you wish to remove. Again, be careful: you will not be prompted to confirm your intentions.
- `sudo`: A program which allows you to run commands with the security privileges of another user. Typically, the program is used to run other programs as `root` – the [superuser⁷⁷](#) of any UNIX-based or UNIX-derived operating system.



There's More!

This is only a brief list of commands. Check out Ubuntu's documentation on [Using the Terminal⁷⁸](#) for a more detailed overview, or the [Cheat Sheet⁷⁹](#) by FOSSwire for a quick, handy reference guide. Like anything else, the more you practice, the more comfortable you will feel working with the terminal.

⁷⁶<http://www.computerhope.com/issues/ch000798.htm>

⁷⁷<http://en.wikipedia.org/wiki/Superuser>

⁷⁸<https://help.ubuntu.com/community/UsingTheTerminal>

⁷⁹<http://fosswire.com/post/2007/08/unixlinux-command-cheat-sheet/>

A Git Crash Course

We strongly recommend that you spend some time familiarising yourself with a **version control**⁸⁰ system for your application's codebase. This chapter provides you with a crash course in how to use **Git**⁸¹, one of the many version control systems available. Originally developed by **Linus Torvalds**⁸², Git is today **one of the most popular version control systems in use**⁸³, and is used by open-source and closed-source projects alike.

This tutorial demonstrates at a high level how Git works, explains the basic commands that you can use, and explains Git's workflow. By the end of this chapter, you'll be able to make contributions to a Git repository, enabling you to work solo, or in a team.

Why Use Version Control?

As your software engineering skills develop, you will find that you can plan and implement solutions to ever more complex problems. As a rule of thumb, the larger the problem specification, the more code you have to write. The more code you write, the greater the emphasis you should put on software engineering practices. Such practices include the use of design patterns and the ***DRY (Don't Repeat Yourself)***⁸⁴ principle.

Think about your experiences with programming thus far. Have you ever found yourself in any of these scenarios?

- Made a mistake to code, realised it was a mistake and wanted to go back?
- Lost code (through a faulty drive), or had a backup that was too old?

⁸⁰https://en.wikipedia.org/wiki/Version_control

⁸¹[http://en.wikipedia.org/wiki/Git_\(software\)](http://en.wikipedia.org/wiki/Git_(software))

⁸²http://en.wikipedia.org/wiki/Linus_Torvalds

⁸³https://en.wikipedia.org/wiki/Comparison_of_source_code_hosting_facilities#Popularity

⁸⁴https://en.wikipedia.org/wiki/Don%27t_repeat_yourself

- Had to maintain multiple versions of a product (perhaps for different organisations)?
- Wanted to see the difference between two (or more) versions of your codebase?
- Wanted to show that a particular change broke (or fixed) a piece of code?
- Wanted to submit a change (patch) to someone else's code?
- Wanted to see how much work is being done (where it was done, when it was done, or who did it)?

Using a version control system makes your life easier in *all* of the above cases. While using version control systems at the beginning may seem like a hassle it will pay off later – so it's good to get into the habit now!

We missed one final (and important) argument for using version control. With ever more complex problems to solve, your software projects will undoubtedly contain a large number of files containing source code. It'll also be likely that you *aren't working alone on the project; your project will probably have more than one contributor*. In this scenario, it can become difficult to avoid conflicts when working on files.

How Git Works

Essentially, Git is comprised of four separate storage locations: your **workspace**, the **local index**, the **local repository** and the **remote repository**. As the name may suggest, the remote repository is stored on some remote server and is the only location stored on a computer other than your own. This means that there are two copies of the repository – your local copy, and the remote copy. Having two copies is one of the main selling points of Git over other version control systems. You can make changes to your local repository when you may not have Internet access, and then apply any changes to the remote repository at a later stage. Only once changes are made to the remote repository can other contributors see your changes.



What is a Repository?

We keep repeating the word *repository*, but what do we mean by that? When considering version control, a repository is a data structure which contains metadata (a set of data that describes other data, hence *meta*) concerning the files which you are storing within the version control system. The kind of metadata that is stored can include aspects such as the historical changes that have taken place within a given file so that you have a record of all changes that take place.

If you want to learn more about the metadata stored by Git, there is a technical tutorial available⁸⁵ for you to read through.

For now, though, let's provide an overview of each of the different aspects of the Git system. We'll recap some of the things we've already mentioned just to make sure it makes sense to you.

- As already explained, the **remote repository** is the copy of your project's repository stored on some remote server. This is particularly important for Git projects that have more than one contributor – you require a central place to store all the work that your team members produce. You could set up a Git server on a computer with Internet access and a properly configured firewall (check out [this Server Fault question⁸⁶](#), for example), or simply use one of many services providing free Git repositories. One of the most widely used services available today is [GitHub⁸⁷](#). In fact this book has a Git **repository⁸⁸** on GitHub!
- The **local repository** is a copy of the remote repository stored on your computer (locally). This is the repository to which you make all your additions, changes and deletions. When you reach a particular milestone, you can then *push* all your local changes to the remote repository. From there, you can instruct your team members to retrieve your changes. This concept is known as *pulling* from the remote repository. We'll subsequently explain pushing and pulling in a bit more detail.

⁸⁵<http://www.sbf5.com/~cduan/technical/git/git-1.shtml>

⁸⁶<http://serverfault.com/questions/189070/what-firewall-ports-need-to-be-open-to-allow-access-to-external-git-repositories>

⁸⁷<https://github.com/>

⁸⁸https://github.com/maxwelld90/tango_with_django_2_code/

- The **local index** is technically part of the local repository. The local index stores a list of files that you want to be managed with version control. This is explained in more detail [later in this chapter](#). You can have a look [here⁸⁹](#) to see a discussion on what exactly a Git index contains.
- The final aspect of Git is your **workspace**. Think of this folder or directory as the place on your computer where you make changes to your version-controlled files. From within your workspace, you can add new files or modify or remove previously existing ones. From there, you then instruct Git to update the repositories to reflect the changes you make in your workspace. This is important – *don't modify the code inside the local repository – you only ever edit files in your workspace.*

Next, we'll be looking at how to [get your Git workspace set up and ready to go](#). We'll also [discuss the basic workflow](#) you should use when using Git.

Setting up Git

We assume that you've got Git installed with the software to go. One easy way to test the software out is to simply issue `git` to your terminal or Command Prompt. If you don't see a `command not found` error, you're good to go. Otherwise, have a look at [how to install Git to your system](#).

⁸⁹<http://stackoverflow.com/questions/4084921/what-does-the-git-index-exactly-contain>



Using Git on Windows

Like Python, Git doesn't come as part of a standard Windows installation. However, Windows implementations of the version control system can be downloaded and installed. You can download the official Windows Git client from the Git [website⁹⁰](#). The installer provides the `git` command line program, which we use in this crash course. You can also download a program called *TortoiseGit*, a graphical extension to the Windows Explorer shell. The program provides a nice right-click Git context menu for files. This makes version control easy to use. You can [download TortoiseGit⁹¹](#) for free. Although we do not cover how to use TortoiseGit in this crash course, many tutorials exist online for it. Check [this tutorial⁹²](#) if you are interested in using it.

We recommend however that you stick to the command line program. We'll be using the commands in this crash course. Furthermore, if you switch to a UNIX/Linux development environment at a later stage, you'll be glad you know the commands!

Setting up your Git workspace is a straightforward process. Once everything is set up, you will begin to make sense of the directory structure that Git uses. Assume that you have signed up for a new account on [GitHub⁹³](#) and [created a new repository on the service⁹⁴](#) for your project. With your remote repository setup, follow these steps to get your local repository and workspace set up on your computer. We'll assume you will be working from your <workspace> directory.

1. Open a terminal and navigate to your home directory (e.g. `$ cd ~`).
2. *Clone* the remote repository – or in other words, make a copy of it. Check out how to do this below.
3. Navigate into the newly-created directory. That's your workspace in which you can add files to be version controlled!

⁹⁰<http://git-scm.com/download/win>

⁹¹<https://code.google.com/p/tortoisegit/>

⁹²<http://robertgreiner.com/2010/02/getting-started-with-git-and-tortoisegit-on-windows/>

⁹³<https://github.com/>

⁹⁴<https://help.github.com/articles/create-a-repo>

How to Clone a Remote Repository

Cloning your repository is a straightforward process with the `git clone` command. Supplement this command with the URL of your remote repository – and if required, authentication details, too. The URL of your repository varies depending on the provider you use. If you are unsure of the URL to enter, it may be worth querying it with your search engine or asking someone in the know.

For GitHub, try the following command, replacing the parts below as appropriate:

```
$ git clone https://github.com/<OWNER>/<REPO_NAME> <workspace>
```

where you replace

- <OWNER> with the username of the person who owns the repository;
- <REPO_NAME> with the name of your project's repository; and
- <workspace> with the name for your workspace directory. This is optional; leaving this option out will simply create a directory with the same name as the repository.

If all is successful, you'll see some text like the example shown below.

```
$ git clone https://github.com/maxwelld90/tango_with_django_2_code/ twd2code
Cloning into 'twd2cc'...
remote: Enumerating objects: 354, done.
remote: Counting objects: 100% (354/354), done.
remote: Compressing objects: 100% (253/253), done.
remote: Total 354 (delta 217), reused 223 (delta 86), pack-reused 0
Receiving objects: 100% (354/354), 1.11 MiB | 1.78 MiB/s, done.
Resolving deltas: 100% (217/217), done.
```

If the output lines end with `done`, everything should have worked. Check your filesystem to see if the directory has been created by running `$ ls` – is the directory now listed?



Not using GitHub?

Many websites provide Git repositories – some free, some paid. While this chapter uses GitHub, you are free to use whatever service you wish. Other providers include [Atlassian Bitbucket](#)⁹⁵ and [Unfuddle](#)⁹⁶. You will have to change the URL from which you clone your repository if you use a service other than GitHub.

The Directory Structure

Once you have cloned your remote repository onto your local computer, navigate into the directory with your terminal, Command Prompt or GUI file browser. If you have cloned an empty repository the workspace directory should appear empty. This directory is your blank workspace with which you can begin to add your project's files.

However, the directory isn't blank at all! On closer inspection, you will notice a hidden directory called `.git`. Stored within this directory are both the local repository and local index. **Do not alter the contents of the `.git` directory.** Doing so could damage your Git setup and break version control functionality. *Your newly created workspace therefore actually contains within it the local repository and index.*

Final Tweaks

With your workspace setup, now would be a good time to make some final tweaks. Here, we discuss two useful features you can try which could make your life (and your team members') a little bit easier.

