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

Available from **www.tangowithdjango.com**

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 2019-08-21

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

Overview

This bookaims toprovideyouwith a practical guideto webdevelopment using*Django2* 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.

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).

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.

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 control2). 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

1https://docs.djangoproject.com/en/2.1/intro/tutorial01/ 2https://en.wikipedia.org/wiki/Inversion_of_control

have provided. If you still do not understand, then check out the Django documentation, go to Stack Overflow3 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!

What you will Learn

In this book, we will be taking an example-based approach to web application development. In the process,wewillshowyouhowtodesigna webapplicationcalled*Rango*(seetheDesignBriefbelow), 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 yourdevelopment environment** – including how to use the terminal, your virtual environment, the `pip` installer, how to work with Git, and more.
• How to **set up a Django project** and **create a basic Django application**.
• How to **configure the Django project** to serve static media and other media files.
• 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.
• 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 these exercises as the subsequent chapters are dependent on them.

3http://stackoverflow.com/questions/tagged/django 4https://en.wikipedia.org/wiki/Object-relational_mapping

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.

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;

• the Git9 version control system;
• GitHub10;
• HTML11;
• CSS12;
• the JavaScript13 programming language;
• the JQuery14 library;
• the Twitter Bootstrap15 framework;
• the Bing Search API16; and
• the PythonAnywhere17 hosting service;

We've selected 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!

5https://github.com/maxwelld90/tango_with_django_2_code 6https://www.python.org 7https://pip.pypa.io/en/stable/

8https://www.djangoproject.com 9https://git-scm.com 10https://github.com 11https://www.w3.org/html/

12https://www.w3.org/Style/CSS/ 13https://www.javascript.com/ 14https://jquery.com 15https://getbootstrap.com/

16https://docs.microsoft.com/en-gb/rest/api/cognitiveservices/bing-web-api-v7-reference 17https://www.pythonanywhere.com

Rango: Initial Design and Specification

The focus of thisbook willbe to develop anapplicationcalled*Rango*. As wedevelop 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 website18.

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 rango19 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/`.
• 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.

18http://www.tangowithdjango.com/ 19https://www.vocabulary.com/dictionary/es/rango

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 Architec- ture** 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)***20 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.

20https://en.wikipedia.org/wiki/Entityâ€“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 goingintoeachbox. Since we arebuilding a web applicationwith 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 willbe 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 toprovide clients withsome idea of whatthe application is goingto look like andwhatfeaturesitwillprovide.Theysavealotoftimeandcanvaryfromhand-drawnsketches 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 reach the 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`.

Aswe buildourapplication,wewillprobablyneed tocreateotherURLmappings.However,theones listedabovewillgetusstartedandgiveusanideaof thedifferentpages.Also,asweprogressthrough the book, we will flesh out how to construct these pages using the Django framework and use its Model-View-Template21 design pattern. However, now that we have a gist of the URL mappings and

21https://docs.djangoproject.com/en/2.1/

what the pages are going to look like, we need to define the data model that will house the data for our web application.

Entity-Relationship Diagram

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 house 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 relationship22. 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.

Take Note!

Getinto thehabitofnoting down anyworking assumptions thatyou make,justlike 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 Database chapter.

Category Model

22https://en.wikipedia.org/wiki/Many-to-many_(data_model)

Field Type `name CharField views IntegerField likes IntegerField`

Page Model

Field Type `category ForeignKey title CharField url URLField views IntegerField`

We willalso 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.

Summary

These high-level design and specifications will serve as a useful reference point when building our webapplication.Whilewe willbe focusingonusingspecifictechnologies,thesesteps arecommonto 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 `$`, we have entered the command `exit`, which, when executed, will close the terminal. Refer to the UNIX chapter for more information.
Whenever you see `>>>`, the following is a command that should be entered into the interactive Python interpreter. This is launched by issuing `$ python`. See what we did there? Once inside the Python interpreter, you can exit it by typing `quit()` or `exit()`.

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 terminal23 (on macOS or UNIX/Linux systems), or the Command Prompt24 (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*.

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.

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) should 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.

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!*

23https://en.wikipedia.org/wiki/Terminal_emulator 24https://en.wikipedia.org/wiki/Cmd.exe

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.

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 tweet25, raise an issue on GitHub26, or e-mail us – our addresses are available on the www.tangowithdjango.com27 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.

25https://twitter.com/tangowithdjango 26https://github.com/leifos/tango_with_django_2/issues 27https://www.tangowithdjango.com

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!

