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

Python web development with Django

Tango with Django

A beginner's guide to web development with **Django 1.9**.

Also compatible with **Django 1.10**

REVISED AND UPDATED FOR django version 1.9

Available from **www.tangowithdjango.com**

How to Tango with Django 1.9

A beginners guide to Python/Django

Leif Azzopardi and David Maxwell

This book is for sale at http://leanpub.com/tangowithdjango19

This version was published on 2018-01-09

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 now ready to Tango with Django @tangowithdjango

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

Contents

1. Overview

The aim of this book is to provide you with a practical guide to web development using *Django* and *Python*. 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 Tutorials 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 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 (e.g. HTML, CSS, JavaScript, etc.).

1.1 Why Work with this Book?

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

This book will lower the learning curve. Web application frameworks can save you a lot of hassle and lot of time. Well, that is 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.

This book will improve your workflow. Using web application frameworks requires you to pick up and run with a particular design pattern - 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 them (i.e. inversion of control). To help you, we've created a number of *workflows* to focus your development process so that you can regain that sense of control and build your web application in a disciplined manner.

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

1.2 What you will Learn

In this book, we will be taking an exampled-based approach. The book will show you how to design a web application called *Rango* (see the Design Brief below). Along the way, we'll show you how to perform the following key tasks.

- **How to setup your development environment** - including how to use the terminal, your virtual environment, the `pip` installer, how to work with Git, and more.
- **Setup a Django project** and create a basic Django application.
- **Configure the Django project** to serve static media and other media files.
- Work with Django's *Model-View-Template* design pattern.
- **Create database models** and use the *object relational mapping (ORM)* functionality provided by Django.
- **Create forms** that can utilise your database models to create dynamically generated web-pages.
- Use the **user authentication** services provided by Django.
- Incorporate **external services** into your Django application.
- Include *Cascading Styling Sheets (CSS)* and *JavaScript* within a web application.
- **Apply CSS** to give your application a professional look and feel.
- Work with **cookies and sessions** with Django.
- Include more advanced functionality like *AJAX* into your application.
- **Deploy your application** to a web server using *PythonAnywhere*.

At the end of each chapter, we have included a number of exercises designed to push you harder and to see if you can apply what you have learned. The later chapters of the book provide a number of open development exercises along with coded solutions and explanations.

Exercises will be clearly delineated like this!

In each chapter we have added a number of exercises to test your knowledge and skill.

You will need to complete these exercises as the subsequent chapters are dependent on them.

Don't worry if you get stuck, though, as you can always check out our solutions to all the exercises on our *GitHub* repository.

1.3 Technologies and Services

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

- the Python programming language;
- the Pip package manager;
- Django;
- the Git version control system;
- GitHub;
- HTML;
- CSS;
- the JavaScript programming language;
- the JQuery library;
- the Twitter Bootstrap framework;
- the Webhose API (referred to as the *search API*); and
- the PythonAnywhere 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 *Webhose API* and deploy your application quickly and easily with *PythonAnywhere*.

1.4 Rango: Initial Design and Specification

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

Design Brief

Your client would like you to create a website called *Rango* that lets users browse through user-defined categories to access various web pages. In Spanish, the word rango is used to mean *"a league ranked by quality"* or *"a position in a social hierarchy"*.

- 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 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*. Visitors to the site should therefore 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 a number of complexities and challenges that need to be addressed. First, let's try and build up a better picture of what needs to be developed by laying down some high-level designs.

Exercises

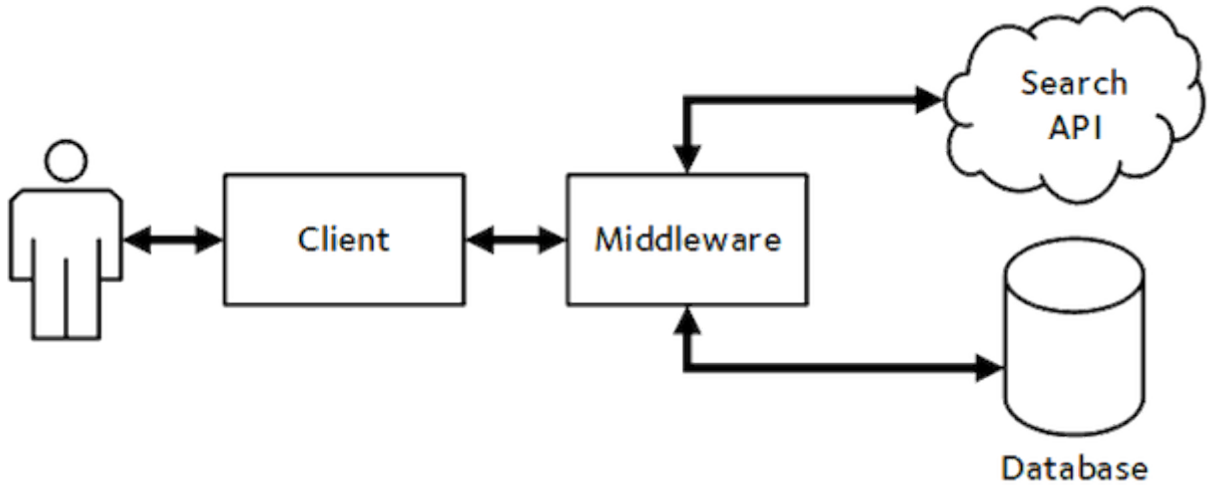
Before going any further, think about these specifications and draw up the following design artefacts.

- An **N-Tier or System Architecture** diagram.
- **Wireframes** of the main and category pages.
- A series of **URL mappings** for the application.
- An ***Entity-Relationship (ER)*** 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 what you are going to build.

N-Tier Architecture

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



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

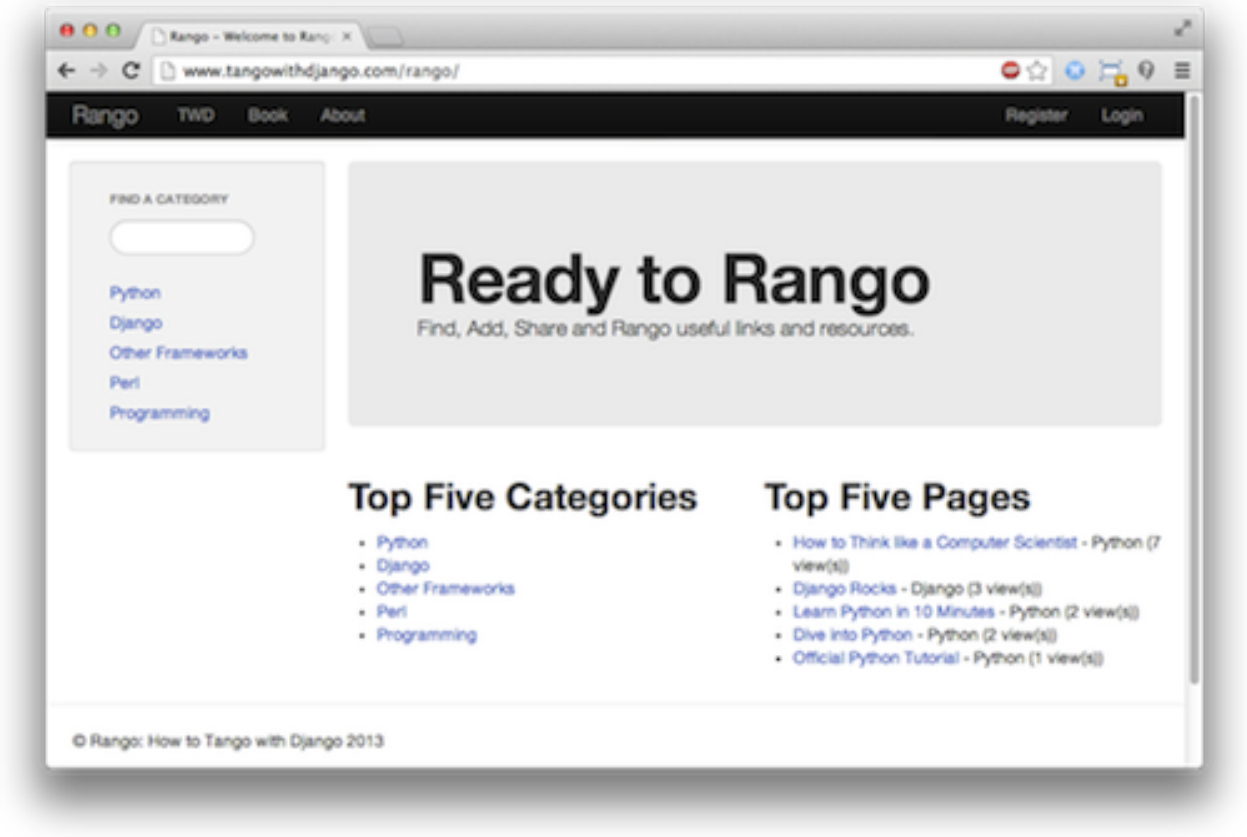
Since we are building a web application with Django, we will use the following technologies for the following tiers.

- The **client** will be a Web browser (such as *Chrome*, *Firefox*, and *Safari*) which will render HTML/CSS pages.
- The **middleware** will be a *Django* application, and will be dispatched through Django's built-in development Web server while we develop.
- The **database** will be the Python-based *SQLite3* Database engine.
- The **search API** will be the search API.

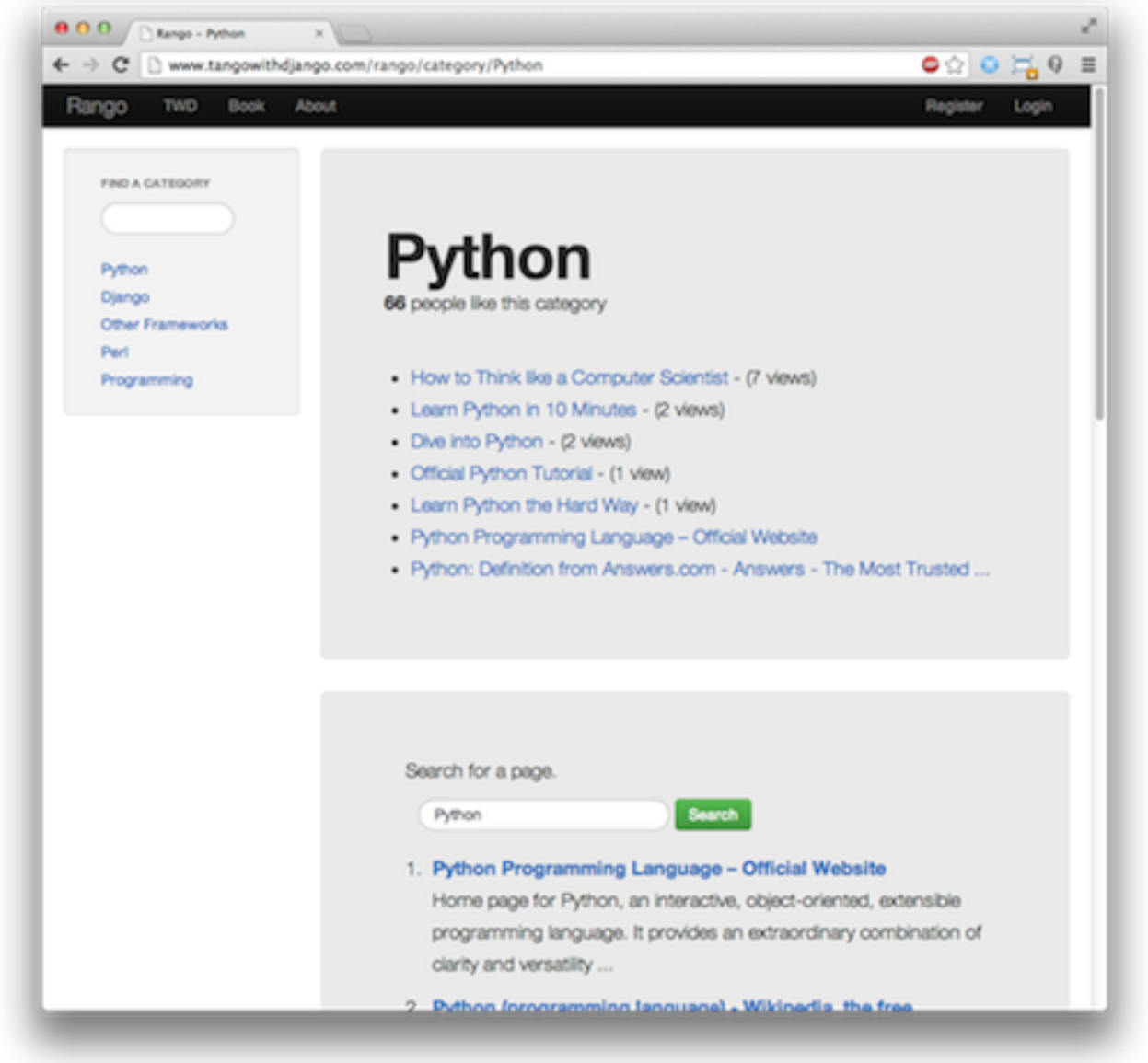
For the most part, this book will focus on developing the middleware. It should however be quite evident from the system architecture diagram that we will have to interface with all the other components.

Wireframes

Wireframes are great way to provide clients with some idea of what the application should look like when complete. They save a lot of time, and can vary from hand drawn sketches to exact mockups depending on the tools that you have at your disposal. For our Rango application, we'd like to make the index page of the site look like the screenshot below. Our category page is also shown below.



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



The category page showing the pages in the category (along with the number of views). Below, a search for *Python* has been conducted, with the results shown underneath.

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 will have to enter into a browser's address bar to reach the given page. The basic URL mappings for Rango are shown below.

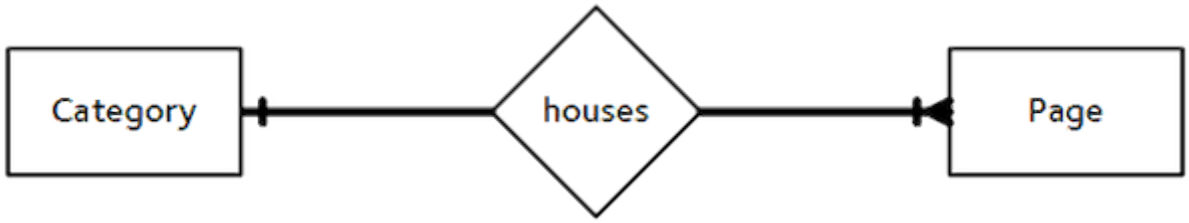
- `/` or `/rango/` will point to the main / index page.

- `/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.

As we build our application, we will probably need to create other URL mappings. However, the ones listed above will get us started and give us an idea of the different pages. Also, as we progress through the book, we will flesh out how to construct these pages using the Django framework and use its Model-View-Template design pattern. However, now that we have a gist of the URL mappings and what the pages are going to look like, we need to define the data model that will house the data for our Web application.

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 relationship. This approach however introduces a number of complexities, so 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, which is not ideal.

Take Note!

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

With this assumption, we then 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

Field	Type
`name`	`CharField`
`views`	`IntegerField`
`likes`	`IntegerField`

Page Model

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

We will also have a model for the `User` so that they can register and login. We have not shown it here, but shall introduce it later in the book when we discuss User Authentication. In the following chapters, will we see how to instantiate these models in Django and how to use the built-in ORM to connect to the database.

1.5 Summary

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

Cut and Paste Coding

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

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 if you're using Python 3 - inconsistent use of tabs and spaces in your code's indentation will lead to a `TabError`.

Most code editors will show the whitespace and whether it is tabs or spaces. If so, turn it on and save yourself a lot of confusion.

2. Getting Ready to Tango

Before we get down to coding, it's really important that we get our development environment setup so that you can *Tango with Django!* You'll need to ensure that you have all the necessary components installed on your computer. This chapter outlines the five key components that you need to be aware of, setup and use. These are listed below.

- Working with the terminal or Command Prompt.
- *Python* and your *Python* installation.
- The Python Package Manager *pip* and *virtual environments*.
- Your *Integrated Development Environment (IDE)*, if you choose to use one.
- A *Version Control System (VCS)*, *Git*.

If you already have Python 2.7/3.4/3.5 and Django 1.9/1.10 installed on your computer, and are familiar with the technologies mentioned, then you can skip straight to the Django Basics chapter. Otherwise, below we provide an overview of the different components and why they are important. We also provide a series of pointers on how to setup the various components.

Your Development Environment

Setting up your development environment is pretty tedious and often frustrating. It's not something that you'd do everyday. Below, we have put together the list of core technologies you need to get started and pointers on how to install them.

From experience, we can also say that it's a good idea when setting your development environment up to note down the steps you took. You'll need them again one day - whether because you have purchased a new computer, or you have been asked to help someone else set their computer up! Taking a note of everything you do will save you time and effort in the future. Don't just think short term!

2.1 Python

To work with Tango with Django, we require you to have installed on your computer a copy of the Python programming language. Any version from the `2.7` family - with a minimum of `2.7.5` - or version `3.4+` will work fine. If you're not sure how to install Python and would like some assistance, have a look at the chapter dealing with installing Python.

Not sure how to use Python?

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

- **Learn Python in 10 Minutes** by Stavros;
- **The Official Python Tutorial**;
- **Think Python: How to Think like a Computer Scientist** by Allen B. Downey; or
- **Learn to Program** by Jennifer Campbell and Paul Gries.

These will get you familiar with the basics of Python so you can start developing using Django. Note you don't need to be an expert in Python to work with Django. Python is awesome and you can pick it up as you go, if you already know another programming language.

2.2 The Python Package Manager

Pip is the python package manager. The package manager allows you install various libraries for the Python programming language to enhance its functionality.

A package manager, whether for Python, your operating system or some other environment, 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. This is opposed to downloading, installing and maintaining software manually. Maintaining Python packages is pretty painful. Most packages often have *dependencies* so these need to be installed too. Then these packages may conflict or require particular versions which need to be resolved. Also, the system path to these packages needs to be specified and maintained. Luckily *pip* handles all this for you - so you can sit back and relax.

Try and run pip with the command `$ pip`. If the command is not found, you'll need to install pip itself - check out the system setup chapter for more information. You should also ensure that the following packages are installed on your system. Run the following commands to install Django and pillow (an image manipulation library for Python).

```
$ pip install -U django==1.9.10
$ pip install pillow
```

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 --global-option="build_ext"
                   --global-option="--disable-jpeg"
```

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

2.3 Virtual Environments

We're almost all set to go! However, before we continue, it's worth pointing out that while this setup is fine to begin with, there are some drawbacks. What if you had another Python application that requires a different version to run, or you wanted to switch to the new version of Django, but still wanted to maintain your Django 1.9 project?

The solution to this is to use virtual environments. Virtual environments allow multiple installations of Python and their relevant packages to exist in harmony. This is the generally accepted approach to configuring a Python setup nowadays.

Setting up a virtual environment is not necessarily but it is highly recommended. The virtual environment chapter details how to setup, create and use virtual environments.

2.4 Integrated Development Environment

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

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 do provide a guide on setting PyCharm up.

2.5 Code Repository

We should also point out that when you develop code, you should always house your code within a version-controlled repository such as SVN or Git. 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 own projects.**

Exercises

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

- Install Python 2.7.5+/3.4+ and Pip.
- Play around with your *command line interface (CLI)* and create a directory called `code`, which we use to create our projects in.
- Setup your Virtual Environment (optional)
- Install the Django and Pillow packages
- Setup an account on a Git Repository site like: GitHub, BitBucket, etc if you haven't already done so.
- Download and setup an Integrated Development Environment like PyCharm

As previously stated, we've made the code for the book and application available on our GitHub repository.

- If you spot any errors or problem, please let us know by making a change request on GitHub.
- If you have any problems with the exercises, you can check out the repository to see how we completed them.

What is a Directory?

In the text above, we refer to creating a *directory*. But what exactly is a *directory*? If you have used a Windows computer up until now, you'll know a directory as a *folder*. The concept of a folder is analogous to a directory - it is a cataloguing structure that contains references to other files and directories.

3. Django Basics

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

3.1 Testing Your Setup

Let's start by checking that your Python and Django installations are correct for this tutorial. To do this, open a new terminal window and issue the following command, which tells you what Python version you have.

```
$ python --version
```

The response should be something like `2.7.11` or `3.5.1`, but any 2.7.5+ or 3.4+ versions of Python should work fine. If you need to upgrade or install Python go to the chapter on setting up your system.

If you are using a virtual environment, then ensure that you have activated it - if you don't remember how go back to 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 2.7.10 (default, Jul 14 2015, 19:46:27)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.39)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

At the prompt, enter the following commands:

```
>>> import django
>>> django.get_version()
'1.9.10'
>>> exit()
```

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

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

Prompts

In this book, there's two things you should look out for when we include code snippets.

Snippets beginning with a dollar sign (`$`) indicates that the remainder of the following line is a terminal or Command Prompt command.

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? You can also exit the Python interpreter by entering `quit()`.

3.2 Creating Your Django Project

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

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

If you don't have a `workspace` directory, then create one, so that you can house your Django projects and other code projects within this directory. We will refer to your `workspace` directory in the code as `<workspace>`. You will have to substitute in the path to your `workspace` directory, for example: `/Users/leifos/Code/` or `/Users/maxwelld90/Workspace/`.

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

Try entering `django-admin` instead. Depending on your setup, some systems may not recognise `django-admin.py`.

While, on Windows, you may have to use the full path to the `django-admin.py` script, for example:

```
python c:\python27\scripts\django-admin.py
       startproject tango_with_django_project
```

as suggested on StackOverflow.

This command will invoke the `django-admin.py` script, which will set up a new Django project called `tango_with_django_project` for you. Typically, we append `_project` to the end of our Django

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
- 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 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 scripts.

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.

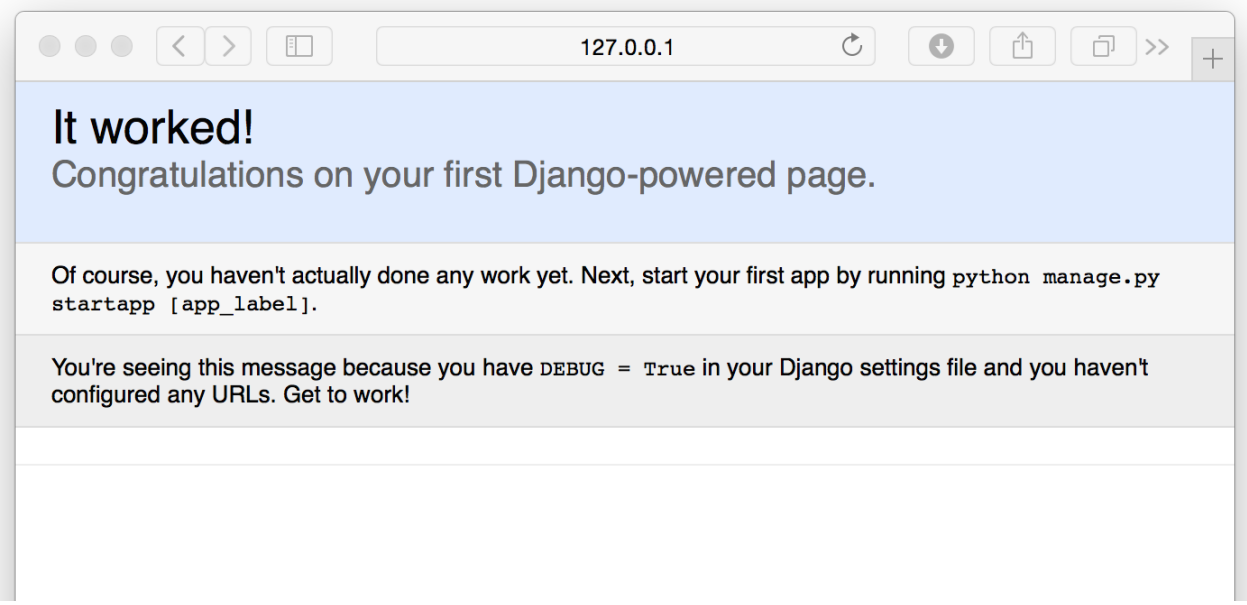
```
$ python manage.py runserver
```

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:

```
$ python manage.py runserver

Performing system checks...

System check identified no issues (0 silenced).

You have unapplied migrations; your app may
not work properly until they are applied.

Run 'python manage.py migrate' to apply them.

October 2, 2016 - 21:45:32
Django version 1.9.10, 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 a number of things. First, there are no issues that stop the application from running. Second, however, you will notice that a warning is raised, i.e. unapplied migrations. We will talk about this in more detail when we setup 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 of the Django development webserver.

Now open up your Web browser and enter the URL `http://127.0.0.1:8000/`. You should see a webpage similar to the one shown in below.



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

You can stop the development server at anytime by pushing `CTRL + C` in your terminal or Command Prompt window. 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 by your operating system.

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

3.3 Creating a Django App

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

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

To 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 a number of Python scripts:

- another `__init__.py`, serving the exact 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 app's code;
- `views.py`, where you can store a series of functions that handle requests and return responses; and
- `migrations` directory, which stores database specific information related to your models.

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

Before you can get started with creating your own 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` tuple. 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.

`startapp` Magic

When creating a new app with the `python manage.py startapp` command, Django may add the new app's name to your `settings.py` `INSTALLED_APPS` list automatically for you. It's nevertheless good practice to check everything is setup correctly before you proceed.

3.4 Creating a View

With our Rango app created, let's now create a simple view. For our first view, let's just send some text back to the client - we won't concern ourselves about using models or templates just yet.

In your favourite IDE, 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.

```
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` 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` 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) to the view.

To create an initial mapping, open `urls.py` located in your project directory and add the following lines of code to the `urlpatterns`:

```
from rango import views

urlpatterns = [
    url(r'^$', views.index, name='index'),
    url(r'^admin/', admin.site.urls),
]
```

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

3.5 Mapping URLs

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

First, open the project's `urls.py` file which is located inside your project configuration directory. As a relative path from your workspace directory, this would be the file `<workspace>/tango_with_django_project/tango_with_django_project/urls.py`. Update the `urlpatterns` list as shown in the example below.

```
from django.conf.urls import url
from django.contrib import admin
from django.conf.urls import include
from rango import views

urlpatterns = [
    url(r'^$', views.index, name='index'),
    url(r'^rango/', include('rango.urls')),
    # above maps any URLs starting
    # with rango/ to be handled by
    # the rango application
    url(r'^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.conf.urls import url
from rango import views

urlpatterns = [
    url(r'^$', views.index, name='index'),
]
```

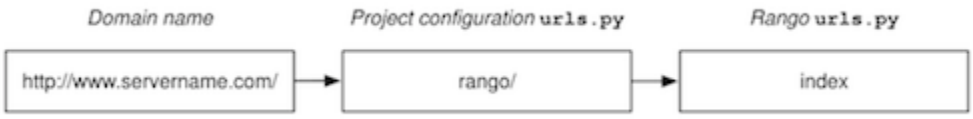
This code imports the relevant Django machinery for URL mappings and the `views` module from `rango`. This allows us to call the function `url` and point to the `index` view for the mapping in `urlpatterns`.

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 would have a URL string of `/rango/about/`.

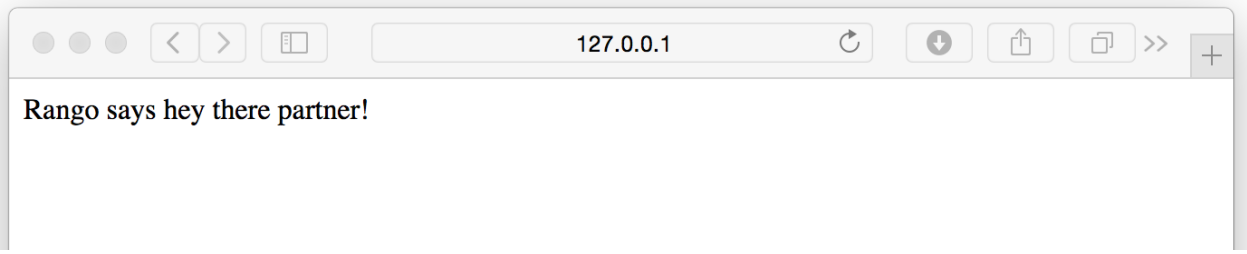
The URL mapping we have created above calls Django's `url()` function, where the first parameter is the regular expression `^$`, which matches to an empty string because `^` denotes starts with, while `$` denotes ends with. As there is nothing in between these characters then it only matches an empty string. Any URL string supplied by the user that matches this pattern means that the view `views.index()` would be invoked by Django. You might be thinking that matching a blank URL is pretty pointless - what use would it serve? Remember that when the URL pattern matching takes place, only a portion of the original URL string is considered. This is because Django will first process the URL patterns in the project processing the original URL string (i.e. `rango/`) and strip away the `rango/` part. Django will then pass on an empty string to the Rango app to handle via the URL patterns in `rango/urls.py`.

The next parameter passed to the `url()` function is the `index` view, which will handle the incoming requests, followed by the optional parameter, `name` that is set to a string `'index'`. By naming our URL mappings we can employ *reverse URL matching* later on. That is we can reference the URL mapping by name rather than by the URL. Later we will explain how to use this when creating templates. But do check out the Official Django documentation on this topic for more information.

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 a number of URL mappings. The initial mapping is quite simple, but as we progress through the book we will create more sophisticated, 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 for further details and further examples.

Note on Regular Expressions

Django URL patterns use regular expressions to perform the matching. It is worthwhile familiarising yourself on how to use regular expressions in Python. The official Python documentation contains a useful guide on regular expressions, while `regexcheatsheet.com` provides a neat summary of regular expressions.

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

3.6 Basic Workflows

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

Creating a new Django Project

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

Creating a new Django App

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

Exercises

Now that you have got Django and your new app up and running, give the following exercises a go 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.
- In the `HttpResponse` in the `about` view include a link back to the main page.
- Now that you have started the book, follow us on Twitter @tangowithdjango, and let us know how you are getting on!

Hints

If you're struggling to get the exercises done, the following hints will hopefully 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(), insert your HTML inside this response.
- The regular expression to match `about/` is `r'^about/'` - so in `rango/urls.py` add in a new mapping to the `about()` view.
- Update your `index()` view to include a link to the `about` view. Keep it simple for now - something like `Rango says hey there partner! <br/> <a href='/rango/about/'>About</a>`.
- Also add the HTML to link back to the index page is into your response from the `about()` view `<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 Tutorial.

4. Templates and Media Files

In this chapter, we'll be introducing the Django template engine, as well as showing how to serve both *static* files and *media* files, both of which can be integrated within your app's webpages.

4.1 Using Templates

Up until this point, we have only connected a URL mapping to a view. The Django framework, however, is based around the *Model-View-Template* architecture. In this section, we will go through the mechanics of how *Templates* work with *Views*, then in the next couple of 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 layouts aids users with navigation, promotes organisation of the website and reinforces a sense of continuity. Django provides templates 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 create a HTML page. This template 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 that never change), complete with special syntax (or *template tags*) which can be overridden and replaced with *dynamic content* that your Django app's views can replace to produce a final HTML response.

Configuring the Templates Directory

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

In your Django project's directory (e.g. `<workspace>/tango_with_django_project/`), create a new directory called `templates`. 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 the

path `<workspace>/tango_with_django_project/templates/rango/` will be the location in which we will store templates associated with our `rango` application.

Keep your Templates Organised

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

To tell the Django project where templates will be stored, open your project's `settings.py` file. Next, locate the `TEMPLATES` data structure. By default, when you create a new Django 1.9 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 are 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 a number of 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 is a software engineering anti-pattern, and will make your project less portable - 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 absolute path of your settings 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 layer, so that `BASE_DIR` contains `<workspace>/tango_with_django_project/`. You can see this process in action, if you are curious, by adding the following lines to your `settings.py` file.

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

Having access to the value of `BASE_DIR` makes it easy for you to reference other aspects of your Django project. As such, 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.

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

Here we make use of `os.path.join()` to mash 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 (or derivative of), forward slashes (`/`) would be used to separate directories, whereas a Windows operating system would use backward slashes (`\`). If you manually append slashes to paths, you may end up with path errors when attempting to run your code on a different operating system, thus reducing your project's portability.

Adding a Template

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

```
<!DOCTYPE html>
<html>

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

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

</html>
```

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

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

In `rango/views.py`, check to see if the following `import` statement exists at the top of the file. 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.

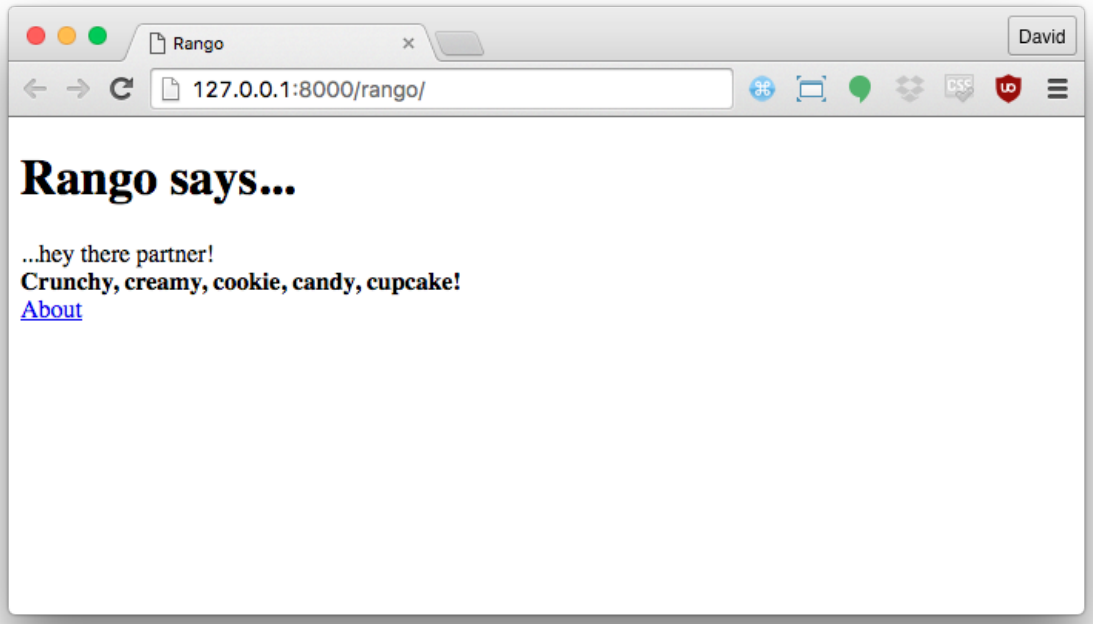
What is the Template Context?

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 example demonstrates how to use templates within your views. However, we have only touched upon 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 documentation.



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.

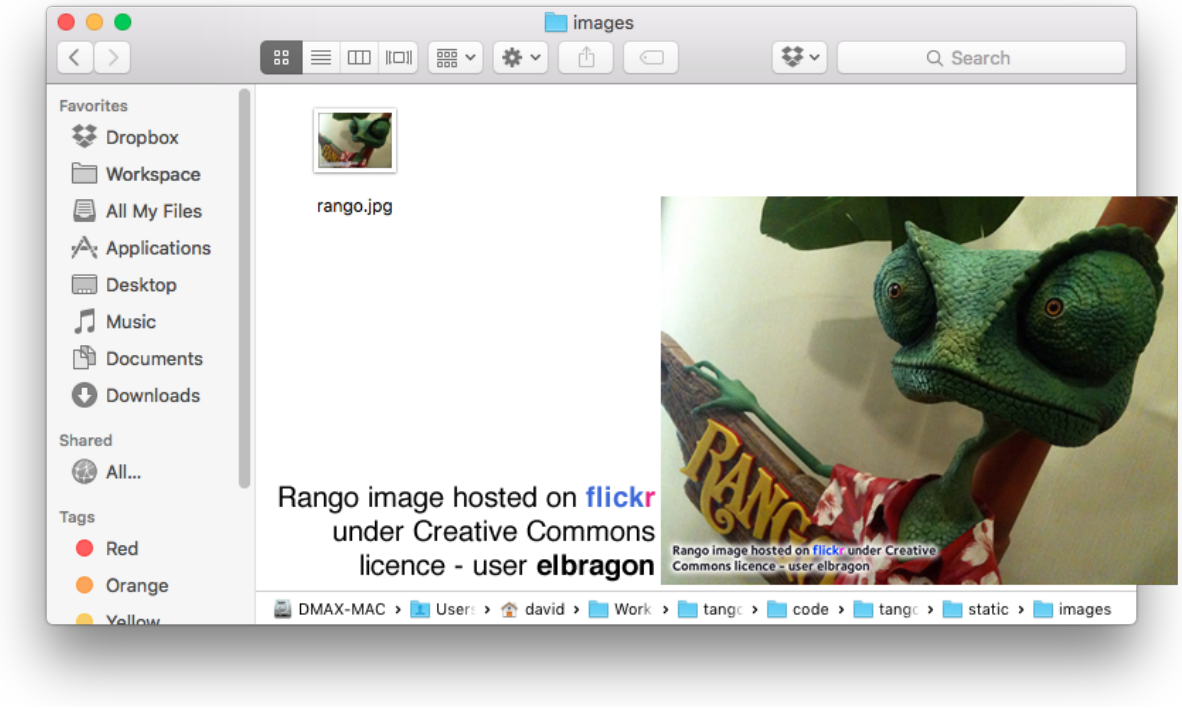
4.2 Serving Static Media Files

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

Configuring the Static Media Directory

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

Next, place an image inside the `images` directory. As shown in below, we chose a picture of the chameleon Rango - 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')
```

This will provide an absolute path to the location `<workspace>/tango_with_django_project/static/`. Once this variable has been created, 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** it doesn't before you create it. If you define it twice, you can start to confuse Django - and yourself.

For this book, we're only going to be using 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.

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

Keep `settings.py` Tidy!

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

Finally, check that the `STATIC_URL` variable is defined within your `settings.py` module. If it is not, then define it as shown below. Note that this variable by default in Django 1.9 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, and the third variable as the location with which clients can access static content.*

Test your Configuration

As a small exercise, test to see if everything is working correctly. Try and view the `rango.jpg` image in your browser when the Django development server is running. If your `STATIC_URL` is set to `/static/` and `rango.jpg` can be found at `images/rango.jpg`, what is the 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.

Don't Forget the Slashes!

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

Serving Static Content

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

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

Static Media Files and Templates

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

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 a HTML comment next to them for easy identification.

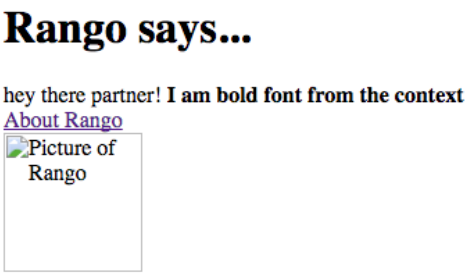
```
<!DOCTYPE html>

{% 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>
    </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 tag. 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:

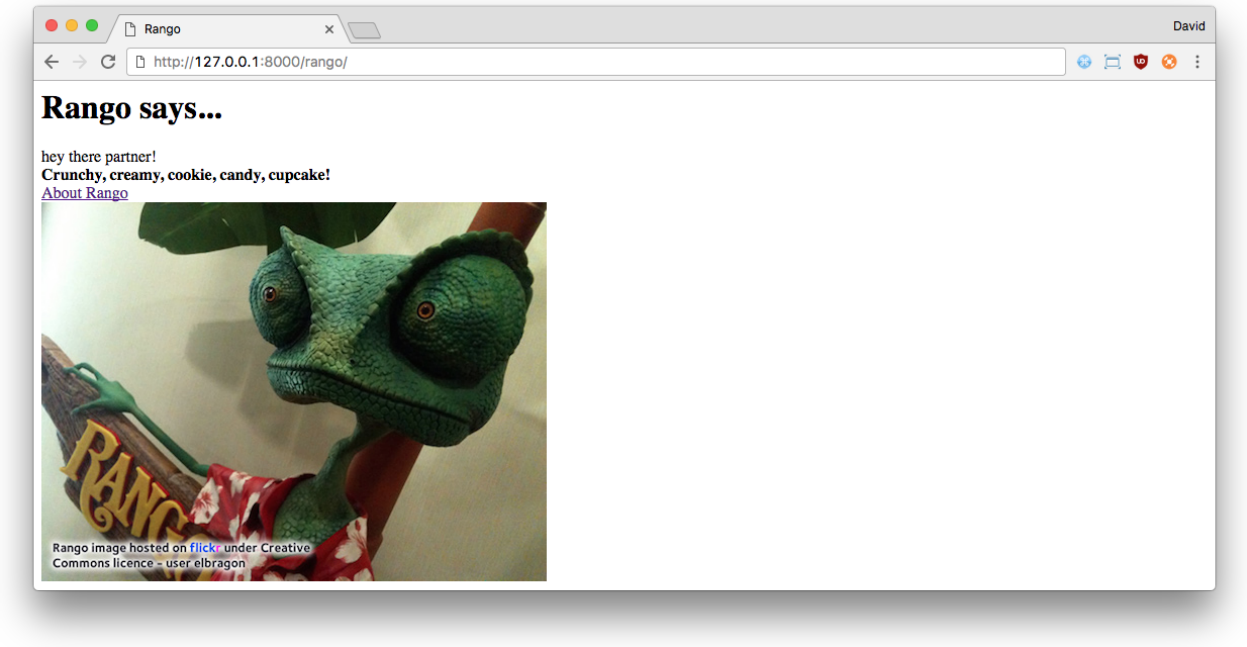
```
<img src="/static/images/rango.jpg" alt="Picture of Rango" />
```

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.

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



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

Templates and `<!DOCTYPE>`

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

Loading other Static Files

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

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

<html>

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

    <body>
        <!-- Image -->
        <img src="{% static "images/rango.jpg" %}" alt="Picture of Rango" />
    </body>

</html>
```

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` error. Try referencing a non-existent file and see what happens. Looking at the output snippet below, notice how the last entry's HTTP status code is `404`.

```
[10/Apr/2016 15:12:48] "GET /rango/ HTTP/1.1" 200 374
[10/Apr/2016 15:12:48] "GET /static/images/rango.jpg HTTP/1.1" 304 0
[10/Apr/2016 15:12:52] "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 templates.

4.3 Serving Media

Static media files can be considered files that don't change and are essential to your application. However, often you will have to store *media files* which are dynamic in nature. 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.

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

Serving Media Files

Like serving static content, Django provides the ability to serve media files in your development environment - 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 by some other means. The deployment chapter will discuss this in more detail.

Modifying `settings.py`

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

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

```
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.

```
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 uploaded file `cat.jpg` will for example be available on your Django development 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* 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 in `settings.py`. Within that list, look for the nested `context_processors` list, and within that list, add a new processor, `django.template.context_processors.media`. Your `context_processors` list should then look similar to 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'
],
```

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, and modifying it by appending a call to the `static()` function to your project's `urlpatterns` list.

```
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.

4.4 Basic Workflow

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

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

1. First, create the template you wish to use and save it within the `templates` directory you specified in your project's `settings.py` module. You may wish to use Django template variables (e.g. `{{ variable_name }}`) or template tags 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. If you haven't already done so, map the view to a URL by modifying your project's `urls.py` file and the application specific `urls.py` file if you have one.

The steps involved for 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" />`.

Exercises

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

- Convert the about page to use a template as well, using a template called `about.html`.
- Within the new `about.html` template, add a picture stored within your project's static files.
- On the about page, include a line that says, `This tutorial has been put together by <your-name>.`
- In your Django project directory, create a new directory called `media`, download a picture of a cat and save it to the `media` directory as `cat.jpg`.
- In your **about page**, add in the `<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 app has working examples of both static and media files.

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, 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.

5. Models and Databases

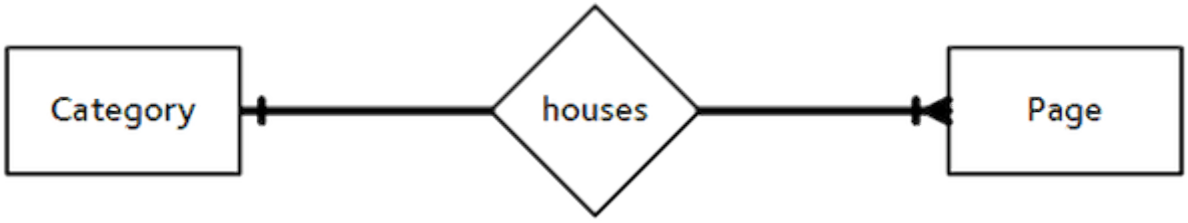
When you think of databases, you will usually think of the *Structured Query Language (SQL)*, the common means with which we query the database for the data we require. With Django, querying an underlying database - which can store all sorts of data, such as your website's user details - is taken care of by the *Object Relational Mapper (ORM)*. In essence, data stored within a database table can be encapsulated within a *model*. A model is a Python object that describes your database table's data. Instead of directly working on the database via SQL, you only need to manipulate the corresponding Python model object.

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 how straightforward it is to get data from the database to the Web browsers of your users.

5.1 Rango's Requirements

Before we get started, let's go over the data requirements for the Rango app that we are developing. Full requirements for the application are provided in detail earlier on, 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 a number of different *webpage categories* with each category housing a number of 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, a number of visits, and a number of likes.
- A page refers to a category, has a title, URL and a number of views.



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

5.2 Telling Django about Your Database

Before we can create any models, we need to set up our database with Django. In Django 1.9, a `DATABASES` variable is automatically created in your `settings.py` module when you set up a new project. It'll look similar to 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, SQLite (see the `ENGINE` option). The `NAME` entry for this database is the path to the database file, which is by default `db.sqlite3` in the root of your Django project.

Git Top Tip

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.

Using other Database Engines

The Django database framework has been created to cater for a variety of different database backends, such as PostgresSQL, MySQL and Microsoft's SQL Server. For other database engines, other keys like `USER`, `PASSWORD`, `HOST` and `PORT` exist for you to configure the database with Django.

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

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

5.3 Creating Models

With your database configured in `settings.py`, let's create the two initial data models for the Rango application. Models for a Django app are stored in the respective `models.py` module. This means that for Rango, models are stored within `rango/models.py`.

For the models themselves, we will create two classes - one class representing each model. Both must inherit 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):  # For Python 2, use __unicode__ too
        return self.name


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

    def __str__(self):  # For Python 2, use __unicode__ too
        return self.title
```

Check `import` Statements

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

`__str__()` or `__unicode__()`?

The `__str__()` and `__unicode__()` methods in Python generate a string representation of the class (similar to the `toString()` method in Java). In Python 2.x, strings are represented in ASCII format in the `__str__()` method. If you want Unicode support, then you need to also implement the `__unicode__()` method.

In Python 3.x, strings are Unicode by default - so you only need to implement the `__str__()` method.

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

Other Field Types

Check out the Django documentation on model fields 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 like 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`) (`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;
- `OneToOneField`, a field type that allows us to define a strict one-to-one relationship; and
- `ManyToManyField`, a field type which allows us to define a many-to-many relationship.

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.

Finally, it is good practice to implement the `__str__()` and/or `__unicode__()` methods. Without this method implemented when you go to `print` the object, it will show as `<Category: Category object>`. This isn't very useful when debugging or accessing the object - instead the code above will print, for example, `<Category: Python>` for the Python category. It is also helpful when we go to use the Admin Interface because Django will display the string representation of the object.