When using your Git repository as part of a team, any changes you make will be associated with the username you use to access your remote Git repository. However, you can also specify your full name and e-mail address to be included with changes that are made by you on the remote repository. Simply open a Command Prompt or terminal and navigate to your workspace. From there, issue two commands: one to tell Git your full name, and the other to tell Git your e-mail address.

⁹⁵<https://bitbucket.org/>

⁹⁶<https://unfuddle.com/>

```
$ git config user.name "Ivan Petrov"  
$ git config user.email "ivanpetrov123@me.bg"
```

Replace the example name and e-mail address with your own – unless your name is John Doe!

The `.gitignore` File

Git also provides you with the capability to stop – or ignore – particular files from being added to version control. For example, you may not wish a file containing unique keys to access web services from being added to version control. If the file were to be added to the remote repository, anyone could theoretically access the file by cloning the repository. With Git, files can be ignored by including them in the `.gitignore` file, which resides in the root of <workspace>. When adding files to version control, Git parses this file. If a file that is being added to version control is listed within `.gitignore`, the file is ignored. Each line of `.gitignore` should be a separate file entry.

Check out the following example of a `.gitignore` file:

```
.pyc  
.DS_Store  
__pycache__/  
db.sqlite3  
*.key  
thumbs.db
```

In this example `.gitignore` file, there are six entries – one on each line. We detail each of the entries below.

- The first entry prompts Git to ignore any file with an extension of `.pyc` – the wildcard `*` denoting *anything*. A `.pyc` file is a [bytecode representation of a Python module](#)⁹⁷, something that Python creates for modules to speed up execution. These can be safely ignored when committing to version control.

⁹⁷<https://stackoverflow.com/a/2998228>

- The second entry, `.DS_Store`, is a hidden file created by macOS systems. These files contain custom attributes that you set when browsing through your filesystem using the Finder. Custom attributes may, for example, represent the position of icons, or the view that you use to show your files in a given directory.
- The third entry represents any directory of the name `__pycache__`. This directory is new to Python 3, and is where the [bytecode representation⁹⁸](#) of your modules live. Again, these directories can be safely ignored.
- `db.sqlite3` is your database. Read the note below for a detailed explanation on why this should always be excluded from your repository.
- `*.key` is the fifth entry. This represents any file with the extension `key`, a convention we use where files with this extension contain sensitive information – like keys for API authentication. These should never be committed – doing so would be the equivalent of publicly sharing your password!
- The final entry is `thumbs.db`. Like the `.DS_Store` file created by macOS, this is the Windows equivalent.

Remember, wildcards are a neat feature to use here. If you ever have a type of file you do not wish to commit, then just use a `*.abc` to denote any filename, with the extension `.abc`.



.gitignore – What else should I ignore?

There are many kinds of files you could safely ignore from being committed and pushed to your Git repositories. Examples include temporary files, databases (that can easily be recreated) and operating system-specific files. Operating system-specific files include configurations for the appearance of the directory when viewed in a given file browser. Windows computers create `thumbs.db` files, while macOS creates `.DS_Store` files.

⁹⁸<https://stackoverflow.com/a/16869074>

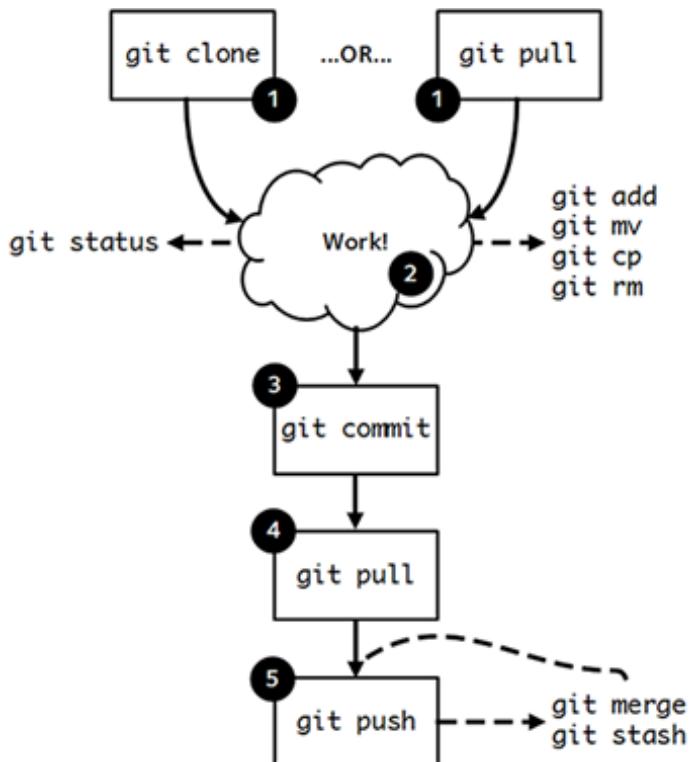


(Not) Committing your Database

You should also take steps to ignore other binary-based files that will frequently change within your repository. As an example in this book, you'll want to consider ignoring the database (as we demonstrate above). Why? When working in a team, everyone's copy of the database is likely going to be different. Different team members may add sample data that will differ from others. If all team members were to continually push up different versions of the database, the chance of creating conflicts increases. This issue is also the reason why we encourage the development of a [population script](#) to quickly fill a new database with sample data! The rule of thumb here is: don't commit a database to your project's repository. Do commit the necessary infrastructure to create the database from scratch, and as a bonus, provide the functionality to fill this blank database with sample data.

Basic Commands and Workflow

With your repository cloned and ready to go on your local computer, you're ready to get to grips with the Git workflow. This section shows you the basic Git workflow, and the associated Git commands you can issue.



A Figure of the Git workflow.

We have provided a representation of the basic Git workflow [as shown above](#). Match each of the numbers in the black circles to the numbered descriptions below to read more about each stage. **Refer to this diagram whenever you're unsure about the next step you should take – it's very useful!**

1. Starting Off

Before you can start work on your project, you must prepare Git. If you haven't yet sorted out your project's Git workspace, you'll need to [clone your repository to set it up](#).

If you've already cloned your repository, it's good practice to get into the habit of updating your local copy by using the `git pull` command. This *pulls* the latest changes from the remote repository onto your computer. By doing this, you'll be working with the most up-to-date version of the repository. This will reduce the possibility of conflicting versions of files, which really can make your life a bit of a nightmare.

To perform a `git pull`, first navigate to your repository directory within your Command Prompt or terminal, then issue the command `git pull`. Check out the snippet below from a Bash terminal to see exactly what you need to do, and what output you should expect to see.

```
$ cd somerepository/
$ git pull
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.
From https://github.com/someuser/somerepository
  86a0b3b..a7cec3d master      -> origin/master
Updating 86a0b3b..a7cec3d
Fast-forward
 README.md | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 README.md
```

This example shows that a `README.md` file has been updated or created from the latest pull.



Getting an Error?

If you receive `fatal: Not a git repository (or any of the parent directories): .git`, you're not in the correct directory. You need `cd` to your local repository directory – the one in which you cloned your repository to.

The majority of Git commands only work when you're in a Git repository.



Pull before you Push!

Always `git pull` on your local copy of your repository before you begin to work. **Always!** Before you are about to `push`, do another `pull` to ensure you have the latest changes.

Remember to talk to your team to coordinate your activity so you are not working on the same files, or use *branching*⁹⁹ to keep things separate. Branching will then at some point involve merging back to a single branch.

⁹⁹<https://git-scm.com/book/en/v2/Git-Branching-Basic-Branching-and-Merging>

2. Doing Some Work!

Once your workspace has been cloned or updated with the latest changes, it's time for you to get some work done! Within your workspace directory, you can take existing files and modify them. You can delete them too, or add new files to be version controlled.

When you modify your repository in any way, you need to keep Git up-to-date of any changes you make – this does not take place automatically. Telling Git what has changed allows you to keep your local index updated. The list of files stored within the local index is then used to perform your next *commit*, which we'll be discussing in the next step. To keep Git informed, several Git commands let you update the local index. Three of the commands are near identical to those that were discussed in the [Unix Crash Course](#) (e.g. `cp`, `mv`), save for the addition of a `git` prefix.

- The first command `git add` allows you to request Git to add a particular file to *staging* for the next commit. For any file that you wish to include in version control – whether it is new or updated, you must `git add` it. The command is invoked by typing `git add <filename>`, where `<filename>` is the name of the file you wish to add to your next commit. Multiple files and directories can be added with the command `git add .` – **but be careful with this¹⁰⁰**.
- `git mv` performs the same function as the Unix `mv` command – it moves files. The only difference between the two is that `git mv` updates the local index for you before moving the file. Specify the filename with the syntax `git mv <current_filename> <new_filename>`. For example, with this command, you can move files to a different directory within your repository. This will be reflected in your next commit. The command is also used to rename files – from the current filename to the new.
- `git cp` allows you to make a copy of a file or directory while adding references to the new files into the local index for you. The syntax is the same as `git mv` above where the filename or directory name is specified thus: `git cp <current_filename> <copied_filename>`.
- The command `git rm` adds a file or directory delete request into the local index. While the `git rm` command does not delete the file straight away, the requested

¹⁰⁰<http://stackoverflow.com/a/16969786>

file or directory is removed from your filesystem and the Git repository upon the next commit. The syntax is similar to the `git add` command, where a filename can be specified thus: `git rm <filename>`. Note that you can add a large number of requests to your local index in one go, rather than removing each file manually. For example, `git rm -rf media/` creates delete requests in your local index for the `media/` directory. The `r` switch enables Git to *recursively* remove each file within the `media/` directory, while `f` allows Git to *forcibly* remove the files. Check out the [Wikipedia page¹⁰¹](#) on the `rm` command for more information.

Lots of changes between commits can make things pretty confusing. You may easily forget what files you've already instructed Git to remove, for example. Fortunately, you can run the `git status` command to see a list of files which have been modified from your current working directory but haven't been added to the local index for processing. Have a look at the typical output from the command below to get a taste of what you can see.



Working with `.gitignore`

If you have [set up your `.gitignore` file correctly](#), you'll notice that files matching those specified within the `.gitignore` file is ignored when you `git add` them. This is the intended behaviour – these files are not supposed to be committed to version control! If you however really do need a file to be included that is in `.gitignore`, you can *force* Git to include it if necessary with the `git add -f <filename>` command.