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 Tutorial**[28];
• **Think Python: How to Think like a Computer Scientist**[29] by Allen B. Downey; or
• **Learn Python in 10 Minutes**[30] by Stavros;
• **Learn to Program**[31] 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.Noteyoudon'tneedtobeanexpertinPython to work with Django –Python is straightforwardto use,andyou can pick itup as you go,especiallyif you already know the ins and outs of at least one other programming language.

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 problemas you would normally only be able to installone versionof the required software ata time. By creating virtual environments for each project, you can then install the respective versions of Django(andanyotherrequiredPythonsoftware)withineachuniqueenvironment.Thisensuresthat 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. Callthe environment `rangoenv`. If youare unsure as tohowtodo this, goto thesupplementary 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.

28https://docs.python.org/3/tutorial/ 29https://greenteapress.com/wp/think-python-2e/ 30https://www.stavros.io/tutorials/python/

31https://www.coursera.org/course/programming1


```
$ 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.

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 system[32] or some other environment[33], 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 `acivated`. Globally, you may have to use the 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.

[32] https://en.wikipedia.org/wiki/Advanced_Packaging_Tool [33] https://docs.npmjs.com/cli/install


```
$ 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.

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.

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*[35] by JetBrains and *PyDev* (a plugin of the Eclipse IDE[36]) standing out as popular choices. The Python Wiki[37] 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

34https://pillow.readthedocs.io/en/stable/installation.html 35http://www.jetbrains.com/pycharm/ 36http://www.eclipse.org/downloads/

37http://wiki.python.org/moin/IntegratedDevelopmentEnvironments

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 documentation34.

not to use an IDE, using a simple text editor like Sublime Text38, TextMate39 or Atom40 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 up41.

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 SVN42 or Git43. 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.**

38https://www.sublimetext.com/ 39https://macromates.com/ 40https://atom.io/

41https://www.jetbrains.com/help/pycharm/2016.1/creating-and-running-your-first-django-project.html 42http://subversion.tigris.org/

43http://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 GitHub44 or BitBucket45 if you haven't already done so.

• Download and set up an IDE like PyCharm46, 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 repository47.

• If you spot anyerrors or problems, please letus know by making an issue on GitHub48.

• If you have any problems with the exercises, you can check out the repository to see how we completed them.

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**.

Theseareavailablefromoursamplecodebaserepository,availableonGitHub49.The `progress_tests` directory on this repository contains a number of different Python modules, each containinga 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.

44https://github.com/ 45https://bitbucket.org/ 46https://www.jetbrains.com/pycharm/

47https://github.com/maxwelld90/tango_with_django_2_code 48https://github.com/leifos/tango_with_django_2/issues

49https://github.com/maxwelld90/tango_with_django_2_code/tree/master/progress_tests

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* whatwe requested inthebook,*thetestwill fail.*This mightseemoverlyharsh, 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.

Howdo 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 repository50 on your computer, or access the individual

test module that you want from the GitHub web interface51.
• 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. Movethe `tests_chapter3.py`moduletoyourproject's `rango`directory.Thisstepdoesnotmake sense right now; as you progress through the book and come back here to refresh yourmemory 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.

50https://github.com/maxwelld90/tango_with_django_2_code

51https://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'dbelookingtodelete `tests_chapter3.py`.Onceyouhaveconfirmedyoursolution 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!

Django Basics

Let'sgetstartedwithDjango!Inthis chapter, we'll be giving youanoverviewof the creationprocess. 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 up and running!

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.

Ifyouareusingavirtualenvironment,thenensurethatyouhaveactivatedit–ifyoudon’tremember 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 Django52.

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
```

Ifyoudon’thavea `workspace`directory,we recommendthatyoucreateone.Thismeansthatyoucan house yourDjango projects (and other code projects) withinthis directory. Itkeeps 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, 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 StackOverflow53. 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`foryou. Typically,we append `_project`to the end of ourDjango project directories so we know exactly what they contain – but the naming convention is 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:
• another directory with the same name as your project, `tango_with_django_project`; and

52https://docs.djangoproject.com/en/2.1/topics/install/

53http://stackoverflow.com/questions/8112630/cant-create-django-project-using-command-prompt

• a Python script called `manage.py`.

For the purposes of this tutorial, we call this nested directory called `tango_with_django_project` 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:

• `__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 that 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 scripts54.

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:

54https://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 – `http://127.0.0.1:8000/`[55]. You should see a webpage similar to the one shown below.

55http://127.0.0.1:8000/

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

Youcan 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 privileged[56] by your operating system.

56http://www.w3.org/Daemon/User/Installation/PrivilegedPorts.html

While you won't be using the lightweight development serverto deploy yourapplication, it's nice to be ableto demoyourapplication onanothermachine in yournetwork.Runningthe serverwithyour 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.**

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 do this, from within your Django project directory (e.g. `<workspace>/tango_with_django_-project`), run the following command.

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

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 documentation57 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.