5.4 Creating and Migrating the Database

With our models defined in `models.py`, we can now let Django work its magic and create the tables in the underlying database. Django provides what is called a *migration tool* 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 creating the database and all the associated tables so that data can then be stored within it. To do this, you must open a terminal or command prompt, and navigate to your project's root directory - where `manage.py` is stored. Run the following command, *bearing in mind that the output may vary from what you see below.*

```
$ python manage.py migrate

Operations to perform:
  Apply all migrations: admin, contenttypes, auth, sessions
Running migrations:
  Rendering model states... DONE
  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 sessions.0001_initial... OK
```

All apps that are installed in your Django project (check `INSTALLED_APPS` in `settings.py`) will update 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 a 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':
  0001_initial.py:
    - Create model Category
    - Create model Page
```

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

Checking the Underlying SQL

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

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

In this example, `rango` is the name of your app, and `0001` is the migration you wish to view the SQL code for. Doing this allows you to get a better understanding of what exactly is going on at the database layer, such as what tables are created. You may 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, rango, contenttypes, auth, sessions
Running migrations:
  Rendering model states... DONE
  Applying rango.0001_initial... OK
```

This output confirms that the database tables have been created in your database, and you are good to go.

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 go through the migration process again.**

5.5 Django Models and the Shell

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

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

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

This will start an instance of the Python interpreter and load in your project's settings for you. You can then interact with the models, with the following terminal session demonstrating this functionality. Check out the inline commentary that we added to see what each command achieves. Note there are slight differences between what Django 1.9 and Django 1.10 return – these are both demonstrated below, complete with commentary.

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

# Show all the current categories
>>> print(Category.objects.all())
# The output examples below are for both Django 1.9 and Django 1.10.
# Both denote the same thing, that no categories have been defined.
[]  # Django 1.9 output -- an empty list.
<QuerySet []>  # Django 1.10 output -- an empty QuerySet object.

# 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())
# The output examples below are for both Django 1.9 and Django 1.10.
# You'll now see a 'test' category in both output examples.
[<Category: test>]  # Django 1.9
<QuerySet [<Category: test>]  # Django 1.10

# 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 your `__str__()` or `__unicode__()` method at work!

Complete the Official Tutorial

The example above is only a very basic taster on database related activities you can perform in the Django shell. If you have not done so already, it's now a good time to complete part two of the official Django Tutorial to learn more about interacting with models. Also check out the official Django documentation on the list of available commands for working with models.

5.6 Configuring the Admin Interface

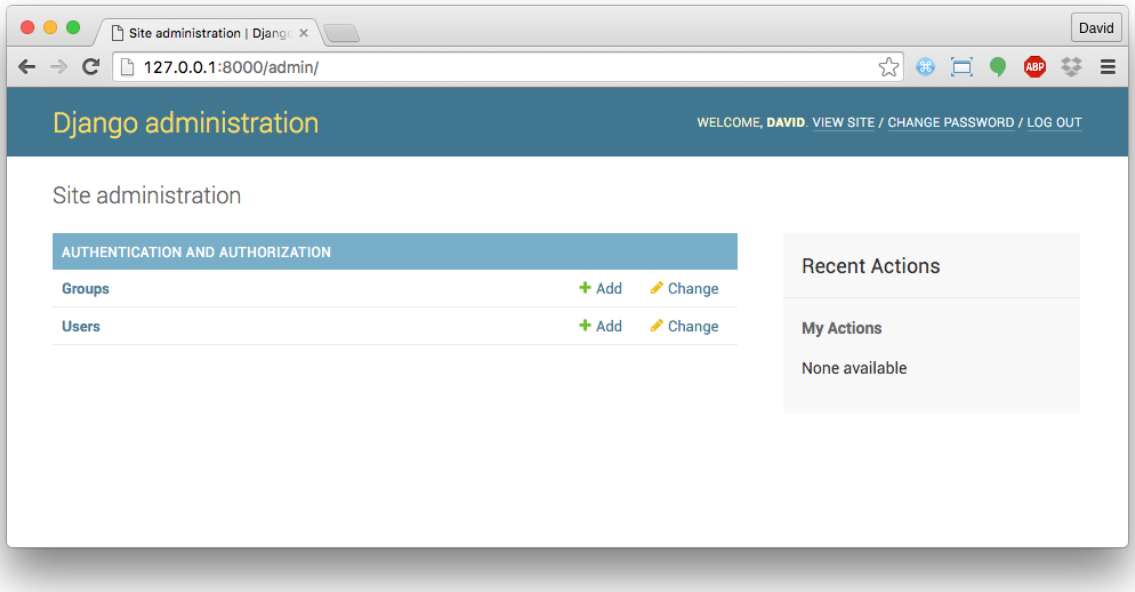
One of the standout features of Django is the built-in, Web-based administrative (or *admin*) interface that allows you to browse, edit and delete data represented as model instances (from the corresponding database tables). In this section, we'll be setting the admin interface up so you can see the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

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

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

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

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



The Django admin interface, sans Rango models.

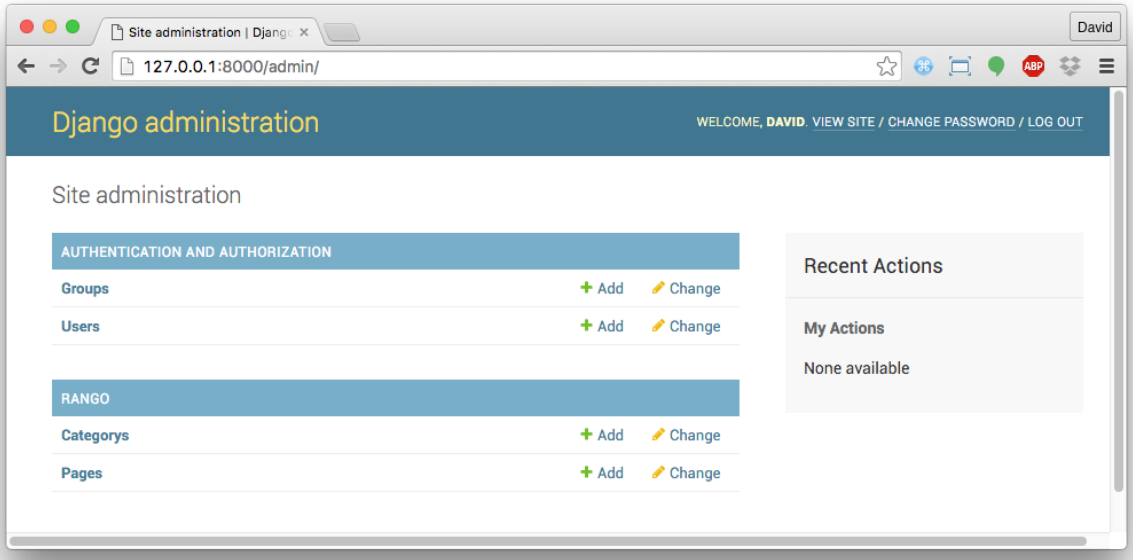
While this looks good, we are missing the `Category` and `Page` models that were defined for the Rango app. To include these models, we need to give Django some help.

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.

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



The Django admin interface, complete with Rango models.

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

Experiment with the Admin Interface

You'll be using the admin interface quite a bit to verify data is stored correctly as you develop the Rango app. 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 more example data.

User Management

The Django admin interface is your port of call for user management, through the Authentication and Authorisation section. Here, you can create, modify and delete user accounts, and varying privilege levels.

Plural vs. Singular Spellings

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

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

    class Meta:
        verbose_name_plural = 'Categories'

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

Expanding `admin.py`

It should be noted that the example `admin.py` module for your Rango app is the most simple, functional example available. However you can customise the Admin interface in a number of ways. Check out the official Django documentation on the admin interface for more information if you're interested.

5.7 Creating a Population Script

Entering test data into your database tends to be a hassle. Many developers will add in some bogus test data by randomly hitting keys, like `wTFzmN00bz7`. Rather than do this, it is better to write a script to automatically populate the database with realistic and credible data. This is because when you go to demo or test your app, you'll see some good examples in the database. Also, if you are deploying the app or sharing it with collaborators, then you/they won't have to go through the process of putting in sample data. 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 `populate_rango.py` file and add the following code.

```
import os
os.environ.setdefault('DJANGO_SETTINGS_MODULE',
                      'tango_with_django_project.settings')

import django
django.setup()
from rango.models import Category, Page

def populate():
    # First, we will create lists of dictionaries containing the pages
    # we want to add into each category.
    # Then we will create a dictionary of dictionaries for our categories.
    # This might seem a little bit confusing, but it allows us to iterate
    # through each data structure, and add the data to our models.

    python_pages = [
        {"title": "Official Python Tutorial",
         "url":"http://docs.python.org/2/tutorial/"},
        {"title":"How to Think like a Computer Scientist",
         "url":"http://www.greenteapress.com/thinkpython/"},
        {"title":"Learn Python in 10 Minutes",
         "url":"http://www.korokithakis.net/tutorials/python/"} ]

    django_pages = [
        {"title":"Official Django Tutorial",
         "url":"https://docs.djangoproject.com/en/1.9/intro/tutorial01/"},
        {"title":"Django Rocks",
         "url":"http://www.djangorocks.com/"},
        {"title":"How to Tango with Django",
         "url":"http://www.tangowithdjango.com/"} ]

    other_pages = [
        {"title":"Bottle",
         "url":"http://bottlepy.org/docs/dev/"},
        {"title":"Flask",
         "url":"http://flask.pocoo.org"} ]

    cats = {"Python": {"pages": python_pages},
            "Django": {"pages": django_pages},
            "Other Frameworks": {"pages": other_pages} }

    # If you want to add more catergories 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      # if you are using Python 2.x then use cats.iteritems() see
48      # http://docs.quantifiedcode.com/python-anti-patterns/readability/
49      # for more information about how to iterate over a dictionary properly.
50
51      for cat, cat_data in cats.items():
52          c = add_cat(cat)
53          for p in cat_data["pages"]:
54              add_page(c, p["title"], p["url"])
55
56      # Print out the categories we have added.
57      for c in Category.objects.all():
58          for p in Page.objects.filter(category=c):
59              print("- {0} - {1}".format(str(c), str(p)))
60
61  def add_page(cat, title, url, views=0):
62      p = Page.objects.get_or_create(category=cat, title=title)[0]
63      p.url=url
64      p.views=views
65      p.save()
66      return p
67
68  def add_cat(name):
69      c = Category.objects.get_or_create(name=name)[0]
70      c.save()
71      return c
72
73  # Start execution here!
74  if __name__ == '__main__':
75      print("Starting Rango population script...")
76      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 explanations below - hopefully you'll learn something new!

You should also note that when you see line numbers along side the code, it indicates that we have listed the entire file, rather than code fragments. It also makes things more difficult for you to copy and paste!

While this looks like a lot of code, what is going on is essentially a series of function calls to two small functions, `add_page()` and `add_cat()` 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__'`, we call the `populate()` function.

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

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

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

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 line 8.

The `for` loop occupying lines 51-54 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 52 above. This is stored because a `Page` requires a `Category` reference. After `add_cat()` and `add_page()` are called in `populate()`, the function concludes by looping through all new `Category` and associated `Page` objects, displaying their names on the terminal.

Creating Model Instances

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

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

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

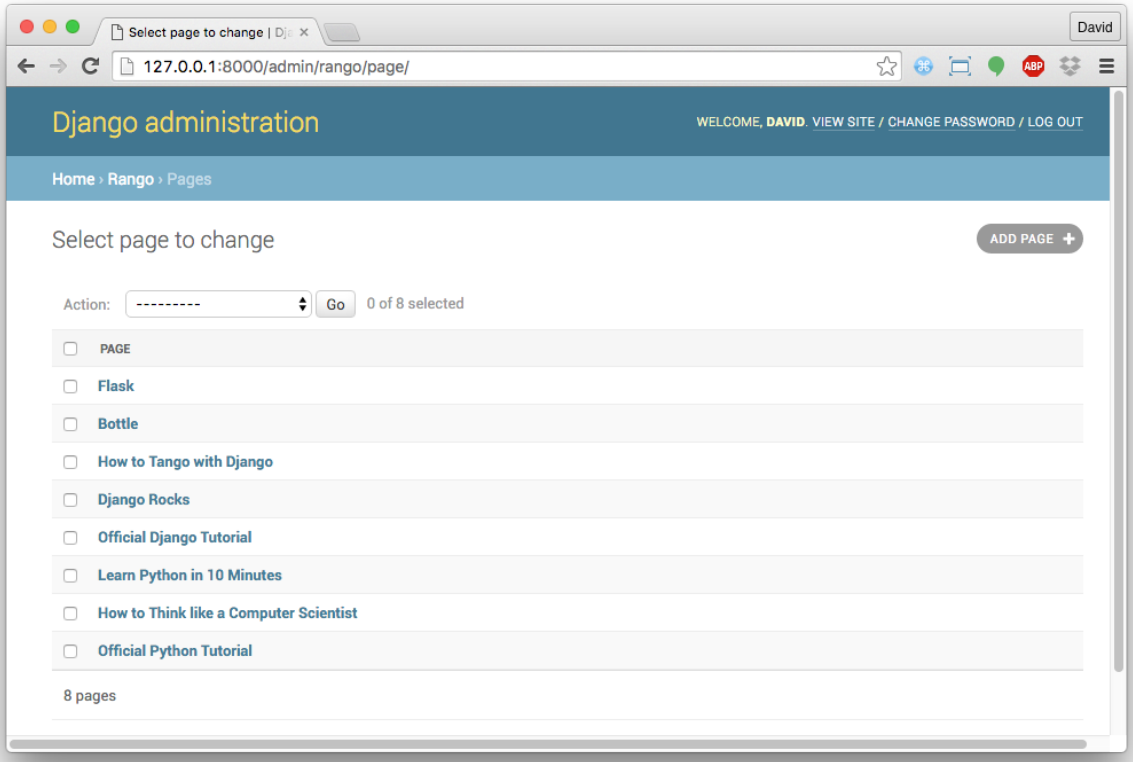
This explanation therefore means that the `[0]` at the end of our call to the `get_or_create()` returns the object reference only. Like most other programming language data structures, Python tuples use zero-based numbering.

You can check out the official Django documentation for more information on the handy `get_or_create()` method.

When saved, you can then run your new populations script by changing the present working directory in a terminal to the Django project's root. It's then a simple case of executing the command `$ python populate_rango.py`. You should then see 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 actually 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 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, you will save yourself time in the long run. When deploying your app elsewhere, running the population script after setting everything up means you can start demonstrating your app straight away. You'll also find it very handy when it comes to unit testing your code.

5.8 Workflow: Model Setup

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

Setting up your Database

With a new Django project, you should first tell Django about the database you intend to use (i.e. configure `DATABASES` in `settings.py`). You can also register any models in the `admin.py` module of your app to 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` to include and register your new model(s).
3. Perform the migration `$ python manage.py makemigrations <app_name>`.
4. Apply the changes `$ python manage.py migrate`. This will create the necessary infrastructure within the database for your new model(s).
5. Create/edit your population script for your new model(s).

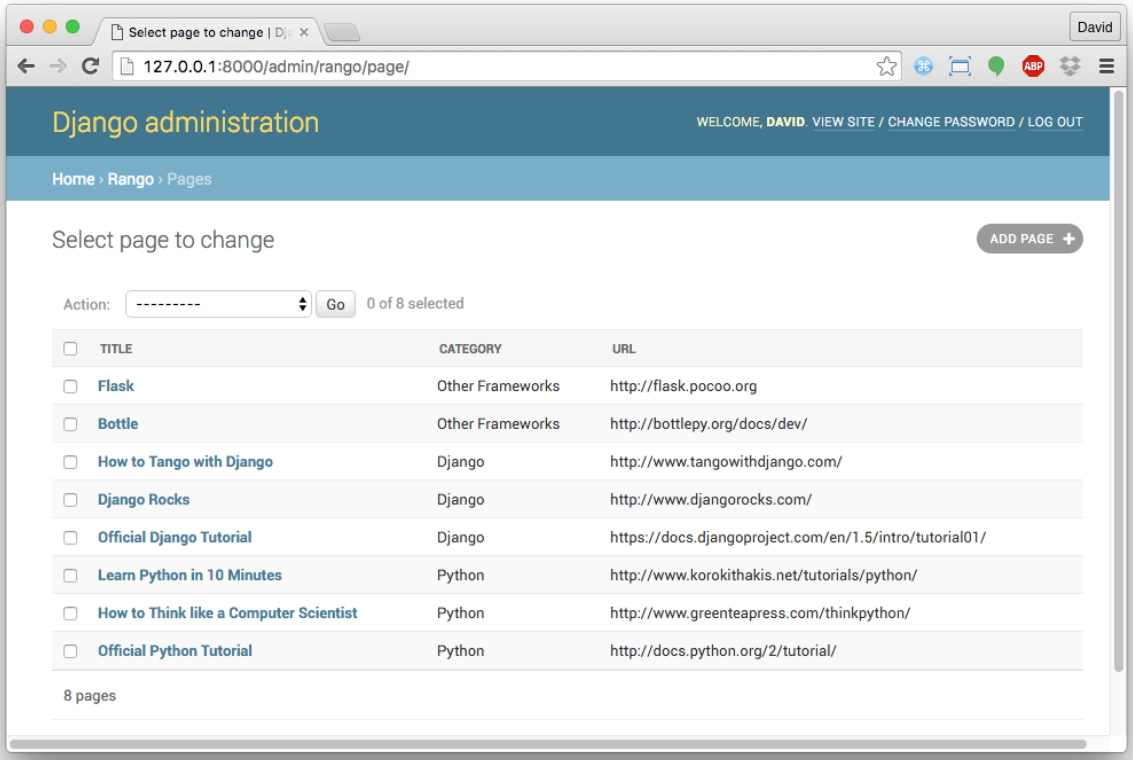
There will be times when you will have to delete your database – sometimes it's easier to just start afresh. When you want to do this, do the the following. Note that for this tutorial, you are using a SQLite database – Django does support a variety of other database engines.

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` 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.

Exercises

Now that you've completed this chapter, try out these exercises to reinforce and practice what you have learnt. **Once again, note that the following chapters will have expected you to have completed these exercises! If you're stuck, there are some hints to help you complete the exercises below**.

- Update the `Category` model to include the additional attributes `views` and `likes` where the `default` values for each are both zero (`0`).
- Make the migrations for your app and then migrate your database to commit the changes.
- 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 and seven of the official Django tutorial. These sections will reinforce what you've learnt on handling databases in Django, and show you additional techniques to customising the Django admin interface.
- 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. You will need to complete the previous exercises or at least go through the official Django Tutorial to complete this exercise.



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

Exercise Hints

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

- Modify the `Category` model by adding two `IntegerFields`: `views` and `likes`.
- In your population script, you can then modify the `add_cat()` function to manipulate the value of the new `views` and `likes` fields in the `Category` model.
  - 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 model instance is assigned to variable `c` in the population script, as defined earlier in this chapter. As an example, you can access the `likes` field using the notation `c.likes`. Don't forget to `save()` the instance!
  - You then need to update the `cats` dictionary in the `populate()` function of your population script. Look at the dictionary. Each key/value pairing represents the *name* of the category as the key, and an additional dictionary containing additional information relating to the category as the *value*. You'll want to modify this 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. 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.

Tests

We have written a few tests to check if you have completed the exercises. To check your work so far, download the `tests.py` script from our GitHub repository, and save it within your `rango` app directory.

To run the tests, issue the following command in the terminal or Command Prompt.

```
$ python manage.py test rango
```

If you are interested in learning about automated testing, now is a good time to check out the chapter on testing. The chapter runs through some of the basics on how you can write tests to automatically check the integrity of your code.

6. Models, Templates and Views

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

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

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

6.2 Showing Categories on Rango's Homepage

One of the requirements regarding the main page was to show the top five rango'ed categories. To fulfil this requirement, we will go through each of the above steps.

Importing Required Models

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

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

Modifying the Index View

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

```
def index(request):
    # Query the database for a list of ALL categories currently stored.
    # Order the categories by no. likes in descending order.
    # Retrieve the top 5 only - or all if less than 5.
    # Place the list in our context_dict dictionary
    # that will be passed to the template engine.

    category_list = Category.objects.order_by('-likes')[:5]
    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. You can see that it uses the `order_by()` method to sort by the number of `likes` in descending order. The `-` in `-likes` denotes that we would like them in descending order (if we removed the `-` then the results would be returned in ascending order). Since a list of `Category` objects will be returned, we used Python's list operators 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.

Warning

For this to work, you will have had to complete the exercises in the previous chapter where you need to add the field `likes` to the `Category` model.

Modifying the Index Template

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

```
<!DOCTYPE html>
{% load staticfiles %}
<html>
<head>
    <title>Rango</title>
</head>

<body>
    <h1>Rango says...</h1>
    <div>hey there partner!</div>

    <div>
    {% if categories %}
    <ul>
        {% for category in categories %}
            <li>{{ category.name }}</li>
        {% endfor %}
    </ul>
    {% else %}
        <strong>There are no categories present.</strong>
    {% endif %}
    </div>

    <div>
        <a href="/rango/about/">About Rango</a><br />
        <img src="{% static "images/rango.jpg" %}" alt="Picture of Rango" />
    </div>
</body>
</html>
```

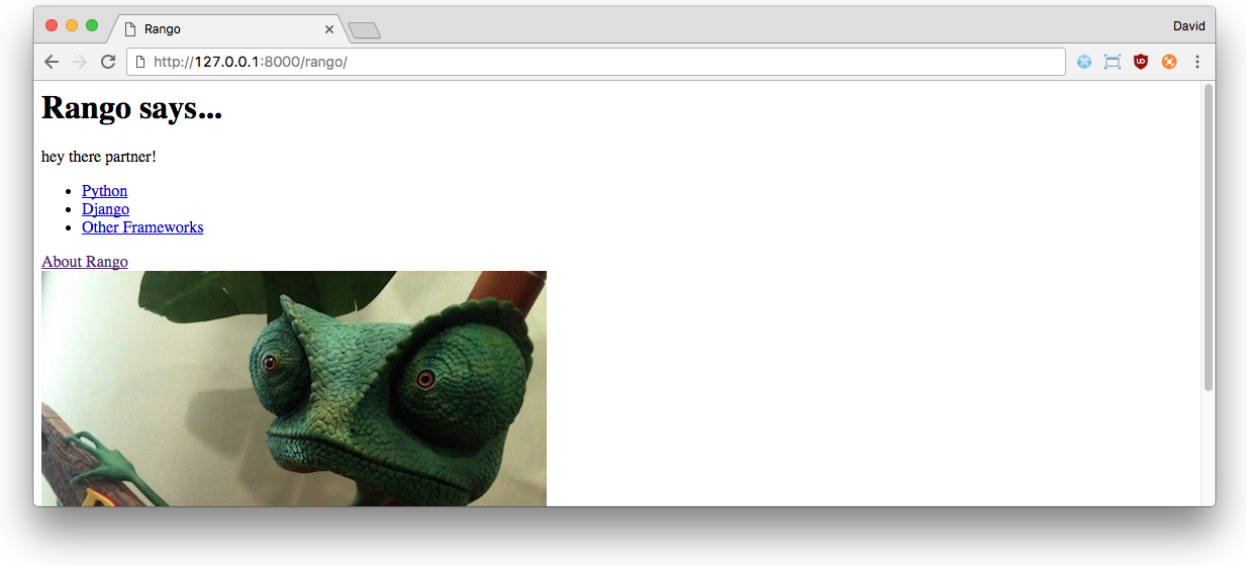
Here, we make use of Django's template language to present the data using `if` and `for` control statements. Within the `<body>` of the page, we test to see if `categories` - the name of the context variable containing our list - actually contains any categories (`{% if categories %}`).

If so, we proceed to construct an unordered HTML list (within the `<ul>` tags). The `for` loop (`{% for category in categories %}`) then iterates through the list of results, and outputs each category's name (`{{ category.name }}`) within a pair of `<li>` tags to indicate a list element.

If no categories exist, a message is displayed instead indicating that no categories are present.

As the example shows in Django's template language, all commands are enclosed within the tags `{%` and `%}`, while variables are referenced within `{{` and `}}` brackets.

If you now visit Rango's homepage at `<http://127.0.0.1:8000/rango/>`, you should see a list of categories underneath the page title just like in the figure below.



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

6.3 Creating a Details Page

According to the specifications for Rango, we also need to show a list of pages that are associated with each category. We have a number of challenges here to overcome. A new view must be created, which should be parameterised. We also need to create URL patterns and URL strings that encode category names.

URL Design and Mapping

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

Instead, we could use the category name as part of the URL. For example, we can imagine that the URL `/rango/category/python/` would lead us to a list of pages related to Python. This is a simple, readable and meaningful URL. If we go with this approach, we'll also have to handle categories that have multiple words, like 'Other Frameworks', etc.

Clean your URLs

Designing clean and readable URLs is an important aspect of web design. See Wikipedia's article on Clean URLs for more details.

To handle this problem we are going to make use of the `slugify` function provided by Django.

Update Category Table with a Slug Field

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

Unsafe URLs

While you can use spaces in URLs, it is considered to be unsafe to use them. Check out the Internet Engineering Task Force Memo on URLs to read more.

Next we need to override the `save()` method of the `Category` model, which we will call the `slugify` method and update the `slug` field with it. Note that every time the category name changes, the slug will also change. Update your model, as shown below, and add in the import.

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

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

    class Meta:
        verbose_name_plural = 'categories'

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

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

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

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

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

```
$ python populate_rango.py
```

Now run the development server with the command `$ python manage.py runserver`, and inspect the data in the models with the admin interface at `http://127.0.0.1:8000/admin/`.

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

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

To solve the first problem, we can either update our model so that the slug field allows blank entries, i.e.:

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

or we can customise the admin interface so that it automatically pre-populates the slug field as you type in the category name. To do this, update `rango/admin.py` with the following code.

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

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

Try out the admin interface and add in a new category.

Now that we have addressed the first problem, we can ensure that the slug field is also unique, by adding the constraint to the slug field.

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

Now that we have added in the slug field we can now use the slugs to uniquely identify each category. We could have added the unique constraint earlier, but if we performed the migration and set everything to be an empty string by default it would have raised an error. This is because the unique constraint would have been violated. We could have deleted the database and then recreated everything - but that is not always desirable.

Migration Woes

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

Sometimes it is just better to just delete the database and recreate everything than try and work out where the conflict is coming from. A neat exercise is to write a script to output the data in the database so that any changes you make can be saved out into a file that can be read in later.

Category Page Workflow

To implement the category pages so that they can be accessed via `/rango/category/<category-name-slug>/` we need to make a number of changes and undertake the following steps:

1. Import the `Page` model into `rango/views.py`.
2. Create a new view in `rango/views.py` called `show_category()`. The `show_category()` view will take an additional parameter, `category_name_url` which will store the encoded category name.

    - We will need helper functions to encode and decode the `category_name_url`.
3. Create a new template, `templates/rango/category.html`.
4. Update Rango's `urlpatterns` to map the new `category` view to a URL pattern in `rango/urls.py`.

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

Category View

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

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

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

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

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

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

        # Adds our results list to the template context under name pages.
        context_dict['pages'] = pages
        # We also add the category object from
        # the database to the context dictionary.
        # We'll use this in the template to verify that the category exists.
        context_dict['category'] = category
    except Category.DoesNotExist:
        # We get here if we didn't find the specified category.
        # Don't do anything -
```

```
        # the template will display the "no category" message for us.
        context_dict['category'] = None
        context_dict['pages'] = None

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

Our new view follows the same basic steps as our `index()` view. We first define a context dictionary and then attempt to extract the data from the models, and add the relevant data to the context dictionary. We determine which category by using the value passed as parameter `category_name_slug` to the `show_category()` view function. If the category slug is found in the `Category` model, we can then pull out the associated pages, and add this to the context dictionary, `context_dict`.

Category Template

Now let's create our template for the new view. In `<workspace>/tango_with_django_project/templates/rango/` directory, create `category.html`. In the new file, add the following code.

```
<!DOCTYPE html>
<html>
<head>
    <title>Rango</title>
</head>
<body>
    <div>
    {% if category %}
        <h1>{{ category.name }}</h1>
        {% if pages %}
            <ul>
            {% for page in pages %}
                <li><a href="{{ page.url }}">{{ page.title }}</a></li>
            {% endfor %}
            </ul>
        {% else %}
            <strong>No pages currently in category.</strong>
        {% endif %}
    {% else %}
        The specified category does not exist!
    {% endif %}
    </div>
</body>
</html>
```

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

If the `category` exists, then we check to see if there are any pages in the category. If so, we iterate through the pages using the `{% for page in pages %}` template tags. For each page in the `pages` list, we present their `title` and `url` attributes. This is displayed in an unordered HTML list (denoted by the `<ul>` tags). If you are not too familiar with HTML then check out the HTML Tutorial by W3Schools.com to learn more about the different tags.

Note on Conditional Template Tags

The Django template conditional tag - `{% if %}` - is a really neat way of determining the existence of an object within the template's context. Make sure you check the existence of an object to avoid errors.

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

Parameterised URL Mapping

Now let's have a look at how we actually pass the value of the `category_name_url` parameter to the `show_category()` function. To do so, we need to modify Rango's `urls.py` file and update the `urlpatterns` tuple as follows.

```
urlpatterns = [
    url(r'^$', views.index, name='index'),
    url(r'^about/$', views.about, name='about'),
    url(r'^category/(?P<category_name_slug>[\w\-]+)/$',
        views.show_category, name='show_category'),
]
```

We have added in a rather complex entry that will invoke `view.show_category()` when the URL pattern `r'^category/(?P<category_name_slug>[\w\-]+)/$'` is matched.

There are two things to note here. First we have added a parameter name with in the URL pattern, i.e. `<category_name_slug>`, which we will be able to access in our view later on. When you create a parameterised URL you need to ensure that the parameters that you include in the URL are declared in the corresponding view. The next thing to note is that the regular expression `[\w\-]+)` will look for any sequence of alphanumeric characters e.g. `a-z`, `A-Z`, or `0-9` denoted by `\w` and any hyphens (-) denoted by `\-`, and we can match as many of these as we like denoted by the `[ ]+` expression.

The URL pattern will match a sequence of alphanumeric characters and hyphens which are between the `rango/category/` and the trailing `/`. This sequence will be stored in the parameter `category_name_slug` and passed to `views.show_category()`. For example, the URL `rango/category/python-books/` would result in the `category_name_slug` having the value, `python-books`. However, if the URL was `rango/category/££££-$$$$$/` then the sequence of characters between `rango/category/` and the trailing `/` would not match the regular expression, and a `404 not found` error would result because there would be no matching URL pattern.

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

Regex Hell

"Some people, when confronted with a problem, think *'I know, I'll use regular expressions.'* Now they have two problems." Jamie Zawinski

Regular expressions may seem horrible and confusing at first, but there are tons of resources online to help you. This cheat sheet is an excellent resource for fixing problems with regular expressions.

Modifying the Index Template

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

```
<!DOCTYPE html>
{% load staticfiles %}
<html>
    <head>
        <title>Rango</title>
    </head>

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

        <div>
            hey there partner!
        </div>
```

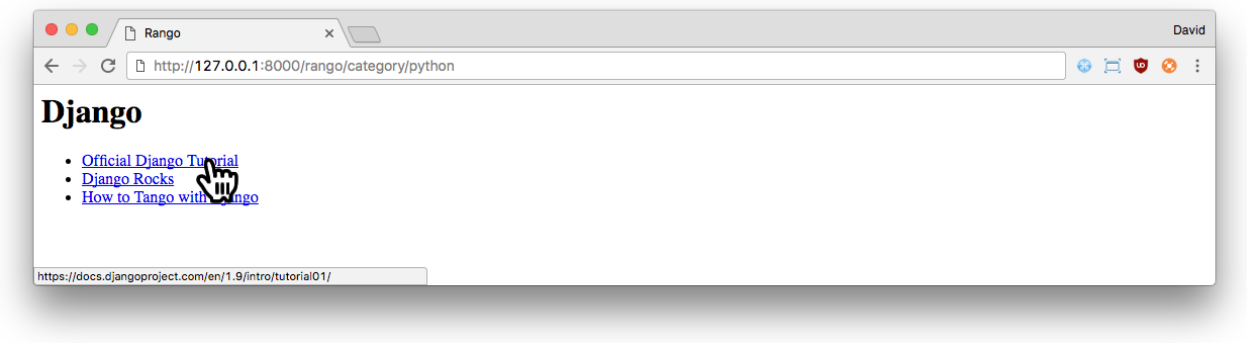
```
        <div>
        {% if categories %}
        <ul>
            {% for category in categories %}
            <!-- Following line changed to add an HTML hyperlink -->
            <li>
            <a href="/rango/category/{{ category.slug }}">{{ category.name }}</a>
            </li>
            {% endfor %}
        </ul>
        {% else %}
            <strong>There are no categories present.</strong>
        {% endif %}
        </div>

        <div>
            <a href="/rango/about/">About Rango</a><br />
            <img src="{% static "images/rango.jpg" %}" alt="Picture of Rango" />
        </div>
    </body>
</html>
```

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

Demo

Let's try everything out now by visiting the Rango homepage. You should see up to five categories on the index page. The categories should now be links. Clicking on `Django` should then take you to the `Django` category page, as shown in the figure below. If you see a list of links like `Official Django Tutorial`, then you've successfully set up the new page.

What happens when you visit a category that does not exist? Try navigating a category which doesn't exist, like `/rango/category/computers/`. Do this by typing the address manually into your browser's address bar. You should see a message telling you that the specified category does not exist.

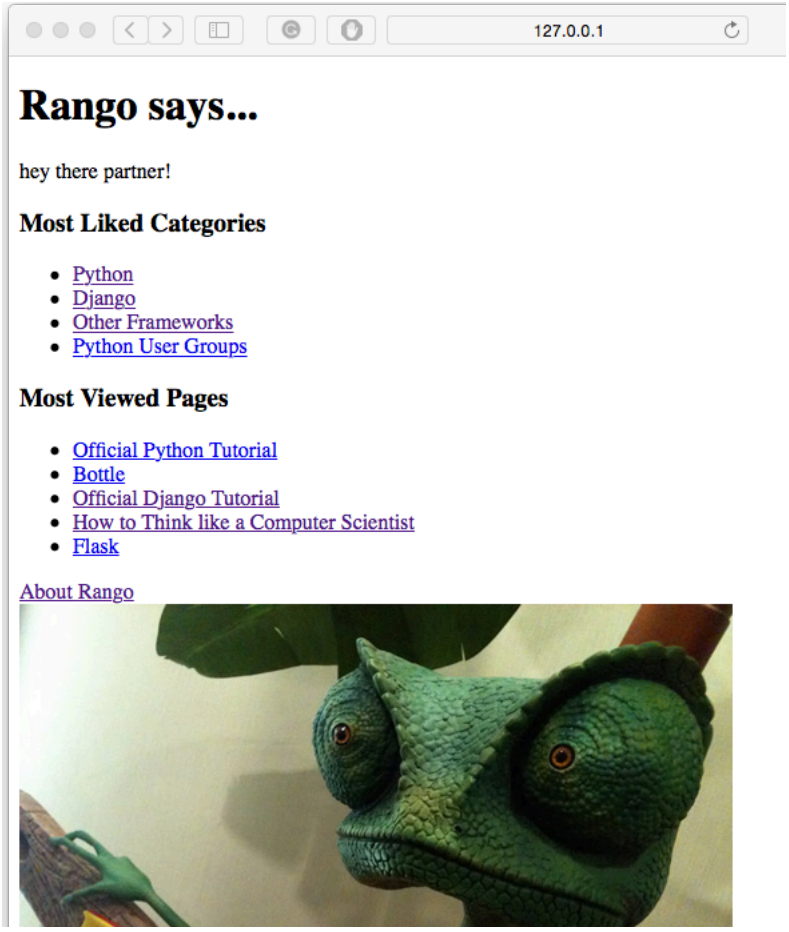


The links to Django pages. Note the mouse is hovering over the first link – you can see the corresponding URL for that link at the bottom left of the Google Chrome window.

Exercises

Reinforce what you've learnt in this chapter by trying out the following exercises.

- Update the population script to add some value to the `views` count for each page.
- Modify the index page to also include the top 5 most viewed pages.
- Include a heading for the "Most Liked Categories" and "Most Viewed Pages".
- Include a link back to the index page from the category page.
- Undertake part three of official Django tutorial if you have not done so already to reinforce what you've learnt here.



The index page after you complete the exercises, showing the most liked categories and most viewed pages.

Hints

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

Model Tips

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

1. Best Practices when working with models by Kostantin Moiseenko. In this post you will find a series of tips and tricks when working with models.
2. How to make your Django Models DRYer by Robert Roskam. In this post you can see how you can use the `property` method of a class to reduce the amount of code needed when accessing related models.

7. Forms

In this chapter, we will run through how to capture data through web forms. Django comes with some neat form handling functionality, making it a pretty straightforward process to collect information from users and save it to the database via the models. According to Django's documentation on forms, the form handling functionality allows you to:

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

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

7.1 Basic Workflow

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

1. If you haven't already got one, create a `forms.py` file within your Django application's directory to store form-related classes.
2. Create a `ModelForm` class for each model that you wish to represent as a form.
3. Customise the forms as you desire.
4. Create or update a view to handle the form
    - including *displaying* the form,
    - *saving* the form data, and
    - *flagging up errors* which may occur when the user enters incorrect data (or no data at all) in the form.
5. Create or update a template to display the form.
6. Add a `urlpattern` to map to the new view (if you created a newone).

This workflow is a bit more complicated than previous workflows, and the views that we have to construct have a lot more complexity as well. However, once you undertake the process a few times it will be pretty clear how everything pieces together.

7.2 Page and Category Forms

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

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

Creating `ModelForm` Classes

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

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

```
1  from django import forms
2  from rango.models import Page, Category
3
4  class CategoryForm(forms.ModelForm):
5      name = forms.CharField(max_length=128,
6                             help_text="Please enter the category name.")
7      views = forms.IntegerField(widget=forms.HiddenInput(), initial=0)
8      likes = forms.IntegerField(widget=forms.HiddenInput(), initial=0)
9      slug = forms.CharField(widget=forms.HiddenInput(), required=False)
10
11     # An inline class to provide additional information on the form.
12     class Meta:
13         # Provide an association between the ModelForm and a model
14         model = Category
15         fields = ('name',)
16
17 class PageForm(forms.ModelForm):
18     title = forms.CharField(max_length=128,
19                             help_text="Please enter the title of the page.")
20     url = forms.URLField(max_length=200,
21                          help_text="Please enter the URL of the page.")
22     views = forms.IntegerField(widget=forms.HiddenInput(), initial=0)
23
24     class Meta:
25         # Provide an association between the ModelForm and a model
```

```
26          model = Page
27
28          # What fields do we want to include in our form?
29          # This way we don't need every field in the model present.
30          # Some fields may allow NULL values, so we may not want to include them.
31          # Here, we are hiding the foreign key.
32          # we can either exclude the category field from the form,
33          exclude = ('category',)
34          # or specify the fields to include (i.e. not include the category field)
35          #fields = ('title', 'url', 'views')
```

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

Django provides us with a number of ways to customise the forms that are created on our behalf. In the code sample above, we've specified the widgets that we wish to use for each field to be displayed. For example, in our `PageForm` class, we've defined `forms.CharField` for the `title` field, and `forms.URLField` for `url` field. Both fields provide text entry for users. Note the `max_length` parameters we supply to our fields - the lengths that we specify are identical to the maximum length of each field we specified in the underlying data models. Go back to the chapter on models to check for yourself, or have a look at Rango's `models.py` file.

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

However, as you can see in the `PageForm`, despite the fact that we have a hidden field, we still need to include the field in the form. If in `fields` we excluded `views`, then the form would not contain that field (despite it being specified) and so the form would not return the value zero for that field. This may raise an error depending on how the model has been set up. If in the model we specified that the `default=0` for these fields then we can rely on the model to automatically populate field with the default value - and thus avoid a `not null` error. In this case, it would not be necessary to have these hidden fields. We have also included the field `slug` in the `CategoryForm`, and set it to use the `widget=forms.HiddenInput()`, but rather than specifying an initial or default value, we have said the field is not required by the form. This is because our model will be responsible on `save()` for populating this field. Essentially, you need to be careful when you define your models and forms to make sure that the form is going to contain and pass on all the data that is required to populate your model correctly.

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

Perhaps the most important aspect of a class inheriting from `ModelForm` is the need to define *which model we're wanting to provide a form for.* We take care of this through our nested `Meta` class. Set the `model` attribute of the nested `Meta` class to the model you wish to use. For example, our `CategoryForm` class has a reference to the `Category` model. This is a crucial step enabling Django to take care of creating a form in the image of the specified model. It will also help in handling flagging up any errors along with saving and displaying the data in the form.

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

More about Forms

Check out the official Django documentation on forms for further information about the different widgets and how to customise forms.

Creating an *Add Category* View

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

```
#Add this import at the top of the file
from rango.forms import CategoryForm
...
def add_category(request):
    form = CategoryForm()

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

        # Have we been provided with a valid form?
        if form.is_valid():
            # Save the new category to the database.
            form.save(commit=True)
            # Now that the category is saved
            # We could give a confirmation message
            # But since the most recent category added is on the index page
            # Then we can direct the user back to the index page.
            return index(request)
        else:
            # The supplied form contained errors -
            # just print them to the terminal.
```

```
            print(form.errors)

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

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

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

`GET` and `POST`

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

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

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

You'll notice from the line in which we call `render()` that we refer to a new template called `add_category.html`. This will contain the relevant Django template code and HTML for the form and page.

Creating the *Add Category* Template

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

```
<!DOCTYPE html>
<html>
    <head>
        <title>Rango</title>
    </head>

    <body>
        <h1>Add a Category</h1>
        <div>
            <form id="category_form" method="post" action="/rango/add_category/">
                {% csrf_token %}
                {% for hidden in form.hidden_fields %}
                    {{ hidden }}
                {% endfor %}
                {% for field in form.visible_fields %}
                    {{ field.errors }}
                    {{ field.help_text }}
                    {{ field }}
                {% endfor %}
                <input type="submit" name="submit" value="Create Category" />
            </form>
        </div>
    </body>
</html>
```

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

- one controlling *hidden* form fields, and
- the other *visible* form fields.

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

Hidden Fields

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

Cross Site Request Forgery Tokens

You should also take note of the code snippet `{% csrf_token %}`. This is a *Cross-Site Request Forgery (CSRF) token*, which helps to protect and secure the HTTP `POST` action that is initiated on the subsequent submission of a form. *The Django framework requires the CSRF token to be present. If you forget to include a CSRF token in your forms, a user may encounter errors when he or she submits the form.* Check out the official Django documentation on CSRF tokens (and this link for Django 1.10) for more information about this.

Mapping the *Add Category* View

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

```
urlpatterns = [
    url(r'^$', views.index, name='index'),
    url(r'about/$', views.about, name='about'),
    url(r'^add_category/$', views.add_category, name='add_category'),
    url(r'^category/(?P<category_name_slug>[\w\-]+)/$',
        views.show_category, name='show_category'),
]
```

Ordering doesn't necessarily matter in this instance. However, take a look at the official Django documentation on how Django process a request for more information. The URL for adding a category is `/rango/add_category/`.

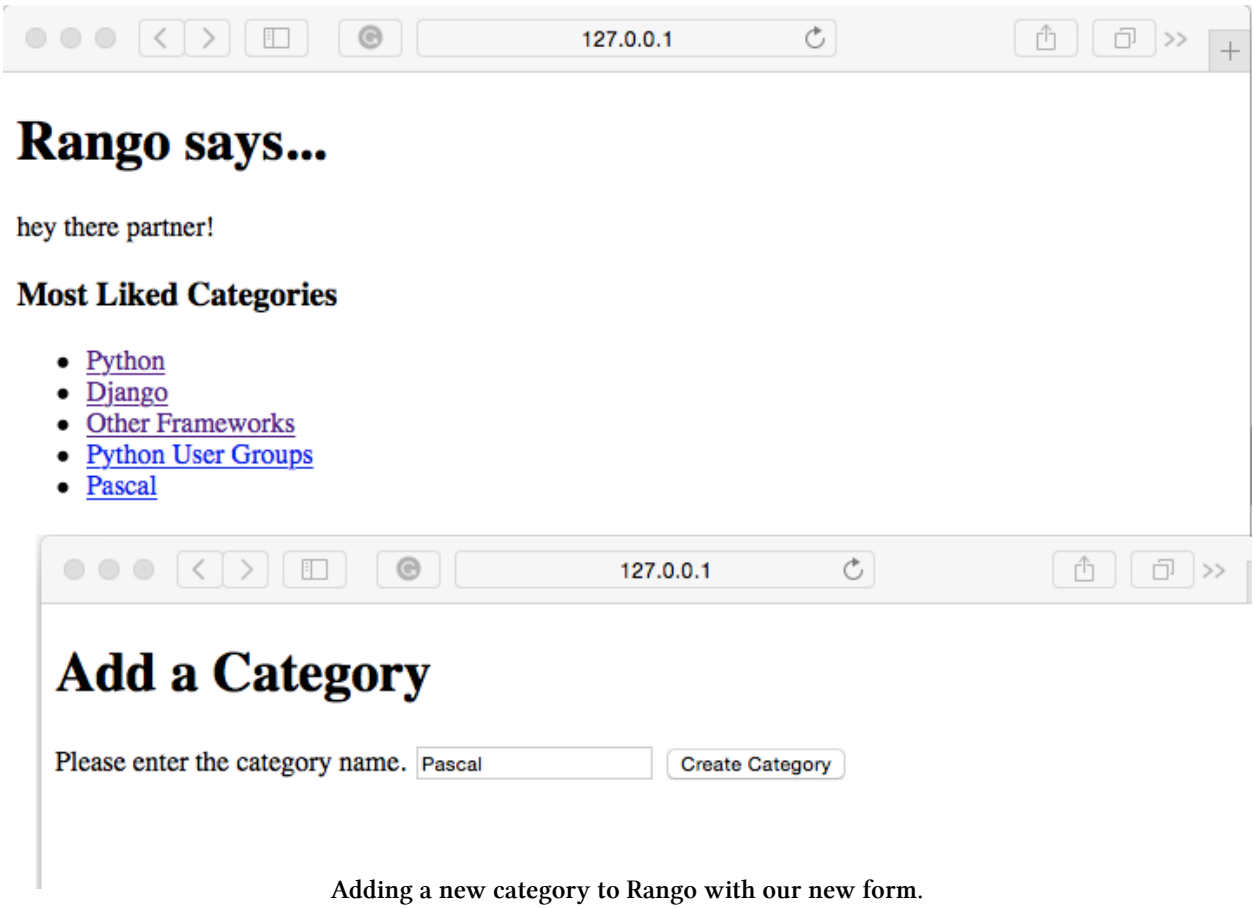
Modifying the Index Page View

As a final step let's put a link on the index page so that we can easily add categories. Edit the template `rango/index.html` and add the following HTML hyperlink in the `<div>` element with the about link.

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

Demo

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



Adding a new category to Rango with our new form.

Missing Categories

If you add a number of categories, they will not always appear on the index page. This is because we are only showing the top five categories on the index page. If you log into the Admin interface, you should be able to view all the categories that you have entered.

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

Cleaner Forms

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

URL Checking

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

In scenarios where user input may not be entirely correct, we can *override* the `clean()` method implemented in `ModelForm`. This method is called upon before saving form data to a new model instance, and thus provides us with a logical place to insert code which can verify - and even fix - any form data the user inputs. We can check if the value of `url` field entered by the user starts with `http://` - and if it doesn't, we can prepend `http://` to the user's input.