¹⁰¹[http://en.wikipedia.org/wiki/Rm_\(Unix\)](http://en.wikipedia.org/wiki/Rm_(Unix))

```
On branch master
Your branch is up-to-date with 'origin/master'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   chapter-git.md
    modified:   chapter-system-setup.md

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

    images/screenshot_template.ai

no changes added to commit (use "git add" and/or "git commit -a")
```

Files `chapter-git.md`, `chapter-system-setup.md` and `images/screenshot_template.ai` are all listed in the example output shown above. All paths are relative from your present working directory, hence the addition of `images/` for the last file – it lives within an `images` subdirectory.

Reading the output of the `git status` command carefully, we can see that the files `chapter-git.md` and `chapter-system-setup.md` are already known to your Git repository, but have been *modified* from the previous commit. The output of the command is telling you that for these changes to be reflected in the next commit, you must `git add` these files. Files that have been deleted or moved concerning the previous commit will also appear here.

There is also a section for *untracked files*. These files have never been committed to version control – and as such will not show up on any Git commits. Files that you have just created will appear in this section. For them to be committed, you simply `git add` each file in turn. If you then run `git status` again, you will see them move up to the first section.



Checking Status

For further information on the `git status` command, check out the [official Git documentation¹⁰²](#).

¹⁰²<https://git-scm.com/docs/git-status>

3. Committing your Changes

We've mentioned *committing* several times in the previous step – but what does it mean? Committing is when you save changes – which are listed in the local index – that you have made within your workspace. The more often you commit, the greater the number of opportunities you'll have to revert to an older version of your code if things go wrong. It's like a checkpoint in a game – reach the checkpoint and you can save your status, perhaps giving you the ability to roll back in the future if you need to.

Make sure you commit often, **but don't commit an incomplete or broken version of a particular module or function. Your colleagues will hate you for it.** There's a lot of discussion as to when the ideal time to commit is. [Have a look at this Stack Overflow page¹⁰³](#) for the opinions of several developers. However, it does make sense to commit only when everything is working. To reiterate, put yourself in the shoes of a developer who has found their colleague has committed something that is broken. How would you feel?

To commit, you issue the `git commit` command. Any changes to existing files that you have indexed will be saved to version control at this point. Additionally, any files that you've requested to be copied, removed, moved or added to version control via the local index will be undertaken at this point. When you commit, you are updating the [`HEAD` of your local repository¹⁰⁴](#).



Commit Requirements

To successfully commit, you need to modify at least one file in your repository and instruct Git to commit it, through the `git add` command. See the previous step for more information on how to do this.

As part of a commit, it's incredibly useful to your future self and others to explain why you committed when you did. You can supply an optional message with your commit if you wish to do so. Instead of simply issuing `git commit`, run the following amended command.

¹⁰³<http://stackoverflow.com/questions/1480723/dvcs-how-often-and-when-to-commit-changes>

¹⁰⁴<http://stackoverflow.com/questions/2304087/what-is-git-head-exactly>

```
$ git commit -m "Chapter 5 exercises completed"
```

From the example above, you can see that using the `-m` switch followed by a string provides you with the opportunity to append a message to your commit. Be as explicit as you can, but don't write too much. People want to see at a glance what you did and do not want to be bored or confused with a long essay. At the same time, don't be too vague. Simply specifying `Updated models.py` may tell a developer what file you modified, but they will require further investigation to see exactly what you changed.



Sensible Commits

Although frequent commits may be a good thing, you will want to ensure that what you have written *works* before you commit. This may sound silly, but it's an incredibly easy thing to not think about. To reiterate (for the third time), **committing code which doesn't roll back work can be infuriating to your team members if they then rollback to a version of your project's codebase which is broken!**

4. Synchronising your Repository



Important when Collaborating

Synchronising your local repository before making changes is crucial to ensure you minimise the chance for conflicts occurring. Make sure you get into the habit of doing a `pull` before you `push`.

After you've committed your local repository and committed your changes, you're just about ready to send your commit(s) to the remote repository by *pushing* your changes. However, what if someone within your group pushes their changes before you do? This means your local repository will be out of sync with the remote repository, meaning that any `git push` command that you issue will fail.

It's therefore always a good idea to check whether changes have been made on the remote repository before updating it. Running a `git pull` command will pull down

any changes from the remote repository, and attempt to place them within your local repository. If no changes have been made, you're clear to push your changes. If changes have been made and cannot be easily rectified, you'll need to do a little bit more work.

In scenarios such as this, you have the option to *merge* changes from the remote repository. After running the `git pull` command, a text editor will appear in which you can add a comment explaining why the merge is necessary. Upon saving the text document, Git will merge the changes from the remote repository to your local repository. The simplest kind of merge is for a file that someone else has changed, but you haven't touched. This is to be expected – and a simple merge of this kind simply merges the other person's edits with your edits to produce a merged version of the repository.

The horrible kind of merge conflict happens when you and someone else work on the same file at the same time. To fix this, you'll need to carefully examine the file and figure out what changes stay, and what changes go. This can be very time consuming, so communication between team members is the key to avoid this scenario.



Editing Merge Logs

If you do see a text editor on your Mac or Linux installation, it's probably the `vi`¹⁰⁵ text editor. If you've never used vi before, check out [this helpful page containing a list of basic commands](#)¹⁰⁶ on the Colorado State University Computer Science Department website. If you don't like vi, [you can change the default text editor](#)¹⁰⁷ that Git calls upon. Windows installations most likely will bring up Notepad.

5. Pushing your Commit(s)

Pushing is the phrase used by Git to describe the sending of any changes in your local repository to the remote repository. This is how your changes become available to your other team members, who can then retrieve them by running the `git`

¹⁰⁵<http://en.wikipedia.org/wiki/Vi>

¹⁰⁶<http://www.cs.colostate.edu/helpdocs/vi.html>

¹⁰⁷<http://git-scm.com/book/en/Customizing-Git-Git-Configuration#Basic-Client-Configuration>

pull command in their respective local workspaces. The git push command isn't invoked as often as committing – *you require one or more commits to perform a push*. You could aim for one push per day, when a particular feature is completed, or at the request of a team member who is after your updated code.

To push your changes, the simplest command to run is:

```
$ git push origin master
```

As explained in [this Stack Overflow question and answer page¹⁰⁸](#) this command instructs the git push command to push your local master branch (where your changes are saved) to the *origin* (the remote server from which you originally cloned). If you are using a more complex setup involving [branching and merging¹⁰⁹](#), alter master to the name of the branch you wish to push.



Important Push?

If your git push is particularly important, you can also alert other team members to the fact they should update their local repositories by pulling your changes. You can do this through a *pull request*. Issue one after pushing your latest changes by invoking the command git request-pull master, where master is your branch name (this is the default value). If you are using a service such as GitHub, the web interface allows you to generate requests without the need to enter the command. Check out [the official GitHub website's tutorial¹¹⁰](#) for more information.

Recovering from Mistakes

This section presents a solution to a coder's worst nightmare: what if you find that your code no longer works? Perhaps a refactoring went wrong, or another team member without discussion changed something. Whatever the reason, using a form of version control always gives you a last resort: rolling back to a previous

¹⁰⁸<http://stackoverflow.com/questions/7311995/what-is-git-push-origin-master-help-with-gits-refs-heads-and-remotes>

¹⁰⁹<http://git-scm.com/book/en/Git-Branching-Basic-Branching-and-Merging>

¹¹⁰<https://help.github.com/articles/using-pull-requests>

commit. This section details how to do just that. We follow the information given from [this Stack Overflow¹¹¹](#) question and answer page.



Changes may be Lost!

You should be aware that this guide will rollback your workspace to a previous iteration. Any uncommitted changes that you have made will be lost, with a very slim chance of recovery! Be wary. If you are having a problem with only one file, you could always view the different versions of the files for comparison. Have a look [at this Stack Overflow page¹¹²](#) to see how to do that.

Rolling back your workspace to a previous commit involves two steps: determining which commit to rollback to, and performing the rollback. To determine what commit to rollback to, you can make use of the `git log` command. Issuing this command within your workspace directory will provide a list of recent commits that you made, your name and the date at which you made the commit. Additionally, the message that is stored with each commit is displayed. **This is why it is highly beneficial to supply commit messages that provide enough information to explain what you do at each commit!** Check out the following output from a `git log` invocation below to see for yourself.

```
commit 88f41317640a2b62c2c63ca8d755feb9f17cf16e          <- Commit hash
Author: John Doe <someaddress@domain.com>                <- Author
Date:   Mon Jul 8 19:56:21 2019 +0100                      <- Date/time
        Nearly finished initial version of the requirements chapter <- Message
commit f910b7d557bf09783b43647f02dd6519fa593b9f
Author: John Doe <someaddress@domain.com>
Date:   Wed Jul 3 11:35:01 2019 +0100
        Added in the Git figures to the requirements chapter.
commit c97bb329259ee392767b87cfe7750ce3712a8bdf
Author: John Doe <someaddress@domain.com>
Date:   Tue Jul 2 10:45:29 2019 +0100
        Added initial copy of Sphinx documentation and tutorial code.
commit 2952efa9a24dbf16a7f32679315473b66e3ae6ad
Author: John Doe <someaddress@domain.com>
Date:   Mon Jul 1 03:56:53 2019 -0700
        Initial commit
```

¹¹¹<http://stackoverflow.com/questions/2007662/rollback-to-an-old-commit-using-git>

¹¹²<http://stackoverflow.com/a/3338145>

From this list, you can choose a commit to rollback to. For the commit you want to rollback to, you must take the commit hash – the long string of letters and numbers. To demonstrate, the top (or `HEAD`) commit hash in the example output above is `88f41317640a2b62c2c63ca8d755feb9f17cf16e`. You can select this in your terminal and copy it to your computer's clipboard.

With your commit hash selected, you can now rollback your workspace to the previous revision. You can do this with the `git checkout` command. The following example command would rollback to the commit with the aforementioned hash.

```
$ git checkout 88f41317640a2b62c2c63ca8d755feb9f17cf16e .
```

Make sure that you run this command from the root of your workspace, and do not forget to include the dot at the end of the command! The dot indicates that you want to apply the changes to the entire workspace directory tree. After this has completed, you should then immediately commit with a message indicating that you performed a rollback. Push your changes and alert your collaborators – perhaps with a pull request. From there, you can start to recover from the mistake by putting your head down and getting on with your project.