```
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.

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.` so that you now have a blank file.

You can now add in the following code.

57https://docs.djangoproject.com/en/2.1/intro/overview/


```
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`58 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`59 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)60 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`–thesecond `tango_with_-` `django_project` directory!) and add the following lines of code to the `urlpatterns` list:

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

This maps thebasicURLto the `index`viewinthe `rango` app. Run the developmentserver (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.

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

58https://docs.djangoproject.com/en/2.1/ref/request-response/#django.http.HttpResponse

59https://docs.djangoproject.com/en/2.1/ref/request-response/#django.http.HttpRequest

60http://en.wikipedia.org/wiki/Uniform_resource_locator

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 the `urlpatterns` list as shown in 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 the rango app.

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 `django.conf.urls`. 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. In this case, as we have used an empty string `''`, then Django will 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 optionalparameter is called `name`.It provides a convenient way to reference the view, and by naming ourURL mappings wecan employ *reverse URLmatching*. 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.

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.

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 URLs[61] 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.Refertothechapterprovidingacrashcourseon Gitifyoucan'trememberthecommands and steps involved in doing this.

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.

61https://docs.djangoproject.com/en/2.1/topics/http/urls/

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`.

• Revise the `HttpResponse` in the `index` view to include a link to the about page.

• Include a link back to the index page in the `about` view's response.

• Now that you have started the book, follow us on Twitter @tangowithdjango62, and let us know how you are getting on!

62https://twitter.com/tangowithdjango

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?

63https://en.wikipedia.org/wiki/Hyperlink 64https://docs.djangoproject.com/en/2.1/intro/tutorial01/

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 *hyperlink*63 to provide a link to the about page – something like `<a href='/rango/about/'>About</a>` 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: `<a href='/rango/'>Index</a>`.

• If you haven't done so already, now's a good time to head off and complete part one of the official Django Tutorial64.

Templates and Media Files

Inthischapter,we'llbeintroducingthe Django templateengine,aswellas showingyouhowtoserve both *static* files and *media* files. Rather than crafting each page, we can use templates to provide the skeleton structure of the page, and then in the view, we can provide the template with the necessary data to render that page. To incorporate JavaScript and CSS – along with images and other media content–wewillusethemachineryprovidedbyDjangotoincludeanddispatchsuchfilestoprovide added functionality (in the case of JavaScript), or to provide styling to our pages.

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 layouts65 aids users with navigation and reinforces a sense of continuity. Django provides templates66 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.

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 thatneverchange), 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.

65http://www.techrepublic.com/blog/web-designer/effective-design-principles-for-web-designers-repetition/

66https://docs.djangoproject.com/en/2.1/ref/templates/

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`. Remember, this is the directory that contains your project's `manage.py` script! Within the new templates directory, create another directory called `rango`. This means that thepath `<workspace>/tango_with_django_project/templates/rango/`willbethe locationinwhich 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 theDjango project where templates will be stored, open yourproject'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 to 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 paths[67] is a software engineering anti-pattern[68], and will make your project less portable[69] - meaning that when you run it on another computer, it probably won't work!

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 module[70]. 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 `<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.

67 http://en.wikipedia.org/wiki/Hard_coding 68 http://sourcemaking.com/antipatterns 69 http://en.wikipedia.org/wiki/Software_portability

70 http://stackoverflow.com/a/9271479


```
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 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 `os.path.join()` to join (concatenate) together the `BASE_DIR` variable and `'templates'`, which would yield `<workspace>/tango_with_django_project/templates/`. This means 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.

```
'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 - but for this book, 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(orderivativeof),forwardslashes(/)wouldbeusedtoseparatedirectories,whereasa 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 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 atthe top of the file.Django should have added it for you when you created the Rango app. If it is not present, add it.

```
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 is the same as {{ 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, 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 exampledemonstrateshowtousetemplateswithinyourviews.However,wehaveonlytouched 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 documentation71.

71https://docs.djangoproject.com/en/2.1/ref/templates/

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.