```
class PageForm(forms.ModelForm):
    ...
    def clean(self):
        cleaned_data = self.cleaned_data
        url = cleaned_data.get('url')

        # If url is not empty and doesn't start with 'http://',
        # then prepend 'http://'.
        if url and not url.startswith('http://'):
            url = 'http://' + url
            cleaned_data['url'] = url

            return cleaned_data
```

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

1. Form data is obtained from the `ModelForm` dictionary attribute `cleaned_data`.

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

This trivial example shows how we can clean the data being passed through the form before being stored. This is pretty handy, especially when particular fields need to have default values - or data within the form is missing, and we need to handle such data entry problems.

Clean Overrides

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

Exercises

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

- What would happen if you don't enter in a category name on the add category form?
- What happens when you try to add a category that already exists?
- What happens when you visit a category that does not exist? A hint for a potential solution to solving this problem can be found below.
- In the section above where we implemented our `ModelForm` classes, we repeated the `max_length` values for fields that we had previously defined in the models chapter. This is bad practice as we are *repeating ourselves!* How can you refactor your code so that you are *not* repeating the `max_length` values?
- If you have not done so already undertake part four of the official Django Tutorial to reinforce what you have learnt here.
- Now let users add pages to each category, see below for some example code and hints.

Creating an *Add Pages* View, Template and URL Mapping

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

- create a new view, `add_page()`,
- create a new template, `rango/add_page.html`,
- add a URL mapping, and
- update the category page/view to provide a link from the category add page functionality.

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

```
from rango.forms import PageForm

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

    form = PageForm()
    if request.method == 'POST':
        form = PageForm(request.POST)
        if form.is_valid():
            if category:
                page = form.save(commit=False)
                page.category = category
                page.views = 0
                page.save()
                return show_category(request, category_name_slug)
        else:
            print(form.errors)

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

Hints

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

- In the `add_page.html` template you can access the slug with `{{ category.slug }}` because the view passes the `category` object through to the template via the context dictionary.
- Ensure that the link only appears when *the requested category exists* - with or without pages. i.e. in the template check with `{% if cat %} .... {% else %} A category by this name does not exist {% endif %}`.
- Update Rango's `category.html` template with a new hyperlink with a line break immediately following it: `<a href="/rango/category/{{category.slug}}/add_page/">Add Page</a> <br/>`
- Make sure that in your `add_page.html` template that the form posts to `/rango/category/{{ category.slug }}/add_page/`.
- Update `rango/urls.py` with a URL mapping (`/rango/category/<category_name_slug>/add_page/`) to handle the above link.
- You can avoid the repetition of `max_length` parameters through the use of an additional attribute in your `Category` class. This attribute could be used to store the value for `max_length`, and then be referenced where required.

If you get *really* stuck, you can always check out our code on GitHub.

8. Working with Templates

So far, we've created several HTML templates for different pages within our Rango application. As you've created more and more templates, you may have noticed that a lot of the HTML code is actually repeated. We are violating the DRY Principle. Furthermore, you might have noticed that the way we have been referring to different pages using *hard coded* URL paths. Taken together, maintaining the site will be nightmare, because if we want to make a change to the general site structure or change a URL path, we will have to modify every template.

In this chapter, we will use *template inheritance* to overcome the first problem, and the *URL template tag* to solve the second problem. We will start with addressing the latter problem first.

8.1 Using Relative URLs in Templates

So far, we have been directly coding the URL of the page or view we want to show within the template, i.e. `<a href="/rango/about/">About</a>`. This kind of hard coding of URLs means that if we change our URL mappings in `urls.py`, then we will have to also change all of these URL references. The preferred way is to use the template tag `url` to look up the URL in the `urls.py` files and dynamically insert the URL path.

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

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

The Django template engine will look up any `urls.py` module for a URL pattern with the attribute `name` set to `about` (`name='about'`), and then reverse match the actual URL. This means if we change the URL mappings in `urls.py`, we don't have to go through all our templates and update them.

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

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

In this example, we must ensure that the app `rango` has the view `about`, contained within its `views.py` module.

In your app's `index.html` template, you will notice that you have a parameterised URL pattern (the `show_category` URL/view takes the `category.slug` as a parameter). To handle this, you can pass the `url` template tag the name of the URL/view and the slug within the template, as follows:

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

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

URLs and Multiple Django Apps

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

Django provides the ability to *namespace* URL configuration modules (e.g. `urls.py`) for each individual app that you employ in your project. Simply adding an `app_name` variable to your app's `urls.py` module is enough. The example below specifies the namespace for the Rango app to be `rango`.

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

app_name = 'rango'
urlpatterns = [
    url(r'^$', views.index, name='index'),
    ...
]
```

Adding an `app_name` variable would then mean that any URL you reference from the `rango` app could be done so like:

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

where the colon in the `url` command separates the namespace from the URL name. Of course, this is an advanced feature for when multiple apps are in presence - but it is a useful trick to know when things start to scale up.

8.2 Dealing with Repetition

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

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

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

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

Reoccurring HTML and The Base Template

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

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

<html>
    <head lang="en">
        <meta charset="UTF-8" />
        <title>Rango</title>
    </head>
    <body>
        <!-- Page specific content goes here -->
    </body>
</html>
```

For the time being, let's make this simple HTML page our app's base template. Save this markup in `base.html` within the `templates/rango/` directory (e.g. `templates/rango/base.html`).

`DOCTYPE` goes First!

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

Template Blocks

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

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

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

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

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

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

Extract Common Structures

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

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

Abstracting Further

Now that you have an understanding of blocks within Django templates, let's take the opportunity to abstract our base template a little bit further. Reopen the `rango/base.html` template and modify it to look like the following.

```
1  <!DOCTYPE html>
2  {% load staticfiles %}
3
4  <html>
5      <head>
6          <title>
7              Rango -
8              {% block title_block %}
9                  How to Tango with Django!
10             {% endblock %}
11         </title>
12     </head>
13     <body>
14         <div>
15             {% block body_block %}
16             {% endblock %}
17         </div>
18         <hr />
```

```
19          <div>
20              <ul>
21                  <li><a href="{% url 'add_category' %}">Add New Category</a></li>
22                  <li><a href="{% url 'about' %}">About</a></li>
23                  <li><a href="{% url 'index' %}">Index</a></li>
24              </ul>
25          </div>
26      </body>
27  </html>
```

From the example above, we have introduced two new features into the base template.

- The first is a template block called `title_block`. This will allow us to specify a custom page title for each page inheriting from our base template. If an inheriting page does not override the block, then the `title_block` defaults to `How to Tango with Django!`, resulting in a complete title of `Rango - How to Tango with Django!`. Look at the contents of the `<title>` tag in the above template to see how this works.
- We have also included the list of links from our current `index.html` template and placed them into a HTML `<div>` tag underneath our `body_block` block. This will ensure the links are present across all pages inheriting from the base template. The links are preceded by a *horizontal rule* (`<hr />`) which provides a visual separation for the user between the content of the `body_block` block and the links.

8.3 Template Inheritance

Now that we've created a base template with blocks, we can now update all the templates we have created so that they inherit from the base template. Let's start by refactoring the template `rango/category.html`.

To do this, first remove all the repeated HTML code leaving only the HTML and template tags/commands specific to the page. Then at the beginning of the template add the following line of code:

```
{% extends 'rango/base.html' %}
```

The `extends` command takes one parameter - the template that is to be extended/inherited from (i.e. `rango/base.html`). The parameter you supply to the `extends` command should be relative from your project's `templates` directory. For example, all templates we use for Rango should extend from `rango/base.html`, not `base.html`. We can then further modify the `category.html` template so it looks like the following complete example.

```
{% extends 'rango/base.html' %}
{% load staticfiles %}

{% block title_block %}
    {{ category.name }}
{% endblock %}

{% block body_block %}
    {% if category %}
        <h1>{{ category.name }}</h1>

        {% if pages %}
            <ul>
            {% for page in pages %}
                <li><a href="{{ page.url }}">{{ page.title }}</a></li>
            {% endfor %}
            </ul>
        {% else %}
            <strong>No pages currently in category.</strong>
        {% endif %}
        <a href="{% url 'add_page' category.slug %}">Add a Page</a>
    {% else %}
        The specified category does not exist!
    {% endif %}
{% endblock %}
```

Loading `staticfiles`

You'll need to make sure you add `{% load staticfiles %}` to the top of **each template** that makes use of static media. If you don't, you'll get an error! Django template modules must be imported individually for each template that requires them. If you've programmed before, this works somewhat differently from object orientated programming languages such as Java, where imports cascade down inheriting classes. Notice how we used the `url` template tag to refer to `rango/<category-name>/add_page/` URL pattern. The `category.slug` is passed through as a parameter to the `url` template tag and Django's Template Engine will produce the correct URL for us.

Now that we inherit from `rango/base.html`, the `category.html` template is much cleaner extending the `title_block` and `body_block` blocks. You don't need a well-formatted HTML document because `base.html` provides all the groundwork for you. All you're doing is plugging in additional content to the base template to create the complete, rendered HTML document that is sent to the client's browser. This rendered HTML document will then conform to the standards, containing components such as the document type declaration on the first line.

More about Templates

Here we have shown how we can minimise the repetition of structure HTML in our templates. However, the Django templating language is very powerful, and even lets you create your own template tags.

Templates can also be used to minimise code within your application's views. For example, if you wanted to include the same database driven content on each page of your application, you could construct a template that calls a specific view to handle the repeating portion of your app's pages. This then saves you from having to call the Django ORM functions that gather the required data for the template in every view that renders it.

If you haven't already done so, now would be a good time to read through the official Django documentation on templates.

Exercises

Now that you've worked through this chapter, there are a number of exercises that you can work through to reinforce what you've learnt regarding Django and templating.

- Update all other previously defined templates in the Rango app to extend from the new `base.html` template. Follow the same process as we demonstrated above. Once completed, your templates should all inherit from `base.html`.
- While you're at it, make sure you remove the links from our `index.html` template. We don't need them anymore! You can also remove the link to Rango's homepage within the `about.html` template.
- When you refactor the `index.html` keep the images that are served up from the static files and media server.
- Update all references to Rango URLs by using the `url` template tag. You can also do this in your `views.py` module too - check out the `reverse()` helper function.

Hints

- Start refactoring the `about.html` template first.
- Update the `title_block` then the `body_block` in each template.
- Have the development server running and check the page as you work on it. Don't change the whole page to find it doesn't work. Changing things incrementally and testing those changes as you go is a much safer solution.
- To reference the links to category pages, you can use the following template code, paying particular attention to the Django template `{% url %}` command.

```
<a href="{% url 'show_category' category.slug %}">{{ category.name }}</a>
```

8.4 The `render()` Method and the `request` Context

When writing views we have used a number of different methods, the preferred way is to use the Django shortcut method `render()`. The `render()` method requires that you pass through the `request` as the first argument. The `request` context houses a lot of information regarding the session, the user, etc, see the Official Django Documentation on Request objects. By passing the `request` through to the template mean that you will also have access to such information when creating templates. In the next chapter we will access information about the `user` - but for now check through all of your views and make sure that they have been implemented using the `render()` method. Otherwise, your templates won't have the information we need later on.

Render and Context

As a quick example of the checks you must carry out, have a look at the `about()` view. Initially, this was implemented with a hard-coded string response, as shown below. Note that we only send the string - we don't make use of the request passed as the `request` parameter.

```
def about(request):
    return HttpResponse('Rango says: Here is the about page.
                         <a href="/rango/">Index</a>')
```

To employ the use of a template, we call the `render()` function and pass through the `request` object. This will allow the template engine to access information such as the request type (e.g. `GET`/`POST`), and information relating to the user's status (have a look at Chapter 9).

```
def about(request):
    # prints out whether the method is a GET or a POST
    print(request.method)
    # prints out the user name, if no one is logged in it prints `AnonymousUser`
    print(request.user)
    return render(request, 'rango/about.html', {})
```

Remember, the last parameter of `render()` is the context dictionary with which you can use to pass additional data to the Django template engine. As we have no additional data to give to the template, we pass through an empty dictionary, `{}`.

8.5 Custom Template Tags

It would be nice to show the different categories that users can browse through in the sidebar on each page. Given what we have learnt so far, we could do the following:

- in the `base.html` template, we could add some code to display an item list of categories; and
- within each view, we could access the `Category` object, get all the categories, and return that in the context dictionary.

However, this is a pretty nasty solution because we will need to be repeatedly including the same code in all views. A DRYer solution would be to create custom template tags that are included in the template, and which can request *their own* data.

Using Template Tags

Create a directory `rango/templatetags`, and create two new modules within it. One must be called `__init__.py`, and this will be kept blank. Name the second module `rango_template_tags.py`. Add the following code to this second module.

```
from django import template
from rango.models import Category

register = template.Library()

@register.inclusion_tag('rango/cats.html')
def get_category_list():
    return {'cats': Category.objects.all()}
```

From this code snippet, you can see a new method called `get_category_list()`. This method returns a list of categories - but is mashed up with the template `rango/cats.html` (as can be seen from the `register.inclusion_tag()` decorator). You can now create this template file, and add the following HTML markup:

```
<ul>
{% if cats %}
    {% for c in cats %}
        <li><a href="{% url 'show_category'  c.slug %}">{{ c.name }}</a></li>
    {% endfor %}
{% else %}
    <li><strong>There are no categories present.</strong></li>
{% endif %}
</ul>
```

To use the template tag in your `base.html` template, first load the custom template tag by including the command `{% load rango_template_tags %}` at the top of the `base.html` template. You can then create a new block to represent the sidebar - and we can call our new template tag with the following code.

```
<div>
    {% block sidebar_block %}
        {% get_category_list %}
    {% endblock %}
</div>
```

Try it out. Now all pages that inherit from `base.html` will also include the list of categories (which we will move to the side later on).

Restart the Server!

You'll need to restart the Django development server (or ensure it restarted itself) every time you modify template tags. If the server doesn't restart, Django won't register the tags.

Parameterised Template Tags

We can also *parameterise* the template tags we create, allowing for greater flexibility. As an example, we'll use parameterisation to highlight which category we are looking at when visiting its page. Adding in a parameter is easy - we can update the `get_category_list()` method as follows.

```
def get_category_list(cat=None):
    return {'cats': Category.objects.all(),
            'act_cat': cat}
```

Note the inclusion of the `cat` parameter to `get_category_list()`, which is optional - and if you don't pass in a category, `None` is used as the subsequent value.

We can then update our `base.html` template which makes use of the custom template tag to pass in the current category - but only if it exists.

```
<div>
    {% block sidebar_block %}
        {% get_category_list category %}
    {% endblock %}
</div>
```

We can also now update the `cats.html` template, too.

```
{% for c in cats %}
    {% if c == act_cat %}
        <li>
        <strong>
              <a href="{% url 'show_category' c.slug %}">{{ c.name }}</a>
        </strong>
        </li>
    {% else  %}
        <li>
            <a href="{% url 'show_category' c.slug %}">{{ c.name }}</a>
        </li>
    {% endif %}
{% endfor %}
```

In the template, we check to see if the category being displayed is the same as the category being passed through during the `for` loop (i.e. `c == act_cat`). If so, we highlight the category name by making it **bold** through use of the `<strong>` tag.

8.6 Summary

In this chapter, we showed how we can:

- reduce coupling between URLs and templates by using the `url` template tag to point to relative URLs;
- reduced the amount of boilerplate code by using template inheritance; and
- avoid repetitive code appearing in views by creating custom templates tags.

Taken together, your template code should be much cleaner and easier to maintain. Of course, Django templates offer a lot more functionality - find out more by visiting the Official Django Documentation on Templates.

9. User Authentication

The aim of this next part of the tutorial is to get you familiar with the user authentication mechanisms provided by Django. We'll be using the `auth` app provided as part of a standard Django installation, located in package `django.contrib.auth`. According to Django's official documentation on Authentication, the application provides the following concepts and functionality.

- The concept of a *User* and the *User* Model.
- *Permissions*, a series of binary flags (e.g. yes/no) that determine what a user may or may not do.
- *Groups*, a method of applying permissions to more than one user.
- A configurable *password hashing system*, a must for ensuring data security.
- *Forms and view tools for logging in users*, or restricting content.

There's lots that Django can do for you regarding user authentication. In this chapter, we'll be covering the basics to get you started. This will help you build your confidence with the available tools and their underlying concepts.

9.1 Setting up Authentication

Before you can begin to play around with Django's authentication offering, you'll need to make sure that the relevant settings are present in your Rango project's `settings.py` file.

Within the `settings.py` file find the `INSTALLED_APPS` list and check that `django.contrib.auth` and `django.contrib.contenttypes` are listed, so that it looks similar to the code below:

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

While `django.contrib.auth` provides Django with access to the provided authentication system, the package `django.contrib.contenttypes` is used by the authentication app to track models installed in your database.

Migrate, if necessary!

If you had to add `django.contrib.auth` and `django.contrib.contenttypes` applications to your `INSTALLED_APPS` tuple, you will need to update your database with the `$ python manage.py migrate` command. This will add the underlying tables to your database e.g. a table for the `User` model.

It's generally good practice to run the `migrate` command whenever you add a new app to your Django project - the app could contain models that'll need to be synchronised to your underlying database.

9.2 Password Hashing

Storing passwords as plaintext within a database is something that should never be done under any circumstances. If the wrong person acquired a database full of user accounts to your app, they could wreak havoc. Fortunately, Django's `auth` app by default stores a hash of user passwords using the PBKDF2 algorithm, providing a good level of security for your user's data. However, if you want more control over how the passwords are hashed, you can change the approach used by Django in your project's `settings.py` module, by adding in a tuple to specify the `PASSWORD_HASHERS`. An example of this is shown below.

```
PASSWORD_HASHERS = (
    'django.contrib.auth.hashers.PBKDF2PasswordHasher',
    'django.contrib.auth.hashers.PBKDF2SHA1PasswordHasher',
)
```

Django considers the order of hashers specified as important, and will pick and use the first password hasher in `PASSWORD_HASHERS` (e.g. `settings.PASSWORD_HASHERS[0]`). If other password hashers are specified in the tuple, Django will also use these if the first hasher doesn't work.

If you want to use a more secure hasher, you can install Bcrypt using `pip install bcrypt`, and then set the `PASSWORD_HASHERS` to be:

```
PASSWORD_HASHERS = [
    'django.contrib.auth.hashers.BCryptSHA256PasswordHasher',
    'django.contrib.auth.hashers.BCryptPasswordHasher',
    'django.contrib.auth.hashers.PBKDF2PasswordHasher',
    'django.contrib.auth.hashers.PBKDF2SHA1PasswordHasher',
]
```

As previously mentioned, Django by default uses the PBKDF2 algorithm to hash passwords. If you do not specify a `PASSWORD_HASHERS` tuple in `settings.py`, Django will use the `PBKDF2PasswordHasher` password hasher, by default. You can read more about password hashing in the official Django documentation on how Django stores passwords.

9.3 Password Validators

As people may be tempted to enter a password that is comparatively easy to guess, a welcome new feature introduced to Django 1.9 is that of password validation. In your Django project's `settings.py` module, you will notice a list of nested dictionaries with the name `AUTH_PASSWORD_VALIDATORS`. From the nested dictionaries, you can clearly see that Django 1.9 comes with a number of pre-built password validators for common password checks, such as length. An `OPTIONS` dictionary can be specified for each validator, allowing for easy customisation. If, for example, you wanted to ensure accepted passwords are at least six characters long, you can set `min_length` of the `MinimumLengthValidator` password validator to `6`. This can be seen in the example shown below.

```
AUTH_PASSWORD_VALIDATORS = [
    ...
    {
        'NAME': 'django.contrib.auth.password_validation.MinimumLengthValidator',
        'OPTIONS': { 'min_length': 6, }
    },
    ...
]
```

It is also possible to create your own password validators. Although we don't cover the creation of custom password validators in this tutorial, refer to the official Django documentation on password validators for more information.

9.4 The `User` Model

The `User` object (located at `django.contrib.auth.models.User`) is considered to be the core of Django's authentication system. A `User` object represents each of the individuals interacting with a

Django application. The Django documentation on User objects states that they are used to allow aspects of the authentication system like access restriction, registration of new user profiles, and the association of creators with site content.

The `User` model has five key attributes. They are:

- the *username* for the user account;
- the account's *password*;
- the user's *email address*;
- the user's *first name*; and
- the user's *surname*.

The `User` model also comes with other attributes such as `is_active`, `is_staff` and `is_superuser`. These are boolean fields used to denote whether the account is active, owned by a staff member, or has superuser privileges respectively. Check out the official Django documentation on the user model for a full list of attributes provided by the base `User` model.

9.5 Additional `User` Attributes

If you would like to include other user related attributes than what is provided by the `User` model, you will needed to create a model that is *associated* with the `User` model. For our Rango app, we want to include two more additional attributes for each user account. Specifically, we wish to include:

- a `URLField`, allowing a user of Rango to specify their own website; and
- a `ImageField`, which allows users to specify a picture for their user profile.

This can be achieved by creating an additional model in Rango's `models.py` file. Let's add a new model called `UserProfile`:

```
class UserProfile(models.Model):
    # This line is required. Links UserProfile to a User model instance.
    user = models.OneToOneField(User)

    # The additional attributes we wish to include.
    website = models.URLField(blank=True)
    picture = models.ImageField(upload_to='profile_images', blank=True)

    # Override the __unicode__() method to return out something meaningful!
    # Remember if you use Python 2.7.x, define __unicode__ too!
    def __str__(self):
        return self.user.username
```

Note that we reference the `User` model using a one-to-one relationship. Since we reference the default `User` model, we need to import it within the `models.py` file:

```
from django.contrib.auth.models import User
```

For Rango, we've added two fields to complete our user profile, and provided a `__str__()` method to return a meaningful value when a unicode representation of a `UserProfile` model instance is requested. Remember, if you are using Python 2, you'll also need to provide a `__unicode__()` method to return a unicode variant of the user's username.

For the two fields `website` and `picture`, we have set `blank=True` for both. This allows each of the fields to be blank if necessary, meaning that users do not have to supply values for the attributes.

Furthermore, it should be noted that the `ImageField` field has an `upload_to` attribute. The value of this attribute is conjoined with the project's `MEDIA_ROOT` setting to provide a path with which uploaded profile images will be stored. For example, a `MEDIA_ROOT` of `<workspace>/tango_with_django_project/media/` and `upload_to` attribute of `profile_images` will result in all profile images being stored in the directory `<workspace>/tango_with_django_project/media/profile_images/`. Recall that in the chapter on templates and media files we set up the media root there.

What about Inheriting to Extend?

It may have been tempting to add the additional fields defined above by inheriting from the `User` model directly. However, because other applications may also want access to the `User` model, it not recommended to use inheritance, but to instead use a one-to-one relationship within your database.

Take the PIL

The Django `ImageField` field makes use of the *Python Imaging Library (PIL)*. If you have not done so already, install PIL via Pip with the command `pip install pillow`. If you don't have `jpeg` support enabled, you can also install PIL with the command `pip install pillow --global-option="build_ext" --global-option="--disable-jpeg"`.

You can check what packages are installed in your (virtual) environment by issuing the command `pip list`.

To make the `UserProfile` model data accessible via the Django admin Web interface, import the new `UserProfile` model into Rango's `admin.py` module.

```
from rango.models import UserProfile
```

Now you can register the new model with the admin interface, with the following line.

```
admin.site.register(UserProfile)
```

Once again, Migrate!

Remember that your database must be updated with the creation of a new model. Run: `$ python manage.py makemigrations rango` from your terminal or Command Prompt to create the migration scripts for the new `UserProfile` model. Then run: `$ python manage.py migrate` to execute the migration which creates the associated tables within the underlying database.

9.6 Creating a *User Registration* View and Template

With our authentication infrastructure laid out, we can now begin to build on it by providing users of our application with the opportunity to create user accounts. We can achieve this by creating a new view, template and URL mapping to handle user registrations.

Django User Registration Applications

It is important to note that there are several off the shelf user registration applications available which reduce a lot of the hassle of building your own registration and login forms.

However, it's a good idea to get a feeling for the underlying mechanics before using such applications. This will ensure that you have some sense of what is going on under the hood. *No pain, no gain.* It will also reinforce your understanding of working with forms, how to extend upon the `User` model, and how to upload media files.

To provide user registration functionality, we will now work through the following steps:

- create a `UserForm` and `UserProfileForm`;
- add a view to handle the creation of a new user;
- create a template that displays the `UserForm` and `UserProfileForm`; and
- map a URL to the view created.

As a final step to integrate our new registration functionality, we will also:

- link the index page to the register page.

Creating the `UserForm` and `UserProfileForm`

In `rango/forms.py`, we now need to create two classes inheriting from `forms.ModelForm`. We'll be creating one for the base `User` class, as well as one for the new `UserProfile` model that we just created. The two `ModelForm`-inheriting classes allow us to display a HTML form displaying the necessary form fields for a particular model, taking away a significant amount of work for us.

In `rango/forms.py`, let's first create our two classes which inherit from `forms.ModelForm`. Add the following code to the module.

```
class UserForm(forms.ModelForm):
    password = forms.CharField(widget=forms.PasswordInput())

    class Meta:
        model = User
        fields = ('username', 'email', 'password')

class UserProfileForm(forms.ModelForm):
    class Meta:
        model = UserProfile
        fields = ('website', 'picture')
```

You'll notice that within both classes, we added a nested `Meta` class. As the name of the nested class suggests, anything within a nested `Meta` class describes additional properties about the particular class to which it belongs. Each `Meta` class must supply a `model` field. In the case of the `UserForm` class the associated model is the `User` model. You also need to specify the `fields` or the fields to `exclude`, to indicate which fields associated with the model should be present (or not) on the rendered form.

Here, we only want to show the fields `username`, `email` and `password` associated with the `User` model, and the `website` and `picture` fields associated with the `UserProfile` model. For the `user` field within `UserProfile` model, we will need to make this association when we register the user. This is because when we create a `UserProfile` instance, we won't yet have the `User` instance to refer to.

You'll also notice that `UserForm` includes a definition of the `password` attribute. While a `User` model instance contains a `password` attribute by default, the rendered HTML form element will not hide the password. If a user types a password, the password will be visible. By updating the `password` attribute, we can specify that the `CharField` instance should hide a user's input from prying eyes through use of the `PasswordInput()` widget.

Finally, remember to include the required classes at the top of the `forms.py` module! We've listed them below for your convenience.

```
from django import forms
from django.contrib.auth.models import User
from rango.models import Category, Page, UserProfile
```

Creating the `register()` View

Next, we need to handle both the rendering of the form and the processing of form input data. Within Rango's `views.py`, add `import` statements for the new `UserForm` and `UserProfileForm` classes.

```
from rango.forms import UserForm, UserProfileForm
```

Once you've done that, add the following new view, `register()`.

```
def register(request):
    # A boolean value for telling the template
    # whether the registration was successful.
    # Set to False initially. Code changes value to
    # True when registration succeeds.
    registered = False

    # If it's a HTTP POST, we're interested in processing form data.
    if request.method == 'POST':
        # Attempt to grab information from the raw form information.
        # Note that we make use of both UserForm and UserProfileForm.
        user_form = UserForm(data=request.POST)
        profile_form = UserProfileForm(data=request.POST)

        # If the two forms are valid...
        if user_form.is_valid() and profile_form.is_valid():
            # Save the user's form data to the database.
            user = user_form.save()

            # Now we hash the password with the set_password method.
            # Once hashed, we can update the user object.
            user.set_password(user.password)
            user.save()

            # Now sort out the UserProfile instance.
            # Since we need to set the user attribute ourselves,
            # we set commit=False. This delays saving the model
            # until we're ready to avoid integrity problems.
```

```
            profile = profile_form.save(commit=False)
            profile.user = user

            # Did the user provide a profile picture?
            # If so, we need to get it from the input form and
            #put it in the UserProfile model.
            if 'picture' in request.FILES:
                profile.picture = request.FILES['picture']

            # Now we save the UserProfile model instance.
            profile.save()

            # Update our variable to indicate that the template
            # registration was successful.
            registered = True
        else:
            # Invalid form or forms - mistakes or something else?
            # Print problems to the terminal.
            print(user_form.errors, profile_form.errors)
    else:
        # Not a HTTP POST, so we render our form using two ModelForm instances.
        # These forms will be blank, ready for user input.
        user_form = UserForm()
        profile_form = UserProfileForm()

    # Render the template depending on the context.
    return render(request,
                  'rango/register.html',
                  {'user_form': user_form,
                   'profile_form': profile_form,
                   'registered': registered})
```

While the view looks pretty complicated, it's actually very similar to how we implemented the add category and add page views. However, here we have to also handle two distinct `ModelForm` instances - one for the `User` model, and one for the `UserProfile` model. We also need to handle a user's profile image, if he or she chooses to upload one.

Furthermore, we need to establish a link between the two model instances that we have created. After creating a new `User` model instance, we reference it in the `UserProfile` instance with the line `profile.user = user`. This is where we populate the `user` attribute of the `UserProfileForm` form, which we hid from users.

Creating the *Registration* Template

Now we need to make the template that will be used by the new `register()` view. Create a new template file, `rango/register.html`, and add the following code.

```
{% extends 'rango/base.html' %}
{% load staticfiles %}

{% block title_block %}
    Register
{% endblock %}

{% block body_block %}
    <h1>Register for Rango</h1>
    {% if registered %}
        Rango says: <strong>thank you for registering!</strong>
        <a href="{% url 'index' %}">Return to the homepage.</a><br />
    {% else %}
        Rango says: <strong>register here!</strong><br />
        <form id="user_form" method="post" action="{% url 'register' %}"
              enctype="multipart/form-data">

        {% csrf_token %}

        <!-- Display each form  -->
        {{ user_form.as_p }}
        {{ profile_form.as_p }}

        <!-- Provide a button to click to submit the form. -->
        <input type="submit" name="submit" value="Register" />
    </form>
    {% endif %}
{% endblock %}
```

Using the `url` Template Tag

Note that we are using the `url` template tag in the above template code e.g. `{% url 'register' %}`. This means we will have to ensure that when we map the URL, we name it `register`.

The first thing to note here is that this template makes use of the `registered` variable we used in our view indicating whether registration was successful or not. Note that `registered` must be `False` in order for the template to display the registration form - otherwise the success message is displayed.

Next, we have used the `as_p` template function on the `user_form` and `profile_form`. This wraps each element in the form in a paragraph (denoted by the `<p>` HTML tag). This ensures that each element appears on a new line.

Finally, in the `<form>` element, we have included the attribute `enctype`. This is because if the user tries to upload a picture, the response from the form may contain binary data - and may be quite large. The response therefore will have to be broken into multiple parts to be transmitted back to the server. As such, we need to denote this with `enctype="multipart/form-data"`. This tells the HTTP client (the web browser) to package and send the data accordingly. Otherwise, the server won't receive all the data submitted by the user.

Multipart Messages and Binary Files

You should be aware of the `enctype` attribute for the `<form>` element. When you want users to upload files from a form, it's an absolute *must* to set `enctype` to `multipart/form-data`. This attribute and value combination instructs your browser to send form data in a special way back to the server. Essentially, the data representing your file is split into a series of chunks and sent. For more information, check out this great Stack Overflow answer.

Furthermore, remember to include the CSRF token, i.e. `{% csrf_token %}` within your `<form>` element! If you don't do this, Django's cross-site forgery protection middleware layer will refuse to accept the form's contents, returning an error.

The `register()` URL Mapping

With our new view and associated template created, we can now add in the URL mapping. In Rango's URLs module `rango/urls.py`, modify the `urlpatterns` tuple as shown below.

```
urlpatterns = [
    url(r'^$', views.index, name='index'),
    url(r'about/$', views.about, name='about'),
    url(r'^add_category/$', views.add_category, name='add_category'),

    url(r'^category/(?P<category_name_slug>[\w\-]+)/$',
        views.show_category,
        name='show_category'),

    url(r'^category/(?P<category_name_slug>[\w\-]+)/add_page/$',
        views.add_page,
        name='add_page'),

    url(r'^register/$',
        views.register,
```

```
        name='register'), # New pattern!
]
```

The newly added pattern (at the bottom of the list) points the URL `/rango/register/` to the `register()` view. Also note the inclusion of a `name` for our new URL, `register`, which we used in the template when we used the `url` template tag, e.g. `{% url 'register' %}`.

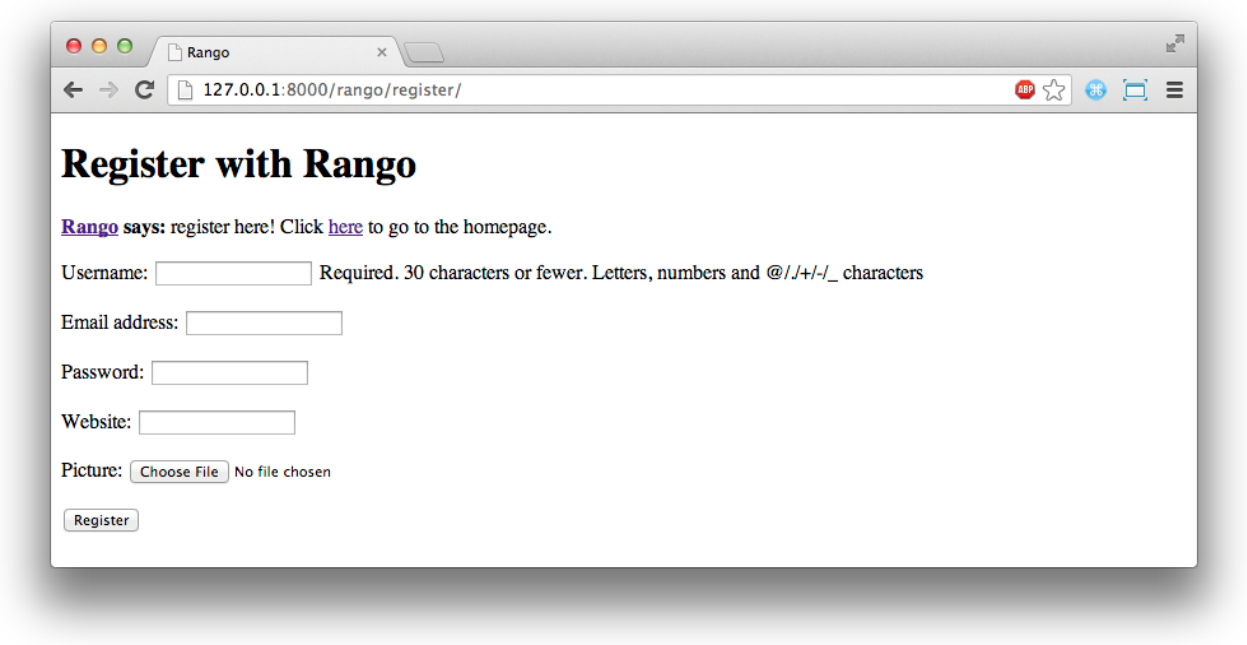
Linking Everything Together

Finally, we can add a link pointing to our new registration URL by modifying the `base.html` template. Update `base.html` so that the unordered list of links that will appear on each page contains a link allowing users to register for Rango.

```
<ul>
        <li><a href="{% url 'add_category' %}">Add a New Category</a></li>
        <li><a href="{% url 'about' %}">About</a></li>
        <li><a href="{% url 'index' %}">Index</a></li>
        <li><a href="{% url 'register' %}">Sign Up</a></li>
</ul>
```

Demo

Now everything is plugged together, try it out. Start your Django development server and try to register as a new user. Upload a profile image if you wish. Your registration form should look like the one illustrated in the figure below.



A screenshot illustrating the basic registration form you create as part of this tutorial.

Upon seeing the message indicating your details were successfully registered, the database should have a new entry in the `User` and `UserProfile` models. Check that this is the case by going into the Django Admin interface.

9.7 Implementing Login Functionality

With the ability to register accounts completed, we now need to provide users of Rango with the ability to login. To achieve this, we'll need to undertake the workflow below:

- Create a login in view to handle the processing of user credentials
- Create a login template to display the login form
- Map the login view to a URL
- Provide a link to login from the index page

Creating the `login()` View

First, open up Rango's views module at `rango/views.py` and create a new view called `user_login()`. This view will handle the processing of data from our subsequent login form, and attempt to log a user in with the given details.

```
def user_login(request):
    # If the request is a HTTP POST, try to pull out the relevant information.
    if request.method == 'POST':
        # Gather the username and password provided by the user.
        # This information is obtained from the login form.
        # We use request.POST.get('<variable>') as opposed
        # to request.POST['<variable>'], because the
        # request.POST.get('<variable>') returns None if the
        # value does not exist, while request.POST['<variable>']
        # will raise a KeyError exception.
        username = request.POST.get('username')
        password = request.POST.get('password')

        # Use Django's machinery to attempt to see if the username/password
        # combination is valid - a User object is returned if it is.
        user = authenticate(username=username, password=password)

        # If we have a User object, the details are correct.
        # If None (Python's way of representing the absence of a value), no user
        # with matching credentials was found.
        if user:
            # Is the account active? It could have been disabled.
            if user.is_active:
                # If the account is valid and active, we can log the user in.
                # We'll send the user back to the homepage.
                login(request, user)
                return HttpResponseRedirect(reverse('index'))
            else:
                # An inactive account was used - no logging in!
                return HttpResponse("Your Rango account is disabled.")
        else:
            # Bad login details were provided. So we can't log the user in.
            print("Invalid login details: {0}, {1}".format(username, password))
            return HttpResponse("Invalid login details supplied.")

    # The request is not a HTTP POST, so display the login form.
    # This scenario would most likely be a HTTP GET.
    else:
        # No context variables to pass to the template system, hence the
        # blank dictionary object...
        return render(request, 'rango/login.html', {})
```

As before, this view may seem rather complex as it has to handle a variety of scenarios. As shown

in previous examples, the `user_login()` view handles form rendering and processing - where the form this time contains `username` and `password` fields.

First, if the view is accessed via the HTTP `GET` method, then the login form is displayed. However, if the form has been posted via the HTTP `POST` method, then we can handle processing the form.

If a valid form is sent via a `POST` request, the username and password are extracted from the form. These details are then used to attempt to authenticate the user. The Django function `authenticate()` checks whether the username and password provided actually match to a valid user account. If a valid user exists with the specified password, then a `User` object is returned, otherwise `None` is returned.

If we retrieve a `User` object, we can then check if the account is active or inactive - if active, then we can issue the Django function `login()`, which officially signifies to Django that the user is to be logged in.

However, if an invalid form is sent - due to the fact that the user did not add both a username and password - the login form is presented back to the user with error messages (i.e. an invalid username/password combination was provided).

You'll also notice that we make use of a new class, `HttpResponseRedirect`. As the name may suggest to you, the response generated by an instance of the `HttpResponseRedirect` class tells the client's Web browser to redirect to the URL you provide as the argument. Note that this will return a HTTP status code of `302`, which denotes a redirect, as opposed to an status code of `200` (success). See the official Django documentation on Redirection to learn more.

Finally, we use another Django method called `reverse` to obtain the URL of the Rango application. This looks up the URL patterns in Rango's `urls.py` module to find a URL called `'index'`, and substitutes in the corresponding pattern. This means that if we subsequently change the URL mapping, our new view won't break.

Django provides all of these functions and classes. As such, you'll need to import them. The following `import` statements must now be added to the top of `rango/views.py`.

```
from django.contrib.auth import authenticate, login
from django.http import HttpResponseRedirect, HttpResponse
from django.core.urlresolvers import reverse
```

Creating a *Login* Template

With our new view created, we'll need to create a new template allowing users to enter their credentials. While we know that the template will live in the `templates/rango/` directory, we'll leave you to figure out the name of the file. Look at the code example above to work out the name based upon the code for the new `user_login()` view. In your new template file, add the following code.

```
{% extends 'rango/base.html' %}
{% load staticfiles %}

{% block title_block %}
    Login
{% endblock %}

{% block body_block %}
<h1>Login to Rango</h1>
<form id="login_form" method="post" action="{% url 'login' %}">
    {% csrf_token %}
    Username: <input type="text" name="username" value="" size="50" />
    <br />
    Password: <input type="password" name="password" value="" size="50" />
    <br />
    <input type="submit" value="submit" />
</form>
{% endblock %}
```

Ensure that you match up the input `name` attributes to those that you specified in the `user_login()` view. For example, `username` matches to the username, and `password` matches to the user's password. Don't forget the `{% csrf_token %}`, either!

Mapping the Login View to a URL

With your login template created, we can now match up the `user_login()` view to a URL. Modify Rango's `urls.py` module so that the `urlpatterns` list contains the following mapping.

```
url(r'^login/$', views.user_login, name='login'),
```

Linking Together

Our final step is to provide users of Rango with a handy link to access the login page. To do this, we'll edit the `base.html` template inside of the `templates/rango/` directory. Add the following link to your list.

```
<ul>
    ...
    <li><a href="{% url 'login' %}">Login</a><li>
</ul>
```

If you like, you can also modify the header of the index page to provide a personalised message if a user is logged in, and a more generic message if the user isn't. Within the `index.html` template, find the message, as shown in the code snippet below.

```
hey there partner!
```

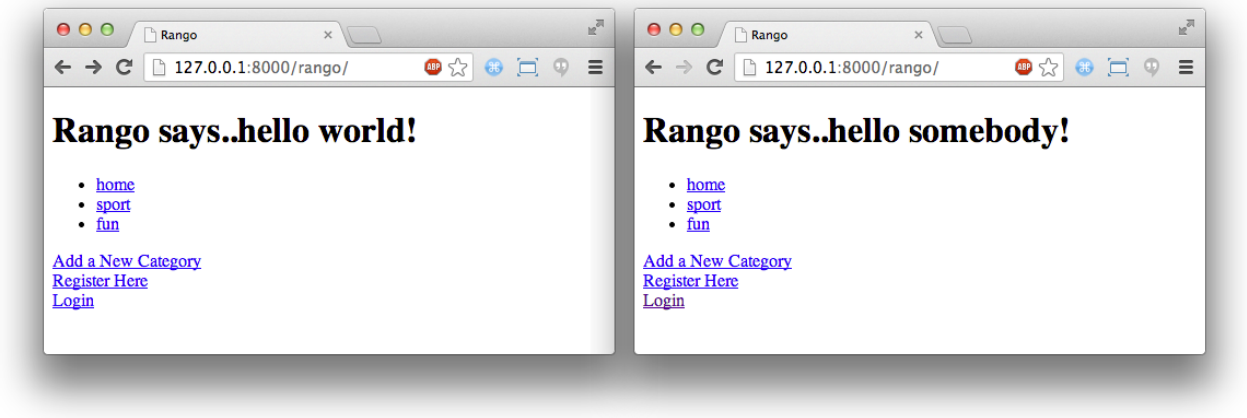
This line can then be replaced with the following code.

```
{% if user.is_authenticated %}
    howdy {{ user.username }}!
{% else %}
    hey there partner!
{% endif %}
```

As you can see, we have used Django's template language to check if the user is authenticated with `{% if user.is_authenticated %}`. If a user is logged in, then Django gives us access to the `user` object. We can tell from this object if the user is logged in (authenticated). If he or she is logged in, we can also obtain details about him or her. In the example about, the user's username will be presented to them if logged in - otherwise the generic `hey there partner!` message will be shown.

Demo

Start the Django development server and attempt to login to the application. The figure below shows the screenshots of the login and index page.



Screenshots illustrating the header users receive when not logged in, and logged in with username `somebody`.

With this completed, user logins should now be working. To test everything out, try starting Django's development server and attempt to register a new account. After successful registration, you should then be able to login with the details you just provided.

9.8 Restricting Access

Now that users can login to Rango, we can now go about restricting access to particular parts of the application as per the specification, i.e. that only registered users can add categories and pages. With Django, there are several ways in which we can achieve this goal.

- In the template, we could use the `{% if user.authenticated %}` template tag to modify how the page is rendered (shown already).
- In the View, we could directly examine the `request` object and check if the user is authenticated.
- Or, we could use a *decorator* function `@login_required` provided by Django that checks if the user is authenticated.

The direct approach checks to see whether a user is logged in, via the `user.is_authenticated()` method. The `user` object is available via the `request` object passed into a view. The following example demonstrates this approach.

```
def some_view(request):
    if not request.user.is_authenticated():
        return HttpResponse("You are logged in.")
    else:
        return HttpResponse("You are not logged in.")
```

The third approach uses Python decorators. Decorators are named after a software design pattern by the same name. They can dynamically alter the functionality of a function, method or class without having to directly edit the source code of the given function, method or class.

Django provides a decorator called `login_required()`, which we can attach to any view where we require the user to be logged in. If a user is not logged in and attempts to access a view decorated with `login_required()`, they are then redirected to another page (that you can set) - typically the login page.

Restricting Access with a Decorator

To try this out, create a view in Rango's `views.py` module called `restricted()`, and add the following code

```
@login_required
def restricted(request):
    return HttpResponse("Since you're logged in, you can see this text!")
```

Note that to use a decorator, you place it *directly above* the function signature, and put a `@` before naming the decorator. Python will execute the decorator before executing the code of your function/method. As a decorator is still a function, you'll still have to import it if it resides within an external module. As `login_required()` exists elsewhere, the following `import` statement is required at the top of `views.py`.

```
from django.contrib.auth.decorators import login_required
```

We'll also need to add in another pattern to Rango's `urlpatterns` list in the `urls.py` file. Add the following line of code.

```
url(r'^restricted/', views.restricted, name='restricted'),
```

We'll also need to handle the scenario where a user attempts to access the `restricted()` view, but is not logged in. What do we do with the user? The simplest approach is to redirect them to a page they can access, e.g. the registration page. Django allows us to specify this in our project's `settings.py` module, located in the project configuration directory. In `settings.py`, define the variable `LOGIN_URL` with the URL you'd like to redirect users to that aren't logged in, i.e. the login page located at `/rango/login/`:

```
LOGIN_URL = '/rango/login/'
```

This ensures that the `login_required()` decorator will redirect any user not logged in to the URL `/rango/login/`.

9.9 Logging Out

To enable users to log out gracefully, it would be nice to provide a logout option to users. Django comes with a handy `logout()` function that takes care of ensuring that the users can properly and securely log out. The `logout()` function will ensure that their session is ended, and that if they subsequently try to access a view that requires authentication then they will not be able to access it, unless they log back in.

To provide logout functionality in `rango/views.py`, add the view called `user_logout()` with the following code.

```
# Use the login_required() decorator to ensure only those logged in can
# access the view.
@login_required
def user_logout(request):
    # Since we know the user is logged in, we can now just log them out.
    logout(request)
    # Take the user back to the homepage.
    return HttpResponseRedirect(reverse('index'))
```

You'll also need to import the `logout` function at the top of `views.py`.

```
from django.contrib.auth import logout
```

With the view created, map the URL `/rango/logout/` to the `user_logout()` view by modifying the `urlpatterns` list in Rango's `urls.py`.

```
url(r'^logout/$', views.user_logout, name='logout'),
```

Now that all the machinery for logging a user out has been completed, we can add some finishing touches. It'd be handy to provide a link from the homepage to allow users to simply click a link to logout. However, let's be smart about this: is there any point providing the logout link to a user who isn't logged in? Perhaps not - it may be more beneficial for a user who isn't logged in to be given the chance to register, for example.