Exercises

If you haven't undertaken what we've been discussing in this chapter already, you should go through everything now to ensure your Git repository is ready to go. To try everything out, you can create a new file `README.md` in the root of your `<workspace>` directory. The file [will be used by GitHub¹¹³](#) to provide information on your project's GitHub homepage.

- Create the file and write some introductory text to your project.
- Add the file to the local index upon completion of writing and commit your changes.
- Push the new file to the remote repository and observe the changes on the GitHub website.

Once you have completed these basic steps, you can then go back and edit the `readme` file some more. Add, commit and push – and then try to revert to the initial version to see if it all works as expected.

¹¹³<https://help.github.com/articles/github-flavored-markdown>



There's More!

There are other more advanced features of Git that we have not covered in this chapter. Examples include **branching** and **merging**, which are useful for projects with different release versions, for example. There are many fantastic tutorials available online if you are interested in taking your super-awesome version control skills a step further. For more details about such features take a look at this [tutorial on getting started with Git¹¹⁴](#), the [Git Guide¹¹⁵](#) or [Learning about Git Branching¹¹⁶](#).

However, if you're only using this chapter as a simple guide to getting to grips with Git, everything that we've covered should be enough. Good luck!

¹¹⁴<https://veerasundar.com/blog/2011/06/git-tutorial-getting-started/>

¹¹⁵<https://rogerdudler.github.io/git-guide/>

¹¹⁶<https://pcottle.github.io/learnGitBranching/>

A CSS Crash Course

In web development, we use *Cascading Style Sheets (CSS)* to describe the presentation of an HTML document (i.e. its look and feel). Each element within an HTML document can be *styled*. The CSS for a given HTML element describes how it is to be rendered on screen. This is done by ascribing *values* to the different *properties* associated with an element to change its appearance. In this chapter, we'll be introducing you to the basics of CSS – and how you can manipulate the elements that appear on your webpages. Consider the material in this chapter supplementary to the main book; we use the Twitter [Bootstrap CSS toolkit](#) to do most of the styling for us in an [earlier chapter](#).



CSS Properties

There are many, many different CSS properties that you can use in your stylesheets. Each provides different functionality. Check out the [W3C website¹¹⁷](#) for a huge list of available properties. [pageresource.com¹¹⁸](#) also has a neat list of properties, with descriptions of what each one does. Check out Section `css-course-reading-label` for a more comprehensive set of links.

CSS works by following a *select and apply pattern*. For a specified element, a set of styling properties are applied. Take a look at the following example in the [figure below](#), where we have some HTML containing `<h1>` tags. In the CSS code example, we specify that all `h1` be styled. We'll come back to [selectors¹¹⁹](#) later on in [this chapter](#). For now, though, you can assume the CSS style defined will be applied to our `<h1>` tags. The style contains four properties:

- `font-size`, setting the size of the font to `16pt`;
- `font-style`, which when set to `italic` italicises the contents of all `<h1>` tags within the document;

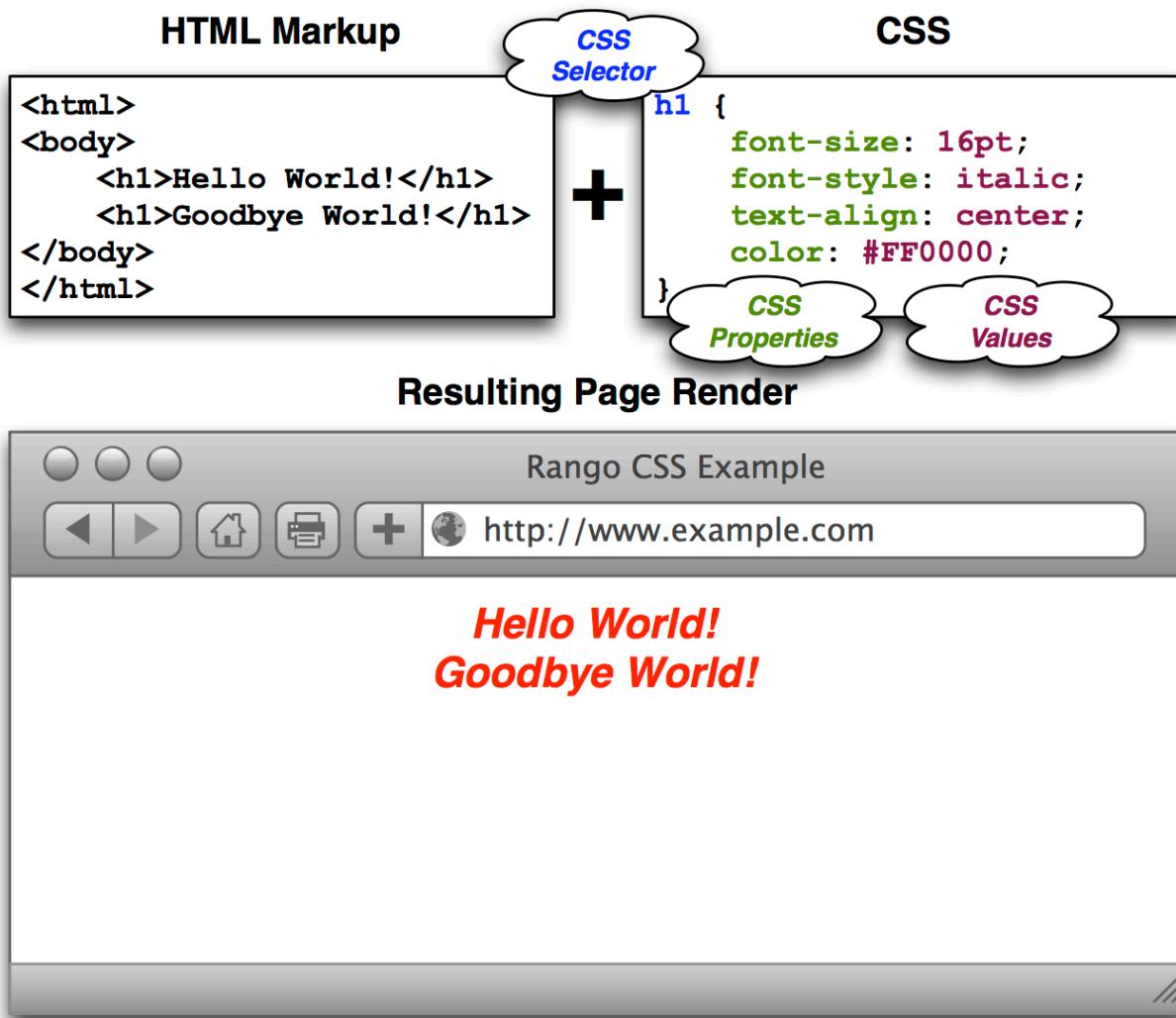
¹¹⁷<https://www.w3.org/Style/CSS/all-properties.en.html>

¹¹⁸<http://www.pageresource.com/dhtml/cssprops.htm>

¹¹⁹https://www.w3schools.com/cssref/css_selectors.asp

- `text-align` centres the text of the `<h1>` tags (when set to `center`); and
- `color`, which sets the colour of the text to red via [hexadecimal code¹²⁰](#) `#FF0000`.

With all of these properties applied, the resultant page render can be seen in the browser as shown in the figure below.



¹²⁰<https://htmlcolorcodes.com/>



What you see is what you (*may or may not*) get

Due to the nature of web development, *what you see isn't necessarily what you'll get*. This is because different browsers have their way of interpreting [web standards¹²¹](#) and so the pages may be rendered differently. Unfortunately, this quirk can lead to plenty of frustration, but today's modern browsers (or developers) are much more in agreement as to how different components of a page should be rendered compared to those of the past. Thank goodness!

One of the classic examples of the disagreement between browsers in terms of interpreting standards is Microsoft's *Internet Explorer*. If you want to know more, have a [look here¹²²](#). Thankfully, Microsoft's modern browser, *Edge*, is much more standards-compliant.

Including Stylesheets

Including stylesheets in your webpages is a relatively straightforward process, and involves including a `<link>` tag within your page's `<head>` tag. Check out the minimal HTML markup sample below for the attributes required within a `<link>` tag.

```
<!DOCTYPE html>
<html>
  <head>
    <link rel="stylesheet" type="text/css" href="URL/TO/stylesheet.css" />
    <title>Sample Title</title>
  </head>

  <body>
    <h1>Hello world!</h1>
  </body>
</html>
```

As can be seen from above, there are at minimum three attributes that you must supply to the `<link>` tag:

¹²¹http://en.wikipedia.org/wiki/Web_standards

¹²²https://www.456bereastreet.com/archive/200612/internet_explorer_and_the_css_box_model/

- `rel`, which allows you to specify the relationship between the HTML document and the resource you're linking to (i.e., a stylesheet);
- `type`, in which you should specify the [MIME type¹²³](#) for CSS; and
- `href`, the attribute which you should point to the URL of the stylesheet you wish to include.

With this tag added, your stylesheet should be included with your HTML page, and the styles within the stylesheet applied to elements within the page. You should be aware that CSS stylesheets are considered as a form of [static media](#), meaning you should place them within your project's static directory.



Inline CSS

You can also add CSS to your HTML documents *inline*, meaning that the CSS is included as part of your HTML page. However, this isn't generally advised because it removes the abstraction between presentational semantics (CSS) and content (HTML).

Basic CSS Selectors

CSS selectors are used to map particular styles to particular HTML elements. In essence, a CSS selector is a *pattern*. Here, we cover three basic forms of CSS selector: *element selectors*, *id selectors* and *class selectors*. [Later on in this chapter](#), we also touch on what are known as *pseudo-selectors*.