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)*72, *JavaScript*73 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.

72http://en.wikipedia.org/wiki/Cascading_Style_Sheets 73https://en.wikipedia.org/wiki/JavaScript

Next, place an image inside the `images` directory. As shown in below, we chose a picture of the chameleon Rango74 - a fitting mascot, if ever there was one.

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')
```

Thiswillprovideanabsolutepathtothelocation`<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 can expect to find static files that can be served. By default, this list does not exist-**check** itdoesn'tbefore youcreate it. If youdefine it twice, youcanstartto confuse Django - and yourself. For this book, we're only going to be using one location to store our project's static files - the path defined in `STATIC_DIR`. As such, we can simply set up `STATICFILES_DIRS` with the following.

74http://www.imdb.com/title/tt1192628/


```
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 justputthings in random places;keep itorganised.Keepyour `DIRS` variables atthe top ofthe 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 is not, then define it as shown below. Note that this variable by default in Django appears close to the end of the module, so you may have to scroll down to find it.

```
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.*

Don't Forget the Slashes!

When setting `STATIC_URL`, check thatyou end the URL you specifywith a forwardslash (e.g. `/static/`, not `/static`). As per the official Django documentation75, 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`).

75https://docs.djangoproject.com/en/2.1/ref/settings/#std:setting-STATIC_URL

Test your Configuration

As asmallexercise,testtosee ifeverything is working correctly.Tryandview 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 URL you enter into your web browser's window?

Try to figure this out before you move on! The answer is coming up if you get stuck.

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 Djangodocumentationon deployment76provides furtherinformation aboutdeployingstatic 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.

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 />
<img src="{% static 'images/rango.jpg' %}"
alt="Picture of Rango" /> <!-- New line --> </div>
```

76https://docs.djangoproject.com/en/2.1/howto/static-files/deployment/


```
</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 tag77. 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:

```
<img src="/static/images/rango.jpg" alt="Picture of Rango" />
```

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.

77https://docs.djangoproject.com/en/2.1/ref/templates/builtins/

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.

Always put `<!DOCTYPE>` First!

When creating the HTML templates, always ensure that the `DOCTYPE` declaration78 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 validation79.

78http://www.w3schools.com/tags/tag_doctype.asp 79https://validator.w3.org/

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

Loading other Static Files

The `{% static %}`templatetagcanbeusedwheneveryouwishtoreferencestaticfileswithin 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 -->
<img src="{% static "images/rango.jpg" %}" alt="Picture of Rango" /> </body>
</html>
```

Don’tupdate the `base.html` template here– this is merelya 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` error80. Try referencing a non-existent file and see what happens. Looking at the outputsnippetbelow, 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 templates81.

80https://en.wikipedia.org/wiki/HTTP_404 81https://docs.djangoproject.com/en/2.1/howto/static-files/#staticfiles-in-templates

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.

Toserve mediafiles successfully,weneedto updatethe Djangoproject’ssettings.This sectiondetails 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 - to 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 bysome other means. The deployment chapter will discuss this in more detail.