Like in the previous section, we'll be modifying Rango's `base.html` template and making use of the `user` object in the template's context to determine what links we want to show. Find your growing list of links at the bottom of the page, and replace it with the following code. Note we also add a link to our restricted page at `/rango/restricted/`.

```
<ul>
{% if user.is_authenticated %}
    <li><a href="{% url 'restricted' %}">Restricted Page</a></li>
    <li><a href="{% url 'logout' %}">Logout</a></li>
{% else %}
    <li><a href="{% url 'login' %}">Sign In</a></li>
    <li><a href="{% url 'register' %}">Sign Up</a></li>
{% endif %}
    <li><a href="{% url 'add_category' %}">Add a New Category</a></li>
    <li><a href="{% url 'about' %}">About</a></li>
    <li><a href="{% url 'index' %}">Index</a></li>
</ul>
```

This code states that when a user is authenticated and logged in, he or she can see the `Restricted Page` and `Logout` links. If he or she isn't logged in, `Register Here` and `Login` are presented. As `About` and `Add a New Category` are not within the template conditional blocks, these links are available to both anonymous and logged in users.

9.10 Taking it Further

In this chapter, we've covered several important aspects of managing user authentication within Django. We've covered the basics of installing Django's `django.contrib.auth` application into our project. Additionally, we have also shown how to implement a user profile model that can provide

additional fields to the base `django.contrib.auth.models.User` model. We have also detailed how to setup the functionality to allow user registrations, login, logout, and to control access. For more information about user authentication and registration consult Django's official documentation on Authentication.

Many Web applications however take the concepts of user authentication further. For example, you may require different levels of security when registering users, by ensuring a valid e-mail address is supplied. While we could implement this functionality, why reinvent the wheel when such functionality already exists? The `django-registration-redux` app has been developed to greatly simplify the process of adding extra functionality related to user authentication. We cover how you can use this package in a following chapter.

Exercises

For now, work on the following two exercises to reinforce what you've learnt in this chapter.

- Customise the application so that only registered users can add categories and pages, while unregistered can only view or use the categories and pages. You'll also have to ensure that the link to add pages appears only if the user browsing the website is logged in.
- Provide informative error messages when users incorrectly enter their username or password.
- Keep your templating know-how up to date by converting the restricted page view to use a template. Call the template `restricted.html`, and ensure that it too extends from Rango's `base.html` template.

10. Cookies and Sessions

In this chapter, we will be touching on the basics of handling *sessions* and storing *cookies*. Both go hand in hand with each other, and are of paramount importance in modern day Web applications. In the previous chapter, the Django framework used sessions and cookies to handle the login and logout functionality. However, all this was done behind the scenes. Here we will explore exactly what is going on under the hood, and how we can use cookies ourselves for other purposes.

10.1 Cookies, Cookies Everywhere!

Whenever a request to a website is made, the webserver returns the content of the requested page. In addition, one or more cookies may also be sent as part of the request. Consider a cookie as a small piece of information sent from the server to the client. When a request is about to be sent, the client checks to see if any cookies that match the address of server exist on the client. If so, they are included in the request. The server can then interpret the cookies as part of the request's context and generate a response to suit.

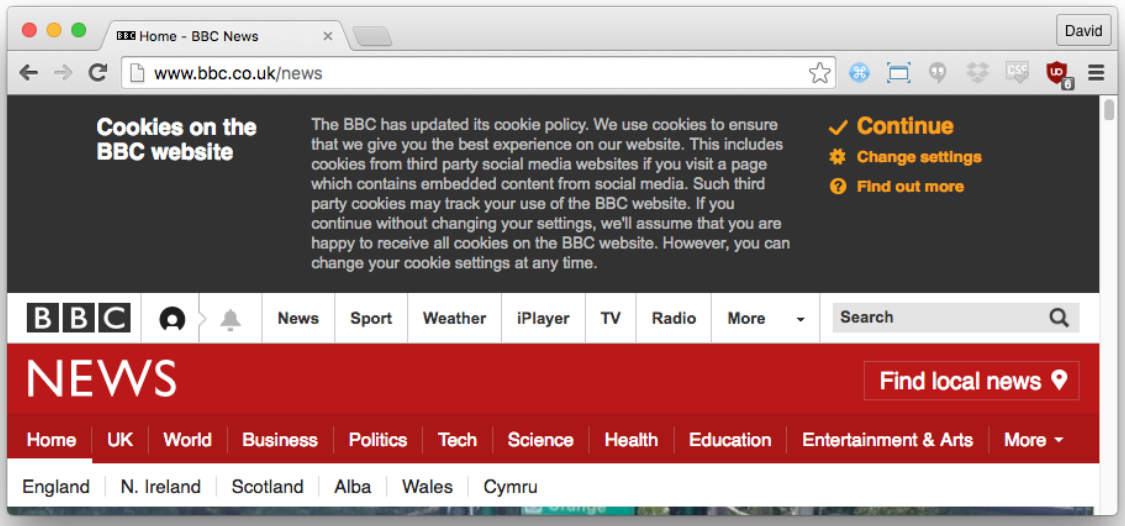
As an example, you may login to a site with a particular username and password. When you have been authenticated, a cookie may be returned to your browser containing your username, indicating that you are now logged into the site. At every request, this information is passed back to the server where your login information is used to render the appropriate page - perhaps including your username in particular places on the page. Your session cannot last forever, however - cookies *have* to expire at some point in time - they cannot be of infinite length. A Web application containing sensitive information may expire after only a few minutes of inactivity. A different Web application with trivial information may expire half an hour after the last interaction - or even weeks into the future.

Cookie Origins

The term *cookie* wasn't actually derived from the food that you eat, but from the term *magic cookie*, a packet of data a program receives and sends again unchanged. In 1994, *MCI* sent a request to *Netscape Communications* to implement a way of implementing persistence across HTTP requests. This was in response to their need to reliably store the contents of a user's virtual shopping basket for an e-commerce solution they were developing. Netscape programmer Lou Montulli took the concept of a magic cookie and applied it to Web communications.

You can find out more about cookies and their history on Wikipedia. Of course, with such a great idea came a software patent - and you can read US patent 5774670 that was submitted by Montulli himself.

The passing of information in the form of cookies can open up potential security holes in your Web application's design. This is why developers of Web applications need to be extremely careful when using cookies. When using cookies, a designer must always ask himself or herself: *does the information you want to store as a cookie* ***really*** *need to be sent and stored on a client's machine?* In many cases, there are more secure solutions to the problem. Passing a user's credit card number on an e-commerce site as a cookie for example would be highly insecure. What if the user's computer is compromised? A malicious program could take the cookie. From there, hackers would have his or her credit card number - all because your Web application's design is fundamentally flawed. This chapter examines the fundamental basics of client-side cookies - and server-side session storage for Web applications.



A screenshot of the BBC News website (hosted in the United Kingdom) with the cookie warning message presented at the top of the page.

Cookies in the EU

In 2011, the European Union (EU) introduced an EU-wide *'cookie law'*, where all hosted sites within the EU should present a cookie warning message when a user visits the site for the first time. The figure above demonstrates such a warning on the BBC News website. You can read about the law here.

If you are developing a site, you'll need to be aware of this law, and other laws especially regarding accessibility.

10.2 Sessions and the Stateless Protocol

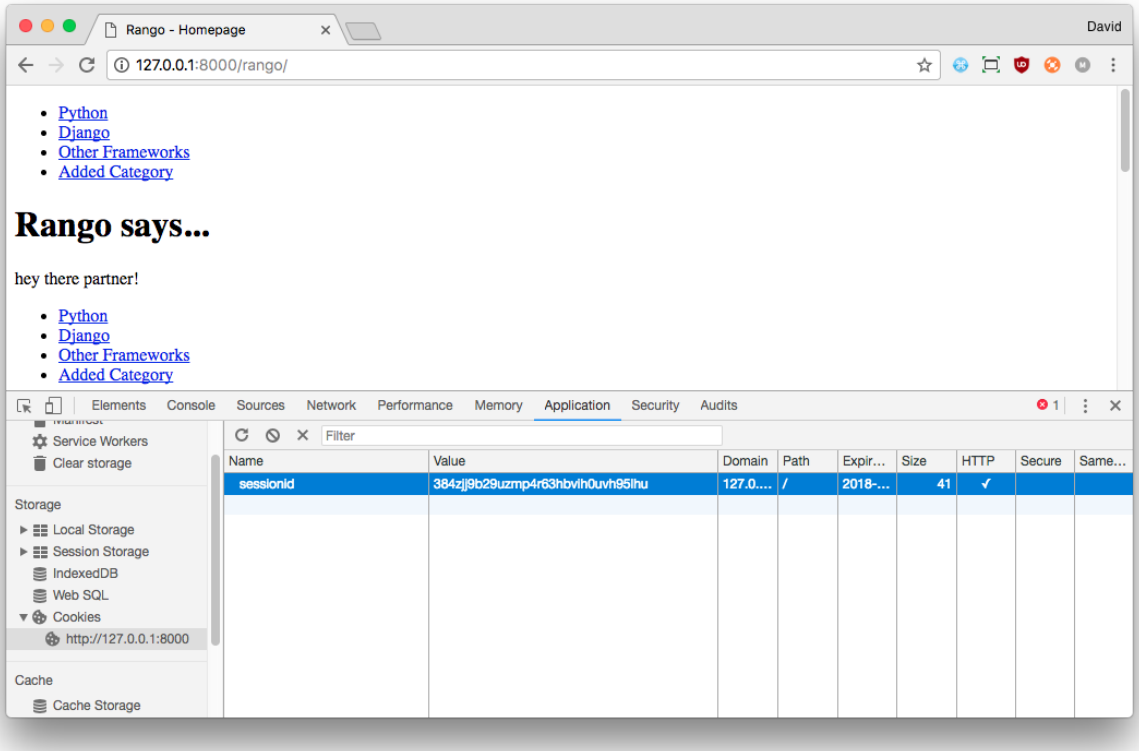
All correspondence between Web browsers (clients) and servers is achieved through the HTTP protocol. As previously mentioned, HTTP is a stateless protocol. This means that a client computer running a Web browser must establish a new network connection (a TCP connection) to the server each time a resource is requested (HTTP `GET`) or sent (HTTP `POST`) [1].

Without a persistent connection between the client and server, the software on both ends cannot simply rely on connections alone to *hold session state*. For example, the client would need to tell the server each time who is logged on to the Web application on a particular computer. This is known as a form of *dialogue* between the client and server, and is the basis of a *session* - a semi-permanent exchange of information. Being a stateless protocol, HTTP makes holding session state pretty challenging, but there are luckily several techniques we can use to circumnavigate this problem.

The most commonly used way of holding state is through the use of a *session ID* stored as a cookie on a client's computer. A session ID can be considered as a token (a sequence of characters, or a *string*) to identify a unique session within a particular Web application. Instead of storing all kinds of information as cookies on the client (such as usernames, names, or passwords), only the session ID is stored, which can then be mapped to a data structure on the Web server. Within that data structure, you can store all of the information you require. This approach is a **much more secure** way to store information about users. This way, the information cannot be compromised by a insecure client or a connection which is being snooped.

If you're using a modern browser that's properly configured, it'll support cookies. Pretty much every website that you visit will create a new *session* for you when you visit. You can see this for yourself now – check out the screenshot below. For Google Chrome, you can view cookies for the currently open website by accessing Chrome's Developer Tools, by accessing *Chrome Settings > More Tools > Developer Tools*. When the Developer Tools pane opens, click the *Application* tab, and look for *Cookies* down the left hand side, within the *Storage* menu. If you're running this with a Rango page open, you should then hopefully see a cookie called `sessionid`. The `sessionid` cookie contains a series of letters and numbers that Django uses to uniquely identify your computer with a given session. With this session ID, all your session details can be accessed – but they are only stored on the *server side*.

[1] The latest version of the HTTP standard HTTP 1.1 actually supports the ability for multiple requests to be sent in one TCP network connection. This provides huge improvements in performance, especially over high-latency network connections (such as via a traditional dial-up modem and satellite). This is referred to as *HTTP pipelining*, and you can read more about this technique on Wikipedia.



A screenshot of Google Chrome's Developer Tools with the `sessionid` cookie highlighted.

Without Cookies

An alternative way of persisting state information *without cookies* is to encode the Session ID within the URL. For example, you may have seen PHP pages with URLs like this one: `http://www.site.com/index.php?sessid=someseeminglyrandomandlongstring1234`. This means you don't need to store cookies on the client machine, but the URLs become pretty ugly. These URLs go against the principles of Django - clean, human-friendly URLs.

10.3 Setting up Sessions in Django

Although this should already be setup and working correctly, it's nevertheless good practice to learn which Django modules provide which functionality. In the case of sessions, Django provides middleware that implements session functionality.

To check that everything is in order, open your Django project's `settings.py` file. Within the file, locate the `MIDDLEWARE` list. You should find within this list a module represented by the string `django.contrib.sessions.middleware.SessionMiddleware`. If you can't see it, add it to the list now. It is the `SessionMiddleware` middleware that enables the creation of unique `sessionid` cookies.

The `SessionMiddleware` is designed to work flexibly with different ways to store session information. There are many approaches that can be taken - you could store everything in a file, in a database, or even in a in-memory cache. The most straightforward approach is to use the `django.contrib.sessions` application to store session information in a Django model/database (specifically, the model `django.contrib.sessions.models.Session`). To use this approach, you'll also need to make sure that `django.contrib.sessions` is in the `INSTALLED_APPS` tuple of your Django project's `settings.py` file. Remember, if you add the application now, you'll need to update your database with the usual migration commands.

Caching Sessions

If you want faster performance, you may want to consider a cached approach for storing session information. You can check out the official Django documentation for advice on cached sessions.

10.4 A Cookie Tasting Session

While all modern Web browsers support cookies, certain cookies may get blocked depending on your browser's security level. Check that you've enabled support for cookies before continuing. It's likely however that everything is ready for you to proceed.

Testing Cookie Functionality

To test out cookies, you can make use of some convenience methods provided by Django's `request` object. The three of particular interest to us are `set_test_cookie()`, `test_cookie_worked()` and `delete_test_cookie()`. In one view, you will need to set the test cookie. In another, you'll need to test that the cookie exists. Two different views are required for testing cookies because you need to wait to see if the client has accepted the cookie from the server.

We'll use two pre-existing views for this simple exercise, `index()` and `about()`. Instead of displaying anything on the pages themselves, we'll be making use of the terminal output from the Django development server to verify whether cookies are working correctly.

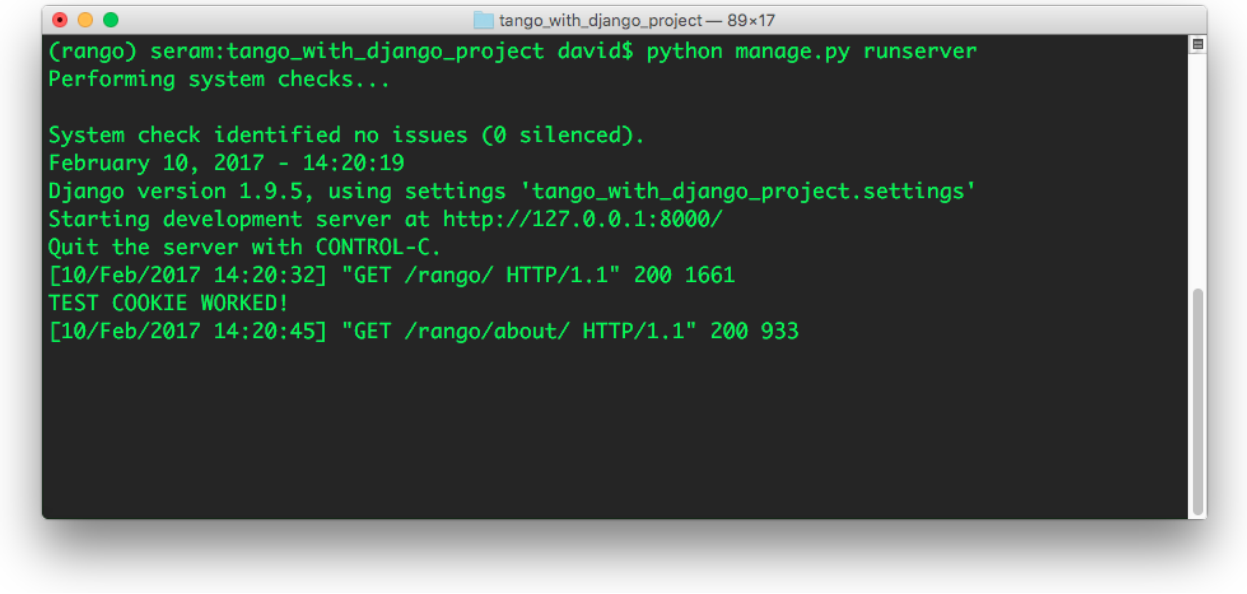
In Rango's `views.py` file, locate your `index()` view. Add the following line to the view. To ensure the line is executed, make sure you put it as the first line of the view.

```
request.session.set_test_cookie()
```

In the `about()` view, add the following three lines to the top of the function.

```
if request.session.test_cookie_worked():
    print("TEST COOKIE WORKED!")
    request.session.delete_test_cookie()
```

With these small changes saved, run the Django development server and navigate to Rango's homepage, `http://127.0.0.1:8000/rango/`. Now navigate to the about page, you should see `TEST COOKIE WORKED!` appear in your Django development server's console, like in the figure below.



A screenshot of the Django development server's console output with the `TEST COOKIE WORKED!` message.

If the message isn't displayed, you'll want to check your browser's security settings. The settings may be preventing the browser from accepting the cookie.

10.5 Client Side Cookies: A Site Counter Example

Now we know how cookies work, let's implement a very simple site visit counter. To achieve this, we're going to be creating two cookies: one to track the number of times the user has visited the Rango app, and the other to track the last time he or she accessed the site. Keeping track of the date and time of the last access will allow us to only increment the site counter once per day (for example) and thus avoid people spamming the site to increment the counter.

The sensible place to assume where a user enters the Rango site is at the index page. Open Rango's `view.py` file. Let's first make a function – given a handle to both the `request` and `response` objects – to handle cookies (`visitor_cookie_handler()`). We can then make use of this function in Rango's `index()` view. In `views.py`, add in the following function. Note that it is not technically a view, because it does not return a `response` object – it is just a *helper function*.

```
def visitor_cookie_handler(request, response):
    # Get the number of visits to the site.
    # We use the COOKIES.get() function to obtain the visits cookie.
    # If the cookie exists, the value returned is casted to an integer.
    # If the cookie doesn't exist, then the default value of 1 is used.
    visits = int(request.COOKIES.get('visits', '1'))

    last_visit_cookie = request.COOKIES.get('last_visit', str(datetime.now()))
    last_visit_time = datetime.strptime(last_visit_cookie[:-7],
                                        '%Y-%m-%d %H:%M:%S')

    # If it's been more than a day since the last visit...
    if (datetime.now() - last_visit_time).days > 0:
        visits = visits + 1
        # Update the last visit cookie now that we have updated the count
        response.set_cookie('last_visit', str(datetime.now()))
    else:
        visits = 1
        # Set the last visit cookie
        response.set_cookie('last_visit', last_visit_cookie)

    # Update/set the visits cookie
    response.set_cookie('visits', visits)
```

This helper function takes the `request` and `response` objects – because we want to be able to access the incoming cookies from the `request`, and add or update cookies in the `response`. In the function, you can see that we call the `request.COOKIES.get()` function, which is a further helper function provided by Django. If the cookie exists, it returns the value. If it does not exist, we can provide a default value. Once we have the values for each cookie, we can calculate whether a day (`.days`) has elapsed between the last visit.

If you want to test this code out without having to wait a day, change `days` to `seconds`. That way the visit counter can be updated every second, as opposed to every day.

Note that all cookie values are returned as strings; *do not assume that a cookie storing whole numbers will return an integer.* You have to manually cast this to the correct type yourself because there's no place in which to store additional information within a cookie telling us of the value's type.

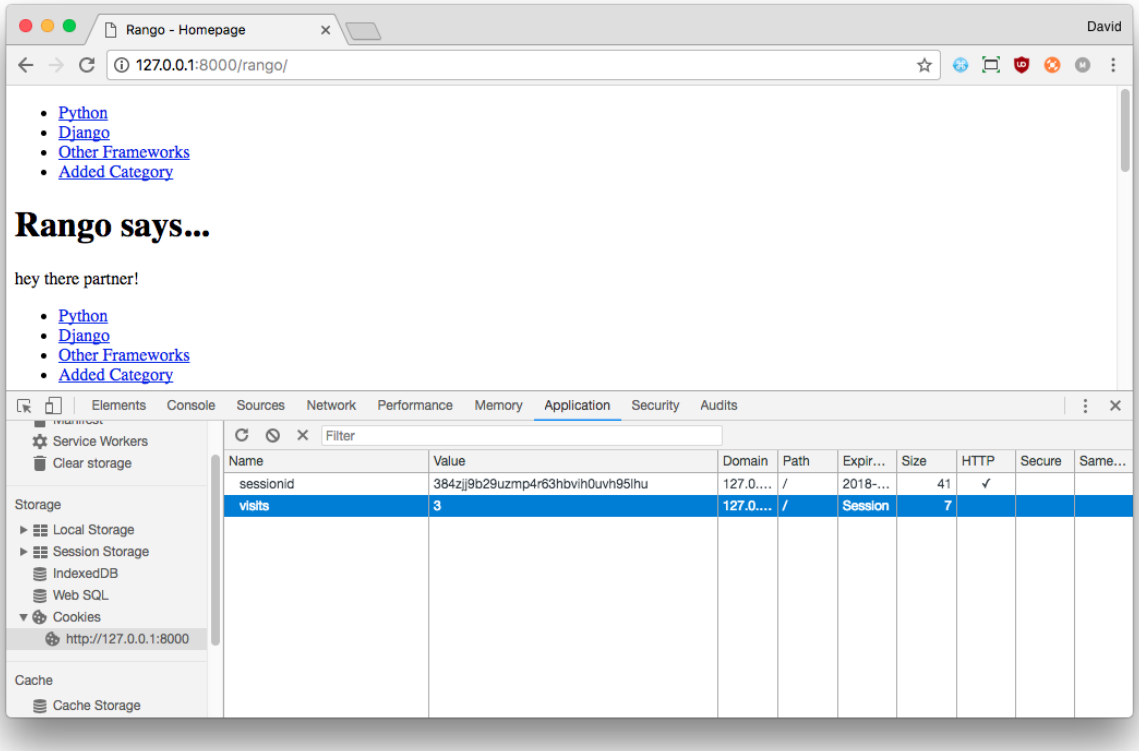
If a cookie does not exist, you can create a cookie with the `set_cookie()` method of the `response` object you create. The method takes in two values, the name of the cookie you wish to create (as a string), and the value of the cookie. In this case, it doesn't matter what type you pass as the value - it will be automatically cast to a string.

Since we are using the `datetime` we need to `import` this into `views.py` at the top of the file.

```
from datetime import datetime
```

Next, update the `index()` view to call the `cookie_handler_function()` helper function. To do this, we need to extract the `response` first.

```
def index(request):
    category_list = Category.objects.order_by('-likes')[:5]
    page_list = Page.objects.order_by('-views')[:5]
    context_dict = {'categories': category_list, 'pages': page_list}

    # Obtain our Response object early so we can add cookie information.
    response = render(request, 'rango/index.html', context_dict)

    # Call the helper function to handle the cookies
    visitor_cookie_handler(request, response)

    # Return response back to the user, updating any cookies that need changed.
    return response
```



A screenshot of Google Chrome with the Developer Tools open showing the cookies for Rango, using the Django development server at `127.0.0.1`. Note the `visits` cookie - the user has visited a total of three times, with each visit at least one day apart.

Now if you visit the Rango homepage and open the cookie inspector provided by your browser (e.g. Google Chrome's Developer Tools), you should be able to see the cookies `visits` and `last_visit`. The figure above demonstrates the cookies in action. Instead of using the developer tools, you could update the `index.html` and add `<p>visits: {{ visits }}</p>` to the template to show the number of visits.

10.6 Session Data

The previous example shows how we can store and manipulate client side cookies - or the data stored on the client. However, a more secure way to save session information is to store any such data on the server side. We can then use the session ID cookie that is stored on the client side (but is effectively anonymous) as the key to access the data.

To use session-based cookies you need to perform the following steps.

1. Make sure that the `MIDDLEWARE_CLASSES` list found in the `settings.py` module contains `django.contrib.sessions.middleware.SessionMiddleware`.

2. Configure your session backend. Make sure that `django.contrib.sessions` is in your `INSTALLED_APPS` in `settings.py`. If not, add it, and run the database migration command, `python manage.py migrate`.
3. By default a database backend is assumed, but you might want to a different setup (i.e. a cache). See the official Django Documentation on Sessions for other backend configurations.

Instead of storing the cookies directly in the request (and thus on the client's machine), you can access server-side data via the method `request.session.get()` and store them with `request.session[]`. Note that a session ID cookie is still used to remember the client's machine (so technically a browser side cookie exists). However, all the user/session data is stored server side. Django's session middleware handles the client side cookie and the storing of the user/session data.

To use the server side data, we need to refactor the code we have written so far. First, we need to update the `visitor_cookie_handler()` function so that it accesses the cookies on the server side. We can do this by calling `request.session.get()`, and store them by placing them in the dictionary `request.session[]`. To help us along, we have made a helper function called `get_server_side_cookie()` that asks the request for a cookie. If the cookie is in the session data, then its value is returned. Otherwise, the default value is returned.

Since all the cookies are stored server side, we won't be changing the response directly. Because of this, we can remove `response` from the `visitor_cookie_handler()` function definition.

```
# A helper method
def get_server_side_cookie(request, cookie, default_val=None):
    val = request.session.get(cookie)
    if not val:
        val = default_val
    return val

# Updated the function definition
def visitor_cookie_handler(request):
    visits = int(get_server_side_cookie(request, 'visits', '1'))
    last_visit_cookie = get_server_side_cookie(request,
                                               'last_visit',
                                               str(datetime.now()))
    last_visit_time = datetime.strptime(last_visit_cookie[:-7],
                                        '%Y-%m-%d %H:%M:%S')

    # If it's been more than a day since the last visit...
    if (datetime.now() - last_visit_time).days > 0:
        visits = visits + 1
        #update the last visit cookie now that we have updated the count
        request.session['last_visit'] = str(datetime.now())
```

```
    else:
        visits = 1
        # set the last visit cookie
        request.session['last_visit'] = last_visit_cookie

    # Update/set the visits cookie
    request.session['visits'] = visits
```

Now that we have updated the handler function, we can now update the `index()` view. First change `visitor_cookie_handler(request, response)` to `visitor_cookie_handler(request)`. Then add in the following line to pass the number of visits to the context dictionary.

```
context_dict['visits'] = request.session['visits']
```

Make sure that these lines are executed before `render()` is called, or your changes won't be executed. The `index()` view should look like the code below.

```
def index(request):
    request.session.set_test_cookie()
    category_list = Category.objects.order_by('-likes')[:5]
    page_list = Page.objects.order_by('-views')[:5]
    context_dict = {'categories': category_list, 'pages': page_list}

    visitor_cookie_handler(request)
    context_dict['visits'] = request.session['visits']

    response = render(request, 'rango/index.html', context=context_dict)
    return response
```

Before you restart the Django development server, delete the existing client side cookies to start afresh. See the warning below for more information.

Avoiding Cookie Confusion

It's highly recommended that you delete any client-side cookies for Rango *before* you start using session-based data. You can do this in your browser's cookie inspector by deleting each cookie individually, or simply clear your browser's cache entirely – ensuring that cookies are deleted in the process.

Data Types and Cookies

An added advantage of storing session data server-side is its ability to cast data from strings to the desired type. This only works however for built-in types, such as `int`, `float`, `long`, `complex` and `boolean`. If you wish to store a dictionary or other complex type, don't expect this to work. In this scenario, you might want to consider pickling your objects.

10.7 Browser-Length and Persistent Sessions

When using cookies you can use Django's session framework to set cookies as either *browser-length sessions* or *persistent sessions*. As the names of the two types suggest:

- browser-length sessions expire when the user closes his or her browser; and
- persistent sessions can last over several browser instances - expiring at a time of your choice. This could be half an hour, or even as far as a month in the future.

By default, browser-length sessions are disabled. You can enable them by modifying your Django project's `settings.py` module. Add the variable `SESSION_EXPIRE_AT_BROWSER_CLOSE`, setting it to `True`.

Alternatively, persistent sessions are enabled by default, with `SESSION_EXPIRE_AT_BROWSER_CLOSE` either set to `False`, or not being present in your project's `settings.py` file. Persistent sessions have an additional setting, `SESSION_COOKIE_AGE`, which allows you to specify the age of which a cookie can live to. This value should be an integer, representing the number of seconds the cookie can live for. For example, specifying a value of `1209600` will mean your website's cookies expire after a two week (14 day) period.

Check out the available settings you can use on the official Django documentation on cookies for more details. You can also check out Eli Bendersky's blog for an excellent tutorial on cookies and Django.

10.8 Clearing the Sessions Database

Sessions accumulate easily, and the data store that contains session information does too. If you are using the database backend for Django sessions, you will have to periodically clear the database that stores the cookies. This can be done using `$ python manage.py clearsessions`. The official Django documentation suggests running this daily as a Cron job. If you don't, you could find your app's performance begin to degrade when it begins to experience more and more users.

10.9 Basic Considerations and Workflow

When using cookies within your Django application, there are a few things you should consider.

- First, consider what type of cookies your Web application requires. Does the information you wish to store need to persist over a series of user browser sessions, or can it be safely disregarded upon the end of one session?

- Think carefully about the information you wish to store using cookies. Remember, storing information in cookies by their definition means that the information will be stored on client's computers, too. This is a potentially huge security risk: you simply don't know how compromised a user's computer will be. Consider server-side alternatives if potentially sensitive information is involved.
- As a follow-up to the previous bullet point, remember that users may set their browser's security settings to a high level that could potentially block your cookies. As your cookies could be blocked, your site may function incorrectly. You *must* cater for this scenario - *you have no control over the client browser's setup.*

If client-side cookies are the right approach for you, then work through the following steps.

1. You must first perform a check to see if the cookie you want exists. Checking the `request` parameter parameter will allow you to do this. The `request.COOKIES.has_key('<cookie_name>')` function returns a boolean value indicating whether a cookie `<cookie_name>` exists on the client's computer or not.
2. If the cookie exists, you can then retrieve its value - again via the `request` parameter - with `request.COOKIES[]`. The `COOKIES` attribute is exposed as a dictionary, so pass the name of the cookie you wish to retrieve as a string between the square brackets. Remember, cookies are all returned as strings, regardless of what they contain. You must therefore be prepared to cast to the correct type (with `int()` or `float()`, for example).
3. If the cookie doesn't exist, or you wish to update the cookie, pass the value you wish to save to the response you generate. `response.set_cookie('<cookie_name>', value)` is the function you call, where two parameters are supplied: the name of the cookie, and the `value` you wish to set it to.

If you need more secure cookies, then use session based cookies.

1. Firstly, ensure that the `MIDDLEWARE_CLASSES` list in your Django project's `settings.py` module contains `django.contrib.sessions.middleware.SessionMiddleware`. If it doesn't, add it to the list.
2. Configure your session backend `SESSION_ENGINE`. See the official Django Documentation on Sessions for the various backend configurations.
3. Check to see if the cookie exists via `requests.sessions.get()`.
4. Update or set the cookie via the session dictionary, `requests.session['<cookie_name>']`.

Exercises

Now you've read through this chapter and tried out the code, give these exercises a go.

- Check that your cookies are server side. Clear the browser's cache and cookies, then check to make sure you can't see the `last_visit` and `visits` variables in the browser. Note you will still see the `sessionid` cookie. Django uses this cookie to look up the session in the database where it stores all the server side cookies about that session.
- Update the *About* page view and template telling the visitors how many times they have visited the site. Remember to call the `visitor_cookie_handler()` before you attempt to get the `visits` cookie from the `request.session` dictionary, otherwise if the cookie is not set it will raise an error.

11. User Authentication with Django-Registration-Redux

In the previous chapter, we added in login and registration functionality by manually coding up the URLs, views and templates. However, such functionality is common to many web application so developers have created numerous add-on applications that can be included in your Django project to reduce the amount of code required to provide login, registration, one-step and two-step authentication, password chaning, password recovery, etc. In this chapter, we will be using the package `django-registration-redux` to provide these facilities.

This will mean we will need to re-factor our code to remove the login and registration functionality we previously created, and then setup and configure our project to include the `django-registration-redux` application. This chapter also will provide you with some experience of using external applications and show you how easily they can be plugged into your Django project.

11.1 Setting up Django Registration Redux

To start we need to first install `django-registration-redux` version 1.4 into your environment using `pip`.

```
pip install -U django-registration-redux==1.4
```

Now that it is installed, we need to tell Django that we will be using this application. Open up the `settings.py` file, and update the `INSTALLED_APPS` list:

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'rango',
    'registration'  # add in the registration package
    ]
```

While you are in the `settings.py` file you can also add the following variables that are part of the registrations package's configuration (these settings should be pretty self explanatory):

```
# If True, users can register
REGISTRATION_OPEN = True
# One-week activation window; you may, of course, use a different value.
ACCOUNT_ACTIVATION_DAYS = 7
# If True, the user will be automatically logged in.
REGISTRATION_AUTO_LOGIN = True
# The page you want users to arrive at after they successfully log in
LOGIN_REDIRECT_URL = '/rango/'
# The page users are directed to if they are not logged in,
# and are trying to access pages requiring authentication
LOGIN_URL = '/accounts/login/'
```

In `tango_with_django_project/urls.py`, you can now update the `urlpatterns` so that it includes a reference to the registration package:

```
url(r'^accounts/', include('registration.backends.simple.urls')),
```

The `django-registration-redux` package provides a number of different registration backends, depending on your needs. For example you may want a two-step process, where user is sent a confirmation email, and a verification link. Here we will be using the simple one-step registration process, where a user sets up their account by entering in a username, email, and password, and is automatically logged in.

11.2 Functionality and URL mapping

The Django Registration Redux package provides the machinery for numerous functions. In the `registration.backend.simple.urls`, it provides the following mappings:

- registration -> `/accounts/register/`
- registration complete -> `/accounts/register/complete/`
- login -> `/accounts/login/`
- logout -> `/accounts/logout/`
- password change -> `/password/change/`
- password reset -> `/password/reset/`

while in the `registration.backends.default.urls` it also provides the functions for activating the account in a two stage process:

- activation complete (used in the two-step registration) -> `activate/complete/`
- activate (used if the account action fails) -> `activate/<activation_key>/`

- activation email (notifies the user an activation email has been sent out)
    - activation email body (a text file, that contains the activation email text)
    - activation email subject (a text file, that contains the subject line of the activation email)

Now the catch. While Django Registration Redux provides all this functionality, it does not provide the templates because these tend to be application specific. So we need to create the templates associated with each view.

11.3 Setting up the Templates

In the Django Registration Redux Quick Start Guide, it provides an overview of what templates are required, but it is not immediately clear what goes within each template. Rather than try and work it out from the code, we can take a look at a set of templates written by Anders Hofstee to quickly get the gist of what we need to code up.

First, create a new directory in the `templates` directory, called `registration`. This is where we will house all the pages associated with the Django Registration Redux application, as it will look in this directory for the templates it requires.

Login Template

In `templates/registration` create the file, `login.html` with the following code:

```
{% extends "rango/base.html" %}
{% block body_block %}
    <h1>Login</h1>
    <form method="post" action=".">
        {% csrf_token %}
        {{ form.as_p }}
        <input type="submit" value="Log in" />
        <input type="hidden" name="next" value="{{ next }}" />
    </form>
    <p>
        Not  a member?
        <a href="{% url 'registration_register' %}">Register</a>
    </p>
{% endblock %}
```

Notice that whenever a URL is referenced, the `url` template tag is used to reference it. If you visit, `http://127.0.0.1:8000/accounts/` then you will see the list of URL mappings, and the names associated with each URL (assuming that `DEBUG=True` in `settings.py`).

Registration Template

In templates/registration create the file, registration_form.html with the following code:

```
{% extends "rango/base.html" %}
{% block body_block %}
    <h1>Register Here</h1>
    <form method="post" action=".">
        {% csrf_token %}
        {{ form.as_p }}
        <input type="submit" value="Submit" />
    </form>
{% endblock %}
```

Registration Complete Template

In templates/registration create the file, registration_complete.html with the following code:

```
{% extends "rango/base.html" %}
{% block body_block %}
    <h1>Registration Complete</h1>
    <p>You are now registered</p>
{% endblock %}
```

Logout Template

In templates/registration create the file, logout.html with the following code:

```
{% extends "rango/base.html" %}
{% block body_block %}
    <h1>Logged Out</h1>
    <p>You are now logged out.</p>
{% endblock %}
```

Try out the Registration Process

Run the server and visit: http://127.0.0.1:8000/accounts/register/ Note how the registration form contains two fields for password - so that it can be checked. Try registering, but enter different passwords.

While this works, not everything is hooked up.

Refactoring your project

Now you will need to update the `base.html` so that the new registration URLs and views are used.

- Update register to point to `<a href="{% url 'registration_register' %}">`.
- Update login to point to `<a href="{% url 'auth_login' %}">`.
- Update logout to point to `<a href="{% url 'auth_logout' %}?next=/rango/">`.
- In `settings.py`, update `LOGIN_URL` to be `'/accounts/login/'`.

Notice that for the logout, we have included a `?next=/rango/`. This is so when the user logs out, it will redirect them to the index page of Rango. If we exclude it, then they will be directed to the log out page (but that would not be very nice).

Next, decommission the `register`, `login`, `logout` functionality from the `rango` application, i.e. remove the URLs, views, and templates (or comment them out).

Modifying the Registration Flow

At the moment, when users register, it takes them to the registration complete page. This feels a bit clunky; so instead, we can take them to the main index page. Overriding the `RegistrationView` provided by `registration.backends.simple.views` can do this. Update the `tango_with_django_project/urls.py` by importing `RegistrationView`, add in the following registration class.

```
from registration.backends.simple.views import RegistrationView

# Create a new class that redirects the user to the index page,
#if successful at logging
class MyRegistrationView(RegistrationView):
    def get_success_url(self, user):
        return '/rango/'
```

Then update the `urlpatterns` list in your Django project's `urls.py` module by adding the following line before the pattern for `accounts`. Note that this is *not* the `urls.py` module within the `rango` directory!

```
url(r'^accounts/register/$',
    MyRegistrationView.as_view(),
        name='registration_register'),
```

This will allow for accounts/register to be matched before any other accounts/ URL. This allows us to redirect accounts/register to our customised registration view.

Exercise and Hints

- Provide users with password change functionality.
- Hint: see Anders Hofstee's Templates to get started.
- Hint: the URL to change passwords is accounts/password/change/ and the URL to denote the password has been changed is: accounts/password/change/done/

12. Bootstrapping Rango

In this chapter, we will be styling Rango using the *Twitter Bootstrap 4 Alpha* toolkit. Bootstrap is the most popular HTML, CSS, JS Framework, which we can use to style our application. The toolkit lets you design and style responsive web applications, and is pretty easy to use once you get familiar with it.

Cascading Style Sheets

If you are not familiar with CSS, have a look at the CSS crash course. We provide a quick guide on the basic of Cascading Style Sheets.

Now take a look at the Bootstrap 4.0 website - it provides you with sample code and examples of the different components and how to style them by added in the appropriate style tags, etc. On the Bootstrap website they provide a number of example layouts which we can base our design on.

To style Rango we have identified that the dashboard style more or less meets our needs in terms of the layout of Rango, i.e. it has a menu bar at the top, a side bar (which we will use to show categories) and a main content pane.

Download and save the HTML source for the Dashboard layout to a file called, `base_bootstrap.html` and save it to your `templates/rango` directory.

Before we can use the template, we need to modify the HTML so that we can use it in our application. The changes that we performed are listed below along with the updated HTML (so that you don't have to go to the trouble).

- Replaced all references of `../../` to be `http://v4-alpha.getbootstrap.com/`.
- Replaced `dashboard.css` with the absolute reference:
  - `http://getbootstrap.com/examples/dashboard/dashboard.css`
- Removed the search form from the top navigation bar.
- Stripped out all the non-essential content from the HTML and replaced it with:
  - `{% block body_block %}{% endblock %}`
- Set the title element to be:
  - `<title> Rango - {% block title %}How to Tango with Django!{% endblock %} </title>`
- Changed `project name` to be `Rango`.
- Added the links to the index page, login, register, etc to the top nav bar.
- Added in a side block, i.e., `{% block side_block %}{% endblock %}`
- Added in `{% load staticfiles %}` after the `DOCTYPE` tag.

12.1 Template

Copying and Pasting

As we said in the introductory chapter, *don't simply copy and paste the code you see here.* Type it in, think about what the HTML markup below is doing as you type. If you don't understand what a particular element does, search for it online. If you don't understand what a given Bootstrap CSS class achieves, check out the documentation.

If you copy and paste over a large portion of code like the template below, you risk including the book's headers and footers, too!

```
<!DOCTYPE html>
{% load staticfiles %}
{% load rango_template_tags %}
<html lang="en">
<head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width,
                                   initial-scale=1, shrink-to-fit=no">
    <meta name="description" content="">
    <meta name="author" content="">
    <link rel="icon" href="{% static 'images/favicon.ico' %}">
    <title>
      Rango - {% block title %}How to Tango with Django!{% endblock %}
    </title>
    <!-- Bootstrap core CSS -->
    <link href="http://v4-alpha.getbootstrap.com/dist/css/bootstrap.min.css"
        rel="stylesheet">
    <!-- Custom styles for this template -->
    <link href=
           "http://v4-alpha.getbootstrap.com/examples/dashboard/dashboard.css"
          rel="stylesheet">
</head>
<body>

<nav class="navbar navbar-toggleable-md navbar-inverse fixed-top bg-inverse">
    <button class="navbar-toggler navbar-toggler-right hidden-lg-up"
            type="button"
            data-toggle="collapse"
```

```
            data-target="#navbar"
            aria-controls="navbar"
            aria-expanded="false"
            aria-label="Toggle navigation">
        <span class="navbar-toggler-icon"></span>
    </button>
    <a class="navbar-brand" href="{% url 'index' %}">Rango</a>

    <div class="collapse navbar-collapse" id="navbar">
        <ul class="navbar-nav mr-auto">
            <li class="nav-item active">
                <a class="nav-link" href="{% url 'index' %}">
                    Home
                </a>
            </li>
            <li class="nav-item">
                <a class="nav-link" href="{% url 'about' %}">
                    About
                </a>
            </li>
            {% if user.is_authenticated %}
            <li class="nav-item">
                <a class="nav-link" href="{% url 'restricted' %}">
                    Restricted Page
                </a>
            </li>
            <li class="nav-item">
                <a class="nav-link" href="{% url 'add_cateory' %}">
                Add a New Category
                </a>
            </li>
            <li class="nav-item">
                <a class="nav-link" href="{% url 'auth_logout' %}?next=/rango/">
                    Logout
                </a>
            </li>
            {% else %}
            <li class="nav-item">
                <a class="nav-link" href="{% url 'registration_register' %}">
                    Register Here
                </a>
            </li>
```

```
            <li class="nav-item">
                <a class="nav-link" href="{% url 'auth_login' %}">
                    Login
                </a>
            </li>
            {% endif %}
        </ul>
    </div>
</nav>

<div class="container-fluid">
    <div class="row">
    <div class="col-sm-3 col-md-2 sidebar">
        {% block sidebar_block %}
            {% get_category_list category %}
        {% endblock %}
    </div>
    <div class="col-sm-9 offset-sm-3 col-md-10 offset-md-2 main">
        {% block body_block %}{% endblock %}
    </div>
    </div>
</div>
<!-- Bootstrap core JavaScript
    ================================================== -->
    <!-- Placed at the end of the document so the pages load faster -->
    <script
      src="https://ajax.googleapis.com/ajax/libs/jquery/2.1.4/jquery.min.js">
    </script>
    <script
      src="http://v4-alpha.getbootstrap.com/dist/js/bootstrap.min.js">
    </script>
    <!-- IE10 viewport hack for Surface/desktop Windows 8 bug -->
    <script
      src=
    "http://v4-alpha.getbootstrap.com/assets/js/ie10-viewport-bug-workaround.js">
    </script>
</body>
</html>
```

Once you have the new template, downloaded the Rango Favicon and saved it to `static/images/`.

If you take a close look at the modified Dashboard HTML source, you'll notice it has a lot of structure in it created by a series of `<div>` tags. Essentially the page is broken into two parts - the top

navigation bar which is contained by `<nav>` tags, and the main content pane denoted by the `<div class="container-fluid">` tag. Within the main content pane, there are two `<div>`s, one for the sidebar and the other for the main content, where we have placed the code for the `sidebar_block` and `body_block`, respectively.

In this new template, we have assumed that you have completed the chapters on User Authentication and used the Django Regisration Redux Package. If not you will need to update the template and remove/modify the references to those links in the navigation bar i.e. in the `<nav>` tags.

Also of note is that the HTML template makes references to external websites to request the required `css` and `js` files. So you will need to be connected to the internet for the style to be loaded when you run the application.

Working Offline?

Rather than including external references to the `css` and `js` files, you could download all the associated files and store them in your static directory. If you do this, simply update the base template to reference the static files stored locally.

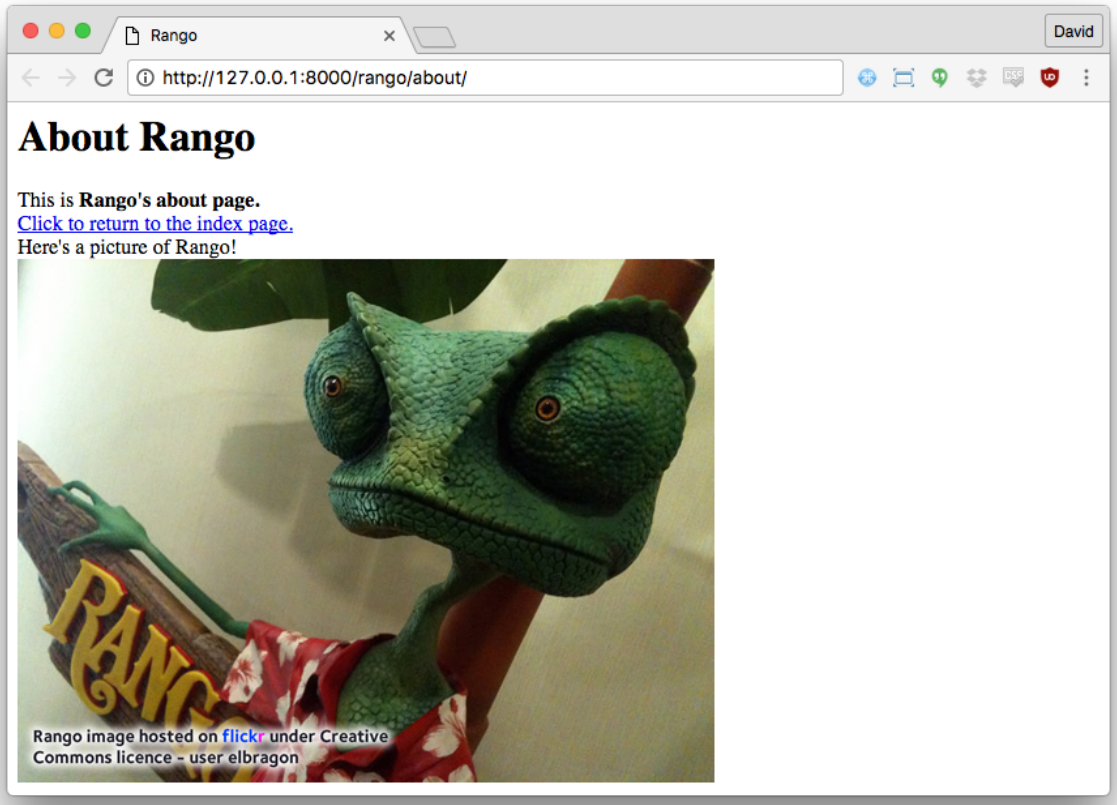
12.2 Quick Style Change

To give Rango a much needed facelift, we can replace the content of the existing `base.html` with the HTML template code in `base_bootstrap.html`. You might want to first comment out the existing code in `base.html` and then copy in the `base_bootstrap.html` code.

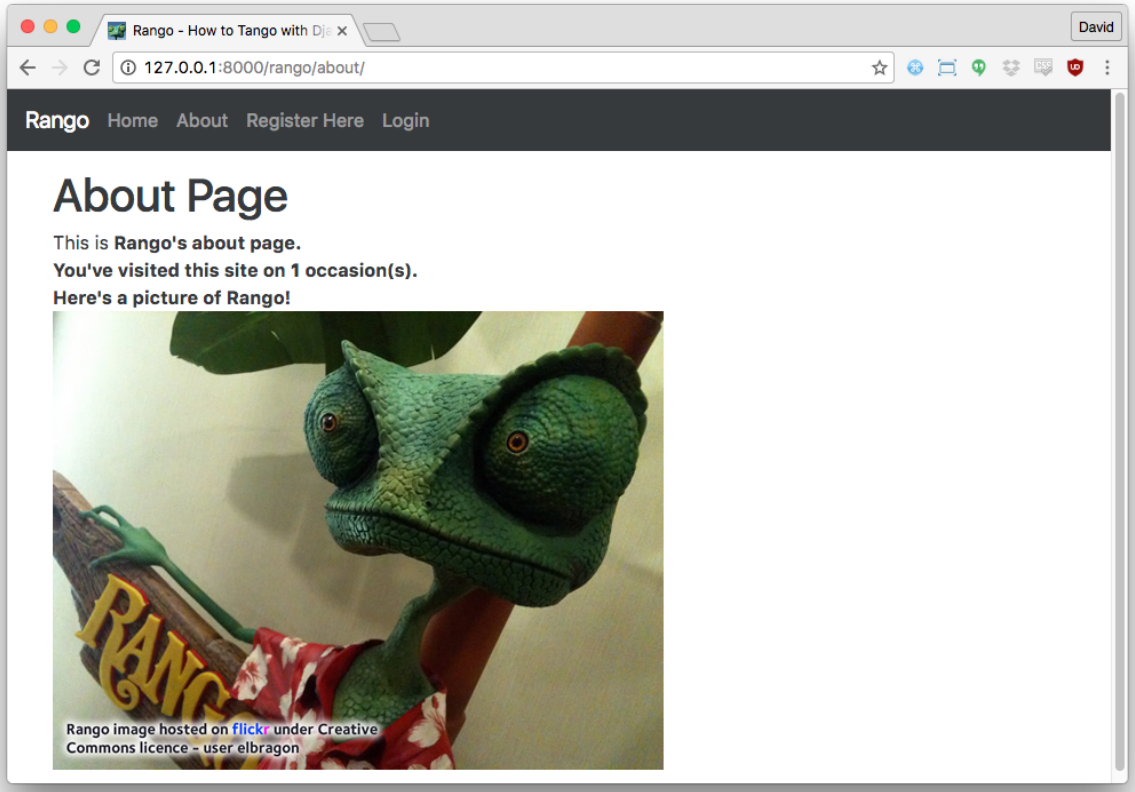
Now reload your application. Pretty nice!

You should notice that your application looks about a hundred times better already. Below we have some screen shots of the about page showing the before and after.

Flip through the different pages. Since they all inherit from base, they will all be looking pretty good, but not perfect! In the remainder of this chapter, we will go through a number of changes to the templates and use various Bootstrap classes to improve the look and feel of Rango.



A screenshot of the About page without styling.



A screenshot of the About page with Bootstrap Styling applied.

The Index Page

For the index page it would be nice to show the top categories and top pages in two separate columns. Looking at the Bootstrap examples, we can see that in the Narrow Jumbotron they have an example with two columns. If you inspect the source, you can see the following HTML that is responsible for the columns.