Element Selectors

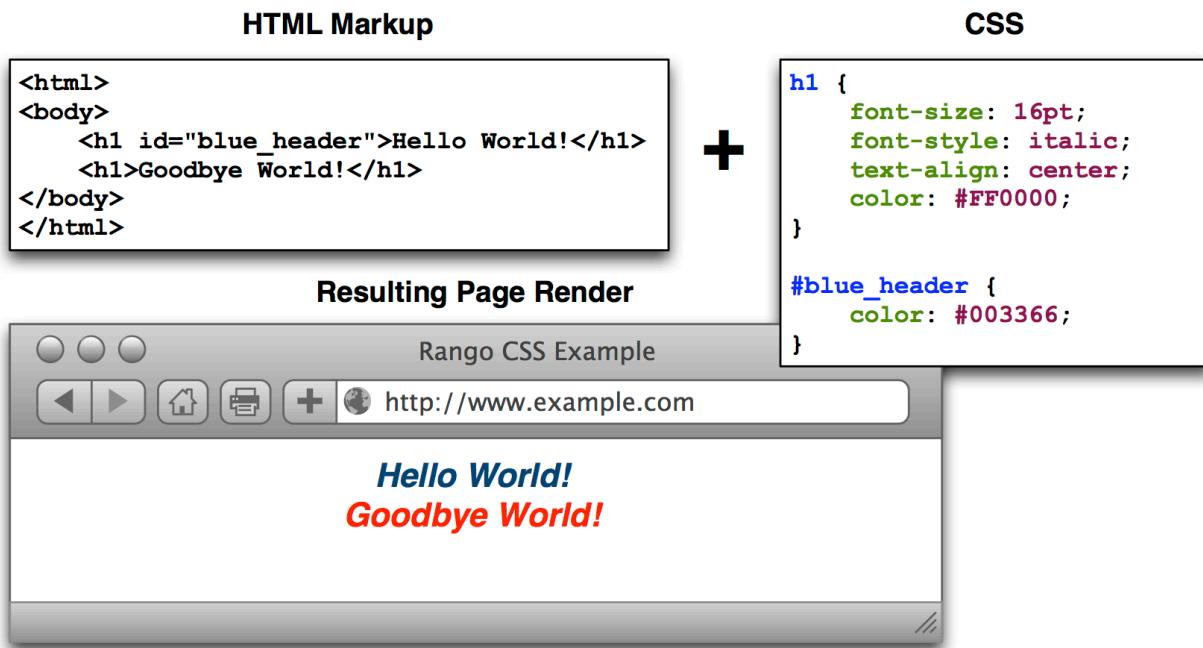
Taking the CSS example from the [rendering example shown above](#), we can see that the selector `h1` matches to any `<h1>` tag. Any selector referencing a tag like this can be called an *element selector*. We can apply element selectors to any HTML element such as `<body>`, `<h1>`, `<h2>`, `<h3>`, `<p>` and `<div>`. These can be all styled similarly. However, using element selectors is pretty crude – styles are applied

¹²³http://en.wikipedia.org/wiki/Internet_media_type

to *all* instances of a particular tag. We usually want a more fine-grained approach to selecting what elements we style, and this is where *id selectors* and *class selectors* come into play.

ID Selectors

The *id selector* is used to map to a unique element on your webpage. Each element on your webpage can be assigned a unique id via the `id` attribute, and it is this identifier that CSS uses to latch styles onto your element. This type of selector begins with a hash symbol (`#`), followed directly by the identifier of the element you wish to match to. Check out the figure below for an example.

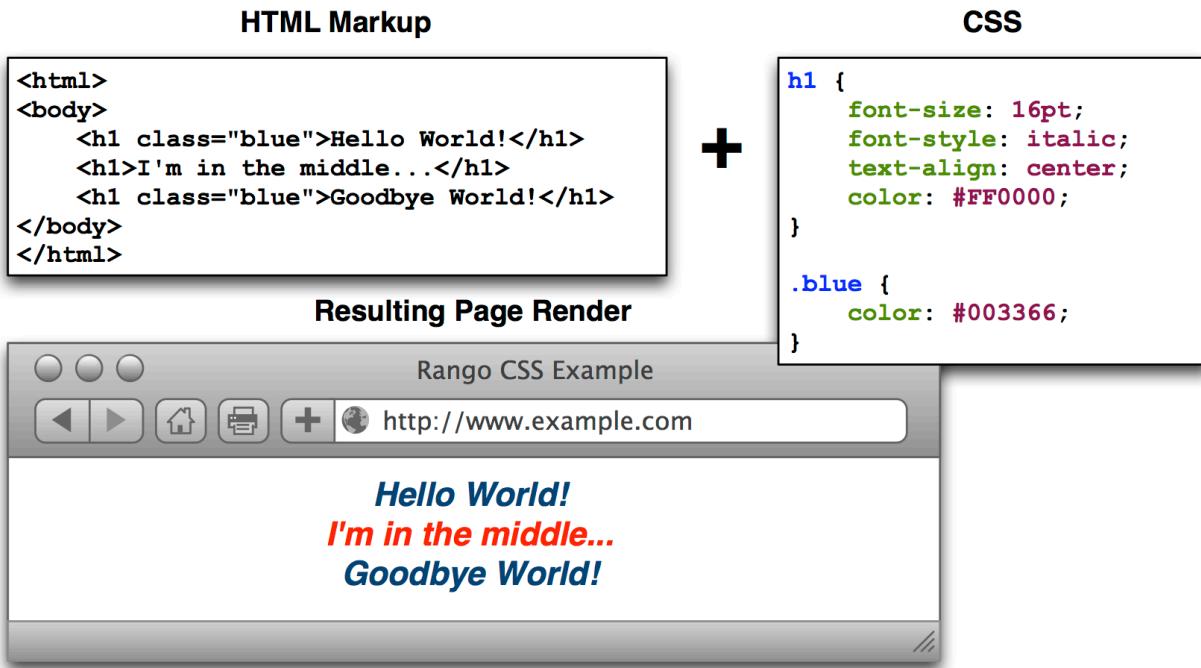


An illustration demonstrating the use of an *id selector* in CSS. Note the blue header has an identifier which matches the CSS attribute `#blue_header`.

Class Selectors

The alternative option is to use *class selectors*. This approach is similar to that of *id selectors*, with the difference that you can legitimately target multiple elements with the same class. If you have a group of HTML elements that you wish to apply the same style, use a class-based approach. The selector for using this method is

to precede the name of your class with a period (.) before opening up the style with curly braces ({}). Check out the [figure below](#) for an example.



An illustration demonstrating the use of a class selector in CSS. The blue headers employ the use of the .blue CSS style to override the red text of the h1 style.



Ensure ids are Unique

Try to use id selectors sparingly. [Ask yourself:](#)¹²⁴ *do I need to apply an identifier to this element in order to target it?* If you need to apply a given set of styles to more than one element, the answer will always be **no**. In cases like this, you should use a class or element selector.

Fonts

Due to the huge number available, using fonts has historically been a pitfall when it comes to web development. Picture this scenario: a web developer has installed and uses a particular font on his or her webpage. The font is pretty arcane – so the probability that the font is present on other computers is relatively small.

¹²⁴<http://net.tutsplus.com/tutorials/html-css-techniques/the-30-css-selectors-you-must-memorize/>

A user who visits the developer's webpage subsequently sees the page rendered incorrectly as the font is not present on their system. CSS tackles this particular issue with the `font-family` property.

The value you specify for `font-family` can be a *list* of possible fonts – and the first one your computer or other device has installed is the font that is used to render the webpage. Text within the specified HTML element subsequently has the selected font applied. The example CSS shown below applies *Arial* if the font exists. If it doesn't, it looks for *Helvetica*. If that font doesn't exist, any available `sans-serif` font¹²⁵ is applied.

```
h1 {  
    font-family: 'Arial', 'Helvetica', sans-serif;  
}
```

In 1996, Microsoft started the [core fonts for the web](#)¹²⁶ initiative to guarantee a particular set of fonts to be present on all computers. However, you can today use pretty much any font you like – check out [Google Fonts](#)¹²⁷ for examples of fonts that you can use, and [this W3Schools article](#)¹²⁸ on how to use such fonts. Always make sure you comply with licencing agreements when using fonts!

Colours and Backgrounds

Colours are important in defining the look and feel of your website. You can change the colour of any element within your webpage, ranging from background colours to borders and text. In this book, we make use of words and *hexadecimal colour codes* to choose the colours we want. As you can see from the list of basic colours shown in the [figure below](#), you can supply either a *hexadecimal* or *RGB (red-green-blue)* value for the colour you want to use. You can also [specify words to describe your colours](#)¹²⁹, such as green, yellow or blue.

¹²⁵<http://en.wikipedia.org/wiki/Sans-serif>

¹²⁶http://en.wikipedia.org/wiki/Core_fonts_for_the_Web

¹²⁷<http://www.google.com/fonts>

¹²⁸https://www.w3schools.com/howto/howto_google_fonts.asp

¹²⁹https://www.w3schools.com/colors/colors_names.asp



Pick Colours Sensibly

Take great care when picking colours to use on your webpages. If you select colours that don't contrast well, people simply won't be able to read your text! There are many websites available that can help you pick out a good colour scheme – try colorcombos.com¹³⁰ for starters.

Applying colours to your elements is a straightforward process. The property that you use depends on the aspect of the element you wish to change! The following subsections explain the relevant properties and how to apply them.

Black: #000000 or rgb(0, 0, 0)
Red: #FF0000 or rgb(255, 0, 0)
Green: #00FF00 or rgb(0, 255, 0)
Blue: #0000FF or rgb(0, 0, 255)
Yellow: #FFFF00 or rgb(255, 255, 0)
Cyan: #00FFFF or rgb(0, 255, 255)
Magenta: #FF00FF or rgb(255, 0, 255)
Grey: #C0C0C0 or rgb(192, 192, 192)
White: #FFFFFF or rgb(255, 255, 255)

Illustration of some basic colours with their corresponding hexadecimal and RGB values.

There are many different websites that you can use to aid you in picking the right hexadecimal codes to enter into your stylesheets. You aren't simply limited to the nine examples above! Try out html-color-codes.com¹³¹ for a simple grid of colours

¹³⁰<http://www.colorcombos.com/>

¹³¹<http://html-color-codes.com/>

and their associated six-character hexadecimal code. You can also try sites such as color-hex.com¹³² which gives you fine-grained control over the colours you can choose.



Hexadecimal Colour Codes

For more information on how colours are coded with hexadecimal, check out [this thorough tutorial](#)¹³³.



Watch your English!