Modifying `settings.py`

First, openyourDjangoproject’s `settings.py`module.Inhere,we’llbeaddinga couple morethings. 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`.

```
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 hosting[82].

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


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

Once again, don't Forget the Slashes!

Like the `STATIC_URL` variable, ensure that `MEDIA_URL` ends with a forward slash (i.e. `/media/`, not `/media`). The extra slash at the end ensures that the root of the URL (e.g. `/media/`) is separated from the 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/stored, and what URL to serve them from (`MEDIA_URL`). With the configuration defined above, the uploadedfile `cat.jpg`will,forexample, be availableonyourDjangodevelopment server at `http://localhost:8000/media/cat.jpg`.

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*[83] 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: `'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,andmodifyingitbyappendinga calltothe `static()`functionto yourproject's `urlpatterns` list.Remember,your**project's** `urls.py`module is theonethatliveswithinthe `tango_with_django_project` directory!

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

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

You'll also need to add the following `import` statements at the top of the `urls.py` module.

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

Once this is complete, 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 a the same level as the `static` directory and the `manage.py` module.

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 specifiedinyourproject's `settings.py`module.YoumaywishtouseDjangotemplatevariables (e.g. `{{ variable_name }}`) or template tags[84] 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!

84https://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 is the directory you specify in your project's `STATICFILES_DIRS` list within `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 `<img />` 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 `<img />` tag like `<img src="{{ MEDIA_URL}}cat.jpg" alt="Picture of a Cat."/>`.

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.`.

• Within the new `about.html` template, add a picture stored within your project's static files. You can just reuse the `rango.jpg` image you used in the index view! Make sure you keep the same `alt` text as the index page!

• 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 `<img>` 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.**

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?

Models and Databases

Typically, web applications require a backend to store the dynamic content that appears on the app's webpages.ForRango,weneedtostorepagesandcategoriesthatarecreated,alongwithotherdetails.

The most convenient way to do this is by employing the services of a relational database – that use the *Structured Query Language (SQL)*. However, Django provides a convenient way in which to access data stored in databases by using an *Object Relational Mapper (ORM)*[85]. 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.

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 – but to refresh your memory, let's quickly summarise our client's requirements.

• Rango is essentially a *web page directory* – a site containing links to other websites.

• There are several different *webpage categories* with each category housing several 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.

85https://en.wikipedia.org/wiki/Object-relational_mapping

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, SQLite86 (see the `ENGINE` option). The `NAME` entry for this database is the path to the database file, which is by default `db.sqlite3` in your project's root directory (i.e. `<workspace>/tango_with_django_project/`).

Don't `git push` your Database!

If you are using Git, you might be tempted to add and commit the database file. This is not a good idea because if you are working on your app with other people, they are likely to change the database and this 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.

86https://www.sqlite.org/

Using other Database Engines

The Django database framework has been created to cater for a variety of different database backends, such as PostgresSQL87, MySQL88 and Microsoft's SQLServer89.For other database engines,otherkeys like `USER`, `PASSWORD`, `HOST`and `PORT` existforyoutoconfigurethedatabase with Django.

While we don'tcover how touse other database engines in this book,there are guides online whichshowyouhowtodothis.Agoodstartingpointis theofficialDjangodocumentation90.

Note that SQLite is sufficientfor demonstrating the functionalityof theDjango 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 robust91.

Creating Models

With your database configured in `settings.py`, let's create the two initial data models for the Rango application. Models fora Django app are storedinthe 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 inherit92 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
```

87http://www.postgresql.org/ 88https://www.mysql.com/

89https://en.wikipedia.org/wiki/Microsoft_SQL_Server 90https://docs.djangoproject.com/en/2.1/ref/databases/#storage-engines

91http://www.sqlite.org/whentouse.html 92https://en.wikipedia.org/wiki/Inheritance_(object-oriented_programming)

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 types[93]. 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.
- `URLField`, much like a `CharField`, but designed forstoringresource 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 fields[94] 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 `NULL`[95]) (`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 relationship[96];
- `OneToOneField`, a field type that allows us to define a strict one-to-one relationship[97]; and

93 https://docs.djangoproject.com/es/2.1/ref/models/fields/#model-field-types

94 https://docs.djangoproject.com/es/2.1/ref/models/fields/#model-field-types 95 https://en.wikipedia.org/wiki/Nullable_type

96 https://en.wikipedia.org/wiki/One-to-many_(data_model) 97 https://en.wikipedia.org/wiki/One-to-one_(data_model)

- `ManyToManyField`, a field type which allows us to define a many-to-many relationship[98].

From our model examples above, the field `category` in model `Page` is of type `ForeignKey`. This allows us to create a one-to-many relationship with model/table `Category`, 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 Djangoto delete thepagesassociatedwiththe categorywhenthe 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 Keys[99] 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!

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 tool*[100] 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.

Setting up

First of all, the database must be *initialised*. This means that creating the database and all the associated tables sothat 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.*

98 https://en.wikipedia.org/wiki/Many-to-many_(data_model)

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

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


```
$ 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
```

AllappsthatareinstalledinyourDjangoproject(check `INSTALLED_APPS`in `settings.py`)willupdate their database representations with this command. 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 scripthasbeencreated.It'scalled `0001_initial.py`,whichcontainsallthenecessarydetailstocreate 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.**

Django Models and the Shell

BeforeweturnourattentiontodemonstratingtheDjangoadmininterface,it'sworthnotingthatyou 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 `__str__()` method at work!

Complete the Official Tutorial

The example above is only a very basic taster on database related activities you can perform intheDjangoshell.Ifyouhavenotdonesoalready,it'snowagoodtimetocompleteparttwo of the official Django Tutorial to learn more about interacting with models101. In addition, have a look at the official Django documentation on the list of available commands102 for working with models.

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.

101 https://docs.djangoproject.com/en/2.1/intro/tutorial02/

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

The Django admin interface, sans Rango models.