```
<div class="row marketing">
    <div class="col-lg-6">
        <h4>Subheading</h4>
        <p>Donec id elit non mi porta gravida at eget metus.
            Maecenas faucibus mollis interdum.</p>
        <h4>Subheading</h4>
    </div>
    <div class="col-lg-6">
        <h4>Subheading</h4>
```

```
        <p>Donec id elit non mi porta gravida at eget metus.
            Maecenas faucibus mollis interdum.</p>
    </div>
</div>
```

Inside the `<div class="row marketing">`, we can see that it contains two `<div>`'s with classes `col-lg-6`. Bootstrap is based on a grid layout, where each container is conceptually broken up into 12 units. The `col-lg-6` class denotes a column that is of size 6, i.e. half the size of its container, `<div class="row marketing">`.

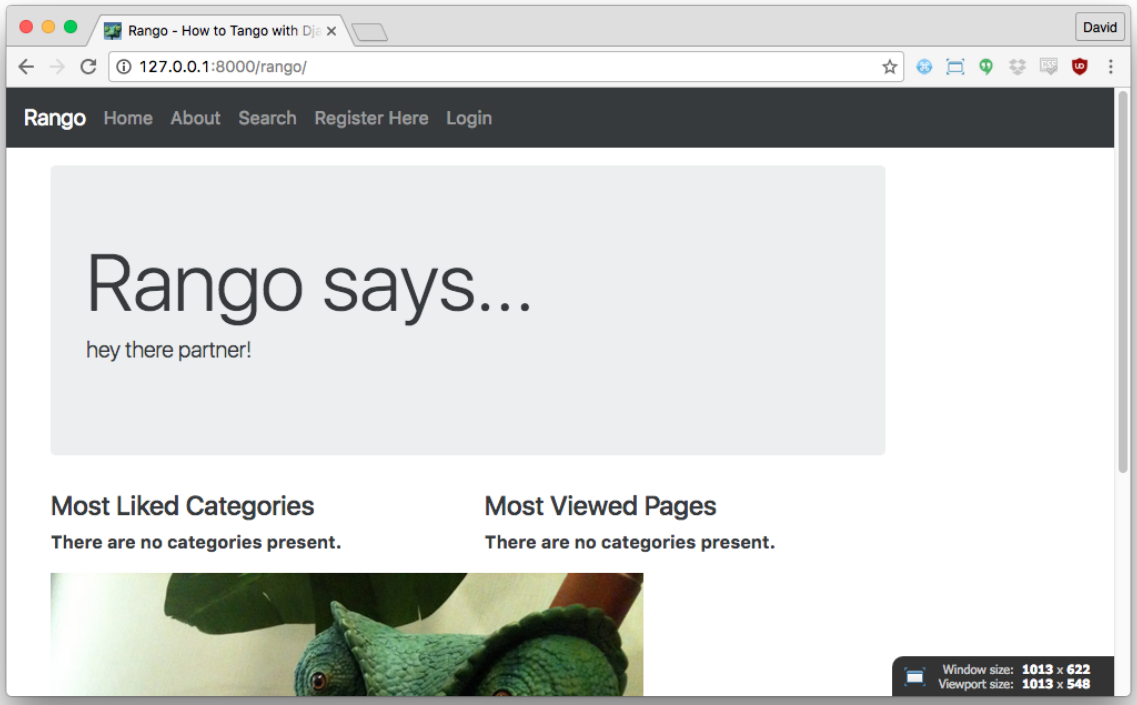
Given this example, we can create columns in `index.html` by updating the template as follows.

```
{% extends 'rango/base.html' %}
{% load staticfiles %}
{% block title_block %}
    Index
{% endblock %}
{% block body_block %}
<div class="jumbotron">
    <h1 class="display-3">Rango says...</h1>
    {% if user.is_authenticated %}
        <h1>hey there {{ user.username }}!</h1>
    {% else %}
        <h1>hey there partner! </h1>
    {% endif %}
</div>
<div class="row marketing">
    <div class="col-lg-6">
    <h4>Most Liked Categories</h4>
    <p>
    {% if categories %}
    <ul>
        {% for category in categories %}
        <li><a href="{% url 'show_category' category.slug %}">
            {{ category.name }}</a></li>
        {% endfor %}
    </ul>
    {% else %}
        <strong>There are no categories present.</strong>
    {% endif %}
    </p>
    </div>
    <div class="col-lg-6">
```

```
        <h4>Most Viewed Pages</h4>
        <p>
        {% if pages %}
        <ul>
            {% for page in pages %}
            <li><a href="{{ page.url }}">{{ page.title }}</a></li>
            {% endfor %}
            </ul>
        {% else %}
            <strong>There are no categories present.</strong>
         {% endif %}
        </p>
    </div>
</div>
<img src="{% static "images/rango.jpg" %}" alt="Picture of Rango" />
{% endblock %}
```

We have also used the `jumbotron` class to make the heading in the page more evident by wrapping the title in a `<div class="jumbotron">`. Reload the page - it should look a lot better now, but the way the list items are presented is pretty horrible.

Let's use the list group styles provided by Bootstrap to improve how they look. We can do this quite easily by changing the `<ul>` elements to `<ul class="list-group">` and the `<li>` elements to `<li class="list-group-item">`. Reload the page, any better?



A screenshot of the Index page with a Jumbotron and Columns.

The Login Page

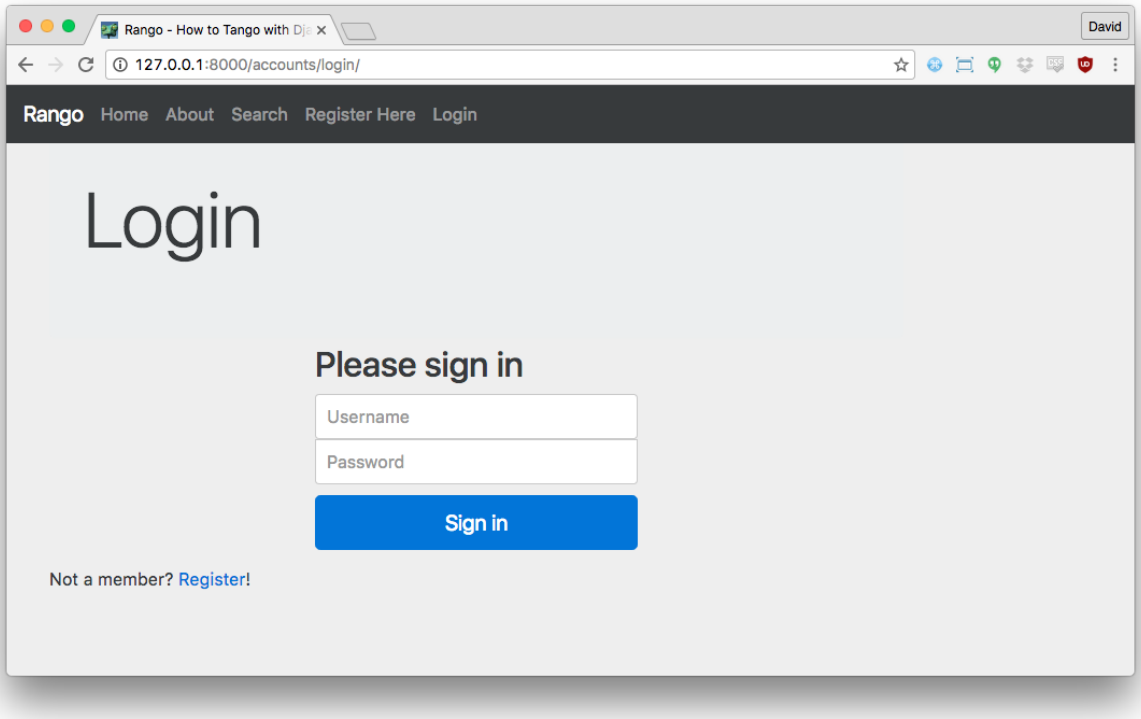
Now let's turn our attention to the login page. On the Bootstrap website you can see they have already made a nice login form. If you take a look at the source, you'll notice that there are a number of classes that we need to include to stylise the basic login form. Update the `body_block` in the `login.html` template as follows:

```
{% block body_block %}
<link href="http://v4-alpha.getbootstrap.com/examples/signin/signin.css"
    rel="stylesheet">
<div class="jumbotron">
    <h1 class="display-3">Login</h1>
</div>
<form class="form-signin" role="form" method="post" action=".">
    {% csrf_token %}
    <h2 class="form-signin-heading">Please sign in</h2>
    <label for="inputUsername" class="sr-only">Username</label>
    <input type="text" name="username" id="id_username" class="form-control"
```

```
            placeholder="Username" required autofocus>
        <label for="inputPassword" class="sr-only">Password</label>
        <input type="password" name="password" id="id_password" class="form-control"
            placeholder="Password" required>
        <button class="btn btn-lg btn-primary btn-block" type="submit"
             value="Submit" />Sign in</button>
</form>
{% endblock %}
```

Besides adding in a link to the bootstrap `signin.css`, and a series of changes to the classes associated with elements, we have removed the code that automatically generates the login form, i.e. `form.as_p`. Instead, we took the elements, and importantly the `id` of the elements generated and associated them with the elements in this bootstrapped form. To find out what these `ids` were, we ran Rango, navigated to the page, and then inspected the source to see what HTML was produced by the `form.as_p` template tag.

In the button, we have set the class to `btn` and `btn-primary`. If you check out the Bootstrap section on buttons you can see there are lots of different colours, sizes and styles that can be assigned to buttons.



A screenshot of the login page with customised Bootstrap Styling.

Other Form-based Templates

You can apply similar changes to `add_cagegory.html` and `add_page.html` templates. For the `add_page.html` template, we can set it up as follows.

```
{% extends "rango/base.html" %}
{% block title %}Add Page{% endblock %}

{% block body_block %}
    {% if category %}
        <form role="form" id="page_form" method="post"
              action="/rango/category/{{category.slug}}/add_page/">
        <h2 class="form-signin-heading"> Add a Page to
            <a href="/rango/category/{{category.slug}}/">
                {{ category.name }}</a></h2>
        {% csrf_token %}
        {% for hidden in form.hidden_fields %}
            {{ hidden }}
        {% endfor %}
        {% for field in form.visible_fields %}
            {{ field.errors }}
            {{ field.help_text }}<br/>
            {{ field }}<br/>
        {% endfor %}
        <br/>
        <button class="btn btn-primary"
                type="submit" name="submit">
            Add Page
        </button>
        </form>
    {%  else %}
        <p>This is category does not exist.</p>
    {%  endif %}
{% endblock %}
```

Exercise

- Create a similar template for the Add Category page called `add_category.html`.

The Registration Template

For the `registration_form.html`, we can update the form as follows:

```
{% extends "rango/base.html" %}
{% block body_block %}

<h2 class="form-signin-heading">Sign Up Here</h2>

<form role="form"  method="post" action=".">
    {% csrf_token %}
    <div class="form-group" >
    <p class="required"><label class="required" for="id_username">
        Username:</label>
        <input class="form-control" id="id_username" maxlength="30"
             name="username" type="text" />
        <span class="helptext">
        Required. 30 characters or fewer. Letters, digits and @/./+/-/_ only.
        </span>
    </p>
    <p class="required"><label class="required" for="id_email">
        E-mail:</label>
        <input class="form-control" id="id_email" name="email"
             type="email" />
    </p>
    <p class="required"><label class="required" for="id_password1">
        Password:</label>
        <input class="form-control" id="id_password1" name="password1"
            type="password" />
    </p>
    <p class="required">
        <label class="required" for="id_password2">
            Password confirmation:</label>
        <input class="form-control" id="id_password2" name="password2"
             type="password" />
        <span class="helptext">
             Enter the same password as before, for verification.
        </span>
    </p>
    </div>
    <button type="submit" class="btn btn-default">Submit</button>
</form>
{% endblock %}
```

Again we have manually transformed the form created by the `{{ form.as_p }}` template tag, and added the various bootstrap classes.

Bootstrap, HTML and Django Kludge

This is not the best solution - we have kind of kludged it together. It would be much nicer and cleaner if we could instruct Django when building the HTML for the form to insert the appropriate classes.

12.3 Using `Django-Bootstrap-Toolkit`

An alternative solution would be to use something like the `django-bootstrap-toolkit`. To install the `django-bootstrap-toolkit`, run:

```
pip install django-bootstrap-toolkit
```

Add, `bootstrap_toolkit` to the `INSTALLED_APPS` tuple in `settings.py`.

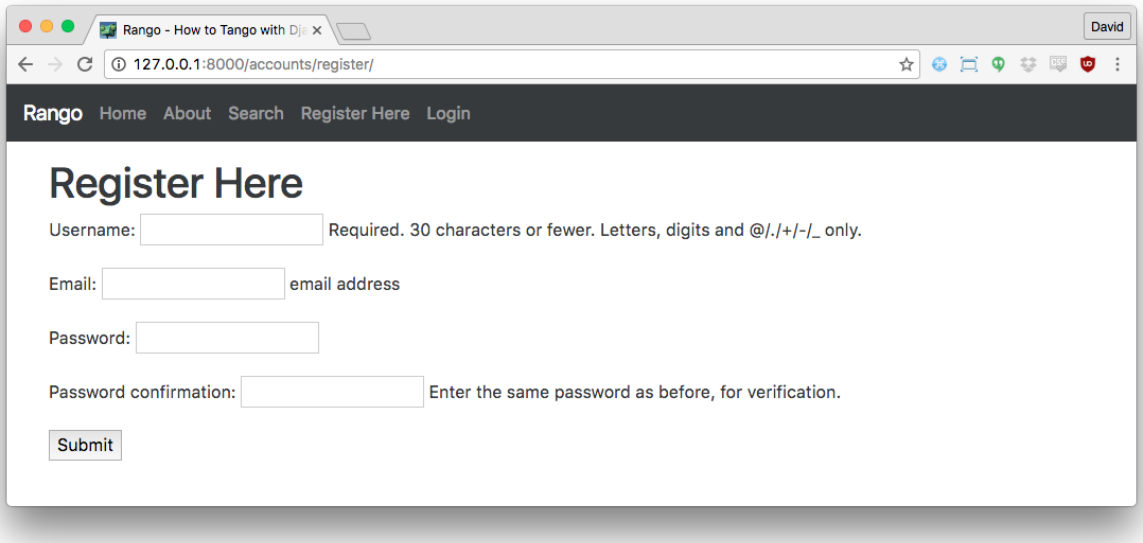
To use the toolkit within our templates, we need to first load the toolkit using the `load` template tag, `{% load bootstrap_toolkit %}`, and then call the function that updates the generated HTML, i.e. `{{ form|as_bootstrap }}`. Updating the `category.html` template, we arrive at the following.

```
{% extends "rango/base.html" %}

{% load bootstrap_toolkit %}
{% block title %}Add Category{% endblock %}
{% block body_block %}
    <form id="category_form" method="post"
        action="{% url 'add_category' %}">
    <h2 class="form-signin-heading">Add a Category</a></h2>
    {% csrf_token %}
    {{ form|as_bootstrap }}
    <br/>
    <button class="btn btn-primary" type="submit"
         name="submit">Create Category</button>
    </form>
{% endblock %}
```

This solution is much cleaner, and automated. However, it does not render as nicely as the first solution. It therefore needs some tweaking to customise it as required, but we'll let you figure out what needs to be done.

Next Steps

In this chapter we have described how to quickly style your Django application using the Bootstrap toolkit. Bootstrap is highly extensible and it is relatively easy to change themes - check out the StartBootstrap Website for a whole series of free themes. Alternatively, you might want to use a different CSS toolkit like: Zurb, Titon, Pure, GroundWorkd or BaseCSS. Now that you have an idea of how to hack the templates and set them up to use a responsive CSS toolkit, we can now go back and focus on finishing off the extra functionality that will really pull the application together.



A screenshot of the Registration page with customised Bootstrap Styling.

Another Style Exercise

While this tutorial uses Bootstrap, an additional, and optional exercise, would be to style Rango using one of the other responsive CSS toolkits. If you do create your own style, let us know and we can link to it to show others how you have improved Rango's styling!

13. Webhose Search

Now that our Rango app is looking good and most of the core functionality has been implemented, we can move onto some of the more advanced functionality. In this chapter, we will connect Rango up to the *Webhose API* so that users can also search for pages, rather than simply browse categories. Before we do so, we need to set up an account with Webhose, and write a wrapper to query and obtain results from their API.

13.1 The Webhose API

The Webhose API provides you with the ability to programmatically query *Webhose*, an online service that collates information from a variety of online sources in real-time. Through a straightforward interface, you can request results for a query in JSON. The returned data can then be interpreted by a JSON parser, with the results then rendered as part of a template within you app.

Although Webhose allows you to obtain results for information that has been recently crawled, we'll be focusing on returning content ranked by its *relevancy* to the query that a user of Rango provides. To use the Webhose API, you'll need an *API key*. The key provides you with 1,000 free queries per month – more than enough for our purposes.

What is an *Application Programming Interface (API)*?

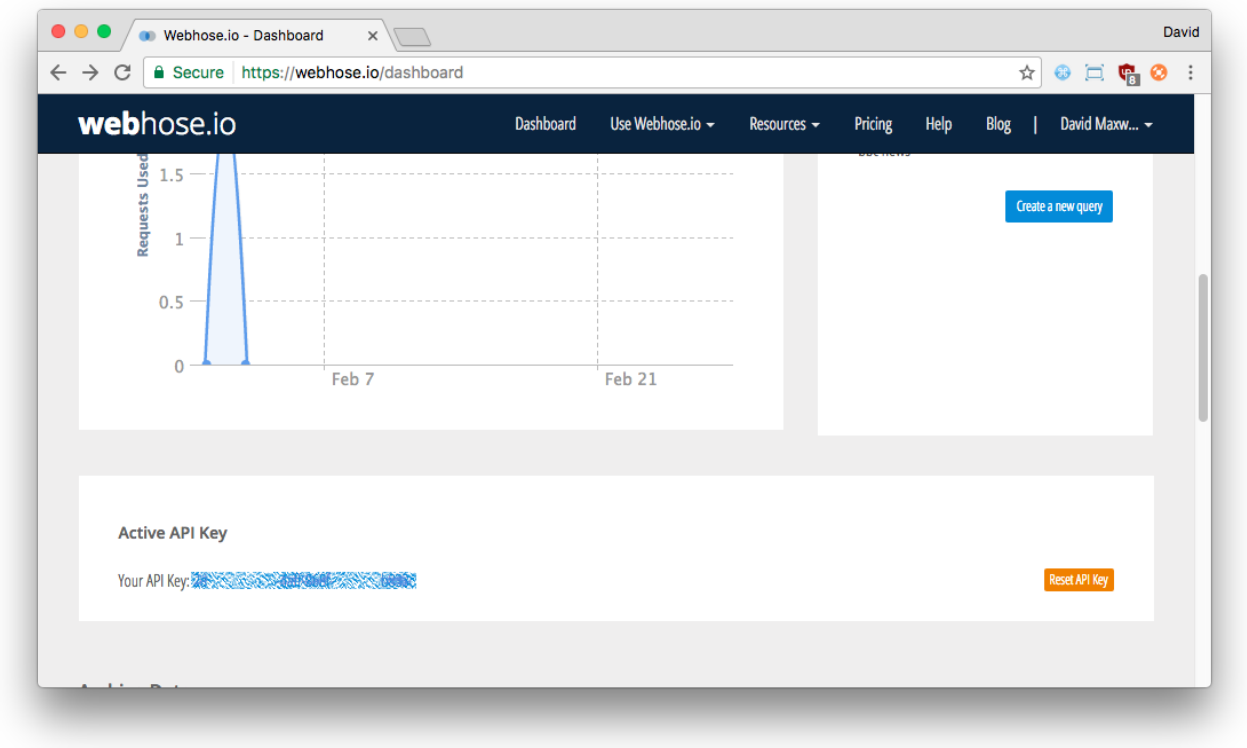
An Application Programming Interface specifies how software components should interact with one another. In the context of web applications, an API is considered as a set of HTTP requests along with a definition of the structures of response messages that each request can return. Any meaningful service that can be offered over the Internet can have its own API - we aren't limited to web search. For more information on web APIs, Luis Rei provides an excellent tutorial on APIs.

Registering for a Webhose API Key

To obtain a Webhose API key, you must first register for a free Webhose account. Head over to `https://www.webhose.io` in your Web browser, and sign up by clicking *'Use it for free'* at the top right of the page. You don't need to provide a company name – and in company e-mail, simply provide a valid e-mail address.

Once you have created your account, you'll be taken to the Webhose *Dashboard*, as can be seen below. From the dashboard, you can see a count of how many queries you have issued over the

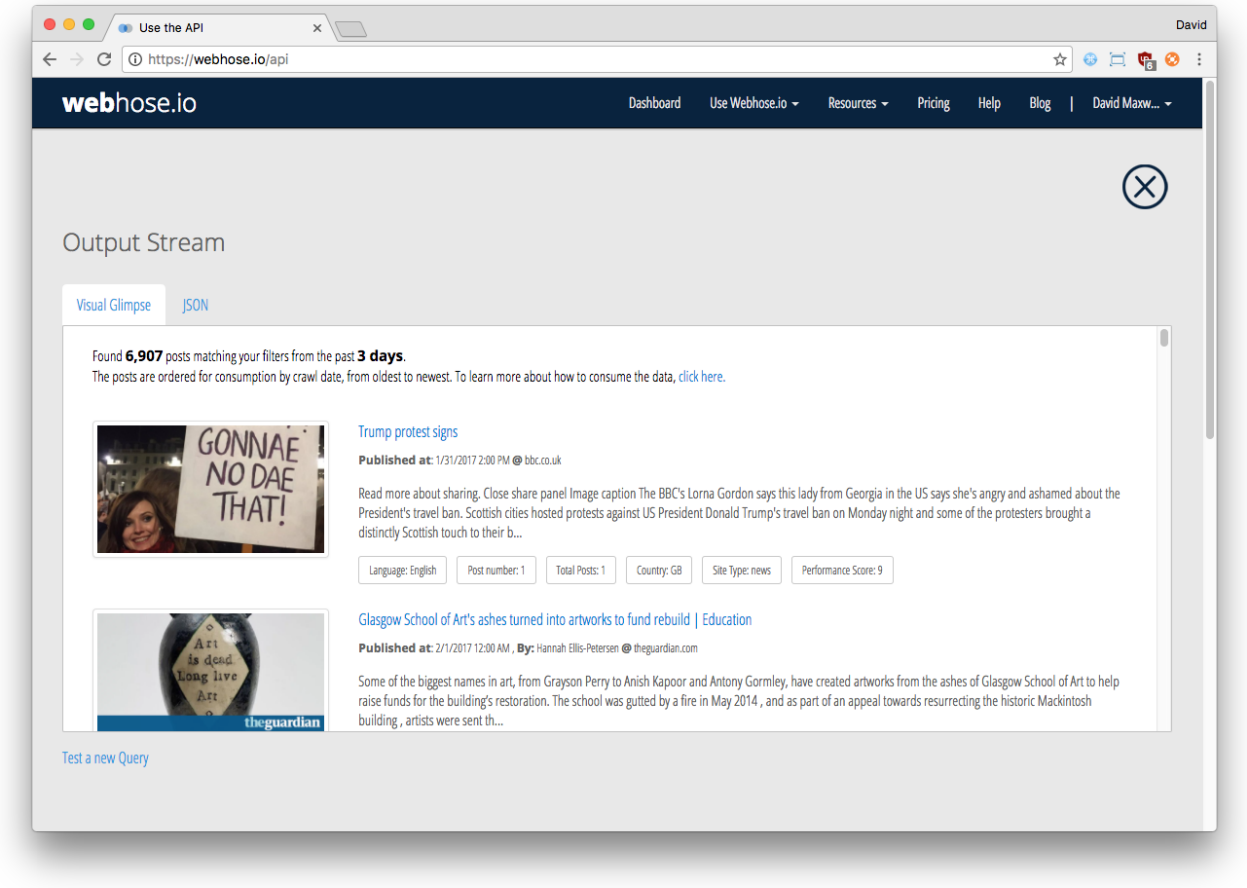
past month, and how many free queries you have remaining. There's also a neat little graph which demonstrates the rate at which you issue queries to Webhose over time. Scroll down the page, and you'll find a section called *Active API Key*. **Take a note of the key shown here by copying it into a blank text file – you'll be needing this later on.** This is your unique key, that when sent with a request to the Webhose API, will identify you to their servers.



The Webhose dashboard, showing where the API key is displayed on the page. You'll most likely have to scroll down to see it. In the screenshot, the API key has been obscured. Keep your key safe and secure!

Once you have your API key, scroll back up to the top of the Webhose dashboard, and click the *Get live data* button, which is in blue. You'll then be taken to the API page, which allows you to play around with the Webhose API interface. Try it out!

1. In the box under *Return posts containing the following keywords*, enter a query.
2. In the *Sort by* dropdown box, choose *Relevancy*.
3. You can then choose a value for *Return posts crawled since*, but leaving it at the 3 day default should be fine.
4. Click *Test the Query*, and you'll then be presented with a series of results to the query you entered. The screenshot below shows example output for the query *Glasgow*.



A sample response from the Webhose API for the query *Glasgow*. Shown is the *Visual Glimpse*; you can also see the raw JSON response from the server by clicking the *JSON* tab.

Have a look at what you get back, and also have a look at the raw JSON response that is returned by the Webhose API. You can do this by clicking on the *JSON* tab. You can try copying and pasting the JSON response in to an online JSON pretty printer to see how it's structured if you want. Close the response by clicking the *X* to the right of the *Output Stream* title, and you'll be returned the API page. You can now scroll down to find the *Integration Examples* section. Make sure the *Endpoint* tab is selected, and have a look at the URL that you are shown. This is the URL that your Rango app will be communicating with to obtain search results – or, in other words, the *endpoint URL*. We'll be making use of it later. An example of the Webhose API endpoint URL – for a given configuration, with the API key and query redacted – is shown below.

```
http://webhose.io/search ? token=<KEY>&format=json&q=<QUERY>&sort=relevancy
```

The URL can essentially be split into two parts - the *base URL* on the left of the question mark, and the querystring to the right. Consider the base URL as the path to the API, and the querystring a series of key and value pairings (e.g. `format` for a key, `json` for a value) that tell the API exactly what you want to get from it. You'll see in the code sample shortly that we take the same process of splitting the URL into a base URL before combining the querystring together to get the complete request URL.

13.2 Adding Search Functionality

Now you've got your Webhose API key, you're ready to implement functionality in Python that issues queries to the Webhose API. Create a new module (file) in the `rango` app directory called `webhose_search.py`, and add the following code – picking the correct one for your Python version. As mentioned earlier in the book, it's better you go through and type the code out – you'll be thinking about how it works as you type (and understanding what's going on), rather than blindly copying and pasting.

Differences between Python 2 and 3

In Python 3, the `urllib` package was refactored, so the way that we connect and work with external web resources has changed from Python 2.7+. Below we have two versions of the code, one for Python 2.7+ and one for Python 3+. Make sure you use the correct one for your environment.

Python 2 Version

```
1  import json
2  import urllib
3  import urllib2
4
5  def read_webhose_key():
6      """
7      Reads the Webhose API key from a file called 'search.key'.
8      Returns either None (no key found), or a string representing the key.
9      Remember: put search.key in your .gitignore file to avoid committing it!
10     """
11     # See Python Anti-Patterns - it's an awesome resource!
12     # Here we are using "with" when opening files.
13     # http://docs.quantifiedcode.com/python-anti-patterns/maintainability/
14     webhose_api_key = None
15
16     try:
17         with open('search.key', 'r') as f:
18             webhose_api_key = f.readline().strip()
19     except:
20         raise IOError('search.key file not found')
21
22     return webhose_api_key
23
```

```
24  def run_query(search_terms, size=10):
25      """
26      Given a string containing search terms (query), and a number of results to
27      return (default of 10), returns a list of results from the Webhose API,
28      with each result consisting of a title, link and summary.
29      """
30      webhose_api_key = read_webhose_key()
31
32      if not webhose_api_key:
33          raise KeyError('Webhose key not found')
34
35      # What's the base URL for the Webhose API?
36      root_url = 'http://webhose.io/search'
37
38      # Format the query string - escape special characters.
39      query_string = urllib.quote(search_terms)
40
41      # Use string formatting to construct the complete API URL.
42      # search_url is a string split over multiple lines.
43      search_url = ('{root_url}?token={key}&format=json&q={query}'
44                    '&sort=relevancy&size={size}').format(
45                      root_url=root_url,
46                      key=webhose_api_key,
47                      query=query_string,
48                      size=size)
49
50      results = []
51
52      try:
53          # Connect to the Webhose API, and convert the response to a
54          # Python dictionary.
55          response = urllib2.urlopen(search_url).read()
56          json_response = json.loads(response)
57
58          # Loop through the posts, appending each to the results list as
59          # a dictionary. We restrict the summary to the first 200
60          # characters, as summary responses from Webhose can be long!
61          for post in json_response['posts']:
62              results.append({'title': post['title'],
63                              'link': post['url'],
64                              'summary': post['text'][:200]})
65      except:
```

```
66          print("Error when querying the Webhose API")
67
68      # Return the list of results to the calling function.
69      return results
```

Python 3 Version

```
1   import json
2   import urllib.parse  # Py3
3   import urllib.request  # Py3
4
5   def read_webhose_key():
6       """
7       Reads the Webhose API key from a file called 'search.key'.
8       Returns either None (no key found), or a string representing the key.
9       Remember: put search.key in your .gitignore file to avoid committing it!
10      """
11      # See Python Anti-Patterns - it's an awesome resource!
12      # Here we are using "with" when opening files.
13      # http://docs.quantifiedcode.com/python-anti-patterns/maintainability/
14      webhose_api_key = None
15
16      try:
17          with open('search.key', 'r') as f:
18              webhose_api_key = f.readline().strip()
19      except:
20          raise IOError('search.key file not found')
21
22      return webhose_api_key
23
24  def run_query(search_terms, size=10):
25      """
26      Given a string containing search terms (query), and a number of results to
27      return (default of 10), returns a list of results from the Webhose API,
28      with each result consisting of a title, link and summary.
29      """
30      webhose_api_key = read_webhose_key()
31
32      if not webhose_api_key:
33          raise KeyError('Webhose key not found')
34
35      # What's the base URL for the Webhose API?
```

```
36      root_url = 'http://webhose.io/search'
37
38      # Format the query string - escape special characters.
39      query_string = urllib.parse.quote(search_terms)  # Py3
40
41      # Use string formatting to construct the complete API URL.
42      # search_url is a string split over multiple lines.
43      search_url = ('{root_url}?token={key}&format=json&q={query}'
44                    '&sort=relevancy&size={size}').format(
45                      root_url=root_url,
46                      key=webhose_api_key,
47                      query=query_string,
48                      size=size)
49
50      results = []
51
52      try:
53          # Connect to the Webhose API, and convert the response to a
54          # Python dictionary.
55          response = urllib.request.urlopen(search_url).read().decode('utf-8')
56          json_response = json.loads(response)
57
58          # Loop through the posts, appending each to the results list as
59          # a dictionary. We restrict the summary to the first 200
60          # characters, as summary responses from Webhose can be long!
61          for post in json_response['posts']:
62              results.append({'title': post['title'],
63                              'link': post['url'],
64                              'summary': post['text'][:200]})
65      except:
66          print("Error when querying the Webhose API")
67
68      # Return the list of results to the calling function.
69      return results
```

In the code samples above, we have implemented two functions: one to retrieve your Webhose API key from a local file (through function `read_webhose_key()`), and another to issue a query to the Webhose API and return results (`run_query()`). Below, we discuss how both of the functions work.

`read_webhose_key()` – Reading the Webhose API Key

The `read_webhose_key()` function reads in your Webhose API key from a file called `search.key`. This file should be located in your Django project's root directory, **not Rango's directory** (i.e.

<workspace>/tango_with_django_project/). We have created this function as it allows you to separate your private API key from the code that utilises it. This is advantageous in scenarios where code is shared publicly (i.e. on GitHub) – you don't want people using your API key!

You should create the `search.key` file now. Take the Webhose API key you copied earlier, and save it into the file `<workspace>/tango_with_django_project/search.key`. The key should be the only contents of the file – nothing else should exist within it. Avoid committing the file to your GitHub repository by updating your repository's `.gitignore` file to exclude any files with a `.key` extension by adding the line `*.key`. This way, your key is only stored locally, and cannot be committed to your remote Git repository by accident.

Keys

Keep them secret, keep them safe!

Don't let anyone use your code. If they misuse it, you could be banned from the service to which it corresponds. Or worse, end up having to pay for the services you did not use.

`run_query()` – Executing the Query

The `run_query()` function takes two parameters: `search_terms`, a string representing a user's query; and `size`, an optional parameter, set to a default of `10`. This second parameter allows us to control the number of results to return from the Webhose API. Given these parameters, the function then communicates with the Webhose API, and returns a series of Python dictionaries within a list, with each dictionary representing an individual result – consisting of a result `title`, `link` and `summary`. The inline commentary in the function definitions above (for both Python 2.7.x and Python 3) explain what's happening at each stage – check out the commentary further to increase your understanding of what is going on.

To summarise, the logic of `run_query()` can be broadly split into seven main tasks, which are explained below.

- First, the function obtains the Webhose API key by calling the `read_webhose_key()` function.
- The function then correctly formats the query string to be sent to the API. This is done by *URL encoding* the string, converting special characters such as spaces to a format that can be understood by Web servers and browsers. As an example, the space character `' '` is converted to `%20`.
- The complete URL for the Webhose API call is then constructed by concatenating the URL encoded `search_terms` string and `size` parameters – as well as your Webhose API key – together into a series of querystring arguments as dictated by the Webhose API documentation.
- We then connect to the Webhose API using the Python `urllib2` (for Python 2.7.x) or `urllib` modules. The string response from the server is then saved in the variable `response`.

- This response is then converted to a Python dictionary object using the Python `json` library.
- The dictionary is then iterated over, with each result returned from the Webhose API saved to the `results` list as a dictionary, consisting of `title`, `link` and `summary` key/value pairings.
- The `results` list is then returned by the function.

Exploring API Options

When starting off with a new API, it's always a good idea to explore the provided documentation to see what options you can play with. We recommend exploring the Webhose API documentation and play around with some of the options that you can vary.

Exercises

Extend your `webhose_search.py` module so that it can be run independently. By this, we mean running `python webhose_search.py` from your terminal or Command Prompt, without running Django's development server. Specifically, you should implement functionality that:

- prompts the user to enter a query, i.e. use `raw_input()`; and
- issues the query via `run_query()`, and prints the results.

For each result, you should display the corresponding `title` and `summary`, with a line break between each result.

You'll also need to modify the `read_webhose_key()` function so that the `search.key` file can be found from the `rango` directory in which `webhose_search.py` is launched. When running the Django development server, Python would expect to find `search.key` in the directory you launched `manage.py`. When you run `webhose_search.py` in the `rango` directory, you'll need to look one directory up. How could you modify the `read_webhose_key()` function to work both ways?

If you are developing Rango on a Windows computer with Python 2.7.x, you'll need to encode the output of each `print` statement using the `str encode()` function with `utf-8`. For example, to display the `title` of a result, you would use `print(result['title']).encode('utf-8')`. This is due to the way that Python calls underlying Windows functions to output content. If you receive a `UnicodeEncodeError`, this may be your solution. Python 3 should be unaffected by this issue.

Hint

You've already done this in your population script for Rango! Try following the Google Python Style Guide to make everything look the part by adding a `main()` function, and calling whatever you need to from there. You should also make use of the following line – if you aren't sure what this line means, have a look online for an answer, or refer back to the section of the tutorial discussing the population script.

```
if __name__ == '__main__':
    main()
```

Using this line will ensure that you can run your module independently of anything else – yet still include the module within another Python program (i.e. your Django project) and not have code automatically executed when you `import`.

To modify your `read_webhose_key()` function, there's a few things you could try. The easiest approach would be to make use of the `os.path.isfile()` function to check if `search.key` is a file that exists in the current directory. If it doesn't exist, then you could assume that the key can be found in `../search.key` – one directory up (`..`) from where your script is running. To access the `isfile()` function, you'll need to make sure the `os` module is imported at the top of `webhose_search.py`.

13.3 Putting Webhose Search into Rango

Now that we have successfully implemented the search functionality module `webhose_search.py`, we need to integrate it into our Rango app. There are three main steps that we need to complete for this to work.

1. We must first create a `search.html` template that extends from Rango's `base.html` template. The `search.html` template will include a HTML `<form>` to capture the user's query, as well as template code to present any results.
2. We then create a Django view to handle the rendering of the `search.html` template for us, as well as calling the `run_query()` function we defined earlier in this chapter. W then allow users to access the new view by mapping it to a new URL within Rango's `urls.py` module.

Adding a Search Template

Let's first start at the beginning, and create a new template called `search.html`. Place it within the `rango` directory of your project's `templates` directory. Add the following HTML markup and Django template code.

```
{% extends 'rango/base.html' %}
{% load staticfiles %}

{% block title %} Search {% endblock %}

{% block body_block %}
<div>
    <h1>Search with Rango</h1>
    <br/>
    <form class="form-inline" id="user_form"
          method="post" action="{% url 'search' %}">
        {% csrf_token %}
        <div class="form-group">
            <input class="form-control" type="text" size="50"
                   name="query" value="" id="query" />
        </div>
        <button class="btn btn-primary" type="submit" name="submit"
                value="Search">Search</button>
    </form>

    <div>
        {% if result_list %}
        <h3>Results</h3>
        <!-- Display search results in an ordered list -->
        <div class="list-group">
        {% for result in result_list %}
            <div class="list-group-item">
                <h4 class="list-group-item-heading">
                    <a href="{{ result.link }}">{{ result.title }}</a>
                    </h4>
                    <p class="list-group-item-text">{{ result.summary }}</p>
            </div>
        {% endfor %}
        </div>
        {% endif %}
    </div>
</div>
{% endblock %}
```

The template code above performs two key tasks.

- The template presents a search box and *Search* button within a HTML `<form>` for users to enter and submit their queries.

- If a `results_list` object is passed to the template's context when rendering, the template then iterates through the `results_list` object, rendering the results contained within. The template expects that each result consists of a `title`, `link` and `summary` – consistent with what is returned from the `run_query()` function defined earlier in this chapter.

To style the page that is rendered, we have made use of Bootstrap panels, list groups, and inline forms.

The Django view in the following subsection will only pass through results to the template above in a context variable called `results_list` when the user issues a query. Initially, no results will be available to show – so `results_list` will not be provided to the template, and thus, no results will be rendered.

Adding the View

With the new template added, we can then add the view that prompts the rendering of our template. Add the following `search()` view to Rango's `views.py` module.