As you may have noticed, CSS uses American English to spell words. As such, there are a few words that are spelt slightly differently compared to their British counterparts, like `color` and `center`. If you have grown up in the United Kingdom or Australia for instance, double-check your spelling and be prepared to spell it the *wrong way!*

Text Colours

To change the colour of text within an element, you must apply the `color` property to the element containing the text you wish to change. The following CSS, for example, changes all the text within an element using class `red` to red.

```
.red {  
    color: #FF0000;  
}
```

You can alter the presentation of a small portion of text within your webpage by wrapping the text within `` tags. Assign a class or unique identifier to the element, and from there you can simply reference the `` tag in your stylesheet while applying the `color` property. In the example markup below, the text `fire engine` would appear in red, using the `red` class defined above.

¹³²<http://www.color-hex.com/color-wheel/>

¹³³http://www.quackit.com/css/css_color_codes.cfm

```
<span>
  I saw a police car and <span class="red">fire engine</span> drive past me.
</span>
```

Borders

You can change the colour of an element's *borders*, too. We'll discuss what borders are discussed as part of the [CSS box model](#). For now, we'll show you how to apply colours to them to make everything look pretty.

Border colours can be specified with the `border-color` property. You can supply one colour for all four sides of your border, or specify a different colour for each side. To achieve this, you'll need to supply different colours, each separated by a space.

```
.some-element {
  border-color: #000000 #FF0000 #00FF00;
}
```

In the example above, we use multiple colours to specify a different colour for three sides. Starting at the top, we rotate clockwise. Thus, the order of colours for each side would be `top right bottom left`.

Our example applies any element with class `some-element` with a black top border, a red right border and a green bottom border. No left border value is supplied, meaning that the left-hand border is left transparent. To specify a colour for only one side of an element's border, consider using the `border-top-color`, `border-right-color`, `border-bottom-color` and `border-left-color` properties where appropriate.

Background Colours

You can also change the colour of an element's background through the use of the `CSS background-color` property. Like the `color` property described above, the `background-color` property can be easily applied by specifying a single colour as its value. Check out the example below which applies a bright green background to the entire webpage. Not very pretty!

```
body {  
    background-color: #00FF00;  
}
```

Background Images

Of course, specifying a colour isn't the only way to change your backgrounds. You can also apply background images to your elements, too. We can achieve this through the `background-image` property.

```
#some-unique-element {  
    background-image: url('../images/filename.png');  
    background-color: #000000;  
}
```

The example above makes use of `filename.png` as the background image for the element with the identifier `some-unique-element`. The path to your image is specified *relative to the path of your CSS stylesheet, not the webpage itself*. Our example above uses the [double dot notation to specify the relative path¹³⁴](#) to the image. We also apply a black background colour to fill the gaps left by our background image – it may not fill the entire size of the element.



Background Image Positioning

By default, background images default to the top-left corner of the relevant element and are repeated on both the horizontal and vertical axes. You can customise this functionality by altering [how the image is repeated¹³⁵](#) with the `background-image` property. You can also specify [where the image is placed¹³⁶](#) with the `background-position` property.

¹³⁴<http://programmers.stackexchange.com/a/186719>

¹³⁵https://www.w3schools.com/cssref/pr_background-repeat.asp

¹³⁶https://www.w3schools.com/cssref/pr_background-position.asp

Containers, Block-Level and Inline Elements

Throughout the crash course thus far, we've introduced you to the `` element but have neglected to tell you what it is. All will become clear in this section as we explain *inline* and *block-level* elements.

A `` is considered to be a so-called *container element*. Along with a `<div>` tag, these elements are themselves meaningless and are provided only for you to *contain* and *separate* your page's content in a logical manner. For example, you may use a `<div>` to contain markup related to a navigation bar, with another `<div>` to contain markup related to the footer of your webpage. Alternatively, you could use the `<nav>` and `<footer>` container elements! As containers themselves are meaningless, styles are usually applied to help control the presentational semantics of your webpage.

Containers come in two kinds: *block-level elements* and *inline elements*. Check out the [figure below](#) for an illustration of the two kinds in action, and read on for a short description of each.

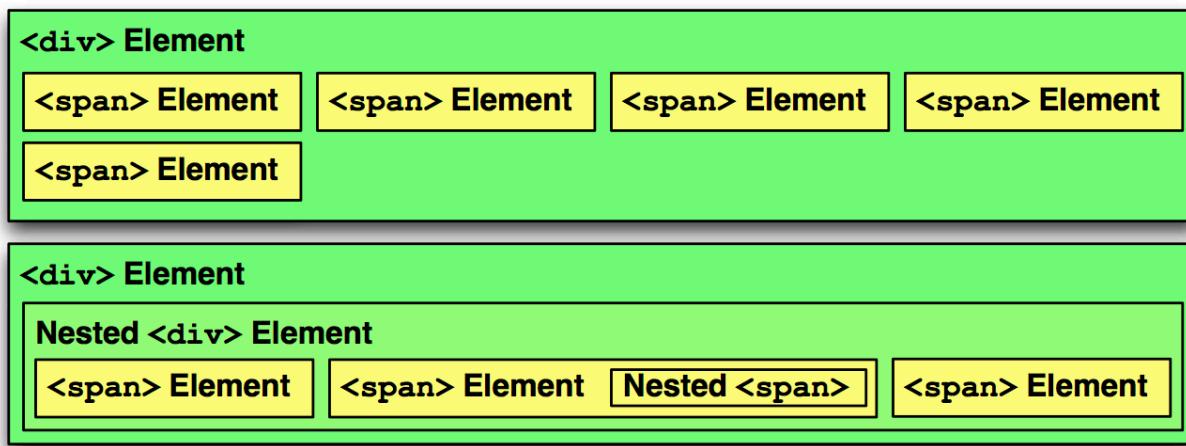


Diagram demonstrating how block-level elements and inline elements are rendered by default. With block-level elements as green, note how a line break is taken between each element. Conversely, inline elements can appear on the same line beside each other. You can also nest block-level and inline elements within each other, but block-level elements cannot be nested within an inline element.

Block-Level Elements

In simple terms, *block-level elements* are by default rectangular in shape and spread across the entire width of the containing element. Block-level elements therefore by default appear underneath each other. The rectangular structure of each block-level element is commonly referred to as the *box model*, which we discuss [later on in this chapter](#). A typical block-level element you will use is the `<div>` tag, short for *division*.

Block-level elements can be nested within other block-level elements to create a hierarchy of elements. You can also nest *inline elements* within block-level elements, but not vice-versa! Read on to find out why.

Inline Elements

An *inline element* does exactly what it says on the tin. These elements appear *inline* to block-level elements on your webpage and are commonly found to be wrapped around the text. You'll find that `` tags are commonly used for this purpose.

This text-wrapping application was explained in the [text colours section](#), where a portion of text could be wrapped in `` tags to change its colour. The corresponding HTML markup would look similar to the example below.

```
<div>
  This is some text wrapped within a block-level element.
  <span class="red">This text is wrapped within an inline element!</span>
  But this text isn't.
</div>
```

Refer back to the [nested blocks figure above](#) to refresh your mind about what you can and cannot nest before you move on.



The CSS `display` Property

The [display property of CSS¹³⁷](#) allows you to change the behaviour of a container. For example, `display: block;` can be set on a `` element to change it to `block` level. By default, it would behave as an `inline` element.

¹³⁷https://www.w3schools.com/cssref/pr_class_display.asp

Basic Positioning

An important concept that we have not yet covered in this CSS crash course regards the positioning of elements within your webpage. Most of the time, you'll be satisfied with inline elements appearing alongside each other, and block-level elements appearing underneath each other. These elements are said to be *positioned statically*.

However, there will be scenarios where you require a little bit more control on where everything goes. In this section, we'll briefly cover three important techniques for positioning elements on your webpage: *floats*, *relative positioning* and *absolute positioning*.

Floats

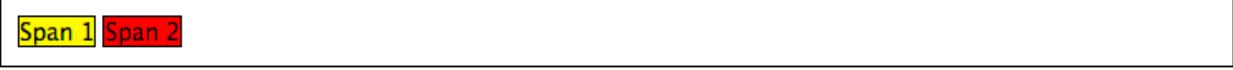
CSS *floats* are one of the most straightforward techniques for positioning elements within your webpage. Using floats allows us to position elements to the left or right of a particular container – or page.

Let's work through an example. Consider the following HTML markup and CSS code.

```
<div class="container">
  <span class="yellow">Span 1</span>
  <span class="red">Span 2</span>
</div>
```

```
.container {  
    border: 1px solid black;  
}  
  
.yellow {  
    background-color: yellow;  
    border: 1px solid black;  
}  
  
.red {  
    background-color: red;  
    border: 1px solid black;  
}
```

This produces the output shown below.



Span 1 Span 2

We can see that each element follows its natural flow: the container element with class `container` spans the entire width of its parent container, while each of the `` elements are enclosed inline within the parent. Now suppose that we wish to then move the red element with the text `Span 2` to the right of its container. We can achieve this by modifying our CSS `.red` class to look like the following example.

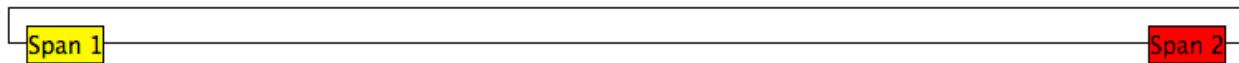
```
.red {  
    background-color: red;  
    border: 1px solid black;  
    float: right;  
}
```

By applying the `float: right;` property and value pairing, we should then see something similar to the example shown below.



Span 1 Span 2

Note how the `.red` element now appears at the right of its parent container, `.container`. We have in effect disturbed the natural flow of our webpage by artificially moving an element! What if we then also applied `float: left;` to the `.yellow `?



This would float the `.yellow` element, removing it from the natural flow of the webpage. In effect, it is now not sitting within the `.container` container. This explains why the parent container does not now fill down with the `` elements like you would expect. You can apply the `overflow: hidden;` property to the parent container as shown below to fix this problem. For more information on how this trick works, have a look at [this QuirksMode.org online article¹³⁸](#).

```
.container {  
    border: 1px solid black;  
    overflow: hidden;  
}
```



Applying `overflow: hidden` ensures that our `.container` pushes down to the appropriate height.

Relative Positioning

Relative positioning can be used if you require a greater degree of control over where elements are positioned on your webpage. As the name may suggest to you, relative positioning allows you to position an element *relative to where it would otherwise be located*. We make use of relative positioning with the `position: relative;` property and value pairing. However, that's only part of the story.

Let's explain how this works. Consider our previous example where two `` elements are sitting within their container.