While this looks good, we aremissing the `Category`and `Page`modelsthatweredefinedforthe 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, either reload the admin web pages or 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.

Tryclickingthe `Categories`linkwithinthe `Rango`section.Fromhere,youshouldseethe `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 categorys` 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.

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 interface[104] for more information if you're interested. We'll be working towards manipulating the `admin.py` module later on in the tutorial.

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 for Rango, start by creating a new Python module within your Django project's root directory (e.g. `<workspace>/tango_with_django_project/`). Create the new, blank `populate_rango.py` file and add the following code.

[103] https://docs.djangoproject.com/en/2.1/topics/db/models/#meta-options [104] https://docs.djangoproject.com/en/2.1/ref/contrib/admin/

Plural vs. Singular Spellings

Notethetypowithintheadmin interface(`Categorys`,not `Categories`).Thistypocanbefixed 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 officialdocumentationonmodels[103]formoreinformationaboutwhatcanbestored 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
```



```
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('- {0} - {1}'.format(str(c),
str(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 thislookslike alotofcode,what isgoingonisessentiallya series of functioncallstotwo 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 75 and 76. This is because, above this point, we define functions; these are not executed *unless* we call them. When the interpreter hits `if __name__ == '__main__'`[105], we call and begin execution of the `populate()` function.

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 the 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.

105 http://stackoverflow.com/a/419185

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

The `__name__ == '__main__'`trickisausefulonethatallows aPythonmoduletoactas 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.

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]` at the end of our call to the `get_or_create()` returns the object reference only. Like most other programming language data structures, Python tuples use zero-based numbering106.

You can check out the official Django documentation107 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 terminalto the Djangoproject'sroot. It's then a simplecase of executingthe command `$ python populate_rango.py`. You should then see output similar to that shown below – the order in which categories are added may vary depending upon how your computer is set up.

```
$ 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 populated the database. Restart the Django development server, navigate to the admin interface (at `http://127.0.0.1:8000/admin/`)and check that you have

106http://en.wikipedia.org/wiki/Zero-based_numbering 107https://docs.djangoproject.com/en/2.1/ref/models/querysets/#get-or-create

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 willsave yourself heaps of time in the longrun.Whendeployingyourappelsewhere,runningthepopulationscriptaftersettingeverything 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.

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 make them accessible via the 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`toincludeandregisteryournewmodel(s)ifyouwanttomakethemaccessible 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 engines[108].

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.

[108] 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 youto havecompletedthese exercises! If you're stuck, there aresome hintsto help you complete the exercises below.**

- Updatethe `Category` model toincludethe additionalattributes `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 two[109] and seven[110] of the official Django tutorial. These sections will reinforce what you've learnt on handling databases in Django, and show you additionaltechniques 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 `name` of the page and the `url` – just like in the screenshot shown below. Complete the official Django tutorial or look at the tip below to complete this particular exercise.

109https://docs.djangoproject.com/en/2.1/intro/tutorial02/ 110https://docs.djangoproject.com/en/2.1/intro/tutorial07/

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

Exercise Hints

Ifyourequire somehelpor inspiration to complete theseexercises done, here aresome hints.
• Modify the `Category` model by adding two `IntegerFields`: `views` and `likes`.
• 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. – 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 modelinstance is assignedtovariable `c` in thepopulation script,as definedearlier 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[111] represents the *name* of the category as the key, and an additional dictionary containing additionalinformationrelatingtothecategoryasthe*value*.You'llwanttomodify this 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 fields to the function call. If you aren't sure how the `for` loop works over dictionaries, check out this online Python tutorial[112]. From here, you can figure out how to access the `views` and `likes` values from your dictionary.
• 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 `admin.site.register(Page, PageAdmin)` in Rango's `admin.py` file.

111https://www.tutorialspoint.com/python/python_dictionary.htm 112https://www.tutorialspoint.com/python/python_dictionary.htm

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.

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.

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

Herewewillcompletesteps two andstep three, whereweneedtomodify theview `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_dict)
```

Here, the expression `Category.objects.order_by('-likes')[:5]` queries the `Category` model to retrieve the top five categories. Youcan 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 operators113 to take the first five objects from the list (`[:5]`) to return a subset of `Category` objects.

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 needtoaddthefield `likes` tothe `Category` model.Likewe have saidalready,we assume you complete all exercises as you progress through this book.

Modifying the Index Template

With the viewupdated,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.

113https://www.quackit.com/python/reference/python_3_list_methods.cfm