```
def search(request):
    result_list = []

    if request.method == 'POST':
        query = request.POST['query'].strip()
        if query:
            # Run our Webhose search function to get the results list!
            result_list = run_query(query)

    return render(request, 'rango/search.html', {'result_list': result_list})
```

By now, the code above should be pretty self explanatory to you. The only major addition here that you wouldn't have done so far is the calling of the `run_query()` function we defined earlier in this chapter. To call it, we are also required to `import` the `webhose_search.py` module, too. Ensure that before you run the Django development server that you add the following `import` statement at the top of Rango's `views.py` module.

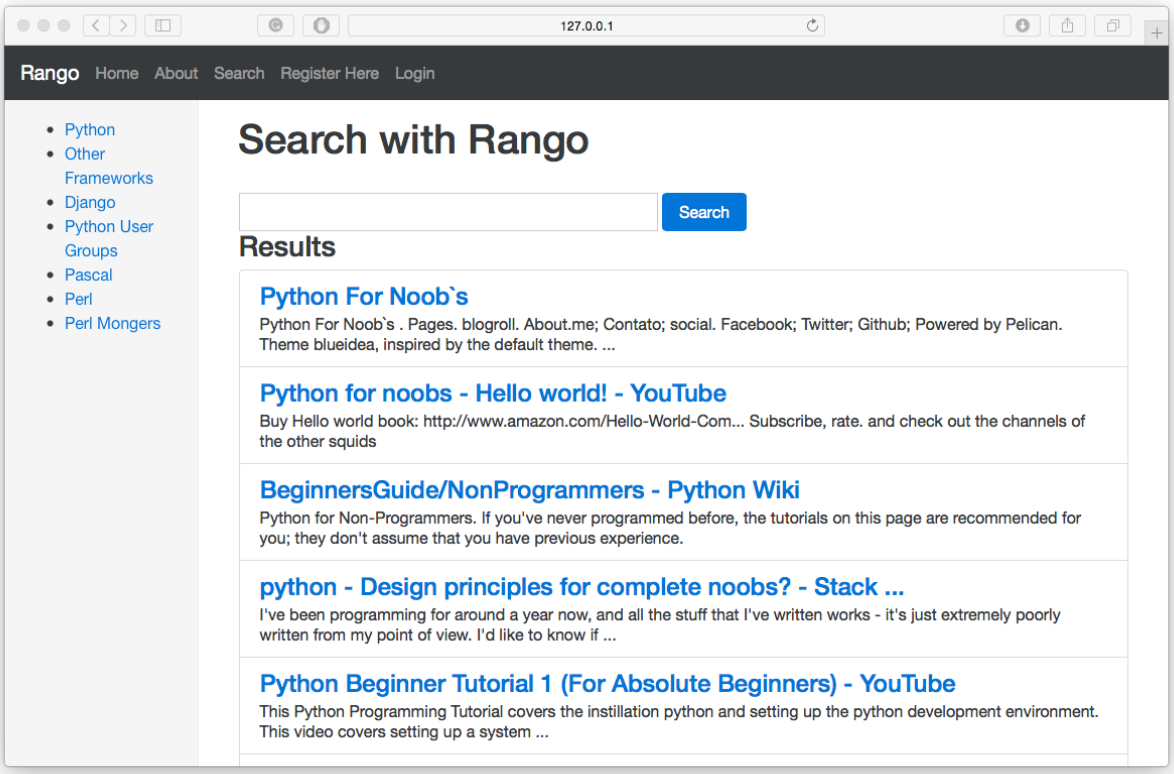
```
from rango.webhose_search import run_query
```

We then need to create the URL mapping between a URL and the `search()` view, as well as make it possible for users to navigate to the search page through Rango's navigation bars.

- Add the URL mapping between the `search()` view and the URL `/rango/search/`, with `name='search'`. This can be done by adding the line `url(r'search/$', views.search, name='search')` to Rango's `urls.py` module.

- Update the `base.html` navigation bar to include a link to the search page. Remember to use the `url` template tag to reference the link, rather than hard coding it into the template.
- Finally, ensure that the `search.key` file is created – with your Webhose API key contained within it – and it is located in your Django project's root directory (i.e. `<workspace>/tango_with_django_project/`, alongside `manage.py`).

Once you have put the URL mapping together and added a link to the search page, you should now be able to issue queries to the Webhose API, with results now showing in the Rango app – as shown in the figure below.



Searching for "Python for Noobs".

Additional Exercise

You may notice that when you issue a query, the query disappears when the results are shown. This is not very user friendly. Update the `search()` view and `search.html` template so that the user's query is displayed within the search box.

Within the view, you will need to put the `query` into the context dictionary. Within the template, you will need to show the query text in the search box.

14. Making Rango Tango! Exercises

So far we have been adding in different pieces of functionality to Rango. We've been building up the application in this manner to get you familiar with the Django Framework, and to learn about how to construct the various parts of an application. However, at the moment, Rango is not very cohesive or interactive. In this chapter, we challenge you to improve the application and its user experience by bringing together some of the functionality that we have already implemented along with some other features.

To make Rango more coherent, integrated and interactive, it would be nice to add the following functionality.

- Track the clickthroughs of Categories and Pages, i.e.:
  - count the number of times a category is viewed
  - count the number of times a page is viewed via Rango, and
  - collect likes for categories (see Django and Ajax Chapter).
- Integrate the browsing and searching within categories, i.e.:
  - instead of having a disconnected search page, let users search for pages on each specific category page, and
  - let users filter the set of categories shown in the side bar (see Django and Ajax Chapter).
- Provide services for Registered Users, i.e.:
  - Assuming you have switched the `django-registration-redux`, we need to setup the registration form to collect the additional information (i.e. website, profile picture)
  - let users view their profile
  - let users edit their profile, and
  - let users see the list of users and their profiles.

Note

We won't be working through all of these tasks right now. Some will be taken care of in the Django and Ajax Chapter, while others will be left to you to complete as additional exercises.

Before we start to add this additional functionality we will make a todo list to plan our workflow for each task. Breaking tasks down into sub-tasks will greatly simplify the implementation so that we are attacking each one with a clear plan. In this chapter, we will provide you with the workflow for a number of the above tasks. From what you have learnt so far, you should be able to fill in the gaps and implement most of it on your own (except those requiring AJAX). In the following chapter, we have included hints, tips and code snippets elaborating on how to implement these features. Of course, if you get really stuck, you can always check out our implementation on GitHub.

14.1 Track Page Clickthroughs

Currently, Rango provides a direct link to external pages. This is not very good if you want to track the number of times each page is clicked and viewed. To count the number of times a page is viewed via Rango you will need to perform the following steps.

- Create a new view called `track_url()`, and map it to URL `/rango/goto/` and name it `'name=goto'`.
- The `track_url()` view will examine the HTTP `GET` request parameters and pull out the `page_id`. The HTTP `GET` requests will look something like `/rango/goto/?page_id=1`.
  - In the view, select/get the `page` with `page_id` and then increment the associated `views` field, and `save()` it.
  - Have the view redirect the user to the specified URL using Django's `redirect` method. Remember to include the import, `from django.shortcuts import redirect`
  - If no parameters are in the HTTP `GET` request for `page_id`, or the parameters do not return a `Page` object, redirect the user to Rango's homepage. Use the `reverse` method from `django.core.urlresolvers` to get the URL string and then redirect. If you are using Django 1.10, then you can import the `reverse` method from `django.shortcuts`.
  - See Django Shortcut Functions for more on `redirect` and `reverse`.
- Update the `category.html` so that it uses `/rango/goto/?page_id=XXX`.
  - Remember to use the `url` template tag instead of using the direct URL i.e.

```
<a href="{% url 'goto' %}?page_id={{page.id}}"\>
```

`GET` Parameters Hint

If you're unsure of how to retrieve the `page_id` *querystring* from the HTTP `GET` request, the following code sample should help you.

```
page_id = None
if request.method == 'GET':
    if 'page_id' in request.GET:
        page_id = request.GET['page_id']
```

Always check the request method is of type `GET` first, then you can access the dictionary `request.GET` which contains values passed as part of the request. If `page_id` exists within the dictionary, you can pull the required value out with `request.GET['page_id']`.

You could also do this without using a *querystring*, but through the URL instead, i.e. `/rango/goto/<page_id>/`. In which case you would need to create a `urlpattern` that pulls out the `page_id`, i.e. `r'goto/(?P<page_id>\d+)/$'`.

14.2 Searching Within a Category Page

Rango aims to provide users with a helpful directory of useful web pages. At the moment, the search functionality is essentially independent of the categories. It would be nicer to have search integrated within the categories. We will assume that a user will first browse through the category of interest. If they can't find a relevant page, they can then search. If they find a page that is relevant, then they can add it to the category. Let's focus on the first problem, of putting search on the category page. To do this, perform the following steps:

- Remove the generic *Search* link from the menu bar, i.e. we are decommissioning the global search functionality.
- Take the search form and results template markup from `search.html` and place it into `category.html`.
- Update the search form so that action refers back to the category page, i.e.:

```
<form class="form-inline" id="user_form"
    method="post" action="{% url 'show_category'  category.slug %}">
```

- Update the category view to handle a HTTP `POST` request. The view must then include any search results in the context dictionary for the template to render.
- Also, lets make it so that only authenticated users can search. So to restrict access within the `category.html` template use:

```
{% if user.authenticated %}
    <!-- Insert search code here -->
{% endif %}
```

14.3 Create and View Profiles

If you have swapped over to the `django-registration-redux` package, then you'll have to collect the `UserProfile` data. To do this, instead of redirecting the user to the Rango index page, you will need to redirect them to a new form, to collect the user's profile picture and URL details. To add the UserProfile registration functionality, you need to:

- create a `profile_registration.html` which will display the `UserProfileForm`;
- create a `UserProfileForm` `ModelForm` class to handle the new form;
- create a `register_profile()` view to capture the profile details;
- map the view to a URL, i.e. `rango/register_profile/`; and

- in the `MyRegistrationView`, update the `get_success_url()` to point to `rango/add_profile/`.

Another useful feature is to let users inspect and edit their own profile. Undertake the following steps to add this functionality.

- First, create a template called `profile.html`. In this template, add in the fields associated with the user profile and the user (i.e. username, email, website and picture).
- Create a view called `profile()`. This view will obtain the data required to render the user profile template.
- Map the URL `/rango/profile/` to your new `profile()` view.
- In the base template add a link called *Profile* into the menu bar, preferably with other user-related links. This should only be available to users who are logged in (i.e. `{% if user.is_authenticated %}`).

To let users browse through user profiles, you can also create a users page that lists all the users. If you click on a user page, then you can see their profile. However, you must make sure that a user is only able to edit their profile!

Referencing Uploaded Content in Templates

If you have successfully completed all of the Templates and Media chapter, your Django setup should be ready to deal with the uploading and serving of user media files. You should be able to reference the `MEDIA_URL` URL (defined in `settings.py`) in your templates through use of the `{{ MEDIA_URL }}` tag, provided by the media template context processor, e.g. `<img src="{{ MEDIA_URL}}cat.jpg">`.

In the next chapter, we provide a series of hints and tips to help you complete the aforementioned features.

15. Making Rango Tango! Hints

Hopefully, you will have been able to complete the exercises given the workflows we provided. If not, or if you need a little help, have a look at the potential solutions we have provided below, and use them within your version of Rango.

Got a better solution?

The solutions provided in this chapter are only one way to solve each problem. They are based on what we have learnt so far. However, if you implement them differently, feel free to share your solutions with us - and tweet links to @tangowithdjango for others to see.

15.1 Track Page Clickthroughs

Currently, Rango provides a direct link to external pages. This is not very good if you want to track the number of times each page is clicked and viewed. To count the number of times a page is viewed via Rango, you'll need to perform the following steps.

Creating a URL Tracking View

Create a new view called `track_url()` in `/rango/views.py` which takes a parameterised HTTP `GET` request (i.e. `rango/goto/?page_id=1`) and updates the number of views for the page. The view should then redirect to the actual URL.

```
from django.shortcuts import redirect

def track_url(request):
    page_id = None
    url = '/rango/'
    if request.method == 'GET':
        if 'page_id' in request.GET:
            page_id = request.GET['page_id']

            try:
                page = Page.objects.get(id=page_id)
                page.views = page.views + 1
```

```
                page.save()
                url = page.url
            except:
                pass

    return redirect(url)
```

Be sure that you import the `redirect()` function to `views.py` if it isn't included already!

```
from django.shortcuts import redirect
```

Mapping URL

In `/rango/urls.py` add the following code to the `urlpatterns` tuple.

```
url(r'^goto/$', views.track_url, name='goto'),
```

Updating the Category Template

Update the `category.html` template so that it uses `rango/goto/?page_id=XXX` instead of providing the direct URL for users to click.

```
{% for page in pages %}
    <li>
        <a href="{% url 'goto' %}?page_id={{page.id}}">{{ page.title }}</a>
        {% if page.views > 1 %}
            ({{ page.views }} views)
        {% elif page.views == 1 %}
            ({{ page.views }} view)
        {% endif %}
    </li>
{% endfor %}
```

Here you can see that in the template we have added some control statements to display `view`, `views` or nothing depending on the value of `page.views`.

Updating Category View

Since we are tracking the number of clickthroughs you can now update the `category()` view so that you order the pages by the number of views:

```
pages = Page.objects.filter(category=category).order_by('-views')
```

Now, confirm it all works, by clicking on links, and then going back to the category page. Don't forget to refresh or click to another category to see the updated page.

15.2 Searching Within a Category Page

Rango aims to provide users with a helpful directory of page links. At the moment, the search functionality is essentially independent of the categories. It would be nicer however to have search integrated into category browsing. Let's assume that a user will first browse their category of interest first. If they can't find the page that they want, they can then search for it. If they find a page that is suitable, then they can add it to the category that they are in. Let's tackle the first part of this description here.

We first need to remove the global search functionality and only let users search within a category. This will mean that we essentially decommission the current search page and search view. After this, we'll need to perform the following.

Decommissioning Generic Search

Remove the generic *Search* link from the menu bar by editing the `base.html` template. You can also remove or comment out the URL mapping in `rango/urls.py`.

Creating a Search Form Template

After the categories add in a new `div` at the bottom of the template in `category.html`, and add in the search form. This is very similar to the template code in the `search.html`, but we have updated the action to point to the `show_category` page. We also pass through a variable called `query`, so that the user can see what query has been issued.

```
<form class="form-inline" id="user_form"
    method="post" action="{% url 'show_category'  category.slug %}">
    {% csrf_token %}
    <div class="form-group">
        <input class="form-control" type="text" size="50"
            name="query" value="{{ query }}" id="query" />
    </div>
    <button class="btn btn-primary" type="submit" name="submit"
        value="Search">Search</button>
</form>
```

After the search form, we need to provide a space where the results are rendered. Again, this code is similar to the template code in `search.html`.

```
<div>
{% if result_list %}
    <h3>Results</h3>
    <!-- Display search results in an ordered list -->
    <div class="list-group">
    {% for result in result_list %}
        <div class="list-group-item">
            <h4 class="list-group-item-heading">
                <a href="{{ result.link }}">{{ result.title }}</a>
            </h4>
            <p class="list-group-item-text">{{ result.summary }}</p>
        </div>
    {% endfor %}
    </div>
{% endif %}
</div>
```

Remember to wrap the search form and search results with `{% if user.authenticated %}` and `{% endif %}`, so that only authenticated users can search. You don't want random users to be wasting your monthly search API's budget!

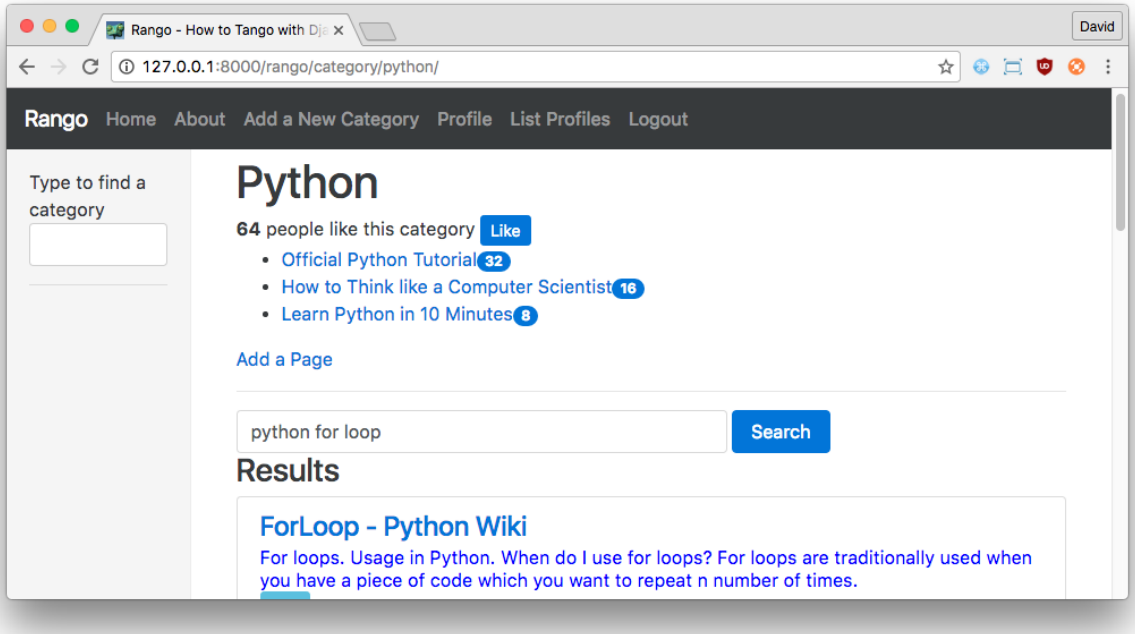
Updating the Category View

Update the category view to handle a HTTP `POST` request (i.e. when the user submits a search) and inject the results list into the context. The following code demonstrates this new functionality.

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

    try:
        # Can we find a category name slug with the given name?
        # If we can't, the .get() method raises a DoesNotExist exception.
        # So the .get() method returns one model instance or raises an exception.
        category = Category.objects.get(slug=category_name_slug)
        # Retrieve all of the associated pages.
        # Note that filter() returns a list of page objects or an empty list
        pages = Page.objects.filter(category=category)
        # Adds our results list to the template context under name pages.
        context_dict['pages'] = pages
        # We also add the category object from
```

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


    # New code added here to handle a POST request

    # create a default query based on the category name
    # to be shown in the search box
    context_dict['query'] = category.name

    result_list = []
    if request.method == 'POST':
        query = request.POST['query'].strip()

        if query:
            # Run our search API function to get the results list!
            result_list = run_query(query)
            context_dict['query'] = query
            context_dict['result_list'] = result_list


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

Notice that the `context_dict` now includes the `result_list` and `query`. If there is no query, we provide a default query, i.e. the category name. The query box then displays this value.



Rango's updated category view, complete with search API search functionality.

15.3 Creating a `UserProfile` Instance

This section provides a solution for creating Rango `UserProfile` accounts. Recall that the standard Django `auth` `User` object contains a variety of standard information regarding an individual user, such as a username and password. We however chose to implement an additional `UserProfile` model to store additional information such as a user's Website and a profile picture. Here, we'll go through how you can implement this, using the following steps.

- Create a `profile_registration.html` that will display the `UserProfileForm`.
- Create a `UserProfileForm` `ModelForm` class to handle the new form.
- Create a `register_profile()` view to capture the profile details.
- Map the view to a URL, i.e. `rango/register_profile/`.
- In the `MyRegistrationView` defined in the Django `registration-redux` chapter, update the `get_success_url()` to point to `rango/add_profile/`.

The basic flow for a registering user here would be:

- clicking the `Register` link;
- filling out the initial Django `registration-redux` form (and thus registering);

- filling out the new `UserProfileForm` form; and
- completing the registration.

This assumes that a user will be registered with Rango *before* the profile form is saved.

Creating a Profile Registration Template

First, let's create a template that'll provide the necessary markup for displaying an additional registration form. In this solution, we're going to keep the Django `registration-redux` form separate from our Profile Registration form - just to delineate between the two. If you can think of a neat way to mix both forms together, why not try it?

Create a template in Rango's templates directory called `profile_registration.html`. Within this new template, add the following markup and Django template code.

```
{% extends "rango/base.html" %}

{% block title_block %}
    Registration - Step 2
{% endblock %}

{% block body_block %}
    <h1>Registration - Step 2</h1>
    <form method="post" action="." enctype="multipart/form-data">
        {% csrf_token %}
        {{ form.as_p }}
        <input type="submit" value="Submit" />
    </form>
{% endblock %}
```

Much like the previous Django `registration-redux` form that we created previously, this template inherits from our `base.html` template, which incorporates the basic layout for our Rango app. We also create an HTML `form` inside the `body_block` block. This will be populated with fields from a `form` object that we'll be passing into the template from the corresponding view (see below).

Don't Forget `multipart/form-data`!

When creating your form, don't forget to include the `enctype="multipart/form-data"` attribute in the `<form>` tag. We need to set this to instruct the Web browser and server that no character encoding should be used - as we are performing *file uploads*. If you don't include this attribute, the image upload component will not work.

Creating the `UserProfileForm` Class

Looking at Rango's `models.py` module, you should see a `UserProfile` model that you implemented previously. We've included it below to remind you of what it contains - a reference to a Django `django.contrib.auth.User` object, and fields for storing a Website and profile image.

```
class UserProfile(models.Model):
    # This line is required. Links UserProfile to a User model instance.
    user = models.OneToOneField(User)
    # The additional attributes we wish to include.
    website = models.URLField(blank=True)
    picture = models.ImageField(upload_to='profile_images', blank=True)

    # Override the __unicode__() method to return out something meaningful!
    def __str__(self):
        return self.user.username
```

In order to provide the necessary HTML markup on the fly for this model, we need to implement a Django `ModelForm` class, based upon our `UserProfile` model. Looking back to the chapter detailing Django forms, we can implement a `ModelForm` for our `UserProfile` as shown in the example below. Perhaps unsurprisingly, we call this new class `UserProfileForm`.

```
class UserProfileForm(forms.ModelForm):
    website = forms.URLField(required=False)
    picture = forms.ImageField(required=False)

    class Meta:
        model = UserProfile
        exclude = ('user',)
```

Note the inclusion of optional (through `required=False`) `website` and `picture` HTML form fields - and the nested `Meta` class that associates the `UserProfileForm` with the `UserProfile` model. The `exclude` attribute instructs the Django form machinery to *not* produce a form field for the `user` model attribute. As the newly registered user doesn't have reference to their `User` object, we'll have to manually associate this with their new `UserProfile` instance when we create it later.

Creating a Profile Registration View

Next, we need to create the corresponding view to handle the processing of a `UserProfileForm` form, the subsequent creation of a new `UserProfile` instance, and instructing Django to render any response with our new `profile_registration.html` template. By now, this should be pretty straightforward to implement. Handling a form means being able to handle a request to render the form (via a HTTP `GET`), and being able to process any entered information (via a HTTP `POST`). A possible implementation for this view is shown below.

```
@login_required
def register_profile(request):
    form = UserProfileForm()

    if request.method == 'POST':
        form = UserProfileForm(request.POST, request.FILES)
        if form.is_valid():
            user_profile = form.save(commit=False)
            user_profile.user = request.user
            user_profile.save()

            return redirect('index')
        else:
            print(form.errors)

    context_dict = {'form':form}

    return render(request, 'rango/profile_registration.html', context_dict)
```

Upon creating a new `UserProfileForm` instance, we then check our `request` object to determine if a `GET` or `POST` was made. If the request was a `POST`, we then recreate the `UserProfileForm`, using data gathered from the `POST` request. As we are also handling a file image upload (for the user's profile image), we also need to pull the uploaded file from `request.FILES`. We then check if the submitted form was valid - meaning that form fields were filled out correctly. In this case, we only really need to check if the URL supplied is valid - since the URL and profile picture fields are marked as optional.

With a valid `UserProfileForm`, we can then create a new instance of the `UserProfile` model with the line `user_profile = form.save(commit=False)`. Setting `commit=False` gives us time to manipulate the `UserProfile` instance before we commit it to the database. This is where can then add in the necessary step to associate the new `UserProfile` instance with the newly created `User` object that has been just created (refer to the flow at the top of this section to refresh your memory). After successfully saving the new `UserProfile` instance, we then redirect the newly created user to Rango's `index` view, using the URL pattern name. If form validation failed for any reason, errors are simply printed to the console. You will probably in your own code want to make the handling of errors more robust.

If the request sent was a HTTP `GET`, the user simply wants to request a blank form to fill out - so we respond by `rendering` the `profile_registration.html` template created above with a blank instance of the `UserProfileForm`, passed to the rendering context dictionary as `form` - thus satisfying the requirement we created in our template. This solution should therefore handle all required scenarios for creating, parsing and saving data from a `UserProfileForm` form.

Can't find `login_required`?

Remember, once a newly registered user hits this view, they will have had a new account created for them - so we can safely assume that he or she is now logged into Rango. This is why we are using the `@login_required` decorator at the top of our view to prevent individuals from accessing the view when they are unauthorised to do so.

If you are receiving an error stating that the `login_required()` function (used as a decorator to our new view) cannot be located, ensure that you have the following `import` statement at the top of your `view.py` module.

```
from django.contrib.auth.decorators import login_required
```

Mapping the View to a URL

Now that our template `ModelForm` and corresponding view have all been implemented, a seasoned Djangoer should now be thinking: *map it!* We need to map our new view to a URL, so that users can access the newly created content. Opening up Rango's `urls.py` module and adding the following line to the `urlpatterns` list will achieve this.

```
url(r'^register_profile/$', views.register_profile, name='register_profile'),
```

This maps our new `register_profile()` view to the URL `/rango/register_profile/`. Remember, the `/rango/` part of the URL comes from your project's `urls.py` module - the remainder of the URL is then handled by the Rango app's `urls.py` module.

Modifying the Registration Flow

Now that everything is (almost) working, we need to tweak the process that users undertake when registering. Back in the Django `registration-redux` chapter, we created a new class-based view called `MyRegistrationView` that changes the URL that users are redirected to upon a successful registration. This needs to be changes from redirecting a user to the Rango homepage (with URL name `index`) to our new user profile registration URL. From the previous section, we gave this the name `register_profile`. This means changing the `MyRegistrationView` class to look like the following example.

```
class MyRegistrationView(RegistrationView):
    def get_success_url(self, user):
        return reverse('register_profile')
```

Now when a user registers, they should be then redirected to the profile registration form – and upon successful completion of that – be redirected to the Rango homepage. It's easy when you know how.

Class-Based Views

In this subsection, we mentioned something called **class-based views**. Class based views are a different, and more elegant, but more sophisticated mechanism, for handling requests. Rather than taking a functional approach as we have done in this tutorial, that is, in our `views.py` we have written functions to handle each request, the class based approach mean inheriting and implementing a series methods to handle the requests. For example, rather than checking if a request was a `get` or a `post`, in the class based approach, you would need to implement a `get()` and `post()` method within the class. When your project and handlers become more complicated, using the Class based approach is more preferable. See the Django Documentation for more information about Class Based Views.

Additional Exercise

- Go through the Django Documentation and study how to create Class-Based Views.
- Update the Rango application to use Class-Based Views.
- Tweet how awesome you are and let us know @tangowithdjango.

15.4 Viewing your Profile

With the creation of a `UserProfile` object now complete, let's implement the functionality to allow a user to view his or her profile and edit it. The process is again pretty similar to what we've done before. We'll need to consider the following aspects:

- the creation of a new template, `profile.html`;
- creating a new view called `profile()` that uses the `profile.html` template; and
- mapping the `profile()` view to a new URL (`/rango/profile`).

We'll also need to provide a new hyperlink in Rango's `base.html` template to access the new view. For this solution, we'll be creating a generalised view that allows you to access the information of any user of Rango. The code will allow logged in users to also edit their profile; but only *their* profile - thus satisfying the requirements of the exercise.

Creating the Template

First, let's create a simple template for displaying a user's profile. The following HTML markup and Django template code should be placed within the new `profile.html` template within Rango's template directory.

```
{% extends 'rango/base.html' %}

{% load staticfiles %}

{% block title %}{{ selecteduser.username }} Profile{% endblock %}

{% block body_block %}

<h1>{{selecteduser.username}} Profile</h1>
<img src="{{ MEDIA_URL }}{{userprofile.picture }}"
     width="300"
     height="300"
     alt="{{selecteduser.username}}" />
<br/>
<div>
    {% if selecteduser.username == user.username %}
        <form method="post" action="." enctype="multipart/form-data">
             {% csrf_token %}
            {{ form.as_p }}
            <input type="submit" value="Update" />
        </form>
    {% else %}
    <p><strong>Website:</strong> <a href="{{userprofile.website}}">
        {{userprofile.website}}</a></p>
    {% endif %}
</div>
<div id="edit_profile"></div>
```

Note that there are a few variables (`selecteduser`, `userprofile` and `form`) that we need to define in the template's context - we'll be doing so in the next section.

The fun part of this template is within the `body_block` block. The template shows the user's profile image at the top. Underneath, the template shows a form allowing the user to change his or her details, which is populated from the `form` variable. This form however is *only shown* when the selected user matches the user that is currently logged in, thus only allowing the presently logged in user to edit his or her profile. If the selected user does not match the currently logged in user, then the selected user's website is displayed - but it cannot be edited.

You should also take not of the fact that we again use `enctype="multipart/form-data"` in the form due to the fact image uploading is used.

Creating the `profile()` View

Based upon the template created above, we can then implement a simple view to handle the viewing of user profiles and submission of form data. In Rango's `views.py` module, create a new view called `profile()`.

```
@login_required
def profile(request, username):
    try:
        user = User.objects.get(username=username)
    except User.DoesNotExist:
        return redirect('index')

    userprofile = UserProfile.objects.get_or_create(user=user)[0]
    form = UserProfileForm(
        {'website': userprofile.website, 'picture': userprofile.picture})

    if request.method == 'POST':
        form = UserProfileForm(request.POST, request.FILES, instance=userprofile)
        if form.is_valid():
            form.save(commit=True)
            return redirect('profile', user.username)
        else:
            print(form.errors)

    return render(request, 'rango/profile.html',
        {'userprofile': userprofile, 'selecteduser': user, 'form': form})
```

This view requires that a user be logged in - hence the use of the `@login_required` decorator. The view begins by selecting the selected `django.contrib.auth.User` from the database - if it exists. If it doesn't, we perform a simple redirect to Rango's homepage rather than greet the user with an error message. We can't display information for a non-existent user! If the user does exist, we can therefore select the user's `UserProfile` instance. If it doesn't exist, we can create a blank one. We then populate a `UserProfileForm` object with the selected user's details if we require it. This is determined by the template as it determines what content is presented to the user.

We then determine if the request is a HTTP `POST` - meaning that the user submitted a form to update their account information. We then extract information from the form into a `UserProfileForm` instance that is able to reference to the `UserProfile` model instance that it is saving to, rather than

creating a new `UserProfile` instance each time. Remember, we are *updating*, not creating *new*. A valid form is then saved. An invalid form or a HTTP `GET` request triggers the rendering of the `profile.html` template with the relevant variables that are passed through to the template via its context.

A Simple Exercise

How can we change the code above to prevent unauthorised users from changing the details of a user account that isn't theirs? What conditional statement do we need to add to enforce this additional check?

Mapping the View to a URL

We then need to map our new `profile()` view to a URL. As usual, this involves the addition of a single line of code to Rango's `urls.py` module. Add the following line to the bottom of the `urlpatterns` list.

```
url(r'^profile/(?P<username>[\w\-]+)/$', views.profile, name='profile'),
```

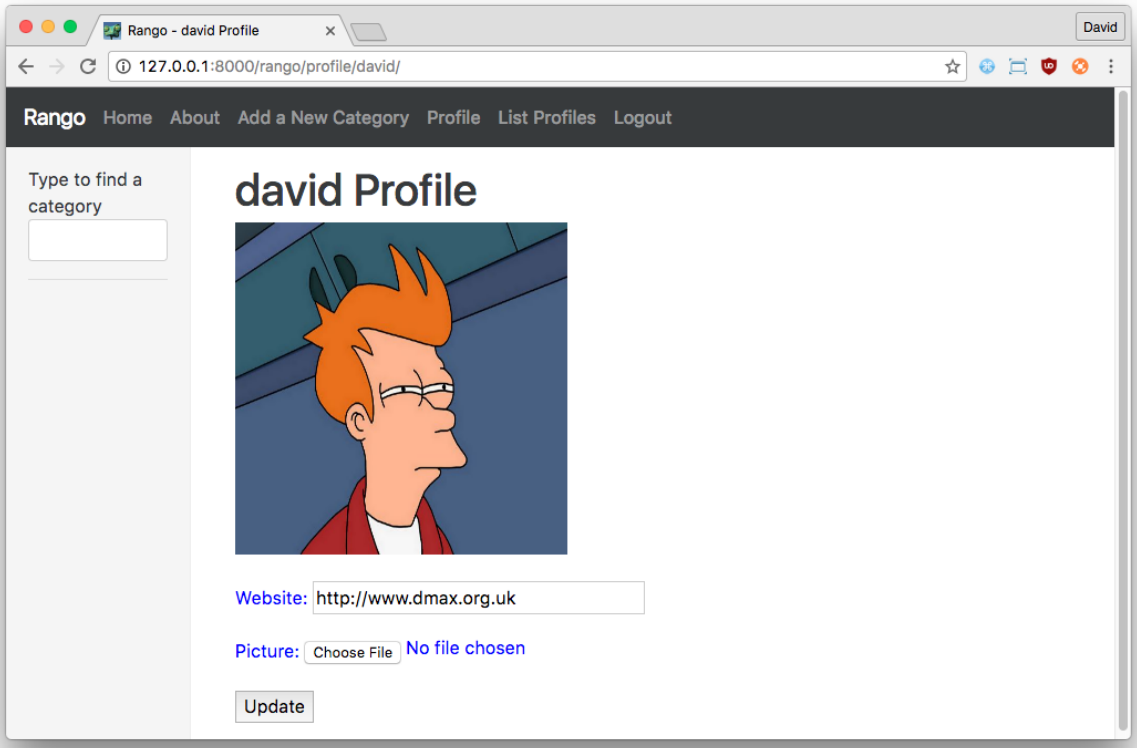
Note the inclusion of a `username` variable which is matched to anything after `/profile/` - meaning that the URL `/rango/profile/maxwelld90` would yield a `username` of `maxwelld90`, which is in turn passed to the `profile()` view as parameter `username`. This is how we are able to determine what user the current user has selected to view.

Tweaking the Base Template

Everything should now be working as expected - but it'd be nice to add a link in Rango's `base.html` template to link the currently logged in user to their profile, providing them with the ability to view or edit it. In Rango's `base.html` template, find the code that lists a series of links in the navigation bar of the page when the *user is logged in*. Add the following hyperlink to this collection.

```
<a href="{% url 'profile' user.username %}">Profile</a>
```

Note that you may want to add additional information to this link, such as adding a `class` attribute to the `<a>` tag to style it correctly. The link called the URL matched to name `profile` (see above), specifying the currently logged in username as the subsequent portion of the URL.



Rango's complete user profile page.

15.5 Listing all Users

Our final challenge is to create another page that allows one to view a list of all users on the Rango app. This one is relatively straightforward - we need to implement another template, view and URL mapping - but the view in this instance is very simplistic. We'll be creating a list of users registered to Rango - and providing a hyperlink to view their profile using the code we implemented in the previous section.

Creating a Template for User Profiles

In Rango's templates directory, create a template called `list_profiles.html`, within the file, add the following HTML markup and Django template code.

```
{% extends 'rango/base_bootstrap.html' %}

{% load staticfiles %}

{% block title %}User Profiles{% endblock %}

{% block body_block %}
<h1>User Profiles</h1>

<div class="panel">
    {% if userprofile_list %}
    <div class="panel-heading">
        <!-- Display search results in an ordered list -->
        <div class="panel-body">
            <div class="list-group">
                {% for listuser in userprofile_list %}
                <div class="list-group-item">
                    <h4 class="list-group-item-heading">
                        <a href="{% url 'profile' listuser.user.username %}">
                            {{ listuser.user.username }}</a>
                    </h4>
                </div>
                {% endfor %}
            </div>
        </div>
    </div>
    {% else %}
        <p>There are no users for the site.</p>
    {% endif %}
</div>
{% endblock %}
```

This template is relatively straightforward - we created a series of `<div>` tags using various Bootstrap classes to style the list. For each user, we display their username and provide a link to their profile page. Notice since we pass through a list of `UserProfile` objects, to access the username of the user, we need to go view the `user` property of the `UserProfile` object to get `username`.

Creating the View

With our template created, we can now create the corresponding view that selects all users from the `UserProfile` model. We also make the assumption that the current user must be logged in to view the other users of Rango. The following view `list_profiles()` can be added to Rango's `views.py` module to provide this functionality.

```
@login_required
def list_profiles(request):
    userprofile_list = UserProfile.objects.all()

    return render(request, 'rango/list_profiles.html',
        {'userprofile_list' : userprofile_list})
```

Mapping the View and Adding a Link

Our final step is to map a URL to the new `list_profiles()` view. Add the following to the `urlpatterns` list in Rango's `urls.py` module to do this.

```
url(r'^profiles/$', views.list_profiles, name='list_profiles'),
```

We could also add a new hyperlink to Rango's `base.html` template, allowing users *who are logged in* to view the new page. Like before, add the following markup to the base template which provides links only to logged in users.

```
<a href="{% url 'list_profiles' %}">List Profiles</a>
```

With this link added you should be able to now view the list of user profiles, and view specific profiles.

Profile Page Exercise

- Update the profile list to include a thumbnail of the user's profile picture.
- If a user does not have a profile picture, then insert a substitute picture by using the service provide by LoremPixel that lets you automatically generate images.

Hint: you can use `<img width="64" height="64" src="http://lorempixel.com/64/64/people/"/>` from LoremPixel to get a picture of `people` that is 64x64 in size. Note that it might take a few seconds for the picture to download.

16. JQuery and Django

JQuery rocks! JQuery is a library written in JavaScript that lets you access the power of JavaScript without the pain. This is because a few lines of JQuery often encapsulates hundreds of lines of JavaScript. Also, JQuery provides a suite of functionality that is mainly focused on manipulating HTML elements. In this chapter, we will describe:

- how to incorporate JQuery within your Django app;
- explain how to interpret JQuery code; and
- and provide a number of small examples.

16.1 Including JQuery in Your Django Project/App

In your *base* template include a reference to:

```
{% load staticfiles %}
<script src="https://ajax.googleapis.com/ajax/libs/jquery/3.0.0/jquery.min.js">
<script src="{% static "js/rango-jquery.js" %}"></script>
```

or if you have downloaded and saved a copy to your static directory, then you can reference it as follows:

```
{% load staticfiles %}
<script src="{% static "js/jquery.min.js" %}"></script>
<script src="{% static "js/rango-jquery.js" %}"></script>
```

Make sure you have your static files set up (see Chapter Templates and Static Media)

In the `static` directory, create a *js* directory and place the JQuery JavaScript file (`jquery.js`) here along with an file called `rango-jquery.js`. This script will house our JavaScript code. In `rango-jquery.js`, add the following JavaScript:

```
$(document).ready(function() {
    // JQuery code to be added in here.
});
```

This piece of JavaScript utilises JQuery. It first selects the document object (with `$(document)`), and then makes a call to `ready()`. Once the document is ready (i.e. the complete page is loaded), the anonymous function denoted by `function() { }` will be executed. It is pretty typical, if not standard, to wait until the document has been finished loading before running the JQuery functions. Otherwise, the code may begin executing before all the HTML elements have been downloaded. See the JQuery Documentation on Ready for more details.

Select and Act Pattern

JQuery requires you to think in a more *functional* programming style, as opposed to the typical JavaScript style which is often written in a more *procedural* programming style. For all the JQuery commands, they follow a similar pattern: **Select and Act**. Select an element, and then perform some action on/with the element.

Example Popup Box on Click

In this example, we want to show you the difference between doing the same functionality in standard JavaScript versus JQuery. In your `about.html` template, add the following piece of code:

```
<button  class="btn btn-primary"
    onClick="alert('You clicked the button using JavaScript.');">
    Click Me - I run JavaScript
</button>
```

As you can see, we are assigning the function `alert()` to the `onClick` handler of the button. Load up the `about` page, and try it out. Now lets do it using JQuery, by first adding another button:

```
<button  class="btn btn-primary" id="about-btn">
    Click Me - I'm JavaScript on Speed</button>
<p>This is a example</p>
<p>This is another example</p>
```

Notice that there is no JavaScript code associated with the button currently. We will be doing that with the following code added to `rango-jquery.js`:

```
$(document).ready( function() {
    $("#about-btn").click( function(event) {
        alert("You clicked the button using JQuery!");
    });
});
```

Reload the page, and try it out. Hopefully, you will see that both buttons pop up an alert.

The JQuery/JavaScript code here first selects the document object, and when it is ready, it executes the functions within its body, i.e. `$("#about-btn").click()`. This code selects the element in the page with an `id` equal to `about-btn`, and then programatically assigns to the `click` event the `alert()` function.

At first, you might think that JQuery is rather cumbersome, as it requires us to include a lot more code to do the same thing. This may be true for a simple function like `alert()`. For more complex functions, it is much cleaner as the JQuery/JavaScript code is maintained in a separate file. This is because we assign the event handler at runtime rather than statically within the code. We achieve separation of concerns between the JQuery/JavaScript code and the HTML markup.

Keep Them Separated

Separation of Concerns is a design principle that is good to keep in mind. In terms of web apps, the HTML is responsible for the page content; CSS is used to style the presentation of the content, while JavaScript is responsible for how the user can interact with the content, and manipulating the content and style.

By keeping them separated, you will have cleaner code and you will reduce maintenance woes in the future.

Put another way, *never mix, never worry!*

Selectors

There are different ways to select elements in JQuery. The above example shows how the `#` selector can be used to find elements with a particular `id` in your HTML document. To find classes, you can use the `.` selector, as shown in the example below.

```
$(".ouch").click( function(event) {
    alert("You clicked me! ouch!");
});
```

Then all elements in the document that have the `class="ouch"` would be selected, and assigned to its on click handler, the `alert()` function. Note that all the elements would be assigned the same function.

HTML tags can also be selected by referring to the tag in the selector:

```
$("p").hover( function() {
    $(this).css('color', 'red');
},
function() {
    $(this).css('color', 'blue');
});
```

Add this JavaScript to your `rango-jquery.js`, and then in the `about.html` template, add a paragraph, `<p>This text is for a JQuery Example</p>`. Try it out, go to the about page and hover over the text.

Here, we are selecting all the `p` HTML elements, and on hover we are associated two functions, one for on hover, and the other for hover off. You can see that we are using another selector called, `this`, which selects the element in question, and then sets its colour to red or blue respectively. Note that the JQuery `hover()` function takes two functions, and the JQuery `click()` function requires the event to be passed through.

Try adding the above code your `rango-jquery.js` file, making sure it is within the `$(document).ready()` function. What happens if you change the `$(this)` to `$(p)`?

Hovering is an example of a mouse move event. For descriptions on other such events, see the JQuery API documentation.

16.2 DOM Manipulation Example

In the above example, we used the `hover` function to assign an event handler to the on hover event, and then used the `css` function to change the colour of the element. The `css` function is one example of DOM manipulation, however, the standard JQuery library provides many other ways in which to manipulate the DOM. For example, we can add classes to elements, with the `addClass` function:

```
$("#about-btn").addClass('btn btn-primary')
```

This will select the element with `id` `#about-btn`, and assign the classes `btn` and `btn-primary` to it. By adding these Bootstrap classes, the button will now appear in the Bootstrap style (assuming you are using the Bootstrap toolkit).

It is also possible to access the inner HTML of a particular element. For example, lets put a `div` in the `about.html` template:

```
<div id="msg">Hello  - I'm here for a JQuery Example too</div>
```

Then add the following JQuery to `rango-jquery.js`:

```
$("#about-btn").click( function(event) {
    msgstr = $("#msg").html()
    msgstr = msgstr + "ooo"
    $("#msg").html(msgstr)
});
```

When the element with `id` `#about-btn` is clicked, we first get the HTML inside the element with `id` `msg` and append `"o"` to it. We then change the HTML inside the element by calling the `html()` function again, but this time passing through string `msgstr` to replace the HTML inside that element.

In this chapter, we have provided a very rudimentary guide to using JQuery and how you can incorporate it within your Django app. From here, you should be able to understand how JQuery operates and experiment with the different functions and libraries provided by JQuery and JQuery developers. In the next chapter, we will be using JQuery to help provide AJAX functionality within Rango.

17. AJAX in Django with JQuery

AJAX essentially is a combination of technologies that are integrated together to reduce the number of page loads. Instead of reloading the full page, only part of the page or the data in the page is reloaded. If you haven't used AJAX before or would like to know more about it before using it, check out the AJAX resources at the Mozilla website.

To simplify the AJAX requests, we will be using the JQuery library. Note that if you are using the Twitter CSS Bootstrap toolkit then JQuery will already be added in. We are using JQuery version 3. Otherwise, download the JQuery library and include it within your application, i.e. save it within your project into the `static/js/` directory.

17.1 AJAX based Functionality

To modernise the Rango application, let's add in a number of features that will use AJAX, such as:

- adding a "Like Button" to let registered users "like" a particular category;
- adding inline category suggestions - so that when a user types they can quickly find a category; and
- adding an "Add Button" to let registered users quickly and easily add a Page to the Category when they perform a search.

Create a new file, called `rango-ajax.js` and add it to your `static/js/` directory. Then in your *base* template include:

```
<script src="{% static "js/jquery.min.js" %}"></script>
<script src="{% static "js/rango-ajax.js" %}"></script>
```

Here we assume you have downloaded a version of the JQuery library, but you can also just directly refer to it:

```
<script
    src="https://ajax.googleapis.com/ajax/libs/jquery/3.0.0/jquery.min.js">
</script>
```

If you are using Bootstrap, then scroll to the bottom of the template code, You will see the JQuery library being imported at the end. You can then add a link to `rango-ajax.js` after the JQuery library import.

Now that we have setup JQuery and have a place to put our client side AJAX code, we can now modify the Rango app.

17.2 Add a Like Button

It would be nice to let users, who are registered, denote that they *"like"* a particular category. In the following workflow, we will let users "like" categories, but we will not be keeping track of what categories they have "liked". A registered user could click the like button multiple times if they refresh the page. If we wanted to keep track of their likes, we would have to add in an additional model, and other supporting infrastructure, but we'll leave that as an exercise for you to complete.

Workflow

To let users "like" certain categories, undertake the following workflow.

- In the `category.html` template:
  - Add in a "Like" button with `id="like"`.
  - Add in a template tag to display the number of likes: `{{% category.likes %}}`
  - Place this inside a div with `id="like_count"`, i.e. `<div id="like_count">{{ category.likes }} </div>`
  - This sets up the template to capture likes and to display likes for the category.
  - Note, since the `category()` view passes a reference to the category object, we can use that to access the number of likes, with `{{ category.likes }}` in the template
- Create a view called, `like_category` which will examine the request and pick out the `category_id` and then increment the number of likes for that category.
  - Don't forgot to add in the url mapping; i.e. map the `like_category` view to `rango/like_category/`. The GET request will then be `rango/like_category/?category_id=XXX`
  - Instead of returning a HTML page have this view will return the new total number of likes for that category.
- Now in `rango-ajax.js` add the JQuery code to perform the AJAX `GET` request.
  - If the request is successful, then update the `#like_count` element, and hide the like button.

Updating Category Template

To prepare the template, we will need to add in the "like" button with `id="like"` and create a `<div>` to display the number of likes `{{% category.likes %}}`. To do this, add the following `<div>` to the *category.html* template after the `<h1>{{ category.name }}</h1>` tag.

```
<div>
<strong id="like_count">{{ category.likes }}</strong> people like this category
{% if user.is_authenticated %}
    <button id="likes" data-catid="{{category.id}}"
        class="btn btn-primary  btn-sm" type="button">
        Like
    </button>
{% endif %}
</div>
```

Create a Like Category View

Create a new view called, `like_category` in `rango/views.py` which will examine the request and pick out the `category_id` and then increment the number of likes for that category.

```
from django.contrib.auth.decorators import login_required

@login_required
def like_category(request):
    cat_id = None
    if request.method == 'GET':
    cat_id = request.GET['category_id']
    likes = 0
    if cat_id:
        cat = Category.objects.get(id=int(cat_id))
        if cat:
            likes = cat.likes + 1
            cat.likes =  likes
            cat.save()
    return HttpResponse(likes)
```

On examining the code, you will see that we are only allowing authenticated users to even access this view because we have put a decorator `@login_required` before our view.

Note that the view assumes that a variable `category_id` has been passed to it via a `GET` request so that we can identify the category to update. In this view, we could also track and record that a particular user has "liked" this category if we wanted - but we are keeping it simple to focus on the AJAX mechanics.

Don't forget to add in the URL mapping, into `rango/urls.py`. Update the `urlpatterns` by adding in:

```
url(r'^like/$', views.like_category, name='like_category'),
```

Making the AJAX request

Now in "rango-ajax.js" you will need to add some JQuery code to perform an AJAX `GET` request. Add in the following code:

```
$('#likes').click(function(){
    var catid;
    catid = $(this).attr("data-catid");
    $.get('/rango/like/', {category_id: catid}, function(data){
        $('#like_count').html(data);
            $('#likes').hide();
    });
});
```

This piece of JQuery/JavaScript will add an event handler to the element with id `#likes`, i.e. the button. When clicked, it will extract the category ID from the button element, and then make an AJAX `GET` request which will make a call to `/rango/like/` encoding the `category_id` in the request. If the request is successful, then the HTML element with ID `like_count` (i.e. the `<strong>`) is updated with the data returned by the request, and the HTML element with ID `likes` (i.e. the `<button>`) is hidden.

There is a lot going on here, and getting the mechanics right when constructing pages with AJAX can be a bit tricky. Essentially, an AJAX request is made given our URL mapping when the button is clicked. This invokes the `like_category` view that updates the category and returns the new number of likes. When the AJAX request receives the response, it updates parts of the page, i.e. the text and the button. The `#likes` button is hidden.

17.3 Adding Inline Category Suggestions

It would be really neat if we could provide a fast way for users to find a category, rather than browsing through a long list. To do this we can create a suggestion component that lets users type in a letter or part of a word, and then the system responds by providing a list of suggested categories, that the user can then select from. As the user types a series of requests will be made to the server to fetch the suggested categories relevant to what the user has entered.

Workflow

To do this you will need to do the following.