¹³⁸<http://www.quirksmode.org/css/clearing.html>

```
<div class="container">
  <span class="yellow">Span 1</span>
  <span class="red">Span 2</span>
</div>
```

```
.container {
  border: 1px solid black;
  height: 200px;
}

.yellow {
  background-color: yellow;
  border: 1px solid black;
}

.red {
  background-color: red;
  border: 1px solid black;
}
```

This produces the following result – just as we would expect. Note that we have artificially increased the height of our container element to 200 pixels. This will allow us more room with which to play.



Now let's attempt to position our `.red` `` element relatively. First, we apply `position: relative` to our `.red` class, like so.

```
.red {  
    background-color: red;  
    border: 1px solid black;  
    position: relative;  
}
```

This does not affect the positioning of our `.red` element. What it does do however is change the positioning of `.red` from static to `relative`. This paves the way for us to specify where – from the original position of our element – we now wish the element to be located.

```
.red {  
    background-color: red;  
    border: 1px solid black;  
    position: relative;  
    left: 150px;  
    top: 80px;  
}
```

By applying the `left` and `top` properties as shown in the example above, we are wanting the `.red` element to be *pushed* 150 pixels *from the left*. In other words, we move the element 150 pixels to the right. Think about that carefully! The `top` property indicates that the element should be pushed 80 pixels from the *top* of the element. The result of our experimentation can be seen below.



From this behaviour, we can see that the properties `right` and `bottom` *push* elements from the right and bottom respectively. We can test this out by applying the properties to our `.yellow` class as shown below.

```
.yellow {  
    background-color: yellow;  
    border: 1px solid black;  
    float: right;  
    position: relative;  
    right: 10px;  
    bottom: 10px;  
}
```

This produces the following output. The `.yellow` container is pushed into the top left-hand corner of our container by pushing up and to the right.



Order Matters

What happens if you apply both a `top` and `bottom` property, or a `left` and `right` property? Interestingly, the *first* property for the relevant axis is applied. For example, if `bottom` is specified before `top`, the `bottom` property is used.

We can even apply relative positioning to elements that are floated. Consider our earlier example where the two `` elements were positioned on either side of the container by floating `.red` to the right.



We can then alter the `.red` class to the following.

```
.red {  
    background-color: red;  
    border: 1px solid black;  
    float: right;  
    position: relative;  
    right: 100px;  
}
```



Span 1

Span 2

This means that relative positioning works from the position at which the element would have otherwise been at – regardless of any other position changing properties being applied.

Absolute Positioning

Our final positioning technique is *absolute positioning*. While we still modify the position parameter of a style, we use absolute as the value instead of relative. In contrast to relative positioning, absolute positioning places an element *relative to its first parent element that has a position value other than static*. This may sound a little bit confusing, but let's go through it step by step to figure out what exactly happens.

First, we can again take our earlier example of the two coloured `` elements within a `<div>` container. The two `` elements are placed side-by-side as they would naturally.

```
<div class="container">  
    <span class="yellow">Span 1</span>  
    <span class="red">Span 2</span>  
</div>
```

```
.container {  
    border: 1px solid black;  
    height: 70px;  
}  
  
.yellow {  
    background-color: yellow;  
    border: 1px solid black;  
}  
  
.red {  
    background-color: red;  
    border: 1px solid black;  
}
```

This produces the output shown below. Note that we again set our `.container` height to an artificial value of 70 pixels to give us more room.



Span 1 Span 2

We now apply absolute positioning to our `.red` element.

```
.red {  
    background-color: red;  
    border: 1px solid black;  
    position: absolute;  
}
```

Like with relative positioning, this has no overall effect on the positioning of our red element in the webpage. We must apply one or more of `top`, `bottom`, `left` or `right` for a new position to take effect. As a demonstration, we can apply `top` and `left` properties to our red element like in the example below.

```
.red {  
    background-color: red;  
    border: 1px solid black;  
    position: absolute;  
    top: 0;  
    left: 0;  
}
```

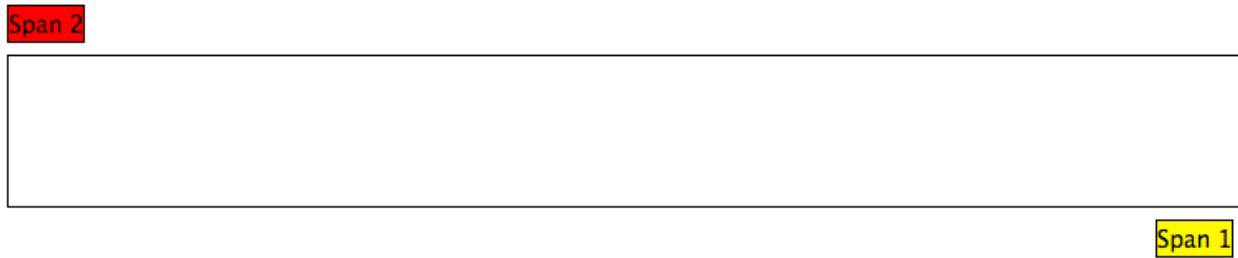


Wow, what happened here? Our red element is now positioned outside of our container! You'll note that if you run this code within your web browser window, the red element appears in the top left-hand corner of the viewport. This therefore means that when applying `top`, `bottom`, `left` and `right` properties to an element with absolute positioning, it is positioned absolutely **within the viewport** – with $(0,0)$ at the top left.

As our container element's position is by default set to `position: static`, the red and yellow elements are moving to the top left and bottom right of our screen respectively. Let's now modify our `.yellow` class to move the yellow `` to 5 pixels from the bottom right-hand corner of our page. The `.yellow` class now looks like the example below.

```
.yellow {  
    background-color: yellow;  
    border: 1px solid black;  
    position: absolute;  
    bottom: 5px;  
    right: 5px;  
}
```

This produces the following result.



But what if we don't want our elements to be positioned absolutely with respect to the entire page? More often than not, we'll be looking to adjusting the positioning of our elements with respect to a container. If we recall our definition for absolute positioning, we will note that absolute positions are calculated *relative to the first parent element that has a position value other than static*. As our container is the only parent for our two `` elements, the container to which the absolutely positioned elements is therefore the `<body>` of our HTML page. We can change this by adding `position: relative` to our `.container` class, just like in the example below.

```
.container {  
    border: 1px solid black;  
    height: 70px;  
    position: relative;  
}
```

This produces the following result. `.container` becomes the first parent element with a position value of anything other than `relative`, meaning our `` elements latch on!



Our elements are now absolutely positioned in relation to `.container`. Great! Now, let's adjust the positioning values of our two `` elements to move them around.

```
.yellow {  
    background-color: yellow;  
    border: 1px solid black;  
    position: absolute;  
    top: 20px;  
    right: 100px;  
}  
  
.red {  
    background-color: red;  
    border: 1px solid black;  
    position: absolute;  
    float: right;  
    bottom: 50px;  
    left: 40px;  
}
```



Note that we also apply `float: right` to our `.red` element. This is to demonstrate that unlike relative positioning, absolute positioning *ignores any other positioning properties applied to an element*. `top: 10px` for example will always ensure that an element appears 10 pixels down from its parent (set with `position: relative`), regardless of whether the element has been floated or not.

The Box Model

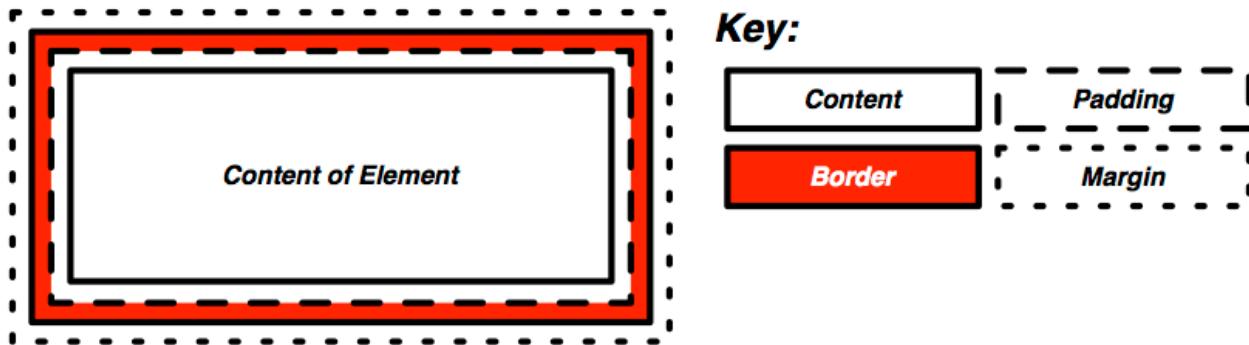
When using CSS, you're never too far away from using *padding*, *borders* and *margins*. These properties are some of the most fundamental styling techniques which you can apply to the elements within your webpages. They are incredibly important and are all related to what we call the *CSS box model*.

Think of each element that you create on a webpage as having a surrounding box. The **CSS box model**¹³⁹ is defined by the **W3C**¹⁴⁰ as a formal means of describing

¹³⁹<http://www.w3.org/TR/CSS2/box.html>

¹⁴⁰<http://www.w3.org/>

the elements or boxes that you create, and how they are rendered in your web browser's viewport. Each element or box consists of *four separate areas*, all of which are illustrated in the [figure below](#). The surrounding box of an element can be split into three divisions, and these are (from inside to outside): the *padding area*, the *border area* and the *margin area*. The actual element itself (whether it be text or an image sits in the middle as the *content area*).



An illustration demonstrating the CSS box model, complete with legend showing the four areas of the model.

For each element within a webpage, you can create a margin, apply some padding or a border with the respective properties `margin`, `padding` and `border`. Margins clear a transparent area around the border of your element; meaning margins are incredibly useful for creating a gap between elements. In contrast, padding creates a gap between the content of an element and its border. This gives the impression that the element appears wider. If you supply a background colour for an element, the background colour is extended with the element's padding. Finally, borders are what you might expect them to be – they provide a border around your element's content and padding.

For more information on the CSS box model, check out Mozilla's excellent explanation of the model¹⁴¹. Why not even order a t-shirt with the box model on it¹⁴²?

¹⁴¹https://developer.mozilla.org/en-US/docs/Learn/CSS/Introduction_to_CSS/Box_model

¹⁴²<http://cssboxmodel.com/>



Watch out for the width!

As you may gather from the [box model illustration](#), the width of an element isn't defined simply by the value you enter as the element's `width`. Rather, you should always consider the width of the border and padding on both sides of your element. This can be represented mathematically as:

```
total_width = content_width + left padding + right padding + left border  
+ left margin + right margin
```

Don't forget this. You'll save yourself a lot of trouble if you don't!



Borders and Padding

If you find yourself working with elements that have a border and/or padding, working out precise widths can be hard. In a scenario where the widths are crucial, you might want to explore the [box-sizing property¹⁴³](#) to better control widths and heights. Read the linked W3CSchools article for more information. This property has saved us on numerous occasions!

Styling Lists

Lists are everywhere. Whether you're reading a list of learning outcomes for a course, or reading a list of times for the train, you know what a list looks like and appreciate its simplicity. If you have a list of items on a webpage, why not use an HTML list? Using lists within your webpages promotes good HTML document structure, allowing text-based browsers, screen readers and other browsers that do not support CSS to render your page sensibly.

Lists however by default do not look particularly appealing to end-users. Take the following HTML list that we'll be styling as we go along trying out different things.

¹⁴³https://www.w3schools.com/css/css3_box-sizing.asp

```
<ul class="sample-list">
  <li>Django</li>
  <li>How to Tango with Django</li>
  <li>Two Scoops of Django</li>
</ul>
```

Rendered without styling, the list looks pretty boring.

- Django
- How to Tango with Django
- Two Scoops of Django

Let's make some modifications. First, let's get rid of the ugly bullet points. With our `` element already (and conveniently) set with class `sample-list`, we can create the following style.

```
.sample-list {
  list-style-type: none;
}
```

This produces the following result. Note the lack of bullet points!

```
Django
How to Tango with Django
Two Scoops of Django
```

Let's now change the orientation of our list. We can do this by altering the `display` property of each of our list's elements (``). The following style maps to this for us.

```
.sample-list li {
  display: inline;
}
```

When applied, our list elements now appear on a single line, just like in the example below.

Django How to Tango with Django Two Scoops of Django

While we may have the correct orientation, our list now looks awful. Where does one element start and the other end? It's a complete mess! Let's adjust our list element style and add some contrast and padding to make things look nicer.

```
.example-list li {  
    display: inline;  
    background-color: #333333;  
    color: #FFFFFF;  
    padding: 10px;  
}
```

When applied, our list looks so much better – and quite professional, too!



Django How to Tango with Django Two Scoops of Django

From the example, it is hopefully clear that lists can be easily customised to suit the requirements of your webpages. For more information and inspiration on how to style lists, you can check out some of the selected links below.

- Have a look at [this excellent tutorial on styling lists on A List Apart¹⁴⁴](https://alistapart.com/article/taminglists/).
- Check out [this advanced tutorial from Web Designer Wall¹⁴⁵](https://webdesignerwall.com/tutorials/advanced-css-menu) that uses graphics to make awesome looking lists. In the tutorial, the author uses Photoshop – you could try using a simpler graphics package if you don't feel confident with Photoshop.
- [This site¹⁴⁶](https://learn.shayhowe.com/html-css/creating-lists/) provides some great inspiration and tips on how you can style lists.

The possibilities of styling lists are endless! You could say it's a never-ending list...

¹⁴⁴<https://alistapart.com/article/taminglists/>

¹⁴⁵<https://webdesignerwall.com/tutorials/advanced-css-menu>

¹⁴⁶<https://learn.shayhowe.com/html-css/creating-lists/>

Styling Links

CSS provides you with the ability to easily style hyperlinks in any way you wish. You can change their colour, their font or any other aspect that you wish – and you can even change how they look when you hover over them!

Hyperlinks are represented within an HTML page through the `<a>` tag, which is short for *anchor*. We can apply styling to all hyperlinks within your webpage as shown in the following example.

```
a {  
    color: red;  
    text-decoration: none;  
}
```

Every hyperlink's text colour is changed to red, with the default underline of the text removed. If we then want to change the `color` and `text-decoration` properties again when a user hovers over a link, we can create another style using the so-called [pseudo-selector](#)¹⁴⁷ `:hover`. Our two styles now look like the example below.

```
a {  
    color: red;  
    text-decoration: none;  
}  
  
a:hover {  
    color: blue;  
    text-decoration: underline;  
}
```

This produces links as shown below. Notice the change in colour of the second link – it is being hovered over.

[Django](#) [How to Tango with Django](#) [Two Scoops of Django](#)

However, you may not wish for the same link styles across the entire webpage. For example, your navigation bar may have a dark background while the rest of your

¹⁴⁷<https://css-tricks.com/pseudo-class-selectors/>

page has a light background. This would necessitate having different link stylings for the two areas of your webpage. The example below demonstrates how you can apply different link styles by using a slightly more complex CSS style selector.

```
#dark {  
    background-color: black;  
}  
  
.light {  
    background-color: white;  
}  
  
.dark a {  
    color: white;  
    text-decoration: underline;  
}  
  
.dark a:hover {  
    color: aqua;  
}  
  
.light a {  
    color: black;  
    text-decoration: none;  
}  
  
.light a:hover {  
    color: olive;  
    text-decoration: underline;  
}
```

We can then construct some simple markup to demonstrate these classes.

```
<div id="dark">  
  <a href="http://www.google.co.uk/">Google Search</a>  
</div>  
  
<div class="light">  
  <a href="http://www.bing.co.uk/">Bing Search</a>  
</div>
```

The resultant output looks similar to the example shown below. Code up the example above, and hover over the links in your browser to see the text colours change!



With a small amount of CSS, you can make some big changes in the way your webpages appear to users.

The Cascade

It's worth pointing out where the *Cascading* in *Cascading Style Sheets* comes into play. Looking back at the CSS rendering example way back at the start of this chapter, you will notice that the red text shown is **bold**, yet no such property is defined in our `h1` style. This is a perfect example of what we mean by *cascading styles*. Most HTML elements have associated with them a *default style* which web browsers apply. For `<h1>` elements, the W3C website provides a typical style that is applied¹⁴⁸. If you check the typical style, you'll notice that it contains a `font-weight: bold;` property and value pairing, explaining where the **bold** text comes from. As we define a further style for `<h1>` elements, typical property/value pairings *cascade* down into our style. If we define a new value for an existing property/value pairing (such as we do for `font-size`), we *override* the existing value. This process can be repeated many times – and the property/value pairings at the end of the process are applied to the relevant element. Check out the figure below for a graphical representation of the cascading process.

¹⁴⁸<http://w3c.github.io/html-reference/h1.html>

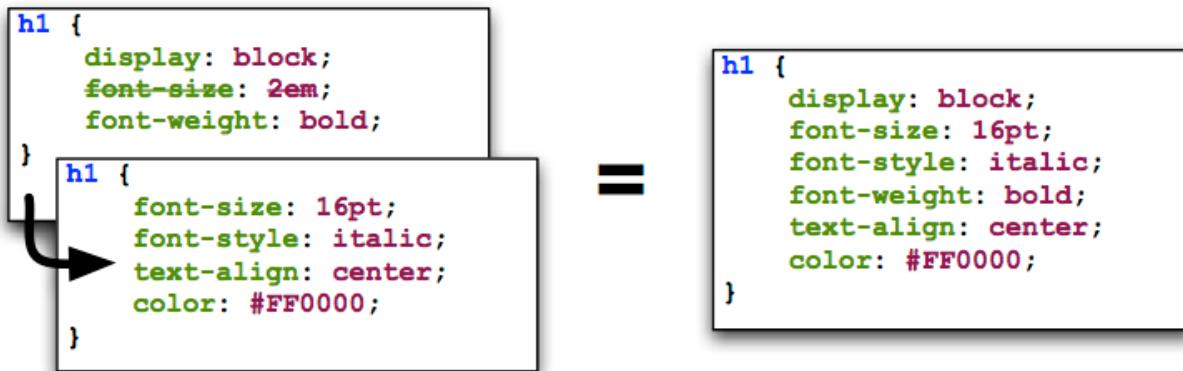


Illustration demonstrating the cascading in Cascading Style Sheets at work. Take note of the `font-size` property in our `h1` style – it is overridden from the default value. The cascading styles produce the resultant style, shown on the right of the illustration.

Additional Reading

What we've discussed in this section is by no means a definitive guide to CSS. There are [books devoted to CSS alone that span over 300 pages!](#)¹⁴⁹ devoted to CSS alone! What we have provided you with here is a very brief introduction showing you the very basics of what CSS is and how you can use it.

As you develop your web applications, you'll undoubtedly run into issues and frustrating problems with styling web content. This is part of the learning experience, and you still have a bit to learn. We strongly recommend that you invest some time trying out several online tutorials about CSS. There are so many resources freely available online, that there isn't any need to buy a book (unless you want to).

- The [W3C provides a neat tutorial on CSS](#)¹⁵⁰, taking you by the hand and guiding you through the different stages required. They also introduce you to several new HTML elements along the way and show you how to style them accordingly.
- [W3Schools also provides some cool CSS tutorials](#)¹⁵¹. Instead of guiding you through the process of creating a webpage with CSS, *W3Schools* has a series of mini-tutorials and code examples to show you to achieve a particular feature,

¹⁴⁹<https://www.amazon.co.uk/CSS-Definitive-Guide-Visual-Presentation/dp/1449393195/>

¹⁵⁰<https://www.w3.org/Style/Examples/011/firstcss.en.html>

¹⁵¹https://www.w3schools.com/css/css_examples.asp

such as setting a background image. We highly recommend that you have a look here.

This list is by no means exhaustive, and a quick web search will indeed yield much more about CSS for you to chew on. Just remember: CSS can be tricky to learn, and there may be times where you feel you want to throw your computer through the window. We say this is pretty normal – but take a break if you get to that stage.



CSS And Browser Compatibility

With an increasing array of devices equipped with more and more powerful processors, we can make our web-based content do more. To keep up, [CSS has constantly evolved¹⁵²](#) to provide new and intuitive ways to express the presentational semantics of our SGML-based markup. To this end, support for relatively new CSS properties¹⁵³ may be limited on several browsers, which can be a source of frustration. The only way to reliably ensure that your website works across a wide range of different browsers and platforms is to [test, test and test some more!](#)¹⁵⁴

¹⁵²<https://blogs.adobe.com/creativecloud/the-evolution-of-css/>

¹⁵³<https://www.quackit.com/css/css3/properties/>

¹⁵⁴<http://browsershots.org/>