- Create a parameterised function called `get_category_list(max_results=0, starts_with='')` that returns all the categories starting with `starts_with` if `max_results=0` otherwise it returns up to `max_results` categories.
  - The function returns a list of category objects annotated with the encoded category denoted by the attribute, `url`
- Create a view called *suggest_category* which will examine the request and pick out the category query string.
  - Assume that a GET request is made and attempt to get the *query* attribute.
  - If the query string is not empty, ask the Category model to get the top 8 categories that start with the query string.
  - The list of category objects will then be combined into a piece of HTML via template.
  - Instead of creating a template called `suggestions.html` re-use the `cats.html` as it will be displaying data of the same type (i.e. categories).
  - To let the client ask for this data, you will need to create a URL mapping; let's call it *suggest*.

With the URL mapping, view, and template in place, you will need to update the `base.html` template to provide a category search box, and then add in some JavaScript/JQuery code to link up everything so that when the user types the suggested categories are displayed.

In the `base.html` template modify the sidebar block so that a div with an id="cats" encapsulates the categories being presented. The JQuery/AJAX will update this element. Before this `<div>` add an input box for a user to enter the letters of a category, i.e.:

```
<input  class="input-medium search-query" type="text"
     name="suggestion" value="" id="suggestion" />
```

With these elements added into the templates, you can add in some JQuery to update the categories list as the user types.

- Associate an on keypress event handler to the *input* with `id="suggestion"`
- `$('#suggestion').keyup(function(){ ... })`
- On keyup, issue an ajax call to retrieve the updated categories list
- Then use the JQuery `.get()` function i.e. `$(this).get( ... )`
- If the call is successful, replace the content of the `<div>` with id="cats" with the data received.
- Here you can use the JQuery `.html()` function i.e. `$('#cats').html( data )`

Exercise

- Update the population script by adding in the following categories: `Pascal`, `Perl`, `PHP`, `Prolog`, `PostScript` and `Programming`. These additional categories will make the demo of the inline category suggestion functionality more impressive.

Parameterising `get_category_list()`

In this helper function, we use a filter to find all the categories that start with the string supplied. The filter we use will be `istartwith`, this will make sure that it doesn't matter whether we use uppercase or lowercase letters. If it on the other hand was important to take into account whether letters was uppercase or not you would use `startswith` instead.

```
def get_category_list(max_results=0, starts_with=''):
    cat_list = []
    if starts_with:
        cat_list = Category.objects.filter(name__istartswith=starts_with)

    if max_results > 0:
        if len(cat_list) > max_results:
            cat_list = cat_list[:max_results]
    return cat_list
```

Create a Suggest Category View

Using the `get_category_list()` function, we can now create a view that returns the top eight matching results as follows:

```
def suggest_category(request):
    cat_list = []
    starts_with = ''

    if request.method == 'GET':
        starts_with = request.GET['suggestion']
    cat_list = get_category_list(8, starts_with)

    return render(request, 'rango/cats.html', {'cats': cat_list })
```

Note here we are reusing the `rango/cats.html` template.

Mapping the View to URL

Add the following code to `urlpatterns` in `rango/urls.py`:

```
url(r'^suggest/$', views.suggest_category, name='suggest_category'),
```

Updating the Base Template

In the base template, in the sidebar `<div>`, add in the following HTML markup:

```
<ul class="nav nav-list flex-column">
    <li class="nav-item">Type to find a category</li>
    <form>
    <li class="nav-item"><input class="search-query form-control" type="text"
        name="suggestion" value="" id="suggestion" />
    </li>
    </form>
</ul>
<hr>
<div id="cats">
</div>
```

Here, we have added in an input box with `id="suggestion"` and div with `id="cats"` in which we will display the response. We don't need to add a button as we will be adding an event handler on keyup to the input box that will send the suggestion request.

Next remove the following lines from the template:

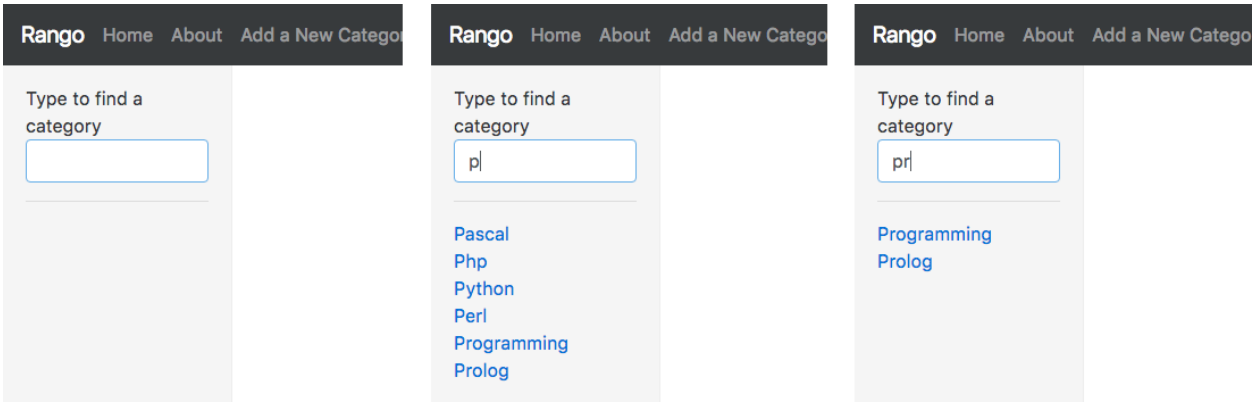
```
{% block sidebar_block %}
    {% get_category_list category %}
{% endblock %}
```

Add AJAX to Request Suggestions

Add the following JQuery code to the `js/rango-ajax.js`:

```
$('#suggestion').keyup(function(){
    var query;
    query = $(this).val();
    $.get('/rango/suggest/', {suggestion: query}, function(data){
        $('#cats').html(data);
    });
});
```

Here, we attached an event handler to the HTML input element with `id="suggestion"` to trigger when a keyup event occurs. When it does, the contents of the input box is obtained and placed into the `query` variable. Then a AJAX `GET` request is made calling `/rango/category_suggest/` with the `query` as the parameter. On success, the HTML element with `id="cats"` (i.e. the `<div>`) is updated with the category list HTML.



An example of the inline category suggestions. Notice how the suggestions populate and change as the user types each individual character.

Exercises

To let registered users quickly and easily add a Page to the Category put an "Add" button next to each search result. - Update the `category.html` template: - Add a small button next to each search result (if the user is authenticated), garnish the button with the title and URL data, so that the JQuery can pick it out. - Put a `<div>` with `id="page"` around the pages in the category so that it can be updated when pages are added. - Remove that link to `add` button, if you like. - Create a view `auto_add_page` that accepts a parameterised `GET` request `(title, url, catid)` and adds it to the category. - Map an URL to the view `url(r'^add/$', views.auto_add_page, name='auto_add_page'),` - Add an event handler to the add buttons using JQuery - when added hide the button. The response could also update the pages listed on the category page, too.

We have included the following code fragments to help you complete the exercises above. The HTML template code for `category.html` that inserts a button, and crucially keeps a record of the category that the button is associated with.

```
{% if user.is_authenticated %}
    <button data-catid="{{category.id}}" data-title="{{ result.title }}"
        data-url="{{ result.link }}"
            class="rango-add btn btn-info btn-sm" type="button">Add</button>
{% endif %}
```

The JQuery code that adds the `click` event handler to every button with the class `rango-add`:

```
$('.rango-add').click(function(){
    var catid = $(this).attr("data-catid");
    var url = $(this).attr("data-url");
    var title = $(this).attr("data-title");
    var me = $(this)
    $.get('/rango/add/',
        {category_id: catid, url: url, title: title}, function(data){
            $('#pages').html(data);
            me.hide();
        });
    });
```

The view code that handles the adding of a link to a category:

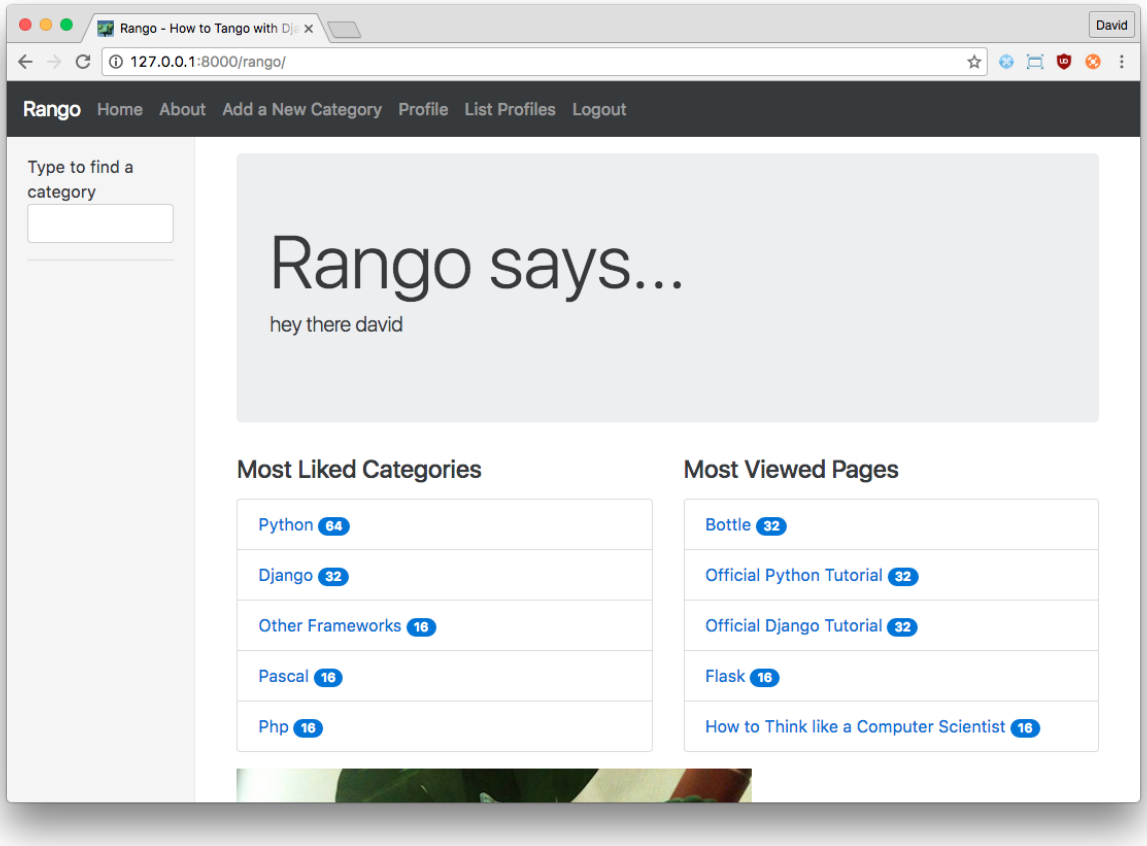
```
@login_required
def auto_add_page(request):
    cat_id = None
    url = None
    title = None
    context_dict = {}
    if request.method == 'GET':
        cat_id = request.GET['category_id']
        url = request.GET['url']
        title = request.GET['title']
        if cat_id:
            category = Category.objects.get(id=int(cat_id))
            p = Page.objects.get_or_create(category=category,
                title=title, url=url)
            pages = Page.objects.filter(category=category).order_by('-views')
            # Adds our results list to the template context under name pages.
            context_dict['pages'] = pages
    return render(request, 'rango/page_list.html', context_dict)
```

The HTML template markup for the new template `page_list.html`:

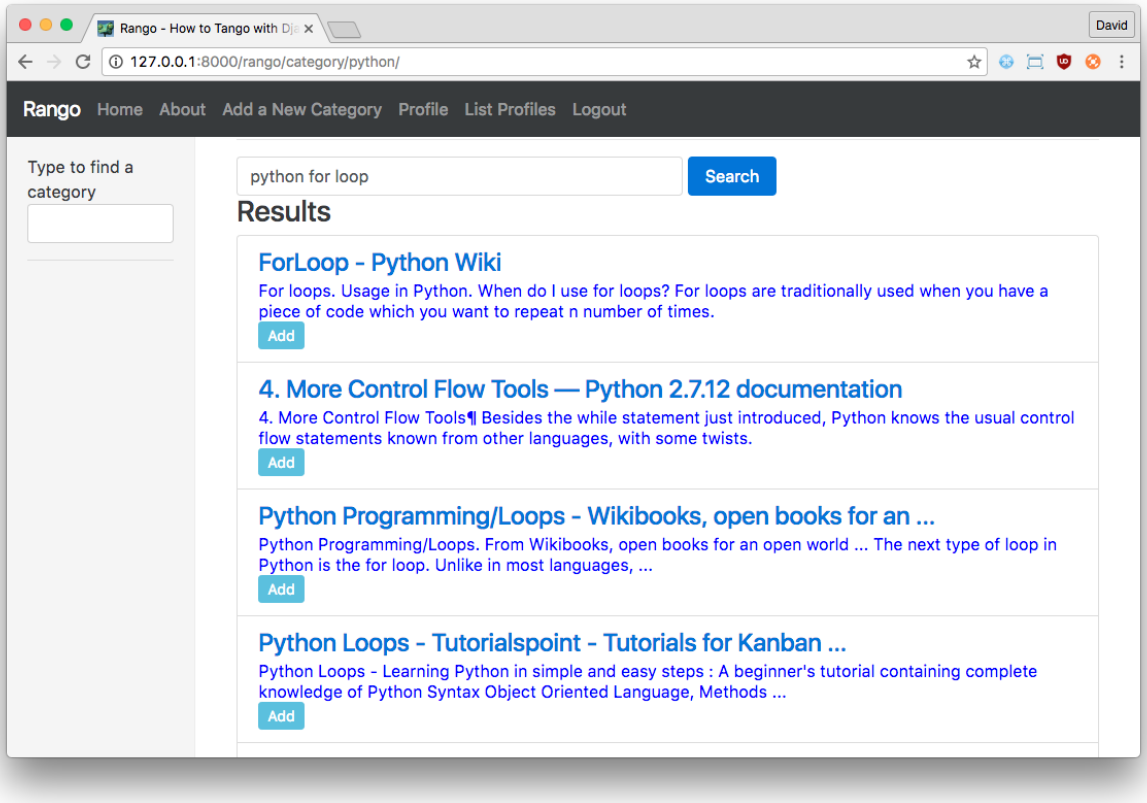
```
{% if pages %}
<ul>
    {% for page in pages %}
    <li><a href="{% url 'goto' %}?page_id={{page.id}}"\>{{ page.title }}</a></li>
    {% endfor %}
</ul>
{% else %}
    <strong>No pages currently in category.</strong>
{% endif %}
```

Finally, don't forget to add in the URL mapping: `url(r'^add/$', views.auto_add_page, name='auto_add_page'),`.

If all has gone well, hopefully, your Rango application will be looking something like screenshots below. But don't stop now, get on with the next chapters and deploy your project!



The main index page of the Rango application.



The category page with the Add Button feature.

18. Automated Testing

It is good practice to get into the habit of writing and developing tests. A lot of software engineering is about writing and developing tests and test suites in order to ensure the software is robust. Of course, most of the time, we are too busy trying to build things to bother about making sure that they work. Or too arrogant to believe it would fail.

According to the Django Tutorial, there are numerous reasons why you should include tests.

- Test will save you time: a change in a complex system can cause failures in unpredictable places.
- Tests don't just identify problems, they prevent them: tests show where the code is not meeting expectations.
- Test make your code more attractive: *"Code without tests is broken by design"* - Jacob Kaplan-Moss, one of Django's original developers.
- Tests help teams work together: they make sure your team doesn't inadvertently break your code.

According to the Python Guide, there are a number of general rules you should try to follow when writing tests. Below are some main rules.

- Tests should focus on one small bit of functionality
- Tests should have a clear purpose
- Tests should be independent.
- Run your tests, before you code, and before your commit and push your code.
- Even better create a hook that tests code on push.
- Use long and descriptive names for tests.

Testing in Django

Currently this chapter provides the very basics of testing and follows a similar format to the Django Tutorial, with some additional notes. We hope to expand this further in the future.

18.1 Running Tests

With Django is a suite of functionality to test apps built. You can test your Rango app by issuing the following command:

```
$ python manage.py test rango

Creating test database for alias 'default'...

----------------------------------------------------------------------
Ran 0 tests in 0.000s

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

This will run through the tests associated with the Rango app. At the moment, nothing much happens. That is because you may have noticed the file `rango/tests.py` only contains an import statement. Every time you create an application, Django automatically creates such a file to encourage you to write tests.

From this output, you might also notice that a database called `default` is referred to. When you run tests, a temporary database is constructed, which your tests can populate, and perform operations on. This way your testing is performed independently of your live database.

Testing the models in Rango

Let's create a test. In the `Category` model, we want to ensure that views are either zero or positive, because the number of views, let's say, can never be less than zero. To create a test for this we can put the following code into `rango/tests.py`:

```
from django.test import TestCase
from rango.models import Category

class CategoryMethodTests(TestCase):
    def test_ensure_views_are_positive(self):
        """
        ensure_views_are_positive should results True for categories
        where views are zero or positive
        """
        cat = Category(name='test',views=-1, likes=0)
        cat.save()
        self.assertEqual((cat.views >= 0), True)
```

The first thing you should notice, if you have not written tests before, is that we have to inherit from `TestCase`. The naming over the method in the class also follows a convention, all tests start with `test_` and they also contain some type of assertion, which is the test. Here we are checking if the values are equal, with the `assertEqual` method, but other types of assertions are also possible.

See the Python 2 Documentation on unit tests or the Python 3 Documentation on unit tests for other commands (i.e. `assertItemsEqual`, `assertListEqual`, `assertDictEqual`, etc). Django's testing machinery is derived from Python's but also provides a number of other asserts and specific test cases.

Now let's run the test:

```
$ python manage.py test rango

Creating test database for alias 'default'...
F
======================================================================
FAIL: test_ensure_views_are_positive (rango.tests.CategoryMethodTests)
----------------------------------------------------------------------
Traceback (most recent call last):
    File "/Users/leif/Code/tango_with_django_project_19/rango/tests.py",
    line 12, in test_ensure_views_are_positive
    self.assertEqual((cat.views>=0), True)
    AssertionError: False != True

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

FAILED (failures=1)
```

As we can see this test fails. This is because the model does not check whether the value is less than zero or not. Since we really want to ensure that the values are non-zero, we will need to update the model to ensure that this requirement is fulfilled. Do this now by adding some code to the `save()` method of the `Category` model. The code should check the value of `views` attribute, and update it accordingly. A simple conditional check of `self.views` where it is less than zero should suffice here.

Once you have updated your model, you can now re-run the test, and see if your code now passes it. If not, try again.

Let's try adding another test that ensures an appropriate slug line is created, i.e. one with dashes, and in lowercase. Add the following code to `rango/tests.py`:

```
def test_slug_line_creation(self):
    """
    slug_line_creation checks to make sure that when we add
    a category an appropriate slug line is created
    i.e. "Random Category String" -> "random-category-string"
    """
    cat = cat('Random Category String')
    cat.save()
    self.assertEqual(cat.slug, 'random-category-string')
```

Does your code still work?

Testing Views

So far we have written tests that focus on ensuring the integrity of the data housed in the models. Django also provides testing mechanisms to test views. It does this with a mock client, that internally makes a calls a Django view via the URL. In the test you have access to the response (including the HTML) and the context dictionary.

Let's create a test that checks that when the index page loads, it displays the message that `There are no categories present`, when the `Category` model is empty.

```
from django.core.urlresolvers import reverse

class IndexViewTests(TestCase):

    def test_index_view_with_no_categories(self):
        """
        If no questions exist, an appropriate message should be displayed.
        """
        response = self.client.get(reverse('index'))
        self.assertEqual(response.status_code, 200)
        self.assertContains(response, "There are no categories present.")
        self.assertQuerysetEqual(response.context['categories'], [])
```

First of all, the Django `TestCase` has access to a `client` object, which can make requests. Here, it uses the helper function `reverse` to look up the URL of the `index` page. Then it tries to get that page, where the `response` is stored. The test then checks a number of things: whether the page loaded OK, whether the response HTML contains the phrase `"There are no categories present."`, and whether the context dictionary contains an empty categories list. Recall that when you run tests, a new database is created, which by default is not populated.

Let's now check the resulting view when categories are present. First add a helper method.

```
from rango.models import Category

def add_cat(name, views, likes):
    c = Category.objects.get_or_create(name=name)[0]
    c.views = views
    c.likes = likes
    c.save()
    return c
```

Then add another method to the `class IndexViewTests(TestCase):`

```
def test_index_view_with_categories(self):
    """
    Check to make sure that the index has categories displayed
    """

    add_cat('test',1,1)
    add_cat('temp',1,1)
    add_cat('tmp',1,1)
    add_cat('tmp test temp',1,1)

    response = self.client.get(reverse('index'))
    self.assertEqual(response.status_code, 200)
    self.assertContains(response, "tmp test temp")

    num_cats =len(response.context['categories'])
    self.assertEqual(num_cats , 4)
```

In this test, we populate the database with four categories, and then check that the loaded page contains the text `tmp test temp` and if the number of categories is equal to `4`. Note that this makes three checks, but is only considered to be one test.

Testing the Rendered Page

It is also possible to perform tests that load up the application and programmatically interact with the DOM elements on the HTML pages by using either Django's test client and/or Selenium, which is are "in-browser" frameworks to test the way the HTML is rendered in a browser.

18.2 Coverage Testing

Code coverage measures how much of your code base has been tested, and how much of your code has been put through its paces via tests. You can install a package called `coverage` via with `pip`

`install coverage` that automatically analyses how much code coverage you have. Once you have `coverage` installed, run the following command:

```
$ coverage run --source='.' manage.py test rango
```

This will run through all the tests and collect the coverage data for the Rango application. To see the coverage report you need to then type:

```
$ coverage report
```

```
Name                                             Stmts   Miss  Cover
------------------------------------------------------------------
manage                                               6      0   100%
populate                                            33     33     0%
rango/__init__                                       0      0   100%
rango/admin                                          7      0   100%
rango/forms                                         35     35     0%
rango/migrations/0001_initial                        5      0   100%
rango/migrations/0002_auto_20141015_1024             5      0   100%
rango/migrations/0003_category_slug                  5      0   100%
rango/migrations/0004_auto_20141015_1046             5      0   100%
rango/migrations/0005_userprofile                    6      0   100%
rango/migrations/__init__                            0      0   100%
rango/models                                        28      3    89%
rango/tests                                         12      0   100%
rango/urls                                          12     12     0%
rango/views                                        110    110     0%
tango_with_django_project/__init__                   0      0   100%
tango_with_django_project/settings                  28      0   100%
tango_with_django_project/urls                       9      9     0%
tango_with_django_project/wsgi                       4      4     0%
------------------------------------------------------------------
TOTAL                                              310    206    34%
```

We can see from the above report that critical parts of the code have not been tested, i.e. `rango/views`. The `coverage` package has many more features that you can explore to make your tests even more comprehensive!

Exercises

Lets say that we want to extend the `Page` to include two additional fields, `last_visit` and `first_visit` that will be of type `timedate`.

- Update the model to include these two fields.
- Update the add page functionality, and the goto functionality.
- Add in a test to ensure the last visit or first visit is not in the future.
- Add in a test to ensure that the last visit equal to or after the first visit.
- Run through Part Five of the official Django Tutorial to learn more about testing.
- Check out the tutorial on test driven development by Harry Percival.

19. Deploying Your Project

This chapter provides a step-by-step guide on how to deploy your Django applications. We'll be looking at deploying applications on PythonAnywhere, an online IDE and web hosting service. The service provides in-browser access to the server-based Python and Bash command line interfaces, meaning you can interact with PythonAnywhere's servers just like you would with a regular terminal instance on your own computer. Currently, PythonAnywhere are offering a free account that sets you up with an adequate amount of storage space and CPU time to get a Django application up and running.

Go Git It!

You can do this chapter independently as we have already implemented Rango and it is available from GitHub. If you haven't used Git/GitHub before, you can check out our chapter on using Git).

19.1 Creating a PythonAnywhere Account

First, sign up for a Beginner PythonAnywhere account. If your application takes off and becomes popular, you can always upgrade your account at a later stage to gain more storage space and CPU time along with a number of other benefits - such as hosting specific domains and SSH abilities, for example.

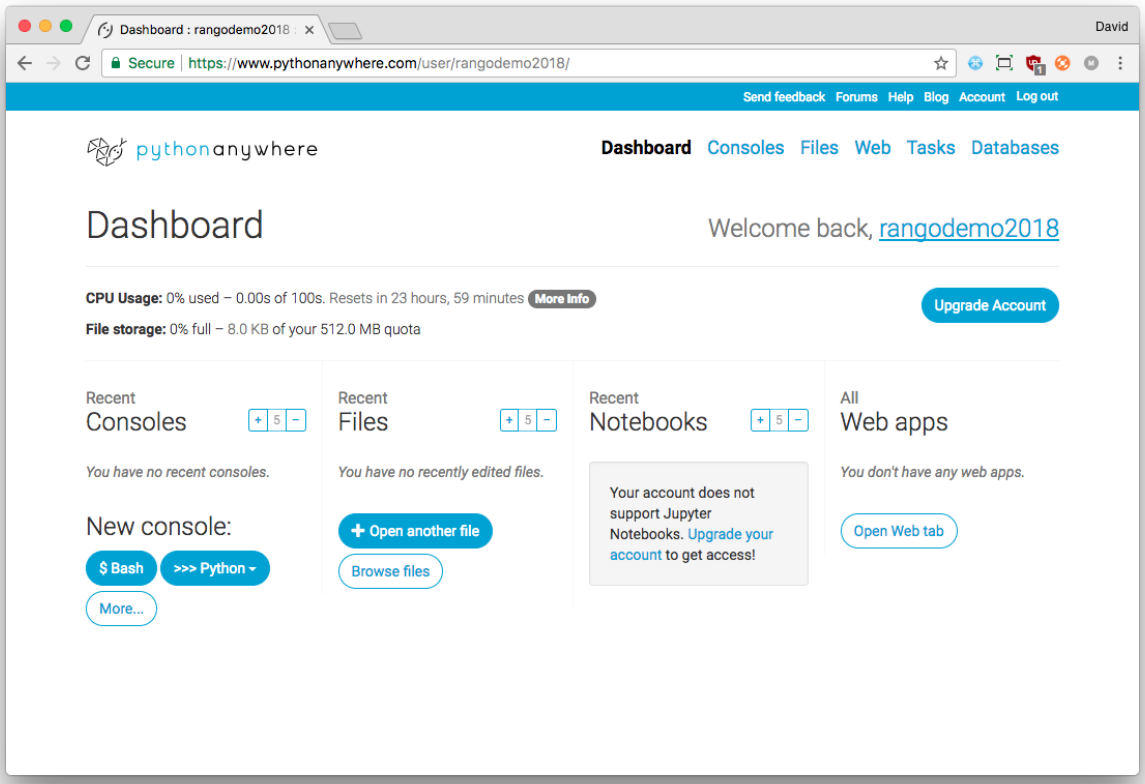
Once your account has been created, you will have your own little slice of the World Wide Web at `http://<username>.pythonanywhere.com`, where `<username>` is your PythonAnywhere username. It is from this URL that your hosted application will be available.

19.2 The PythonAnywhere Web Interface

The PythonAnywhere web interface contains a *dashboard* which in turn provides a series of different components allowing you to manage your application. The components as illustrated in the figure below include:

- *Consoles*, allowing you to create and interact with Python and Bash console instances;
- *Files*, which allows you to upload to and organise files within your disk quota; and
- *Web Apps*, allowing you to configure settings for your hosted web application;

Other components exist, such as *Notebooks*, but we won't be using them here – we'll be working primarily with the *consoles* and *web* components. The PythonAnywhere Wiki provides a series of detailed explanations on how to use the other components if you are interested in finding out more.



The PythonAnywhere dashboard, showing the main components you can use.

19.3 Creating a Virtual Environment

As part of its standard default Bash environment, PythonAnywhere comes with Python 2.7.6 and a number of pre-installed Python Packages (including *Django 1.3.7* and *Django-Registration 0.8*). Since we are using a different setup, we need to select a particular Python version and setup a virtual environment for our application.

First, open a Bash console. From the PythonAnywhere dashboard, click the *$Bash$* button under the *Consoles* component. You will then be taken to a black screen where a terminal will be initialised. When the terminal is ready for you to use (i.e. you see the time presented to you), enter the following commands.

```
$ mkvirtualenv --python=<python-version> rango
```

If you've coded up the tutorial using Python 3.x, then change `<python-version>` to either `python3.4`, `python3.5` or `python3.6` (double check what version you used by running `python --version` on your own computer!). If you are using Python 2.7.x, then change `<python-version>` to `python2.7`. The command you enter creates a new virtual environment called `rango` using the version of Python that you specified. For example, below is the output for when we created a Python 2.7 virtual environment.

```
11:45 ~ $ mkvirtualenv --python=python3.6 rango
Running virtualenv with interpreter /usr/bin/python3.6
New python executable in /home/rangodemo2018/.virtualenvs/rango/bin/python3.6
Also creating executable in /home/rangodemo2018/.virtualenvs/rango/bin/python
Installing setuptools, pip, wheel...done.
virtualenvwrapper creating /home/rangodemo2018/.virtualenvs/.../predeactivate
virtualenvwrapper creating /home/rangodemo2018/.virtualenvs/.../postdeactivate
virtualenvwrapper creating /home/rangodemo2018/.virtualenvs/.../preactivate
virtualenvwrapper creating /home/rangodemo2018/.virtualenvs/.../postactivate
virtualenvwrapper creating /home/rangodemo2018/.virtualenvs/.../get_env_details
```

Note in the example above, the PythonAnywhere username used is `rangodemo2018` - this will be replaced with your own username. The process of creating the virtual environment will take a little while to complete, after which you will be presented with a slightly different prompt.

```
(rango) 11:45 ~ $
```

Note the inclusion of `(rango)` compared to the previous command prompt. This signifies that the `rango` virtual environment has been activated, so any package installations will be done within that virtual environment, leaving the wider system setup alone. If you issue the command `ls -la`, you will see that a directory called `.virtualenvs` has been created. This is the directory in which all of your virtual environments and associated packages will be stored. To confirm the setup, issue the command `which pip`. This will print the location in which the active `pip` binary is located - hopefully within `.virtualenvs` and `rango`, as shown in the example below.

```
/home/<username>/.virtualenvs/test/bin/pip
```

To see what packages are already installed, enter `pip list`. Now we can customise the virtual environment by installing the required packages for our Rango application. Install all the required packages, by issuing the following commands.

```
$ pip install -U django==1.9.10
$ pip install pillow
$ pip install django-registration-redux
$ pip install django-bootstrap-toolkit
```

Ensure that you replace the version of Django specified above with the one you used during development. Not doing so will in all probability cause you to get weird, weird exceptions and will result in you pulling out your hair in frustration. You could alternatively use `pip freeze > requirements.txt` to save your current development environment, and then on PythonAnywhere, run `pip install -r requirements.txt` to install all the packages in one go.

Waiting to Download...

Installing all theses packages may take some time, so you can relax, call a friend, or tweet about our tutorial `@tangowithdjango`!

Once installed, check if Django has been installed with the command `which django-admin.py`. You should receive output similar to the following example.

```
/home/<username>/.virtualenvs/rango/bin/django-admin.py
```

Virtual Environments on PythonAnywhere

PythonAnywhere also provides instructions on how to setup virtual environments. Check out their Wiki documentation for more information.

Virtual Environment Switching

Moving between virtual environments can be done pretty easily. PythonAnywhere should have this covered for you. Below, we provide you with a quick tutorial on how to switch between virtual environments.

At your terminal, you can launch into a pre-existing virtual environment with the `workon` command. For example, to load up the `rango` environment, enter:

```
16:48 ~ $ workon rango
```

where `rango` can be replaced with the name of the virtual environment you wish to use. Your prompt should then change to indicate you are working within a virtual environment. This is shown by the addition of `(rango)` to your prompt.

```
(rango) 16:49 ~ $
```

You can then leave the virtual environment using the `deactivate` command. Your prompt should then be missing the `(rango)` prefix, with an example shown below.

```
(rango) 16:49 ~ $ deactivate
16:51 ~ $
```

Cloning your Git Repository

Now that your virtual environment for Rango is all setup, you can now clone your Git repository to obtain a copy of your project's files. Clone your repository by issuing the following command from your home directory:

```
$ git clone https://<USERNAME>:<PASSWORD>@github.com/<OWNER>/<REPO_NAME>.git
```

where you replace - `<USERNAME>` with your GitHub username; - `<PASSWORD>` with your GitHub password; - `<OWNER>` with the username of the person who owns the repository; and - `<REPO_NAME>` with the name of your project's repository.

Setting Up the Database

With your files cloned, you must then prepare your database. We'll be using the `populate_rango.py` module that we created earlier in the book. As we'll be running the module, you must ensure that you are using the `rango` virtual environment (i.e. you see `(rango)` as part of your prompt - if not, invoke `workon rango`). From your home directory, move into the `tango_with_django_19` directory, then to the `code` directory. This directory will be named after your git repository, so the name may differ. For example, if you called your repository `twd`, the directory will also be called `twd`. Depending upon how you configured your repository, you should also `cd` into the directory with `manage.py` in it - `tango_with_django_project`. Now issue the following commands.

```
(rango) 16:55 ~/tango_with_django $ python manage.py makemigrations rango
(rango) 16:55 ~/tango_with_django $ python manage.py migrate
(rango) 16:56 ~/tango_with_django $ python populate_rango.py
(rango) 16:57 ~/tango_with_django $ python manage.py createsuperuser
```

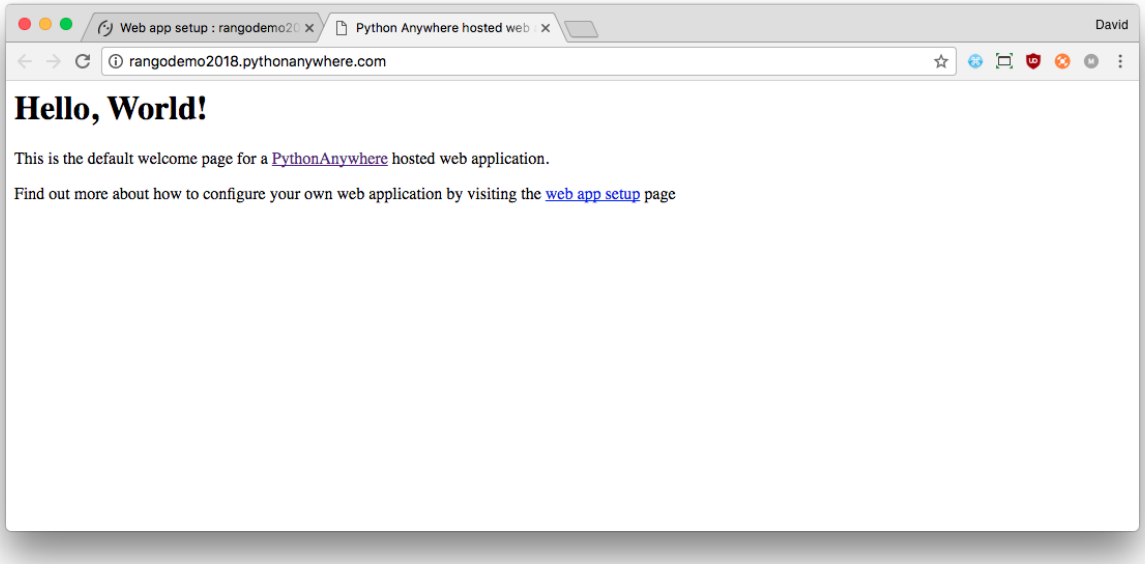
As discussed earlier in the book, the first command creates the migrations for the `rango` app, then the `migrate` command creates the *SQLlite3* database. Once the database is created, the database can be populated and a superuser created.

19.4 Setting up Your Web Application

Now that the database is setup, we need to configure the PythonAnywhere *NGINX* Web server to serve up your application. From PythonAnywhere's dashboard, open up the *Web* tab. On the left of the page that appears, click *Add a new web app.*

A popup box will then appear. Follow the instructions on-screen, and when the time comes, select the *manual configuration* option and complete the wizard. Make sure you select the same Python version as the one you selected earlier.

In a new tab or window in your Web browser, go visit your PythonAnywhere subdomain at the address `http://<username>.pythonanywhere.com`. You should be presented with the default `Hello, World!` webpage, as shown below. This is because the WSGI script is currently serving up this page, and not your Django application. This is what we need to change next.



The default PythonAnywhere *hello world* webpage.

Configure the Virtual Environment

To set the virtual environment for your app, navigate again to the *Web* tab in PythonAnywhere's interface. From there, scroll all the way down under you see the heading *Virtualenv.*

Enter in the path to your virtual environment. Assuming you created a virtual environment called `rango` the path would be:

```
/home/<username>/.virtualenvs/rango
```

You can start a console to check if it is successful.

Now in the *Code* section, you can set the path to your web applications source code.

```
/home/<username>/<path-to>/tango_with_django_project/
```

Note that this path should be pointing to the directory with your project's `manage.py` file within it. If for example you cloned a repository called `tango_with_django19` into your account's home directory, the path will be something like:

```
/home/<username>/tango_with_django_19/code/tango_with_django_project/
```

Configuring the WSGI Script

The Web Server Gateway Interface, a.k.a. *WSGI* provides a simple and universal interface between Web servers and Web applications. PythonAnywhere uses WSGI to bridge the server-application link and map incoming requests to your subdomain to your web application.

To configure the WSGI script, navigate to the *Web* tab in PythonAnywhere's interface. Under the Code heading you can see a link to the WSGI configuration file in the Code section: e.g. `/var/www/<username>_pythonanywhere_com_wsgi.py`.

The people at PythonAnywhere have set up a sample WSGI file for us with several possible configurations. For your Web application, you'll need to configure the Django section of the file by clicking on the link to open a simple editor. The example below demonstrates a possible configuration for your application.

```
import os
import sys

# Add your project's directory the PYTHONPATH
path = '/home/<username>/<path-to>/tango_with_django_project/'
if path not in sys.path:
    sys.path.append(path)

# Move to the project directory
os.chdir(path)

# Tell Django where the settings.py module is located
os.environ.setdefault('DJANGO_SETTINGS_MODULE',
                      'tango_with_django_project.settings')
```

```
# Import your Django project's configuration
import django
django.setup()

# Import the Django WSGI to handle any requests
import django.core.handlers.wsgi
application = django.core.handlers.wsgi.WSGIHandler()
```

Ensure that you replace `<username>` with your PythonAnywhere username, and update any other path settings to suit your application. You should also remove all other code from the WSGI configuration script to ensure no conflicts take place.

The script adds your project's directory to the `PYTHONPATH` for the Python instance that runs your web application. This allows Python to access your project's modules. If you have additional paths to add, you can easily insert them here. You can then specify the location of your project's `settings.py` module. The final step is to include the Django WSGI handler and invoke it for your application.

When you have completed the WSGI configuration, click the *Save* button at the top-right of the webpage. Navigate back to the *Web* tab within the PythonAnywhere interface, and click the *Reload* button at the top of the page (the big green one). When the application is reloaded, you can then revisit your PythonAnywhere subdomain at `http://<username>.pythonanywhere.com`. Hopefully, if all went well, you should see your application up and running. If not, check through your scripts and paths carefully. Double check your paths by actually visiting the directories, and use `pwd` to confirm the path. If you see a `DisallowedHost` exception, you need to follow the steps below.

Bad Gateway Errors

During testing, we noted that you can sometimes receive `HTTP 502 - Bad Gateway` errors instead of your application. Try reloading your application again, and then waiting a longer. If the problem persists, try reloading again. If the problem still persists, check out your log files to see if any accesses/errors are occurring, before contacting the PythonAnywhere support.

Allowing your Hostname

A security feature in more recent versions of Django is that of allowed hosts. By only allowing a particular set of domains to be served by your web server, this reduces the chance that your app could be part of a so-called HTTP Host Header attack. You may find that when you run your app for the first time, you see a `DisallowedHost` exception stopping your app from loading.

This is a simple problem to fix, and involves a change in your project's `settings.py` module. First, work out your app's URL on PythonAnywhere. For a basic account, this will be

`http://<username>.pythonanywhere.com`, where `<username>` is replaced with your PythonAnywhere username. It's a good idea to edit this file locally (on your own computer), then `git add`, `git commit` and `git push` your changes to your Git repository, before downloading the changes to your PythonAnywhere account. Alternatively, you can edit the file directly on PythonAnywhere by editing the file in the Web interface's files component – or using a text editor in the terminal, like `nano` or `vi`.

With this information, open your project's `settings.py` module and locate the `ALLOWED_HOSTS` list, which by default will be empty (and found near the top of the file). Add a string with your PythonAnywhere URL into that list – such that it now looks like the following example.

```
ALLOWED_HOSTS = ['http://<username>.pythonanywhere.com']
```

If you have edited the file on your own computer, you can now go through the (hopefully) now familiar process of running the `git add settings.py`, `git commit` and `git push` commands to make changes to your Git repository. Once done, you should then run the `git pull` command to retrieve the changes on your PythonAnywhere account. If you have edited the file directly on PythonAnywhere, simply save the file.

All that then remains is for you to reload your PythonAnywhere app. This can be done by clicking the *Reload* button in the PythonAnywhere web components page. Once you've done this, access your app's URL, and you should see your app working, but without static media!

Assigning Static Paths

We're almost there. One issue that we still have to address is to sort out paths for our application. Doing so will allow PythonAnywhere's servers to serve your static content, for example From the PythonAnywhere dashboard, click the URL of your app and wait for the page to load.

Once loaded, perform the following under the *Static files* header. We essentially are adding in the correct URLs and filesystem paths to allow PythonAnywhere's web server to find and serve your static media files.

First, we should set the location of the Django admin interface's static media files. Click the *Enter URL* link, and type in `/static/admin/`. Click the tick to confirm your input, and then click the *Enter path* link, entering the following long-winded filesystem path (all on a single line).

```
/home/<username>/.virtualenvs/rango/lib/<python-version>/site-packages/django/
  contrib/admin/static/admin
```

As usual, replace `<username>` with your PythonAnywhere username. `<python-version>` should also be replaced with `python2.7`, `python3.6`, etc., depending on which Python version you selected. You may also need to change `rango` if this is not the name of your application's virtual environment. Remember to hit return to confirm the path, or click the tick.

Repeat the two steps above for the URL `/static/` and filesystem path `/home/<username>/<path-to>/tango_with_django_project/static`, with the path setting pointing to the `static` directory of your Web application.

With these changes saved, reload your web application by clicking the *Reload* button at the top of the page. Don't forget the about potential for `HTTP 502 - Bad Gateway` errors. Setting the static directories means that when you visit the `admin` interface, it has the predefined Django stylesheets, and that you can access images and scripts. Reload your Web application, and you should now notice that your images are present.

Search API Key

Add your search API key to `search.key` to enable the search functionality in Rango.

Turning off `DEBUG` Mode

When you application is ready to go, it's a good idea to instruct Django that your application is now hosted on a production server. To do this, open your project's `settings.py` file and change `DEBUG = True` to `DEBUG = False`. This disables Django's debug mode, and removes explicit error messages.

Why do this? Disabling explicit error messages stops people from viewing them. Individuals will find this to be unprofessional if they see detailed error logs being displayed on a production website. Furthermore, malicious individuals could use the information the error message displays to potentially exploit weaknesses in your app. It therefore makes sense to simple turn off this functionality.

Error messages are however still logged. If something goes wrong, you can see the Python stack track to work out how to fix it. You can view your log files by opening the PythonAnywhere Web component, displaying the page for your app. Look for the *Log files* header – and there, you'll see an *Error log* link. If you click this, you'll see your app's stack traces that are generated when an exception is raised and unhandled.

19.5 Log Files

Deploying your Web application to an online environment introduces another layer of complexity. It is likely that you will encounter new and bizarre errors due to unsuspecting problems. When facing such errors, vital clues may be found in one of the three log files that the Web server on PythonAnywhere creates.

Log files can be viewed via the PythonAnywhere web interface by clicking on the *Web* tab, or by viewing the files in `/var/log/` within a Bash console instance. The files provided are:

- `access.log`, which provides a log of requests made to your subdomain;

- `error.log`, which logs any error messages produced by your web application; and
- `server.log`, providing log details for the UNIX processes running your application.

Note that the names for each log file are prepended with your subdomain. For example, `access.log` will have the name `<username>.pythonanywhere.com.access.log`.

When debugging, you may find it useful to delete or move the log files so that you don't have to scroll through a huge list of previous attempts. If the files are moved or deleted, they will be recreated automatically when a new request or error arises.

Exercises

Congratulations, you've successfully deployed Rango!

- Tweet a link of your application to @tangowithdjango.
- Tweet or e-mail us to let us know your thoughts on the tutorial!

20. Final Thoughts

In this book, we have gone through the process of web development from specification to deployment. Along the way we have shown how to use the Django framework to construct the models, views and templates associated with a Web application. We have also demonstrated how toolkits and services like Bootstrap, JQuery and PythonAnywhere. can be integrated within an application. However, the road doesn't stop here. While, as we have only painted the broad brush strokes of a web application - as you have probably noticed there are lots of improvements that could be made to Rango - and these finer details often take a lot more time to complete as you polish the application. By developing your application on a firm base and good setup you will be able to construct up to 80% of your site very rapidly and get a working demo online.

In future versions of this book we intend to provide some more details on various aspects of the framework, along with covering the basics of some of the other fundamental technologies associated with web development. If you have any suggestions or comments about how to improve the book please get in touch.

Please report any bugs, problems, etc., or submit change requests via GitHub. Thank you!

20.1 Acknowledgements

This book was written to help teach web application development to computing science students. In writing the book and the tutorial, we have had to rely upon the awesome Django community and the Django Documentation for the answers and solutions. This book is really the combination of that knowledge pieced together in the context of building Rango.

We would also like to thank all the people who have helped to improve this resource by sending us comments, suggestions, Git issues and pull requests. If you've sent in changes over the years, please do remind us if you are not on the list!

Adam Kikowski, Adam Mertz, **Alessio Oxilia**, Ally Weir, **bernieyangmh**, Breakerfall, **Brian**, Burak K., **Burak Karaboga**, Can Ibanoglu, **Charlotte**, Claus Conrad, **Codenius**, cspollar, **Dan C**, Darius, **David Manlove**, David Skillman, **Deep Sukhwani** Devin Fitzsimons, **Dhiraj Thakur**, Duncan Drizy, **Gerardo A-C**, Giles T., **Grigoriy M**, James Yeo, **Jan Felix Trettow**, Joe Maskell, **Jonathan Sundqvist**, Karen Little, **Kartik Singhal**, koviusesGitHub, **Krace Kumar**, ma-152478, **Manoel Maria**, Martin de G., **Matevz P.**, mHulb, **Michael Herman**, Michael Ho Chum, **Mickey P.**, Mike Gleen, **nCrazed**, Nitin Tulswani, **nolan-m**, Oleg Belausov, **pawonfire**, pdehaye, **Peter Mash**, Pierre-Yves Mathieu, **Praestgias**, pzkpfwVI, **Ramdog**, Rezha Julio, **rnevius**, Sadegh Kh, **Saex**, Saurabh Tandon, **Serede Sixty Six**, Svante Kvarnstrom, **Tanmay Kansara**, Thomas Murphy, **Thomas Whyyou**, William Vincent, and **Zhou**.

Thank you all very much!

Appendices

Setting up your System

This chapter provides a brief overview of the different components that you need to have working in order to develop Django apps.

Choosing a Python Version

Django supports both the Python 2.7.x and 3 programming languages. While they both share the same name, they are fundamentally different programming languages. In this chapter, we assume you are setting up Python 2.7.5 - you can change the version number as you require.

Installing Python

So, how do you go about installing Python 2.7/3.4 on your computer? You may already have Python installed on your computer - and if you are using a Linux distribution or OS X, you will definitely have it installed. Some of your operating system's functionality is implemented in Python, hence the need for an interpreter!

Unfortunately, nearly all modern operating systems utilise a version of Python that is older than what we require for this tutorial. There's many different ways in which you can install Python, and many of them are sadly rather tricky to accomplish. We demonstrate the most commonly used approaches, and provide links to additional reading for more information.

Do not remove your default Python installation

This section will detail how to run Python 2.7.5 *alongside* your current Python installation. It is regarded as poor practice to remove your operating system's default Python installation and replace it with a newer version. Doing so could render aspects of your operating system's functionality broken!

Apple mac OS/OS X

The most simple way to get Python 2.7.5 installed on your Mac is to download and run the simple installer provided on the official Python website. You can download the installer by visiting the webpage at http://www.python.org/getit/releases/2.7.5/.

Make sure you have the correct version for your Mac

Ensure that you download the `.dmg` file that is relevant to your particular mac OS/OS X installation!

1. Once you have downloaded the `.dmg` file, double-click it in the Finder.
2. The file mounts as a separate disk and a new Finder window is presented to you.
3. Double-click the file `Python.mpkg`. This will start the Python installer.
4. Continue through the various screens to the point where you are ready to install the software. You may have to provide your password to confirm that you wish to install the software.
5. Upon completion, close the installer and eject the Python disk. You can now delete the downloaded `.dmg` file.

You should now have an updated version of Python installed, ready for Django! Easy, huh? You can also install Python 3.4+ in a similar version, if you prefer to use Python 3.

Linux Distributions

Unfortunately, there are many different ways in which you can download, install and run an updated version of Python on your Linux distribution. To make matters worse, methodologies vary from distribution to distribution. For example, the instructions for installing Python on Fedora may differ from those to install it on an Ubuntu installation.

However, not all hope is lost. An awesome tool (or a *Python environment manager*) called pythonbrew can help us address this difficulty. It provides an easy way to install and manage different versions of Python, meaning you can leave your operating system's default Python installation alone.

Taken from the instructions provided from the pythonbrew GitHub page and this Stack Overflow question and answer page, the following steps will install Python 2.7.5 on your Linux distribution.

1. Open a new terminal instance.
2. Run the command `curl -kL http://xrl.us/pythonbrewinstall | bash`. This will download the installer and run it within your terminal for you. This installs pythonbrew into the directory `~/.pythonbrew`. Remember, the tilde (~) represents your home directory!
3. You then need to edit the file `~/.bashrc`. In a text editor (such as `gedit`, `nano`, `vi` or `emacs`), add the following to a new line at the end of `~/.bashrc`: `[[ -s $HOME/.pythonbrew/etc/bashrc ]] && source $HOME/.pythonbrew/etc/bashrc`
4. Once you have saved the updated `~/.bashrc` file, close your terminal and open a new one. This allows the changes you make to take effect.
5. Run the command `pythonbrew install 2.7.5` to install Python 2.7.5.

6. You then have to *switch* Python 2.7.5 to the *active* Python installation. Do this by running the command `pythonbrew switch 2.7.5`.
7. Python 2.7.5 should now be installed and ready to go.

Hidden Directories and Files

Directories and files beginning with a period or dot can be considered the equivalent of *hidden files* in Windows. Dot files are not normally visible to directory-browsing tools, and are commonly used for configuration files. You can use the `ls` command to view hidden files by adding the `-a` switch to the end of the command, giving the command `ls -a`.

Windows

By default, Microsoft Windows comes with no installations of Python. This means that you do not have to worry about leaving existing versions be; installing from scratch should work just fine. You can download a 64-bit or 32-bit version of Python from the official Python website. If you aren't sure which one to download, you can determine if your computer is 32-bit or 64-bit by looking at the instructions provided on the Microsoft website.

1. When the installer is downloaded, open the file from the location to which you downloaded it.
2. Follow the on-screen prompts to install Python.
3. Close the installer once completed, and delete the downloaded file.

Once the installer is complete, you should have a working version of Python ready to go. By default, Python 2.7.5 is installed to the directory `C:\Python27`. We recommend that you leave the path as it is.

Upon the completion of the installation, open a Command Prompt and enter the command `python`. If you see the Python prompt, installation was successful. However, in certain circumstances, the installer may not set your Windows installation's `PATH` environment variable correctly. This will result in the `python` command not being found. Under Windows 7, you can rectify this by performing the following:

1. Click the *Start* button, right click *My Computer* and select *Properties*.
2. Click the *Advanced* tab.
3. Click the *Environment Variables* button.
4. In the *System variables* list, find the variable called *Path*, click it, then click the *Edit* button.
5. At the end of the line, enter `;C:\python27;C:\python27\scripts`. Don't forget the semicolon - and certainly *do not* add a space.

6. Click OK to save your changes in each window.
7. Close any Command Prompt instances, open a new instance, and try run the `python` command again.

This should get your Python installation fully working. Things might differ ever so slightly on Windows 10.

Setting Up the `PYTHONPATH`

With Python now installed, we now need to check that the installation was successful. To do this, we need to check that the `PYTHONPATH` environment variable is setup correctly. `PYTHONPATH` provides the Python interpreter with the location of additional Python packages and modules which add extra functionality to the base Python installation. Without a correctly set `PYTHONPATH`, we'll be unable to install and use Django!

First, let's verify that our `PYTHONPATH` variable exists. Depending on the installation technique that you chose, this may or may not have been done for you. To do this on your UNIX-based operating system, issue the following command in a terminal.

```
$ echo $PYTHONPATH
```

On a Windows-based machine, open a Command Prompt and issue the following.

```
$ echo %PYTHONPATH%
```

If all works, you should then see output that looks something similar to the example below. On a Windows-based machine, you will obviously see a Windows path, most likely originating from the C drive.

```
/opt/local/Library/Frameworks/Python.framework/
    Versions/2.7/lib/python2.7/site-packages:
```

This is the path to your Python installation's `site-packages` directory, where additional Python packages and modules are stored. If you see a path, you can continue to the next part of this tutorial. If you however do not see anything, you'll need to do a little bit of detective work to find out the path. On a Windows installation, this should be a trivial exercise: `site-packages` is located within the `lib` directory of your Python installation directory. For example, if you installed Python to `C:\Python27`, `site-packages` will be at `C:\Python27\Lib\site-packages\`.

UNIX-based operating systems however require a little bit of detective work to discover the path of your `site-packages` installation. To do this, launch the Python interpreter. The following terminal session demonstrates the commands you should issue.

```
$ python

Python 2.7.5 (v2.7.5:ab05e7dd2788, May 13 2013, 13:18:45)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.

>>> import site
>>> print(site.getsitepackages()[0])

'/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/site-packages'

>>> quit()
```

Calling `site.getsitepackages()` returns a list of paths that point to additional Python package and module stores. The first typically returns the path to your `site-packages` directory - changing the list index position may be required depending on your installation. If you receive an error stating that `getsitepackages()` is not present within the `site` module, verify you're running the correct version of Python. Version 2.7.5 should include this function. Previous versions of the language do not include this function.

The string which is shown as a result of executing `print site.getsitepackages()[0]` is the path to your installation's `site-packages` directory. Taking the path, we now need to add it to your configuration. On a UNIX-based or UNIX-derived operating system, edit your `.bashrc` file once more, adding the following to the bottom of the file.

```
export PYTHONPATH=$PYTHONPATH:<PATH_TO_SITE-PACKAGES>
```

Replace `<PATH_TO_SITE-PACKAGES>` with the path to your `site-packages` directory. Save the file, and quit and reopen any instances of your terminal.

On a Windows-based computer, you must follow the instructions shown above to bring up the environment variables settings dialog. Add a `PYTHONPATH` variable with the value being set to your `site-packages` directory, which is typically `C:\Python27\Lib\site-packages\`.

Using `setuptools` and `pip`

Installing and setting up your development environment is a really important part of any project. While it is possible to install Python Packages such as Django separately, this can lead to numerous problems and hassles later on. For example, how would you share your setup with another developer? How would you set up the same environment on your new machine? How would you upgrade to the latest version of the package? Using a package manager removes much of the hassle involved in setting up and configuring your environment. It will also ensure that the package you

install is the correct for the version of Python you are using, along with installing any other packages that are dependent upon the one you want to install.

In this book, we use `pip`. `pip` is a user friendly wrapper over the `setuptools` Python package manager. Because `pip` depends on `setuptools`, we are required to ensure that both are installed on your computer.

To start, we should download `setuptools` from the official Python package website. You can download the package in a compressed `.tar.gz` file. Using your favourite file extracting program, extract the files. They should all appear in a directory called `setuptools-1.1.6` - where `1.1.6` represents the `setuptools` version number. From a terminal instance, you can then change into the directory and execute the script `ez_setup.py` as shown below.

```
$ cd setuptools-1.1.6
$ sudo python ez_setup.py
```

In the example above, we also use `sudo` to allow the changes to become system wide. The second command should install `setuptools` for you. To verify that the installation was successful, you should be able to see output similar to that shown below.

```
Finished processing dependencies for setuptools==1.1.6
```

Of course, `1.1.6` is substituted with the version of `setuptools` you are installing. If this line can be seen, you can move onto installing `pip`. This is a trivial process, and can be completed with one simple command. From your terminal instance, enter the following.

```
$ sudo easy_install pip
```

This command should download and install `pip`, again with system wide access. You should see the following output, verifying `pip` has been successfully installed.

```
Finished processing dependencies for pip
```

Upon seeing this output, you should be able to launch `pip` from your terminal. To do so, just type `pip`. Instead of an unrecognised command error, you should be presented with a list of commands and switches that `pip` accepts. If you see this, you're ready to move on!

No Sudo on Windows!

On Windows computers, follow the same basic process. You won't need to enter the `sudo` command, however.

Virtual Environments

We're almost all set to go! However, before we continue, it's worth pointing out that while this setup is fine to begin with, there are some drawbacks. What if you had another Python application that requires a different version to run? Or you wanted to switch to the new version of Django, but still wanted to maintain your Django 1.7 project?

The solution to this is to use virtual environments. Virtual environments allow multiple installations of Python and their relevant packages to exist in harmony. This is the generally accepted approach to configuring a Python setup nowadays. They are pretty easy to setup, once you have `pip` installed, and you know the right commands. You need to install a couple of additional packages.

```
$ pip install virtualenv
$ pip install virtualenvwrapper
```

The first package provides you with the infrastructure to create a virtual environment. See a non-magical introduction to `pip` and Virtualenv for Python Beginners by Jamie Matthews for details about using virtualenv. However, using just *virtualenv* alone is rather complex. The second package provides a wrapper to the functionality in the virtualenv package and makes life a lot easier.

If you are using a Linux/UNIX based OS, then to use the wrapper you need to call the following shell script from your command line: :

```
$ source virtualenvwrapper.sh
```

It is a good idea to add this to your bash/profile script. So you don't have to run it each and every time you want to use virtual environments. However, if you are using windows, then install the virtualenvwrapper-win package:

```
$ pip install virtualenvwrapper-win
```

Now you should be all set to create a virtual environment:

```
$ mkvirtualenv rango
```

You can list the virtual environments created with `lsvirtualenv`, and you can activate a virtual environment as follows:

```
$ workon rango
(rango)$
```

Your prompt with change and the current virtual environment will be displayed, i.e. `rango`. Now within this environment you will be able to install all the packages you like, without interfering with your standard or other environments. Try `pip list` to see you don't have Django or Pillow installed in your virtual environment. You can now install them with `pip` so that they exist in your virtual environment.

Version Control

We should also point out that when you develop code, you should always house your code within a version controlled repository such as SVN or Git. We have provided a chapter on using Git if you haven't used Git and GitHub before. We highly recommend that you set up a Git repository for your own projects. Doing so could save you from disaster.

Exercises

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

- Install Python 2.7.5+ or Python 3.4+ and `pip`.
- Play around with your CLI and create a directory called `code`, which we use to create our projects in.
- Install the Django and Pillow packages.
- Setup your Virtual Environment
- Setup your account on GitHub
- Download and setup a Integrated Development Environment like PyCharm Edu.
- We have made the code for the book and application that you build available on GitHub, see Tango With Django Book and Rango Application.
- If you spot any errors or problem with the book, you can make a change request!
- If you have any problems with the exercises, you can check out the repository and see how we completed them.

A Crash Course in UNIX-based Commands

Depending on your computing background, you may or may not have encountered a UNIX based system, or a derivative of. This small crash course focuses on getting you up to speed with the *terminal*, an application in which you issue commands for the computer to execute. This differs from a point-and-click *Graphical User Interface (GUI)*, the kind of interface that has made computing so much more accessible. A terminal based interface may be more complex to use, but the benefits of using such an interface include getting things done quicker, and more accurately, too.

Not for Windows!

Note that we're focusing on the Bash shell, a shell for UNIX-based operating systems and their derivatives, including OS X and Linux distributions. If you're a Windows user, you can use the Windows Command Prompt or Windows PowerShell. Users of Windows 10 with the 2016 Anniversary Update will also be able to issue Bash commands directly to the Command Prompt. You could also experiment by installing Cygwin to bring Bash commands to Windows.

Using the Terminal

UNIX based operating systems and derivatives - such as OS X and Linux distributions - all use a similar looking terminal application, typically using the Bash shell. All possess a core set of commands that allow you to navigate through your computer's filesystem and launch programs - all without the need for any graphical interface.

Upon launching a new terminal instance, you'll be typically presented with something resembling the following.

```
sibu:~ david$
```

What you see is the *prompt*, and indicates when the system is waiting to execute your every command. The prompt you see varies depending on the operating system you are using, but all look generally very similar. In the example above, there are three key pieces of information to observe:

- your username and computer name (username of `david` and computer name of `sibu`);

- your *present working directory* (the tilde, or ~); and
- the privilege of your user account (the dollar sign, or $).

What is a Directory?

In the text above, we refer to your present working directory. But what exactly is a *directory*? If you have used a Windows computer up until now, you'll probably know a directory as a *folder*. The concept of a folder is analogous to a directory - it is a cataloguing structure that contains references to other files and directories.

The dollar sign (`$`) typically indicates that the user is a standard user account. Conversely, a hash symbol (`#`) may be used to signify the user logged in has root privileges. Whatever symbol is present is used to signify that the computer is awaiting your input.

Prompts can Differ

The information presented by the prompt on your computer may differ from the example shown above. For example, some prompts may display the current date and time, or any other information. It all depends how your computer is set up.

When you are using the terminal, it is important to know where you are in the file system. To find out where you are, you can issue the command `pwd`. This will display your *Present Working Directory* (hence `pwd`). For example, check the example terminal interactions below.

```
Last login: Wed Mar 23 15:01:39 2016
sibu:~ david$ pwd
/users/grad/david
sibu:~ david$
```

You can see that the present working directory in this example is `/users/grad/david`.

You'll also note that the prompt indicates that the present working directory is a tilde ~. The tilde is used a special symbol which represents your *home directory*. The base directory in any UNIX based file system is the *root directory*. The path of the root directory is denoted by a single forward slash (`/`). As folders (or directories) are separated in UNIX paths with a `/`, a single `/` denotes the root!

If you are not in your home directory, you can *Change Directory* (`cd`) by issuing the following command:

```
sibu:/ david$ cd ~
sibu:~ david$
```

Note how the present working directory switches from `/` to ~ upon issuing the `cd ~` command.

Path Shortcuts

UNIX shells have a number of different shorthand ways for you to move around your computer's filesystem. You've already seen that a forward slash (`/`) represents the root directory, and the tilde (~) represents your home directory in which you store all your personal files. However, there are a few more special characters you can use to move around your filesystem in conjunction with the `cd` command.

- Issuing `cd ~` will always return you to your home directory. On some UNIX or UNIX derivatives, simply issuing `cd` will return you to your home directory, too.
- Issuing `cd ..` will move your present working directory **up one level** of the filesystem hierarchy. For example, if you are currently in `/users/grad/david/code/`, issuing `cd ..` will move you to `/users/grad/david/`.
- Issuing `cd -` will move you to the **previous directory you were working in**. Your shell remembers where you were, so if you were in `/var/tmp/` and moved to `/users/grad/david/`, issuing `cd -` will move you straight back to `/var/tmp/`. This command obviously only works if you've move around at least once in a given terminal session.

Now, let's create a directory within the home directory called `code`. To do this, you can use the *Make Directory* command, called `mkdir`.

```
sibu:~ david$ mkdir code
sibu:~ david$
```

There's no confirmation that the command succeeded. We can change the present working directory with the `cd` command to change to `code`. If this succeeds, we will know the directory has been successfully created.

```
sibu:~ david$ cd code
sibu:code david$
```

Issuing a subsequent `pwd` command to confirm our present working directory yields `/users/grad/david/code` - our home directory, with `code` appended to the end. You can also see from the prompt in the example above that the present working directory changes from ~ to `code`.

Change Back

Now issue the command to change back to your home directory. What command do you enter?

From your home directory, let's now try out another command to see what files and directories exist. This new command is called `ls`, shorthand for *list*. Issuing `ls` in your home directory will yield something similar to the following.

```
sibu:~ david$ ls
code
```

This shows us that there's something present our home directory called `code`, as we would expect. We can obtain more detailed information by adding a `l` switch to the end of the `ls` command - with `l` standing for *list*.

```
sibu:~ david$ ls -l
drwxr-xr-x  2 david  grad  68  2 Apr 11:07 code
```

This provides us with additional information, such as the modification date (`2 Apr 11:07`), whom the file belongs to (user `david` of group `grad`), the size of the entry (68 bytes), and the file permissions (`drwxr-xr-x`). While we don't go into file permissions here, the key thing to note is the `d` at the start of the string that denotes the entry is a directory. If we then add some files to our home directory and reissue the `ls -l` command, we then can observe differences in the way files are displayed as opposed to directories.

```
sibu:~ david$ ls -l
drwxr-xr-x  2 david  grad      68  2 Apr 11:07 code
-rw-r--r--@ 1 david  grad  303844  1 Apr 16:16 document.pdf
-rw-r--r--  1 david  grad      14  2 Apr 11:14 readme.md
```

One final useful switch to the `ls` command is the `a` switch, which displays *all* files and directories. This is useful because some directories and files can be *hidden* by the operating system to keep things looking tidy. Issuing the command yields more files and directories!

```
sibu:~ david$ ls -la
-rw-r--r--   1  david  grad     463 20 Feb 19:58 .profile
drwxr-xr-x   16 david  grad     544 25 Mar 11:39 .virtualenvs
drwxr-xr-x   2  david  grad      68  2 Apr 11:07 code
-rw-r--r--@  1  david  grad  303844  1 Apr 16:16 document.pdf
-rw-r--r--   1  david  grad      14  2 Apr 11:14 readme.md
```

This command shows a hidden directory `.virtualenvs` and a hidden file `.profile`. Note that hidden files on a UNIX based computer (or derivative) start with a period (`.`). There's no special `hidden` file attribute you can apply, unlike on Windows computers.

Combining `ls` Switches

You may have noticed that we combined the `l` and `a` switches in the above `ls` example to force the command to output a list displaying all hidden files. This is a valid command - and there are even more switches you can use to customise the output of `ls`.

Creating files is also easy to do, straight from the terminal. The `touch` command creates a new, blank file. If we wish to create a file called `new.txt`, issue `touch new.txt`. If we then list our directory, we then see the file added.

```
sibu:~ david$ ls -l
drwxr-xr-x  2 david  grad      68  2 Apr 11:07 code
-rw-r--r--@ 1 david  grad  303844  1 Apr 16:16 document.pdf
-rw-r--r--  1 david  grad       0  2 Apr 11:35 new.txt
-rw-r--r--  1 david  grad      14  2 Apr 11:14 readme.md
```

Note the filesize of `new.txt` - it is zero bytes, indicating an empty file. We can start editing the file using one of the many available text editors that are available for use directly from a terminal, such as `nano` or `vi`. While we don't cover how to use these editors here, you can have a look online for a simple how-to tutorial. We suggest starting with `nano` - while there are not as many features available compared to other editors, using `nano` is much simpler.

Core Commands

In the short tutorial above, you've covered a few of the core commands such as `pwd`, `ls` and `cd`. There are however a few more standard UNIX commands that you should familiarise yourself with before you start working for real. These are listed below for your reference, with most of them focusing upon file management. The list comes with an explanation of each, and an example of how to use them.

- `pwd`: As explained previously, this command displays your *present working directory* to the terminal. The full path of where you are presently is displayed.
- `ls`: Displays a list of files in the current working directory to the terminal.
- `cd`: In conjunction with a path, `cd` allows you to change your present working directory. For example, the command `cd /users/grad/david/` changes the current working directory to `/users/grad/david/`. You can also move up a directory level without having to provide the absolute path by using two dots, e.g. `cd ..`.
- `cp`: Copies files and/or directories. You must provide the *source* and the *target*. For example, to make a copy of the file `input.py` in the same directory, you could issue the command `cp input.py input_backup.py`.

- `mv`: Moves files/directories. Like `cp`, you must provide the *source* and *target*. This command is also used to rename files. For example, to rename `numbers.txt` to `letters.txt`, issue the command `mv numbers.txt letters.txt`. To move a file to a different directory, you would supply either an absolute or relative path as part of the target - like `mv numbers.txt /home/david/numbers.txt`.
- `mkdir`: Creates a directory in your current working directory. You need to supply a name for the new directory after the `mkdir` command. For example, if your current working directory was `/home/david/` and you ran `mkdir music`, you would then have a directory `/home/david/music/`. You will need to then `cd` into the newly created directory to access it.
- `rm`: Shorthand for *remove*, this command removes or deletes files from your filesystem. You must supply the filename(s) you wish to remove. Upon issuing a `rm` command, you will be prompted if you wish to delete the file(s) selected. You can also remove directories using the recursive switch. Be careful with this command - recovering deleted files is very difficult, if not impossible!
- `rmdir`: An alternative command to remove directories from your filesystem. Provide a directory that you wish to remove. Again, be careful: you will not be prompted to confirm your intentions.
- `sudo`: A program which allows you to run commands with the security privileges of another user. Typically, the program is used to run other programs as `root` - the superuser of any UNIX-based or UNIX-derived operating system.

There's More!

This is only a brief list of commands. Check out Ubuntu's documentation on Using the Terminal for a more detailed overview, or the Cheat Sheet by FOSSwire for a quick, handy reference guide. Like anything else, the more you practice, the more comfortable you will feel working with the terminal.

A Git Crash Course

We strongly recommend that you spend some time familiarising yourself with a version control system for your application's codebase. This chapter provides you with a crash course in how to use Git, one of the many version control systems available. Originally developed by Linus Torvalds, Git is today one of the most popular version control systems in use, and is used by open-source and closed-source projects alike.

This tutorial demonstrates at a high level how Git works, explains the basic commands that you can use, and provides an explanation of Git's workflow. By the end of this chapter, you'll be able to make contributions to a Git repository, enabling you to work solo, or in a team.

Why Use Version Control?

As your software engineering skills develop, you will find that you are able to plan and implement solutions to ever more complex problems. As a rule of thumb, the larger the problem specification, the more code you have to write. The more code you write, the greater the emphasis you should put on software engineering practices. Such practices include the use of design patterns and the *DRY (Don't Repeat Yourself)* principle.

Think about your experiences with programming thus far. Have you ever found yourself in any of these scenarios?

- Made a mistake to code, realised it was a mistake and wanted to go back?
- Lost code (through a faulty drive), or had a backup that was too old?
- Had to maintain multiple versions of a product (perhaps for different organisations)?
- Wanted to see the difference between two (or more) versions of your codebase?
- Wanted to show that a particular change broke of fixed a piece of code?
- Wanted to submit a change (patch) to someone else's code?
- Wanted to see how much work is being done (where it was done, when it was done, or who did it)?

Using a version control system makes your life easier in *all* of the above cases. While using version control systems at the beginning may seem like a hassle it will pay off later - so it's good to get into the habit now!

We missed one final (and important) argument for using version control. With ever more complex problems to solve, your software projects will undoubtedly contain a large number of files containing source code. It'll also be likely that you *aren't working alone on the project; your project will probably have more than one contributor.* In this scenario, it can become difficult to avoid conflicts when working on files.

How Git Works

Essentially, Git comprises of four separate storage locations: your **workspace**, the **local index**, the **local repository** and the **remote repository**. As the name may suggest, the remote repository is stored on some remote server, and is the only location stored on a computer other than your own. This means that there are two copies of the repository - your local copy, and the remote copy. Having two copies is one of the main selling points of Git over other version control systems. You can make changes to your local repository when you may not have Internet access, and then apply any changes to the remote repository at a later stage. Only once changes are made to the remote repository can other contributors see your changes.

What is a Repository?

We keep repeating the word *repository*, but what do we actually mean by that? When considering version control, a repository is a data structure which contains metadata (a set of data that describes other data, hence *meta*) concerning the files which you are storing within the version control system. The kind of metadata that is stored can include aspects such as the historical changes that have taken place within a given file, so that you have a record of all changes that take place.

If you want to learn more about the metadata stored by Git, there is a technical tutorial available for you to read through.

For now though, let's provide an overview of each of the different aspects of the Git system. We'll recap some of the things we've already mentioned just to make sure it makes sense to you.

- As already explained, the **remote repository** is the copy of your project's repository stored on some remote server. This is particularly important for Git projects that have more than one contributor - you require a central place to store all the work that your team members produce. You could set up a Git server on a computer with Internet access and a properly configured firewall (check out this Server Fault question, for example), or simply use one of many services providing free Git repositories. One of the most widely used services available today is GitHub. In fact, this book has a Git repository on GitHub!
- The **local repository** is a copy of the remote repository stored on your computer (locally). This is the repository to which you make all your additions, changes and deletions. When you reach a particular milestone, you can then *push* all your local changes to the remote repository. From there, you can instruct your team members to retrieve your changes. This concept is known as *pulling* from the remote repository. We'll subsequently explain pushing and pulling in a bit more detail.
- The **local index** is technically part of the local repository. The local index stores a list of files that you want to be managed with version control. This is explained in more detail later in this chapter. You can have a look here to see a discussion on what exactly a Git index contains.

- The final aspect of Git is your **workspace**. Think of this folder or directory as the place on your computer where you make changes to your version controlled files. From within your workspace, you can add new files or modify or remove previously existing ones. From there, you then instruct Git to update the repositories to reflect the changes you make in your workspace. This is important - *don't modify code inside the local repository - you only ever edit files in your workspace.*

Next, we'll be looking at how to get your Git workspace set up and ready to go. We'll also discuss the basic workflow you should use when using Git.

Setting up Git

We assume that you've got Git installed with the software to go. One easy way to test the software out is to simply issue `git` to your terminal or Command Prompt. If you don't see a `command not found` error, you're good to go. Otherwise, have a look at how to install Git to your system.

Using Git on Windows

Like Python, Git doesn't come as part of a standard Windows installation. However, Windows implementations of the version control system can be downloaded and installed. You can download the official Windows Git client from the Git website. The installer provides the `git` command line program, which we use in this crash course. You can also download a program called *TortoiseGit*, a graphical extension to the Windows Explorer shell. The program provides a really nice right-click Git context menu for files. This makes version control really easy to use. You can download TortoiseGit for free. Although we do not cover how to use TortoiseGit in this crash course, many tutorials exist online for it. Check this tutorial if you are interested in using it.

We recommend however that you stick to the command line program. We'll be using the commands in this crash course. Furthermore, if you switch to a UNIX/Linux development environment at a later stage, you'll be glad you know the commands!

Setting up your Git workspace is a straightforward process. Once everything is set up, you will begin to make sense of the directory structure that Git uses. Assume that you have signed up for a new account on GitHub and created a new repository on the service for your project. With your remote repository setup, follow these steps to get your local repository and workspace setup on your computer. We'll assume you will be working from your `<workspace>` directory.

1. Open a terminal and navigate to your home directory (e.g. `$ cd ~`).
2. *Clone* the remote repository - or in other words, make a copy of it. Check out how to do this below.
3. Navigate into the newly created directory. That's your workspace in which you can add files to be version controlled!

How to Clone a Remote Repository

Cloning your repository is a straightforward process with the `git clone` command. Supplement this command with the URL of your remote repository - and if required, authentication details, too. The URL of your repository varies depending on the provider you use. If you are unsure of the URL to enter, it may be worth querying it with your search engine or asking someone in the know.

For GitHub, try the following command, replacing the parts below as appropriate:

```
$ git clone https://<USER>:<PASS>@github.com/<OWNER>/<REPO_NAME>.git <workspace>
```

where you replace

- `<USER>` with your GitHub username;
- `<PASS>` with your GitHub password;
- `<OWNER>` with the username of the person who owns the repository;
- `<REPO_NAME>` with the name of your project's repository; and
- `<workspace>` with the name for your workspace directory. This is optional; leaving this option out will simply create a directory with the same name as the repository.

If all is successful, you'll see some text like the example shown below.

```
$ git clone https://github.com/leifos/tango_with_django_19
Cloning into 'tango_with_django_19'...
remote: Counting objects: 18964, done.
remote: Total 18964 (delta 0), reused 0 (delta 0), pack-reused 18964
Receiving objects: 100% (18964/18964), 99.69 MiB | 3.51 MiB/s, done.
Resolving deltas: 100% (13400/13400), done.
Checking connectivity... done.
```

If the output lines end with `done`, everything should have worked. Check your filesystem to see if the directory has been created.

Not using GitHub?

There are many websites that provide Git repositories - some free, some paid. While this chapter uses GitHub, you are free to use whatever service you wish. Other providers include Atlassian Bitbucket and Unfuddle. You will of course have to change the URL from which you clone your repository if you use a service other than GitHub.

The Directory Structure

Once you have cloned your remote repository onto your local computer, navigate into the directory with your terminal, Command Prompt or GUI file browser. If you have cloned an empty repository the workspace directory should appear empty. This directory is therefore your blank workspace with which you can begin to add your project's files.

However, the directory isn't blank at all! On closer inspection, you will notice a hidden directory called `.git`. Stored within this directory are both the local repository and local index. **Do not alter the contents of the `.git` directory**. Doing so could damage your Git setup and break version control functionality. *Your newly created workspace therefore actually contains within it the local repository and index.*

Final Tweaks

With your workspace setup, now would be a good time to make some final tweaks. Here, we discuss two cool features you can try which could make your life (and your team members') a little bit easier.

When using your Git repository as part of a team, any changes you make will be associated with the username you use to access your remote Git repository. However, you can also specify your full name and e-mail address to be included with changes that are made by you on the remote repository. Simply open a Command Prompt or terminal and navigate to your workspace. From there, issue two commands: one to tell Git your full name, and the other to tell Git your e-mail address.

```
$ git config user.name "John Doe"
$ git config user.email "johndoe123@me.com"
```

Obviously, replace the example name and e-mail address with your own - unless your name actually is John Doe.

Git also provides you with the capability to stop - or ignore - particular files from being added to version control. For example, you may not wish a file containing unique keys to access web services from being added to version control. If the file were to be added to the remote repository, anyone could theoretically access the file by cloning the repository. With Git, files can be ignored by including them in the `.gitignore` file, which resides in the root of `<workspace>`. When adding files to version control, Git parses this file. If a file that is being added to version control is listed within `.gitignore`, the file is ignored. Each line of `.gitignore` should be a separate file entry.

Check out the following example of a `.gitignore` file:

```
`config/api_keys.py`
`*.pyc`
```

In this example file, there are two entries - one on each line. The first entry prompts Git to ignore the file `api_keys.py` residing within the `config` directory of your repository. The second entry then prompts Git to ignore *all* instance of files with a `.pyc` extension, or compiled Python files. This is a really nice feature: you can use *wildcards* to make generic entries if you need to!

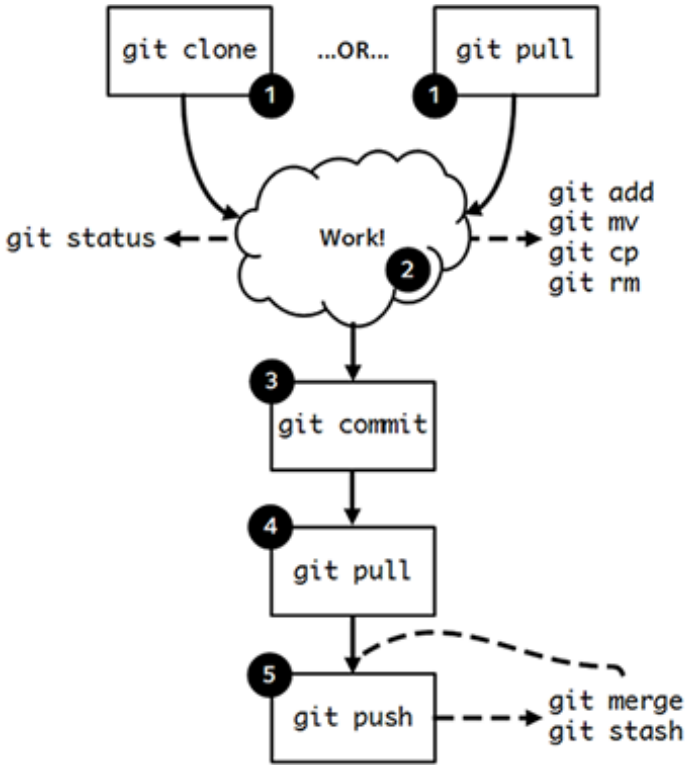
`.gitignore` - What else should I ignore?

There are many kinds of files you could safely ignore from being committed and pushed to your Git repositories. Examples include temporary files, databases (that can easily be recreated) and operating system-specific files. Operating system-specific files include configurations for the appearance of the directory when viewed in a given file browser. Windows computers create `thumbs.db` files, while OS X creates `.DS_Store` files.

When you create a new repository on GitHub, the service can offer to create a `.gitignore` file based upon the languages you will use in your project, which can save you some time setting everything up.

Basic Commands and Workflow

With your repository cloned and ready to go on your local computer, you're ready to get to grips with the Git workflow. This section shows you the basic Git workflow - and the associated Git commands you can issue.



A Figure of the Git Workflow

We have provided a pictorial representation of the basic Git workflow as shown above. Match each of the numbers in the black circles to the numbered descriptions below to read more about each stage. **Refer to this diagram whenever you're unsure about the next step you should take - it's very useful!**

1. Starting Off

Before you can start work on your project, you must prepare Git. If you haven't yet sorted out your project's Git workspace, you'll need to `clone` your repository to set it up.

If you've already cloned your repository, it's good practice to get into the habit of updating your local copy by using the `git pull` command. This *pulls* the latest changes from the remote repository onto your computer. By doing this, you'll be working from the same page as your team members. This will reduce the possibility of conflicting versions of files, which really does make your life a bit of a nightmare.

To perform a `git pull`, first navigate to your `<workspace>` directory within your Command Prompt or terminal, then issue `git pull`. Check out the snippet below from a Bash terminal to see exactly what you need to do, and what output you should expect to see.

```
$ cd <workspace>
$ git pull
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.
From https://github.com/someuser/somerepository
    86a0b3b..a7cec3d  master     -> origin/master
Updating 86a0b3b..a7cec3d
Fast-forward
  README.md | 1 +
  1 file changed, 1 insertion(+)
  create mode 100644 README.md
```

This example shows that a `README.md` file has been updated or created from the latest pull.

Getting an Error?

If you receive `fatal: Not a git repository (or any of the parent directories): .git`, you're not in the correct directory. You need `cd` to your workspace directory - the one in which you cloned your repository to. A majority of Git commands only work when you're in a Git repository.

Pull before you Push!

Always `git pull` on your local copy of your repository before you begin to work. **Always!**

Before you are about to push, do another pull.

Remember to talk to your team to coordinate your activity so you are not working on the same files, or using branching.

2. Doing Some Work!

Once your workspace has been cloned or updated with the latest changes, it's time for you to get some work done! Within your workspace directory, you can take existing files and modify them. You can delete them too, or add new files to be version controlled.

When you modify your repository in any way, you need to keep Git up-to-date of any changes. Doing so allows Git to update your local index. The list of files stored within the local index are then used to perform your next *commit*, which we'll be discussing in the next step. To keep Git informed, there are several Git commands that let you update the local index. Three of the commands are near identical to those that were discussed in the Unix Crash Course (e.g. `cp`, `mv`), with the addition of a `git` prefix.

- The first command `git add` allows you to request Git to add a particular file to the next commit for you. A common newcomer mistake is to assume that `git add` is used for adding new files to your repository only - *this is not the case. You must tell Git what modified files you wish to commit, too.* The command is invoked by typing `git add <filename>`, where `<filename>` is the name of the file you wish to add to your next commit. Multiple files and directories can be added with the command `git add .` - **but be careful with this**.
- `git mv` performs the same function as the Unix `mv` command - it moves files. The only difference between the two is that `git mv` updates the local index for you before moving the file. Specify the filename with the syntax `git mv <current_filename> <new_filename>`. For example, with this command you can move files to a different directory within your repository. This will be reflected in your next commit. The command is also used to rename files - from the old filename to the new.
- `git cp` allows you to make a copy of a file or directory while adding references to the new files into the local index for you. The syntax is the same as `git mv` above where the filename or directory name is specified thus: `git cp <current_filename> <copied_filename>`.
- The command `git rm` adds a file or directory delete request into the local index. While the `git rm` command does not delete the file straight away, the requested file or directory is removed from your filesystem and the Git repository upon the next commit. The syntax is similar to the `git add` command, where a filename can be specified thus: `git rm <filename>`. Note that you can add a large number of requests to your local index in one go, rather than removing each file manually. For example, `git rm -rf media/` creates delete requests in your local index for the `media/` directory. The `r` switch enables Git to *recursively* remove each file within the `media/` directory, while `f` allows Git to *forcibly* remove the files. Check out the Wikipedia page on the `rm` command for more information.

Lots of changes between commits can make things pretty confusing. You may easily forget what files you've already instructed Git to remove, for example. Fortunately, you can run the `git status` command to see a list of files which have been modified from your current working directory, but haven't been added to the local index for processing. Check out typical output from the command below to get a taste of what you can see.

Working with `.gitignore`

If you have set up your `.gitignore` file correctly, you'll notice that files matching those specified within the `.gitignore` file are...ignored when you `git add` them. This is the intended behaviour - these files are not supposed to be committed to version control! If you however do need a file to be included that is in `.gitignore`, you can *force* Git to include it if necessary with the `git add -f <filename>` command.

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
    modified:   chapter-unix.md
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)
    modified:   chapter-git.md
```

From this example above, we can see that the file `chapter-unix.md` has been added to the latest commit, and will therefore be updated in the next `git push`. The file `chapter-git.md` has been updated, but `git add` hasn't been run on the file, so the changes won't be applied to the repository.

Checking Status

For further information on the `git status` command, check out the official Git documentation.

3. Committing your Changes

We've mentioned *committing* several times in the previous step - but what does it mean? Committing is when you save changes - which are listed in the local index - that you have made within your workspace. The more often you commit, the greater the number of opportunities you'll have to revert back to an older version of your code if things go wrong. Make sure you commit often, but don't commit an incomplete or broken version of a particular module or function. There's a lot of discussion as to when the ideal time to commit is. Have a look at this Stack Overflow page for the opinions of several developers. It does however make sense to commit only when everything is working. If you find you need to roll back to a previous commit only to find nothing works, you won't be too happy.

To commit, you issue the `git commit` command. Any changes to existing files that you have indexed will be saved to version control at this point. Additionally, any files that you've requested to be

copied, removed, moved or added to version control via the local index will be undertaken at this point. When you commit, you are updating the *HEAD* of your local repository.

Commit Requirements

In order to successfully commit, you need to modify at least one file in your repository and instruct Git to commit it, through the `git add` command. See the previous step for more information on how to do this.

As part of a commit, it's incredibly useful to your future self and others to explain why you committed when you did. You can supply an optional message with your commit if you wish to do so. Instead of simply issuing `git commit`, run the following amended command.

```
$ git commit -m "Updated helpers.py to include a Unicode conversion function."
```

From the example above, you can see that using the `-m` switch followed by a string provides you with the opportunity to append a message to your commit. Be as explicit as you can, but don't write too much. People want to see at a glance what you did, and do not want to be bored or confused with a long essay. At the same time, don't be too vague. Simply specifying `Updated helpers.py` may tell a developer what file you modified, but they will require further investigation to see exactly what you changed.

Sensible Commits

Although frequent commits may be a good thing, you will want to ensure that what you have written actually *works* before you commit. This may sound silly, but it's an incredibly easy thing to not think about. To reiterate, committing code which doesn't actually work can be infuriating to your team members if they then rollback to a version of your project's codebase which is broken!

4. Synchronising your Repository

Important when Collaborating

Synchronising your local repository before making changes is crucial to ensure you minimise the chance for conflicts occurring. Make sure you get into the habit of doing a `pull` before you `push`.

After you've committed your local repository and committed your changes, you're just about ready to send your commit(s) to the remote repository by *pushing* your changes. However, what if someone

within your group pushes their changes before you do? This means your local repository will be out of sync with the remote repository, meaning that any `git push` command that you issue will fail.

It's therefore always a good idea to check whether changes have been made on the remote repository before updating it. Running a `git pull` command will pull down any changes from the remote repository, and attempt to place them within your local repository. If no changes have been made, you're clear to push your changes. If changes have been made and cannot be easily rectified, you'll need to do a little bit more work.

In scenarios such as this, you have the option to *merge* changes from the remote repository. After running the `git pull` command, a text editor will appear in which you can add a comment explaining why the merge is necessary. Upon saving the text document, Git will merge the changes from the remote repository to your local repository.

Editing Merge Logs

If you do see a text editor on your Mac or Linux installation, it's probably the `vi` text editor. If you've never used vi before, check out this helpful page containing a list of basic commands on the Colorado State University Computer Science Department website. If you don't like vi, you can change the default text editor that Git calls upon. Windows installations most likely will bring up Notepad.

5. Pushing your Commit(s)

Pushing is the phrase used by Git to describe the sending of any changes in your local repository to the remote repository. This is the way in which your changes become available to your other team members, who can then retrieve them by running the `git pull` command in their respective local workspaces. The `git push` command isn't invoked as often as committing - *you require one or more commits to perform a push.* You could aim for one push per day, when a particular feature is completed, or at the request of a team member who is after your updated code.

To push your changes, the simplest command to run is:

```
$ git push origin master
```

As explained on this Stack Overflow question and answer page this command instructs the `git push` command to push your local master branch (where your changes are saved) to the *origin* (the remote server from which you originally cloned). If you are using a more complex setup involving branching and merging, alter `master` to the name of the branch you wish to push.

Important Push?

If your `git push` is particularly important, you can also alert other team members to the fact they should really update their local repositories by pulling your changes. You can do this through a *pull request.* Issue one after pushing your latest changes by invoking the command `git request-pull master`, where master is your branch name (this is the default value). If you are using a service such as GitHub, the web interface allows you to generate requests without the need to enter the command. Check out the official GitHub website's tutorial for more information.

Recovering from Mistakes

This section presents a solution to a coder's worst nightmare: what if you find that your code no longer works? Perhaps a refactoring went terribly wrong, or another team member without discussion changed something. Whatever the reason, using a form of version control always gives you a last resort: rolling back to a previous commit. This section details how to do just that. We follow the information given from this Stack Overflow question and answer page.

Changes may be Lost!

You should be aware that this guide will rollback your workspace to a previous iteration. Any uncommitted changes that you have made will be lost, with a very slim chance of recovery! Be wary. If you are having a problem with only one file, you could always view the different versions of the files for comparison. Have a look at this Stack Overflow page to see how to do that.

Rolling back your workspace to a previous commit involves two steps: determining which commit to roll back to, an performing the rollback. To determine what commit to rollback to, you can make use of the `git log` command. Issuing this command within your workspace directory will provide a list of recent commits that you made, your name and the date at which you made the commit. Additionally, the message that is stored with each commit is displayed. This is where it is highly beneficial to supply commit messages that provide enough information to explain what is going on. Check out the following output from a `git log` invocation below to see for yourself.

```
commit 88f41317640a2b62c2c63ca8d755feb9f17cf16e                <- Commit hash
Author: John Doe <someaddress@domain.com>                      <- Author
Date:   Mon Jul 8 19:56:21 2013 +0100                          <- Date/time
    Nearly finished initial version of the requirements chapter <- Message
commit f910b7d557bf09783b43647f02dd6519fa593b9f
Author: John Doe <someaddress@domain.com>
Date:   Wed Jul 3 11:35:01 2013 +0100
    Added in the Git figures to the requirements chapter.
commit c97bb329259ee392767b87cfe7750ce3712a8bdf
Author: John Doe <someaddress@domain.com>
Date:   Tue Jul 2 10:45:29 2013 +0100
    Added initial copy of Sphinx documentation and tutorial code.
commit 2952efa9a24dbf16a7f32679315473b66e3ae6ad
Author: John Doe <someaddress@domain.com>
Date:   Mon Jul 1 03:56:53 2013 -0700
    Initial commit
```

From this list, you can choose a commit to rollback to. For the selected commit, you must take the commit hash - the long string of letters and numbers. To demonstrate, the top (or `HEAD`) commit hash in the example output above is `88f41317640a2b62c2c63ca8d755feb9f17cf16e`. You can select this in your terminal and copy it to your computer's clipboard.

With your commit hash selected, you can now rollback your workspace to the previous revision. You can do this with the `git checkout` command. The following example command would rollback to the commit with hash `88f41317640a2b62c2c63ca8d755feb9f17cf16e`.

```
$ git checkout 88f41317640a2b62c2c63ca8d755feb9f17cf16e .
```

Make sure that you run this command from the root of your workspace, and do not forget to include the dot at the end of the command! The dot indicates that you want to apply the changes to the entire workspace directory tree. After this has completed, you should then immediately commit with a message indicating that you performed a rollback. Push your changes and alert your collaborators - perhaps with a pull request. From there, you can start to recover from the mistake by putting your head down and getting on with your project.

Exercises

If you haven't undertaken what we've been discussing in this chapter already, you should go through everything now to ensure your Git repository is ready to go. To try everything out, you can create a new file `README.md` in the root of your `<workspace>` directory. The file will be used by GitHub to provide information on your project's GitHub homepage.

- Create the file, and write some introductory text to your project.
- Add the file to the local index upon completion of writing, and commit your changes.
- Push the new file to the remote repository and observe the changes on the GitHub website.

Once you have completed these basic steps, you can then go back and edit the readme file some more. Add, commit and push - and then try to revert to the initial version to see if it all works as expected.

There's More!

There are other more advanced features of Git that we have not covered in this chapter. Examples include **branching** and **merging**, which are useful for projects with different release versions, for example. There are many fantastic tutorials available online if you are interested in taking you super-awesome version control skills a step further. For more details about such features take a look at this tutorial on getting started with Git, the Git Guide or Learning about Git Branching.

However, if you're only using this chapter as a simple guide to getting to grips with Git, everything that we've covered should be enough. Good luck!

A CSS Crash Course

In Web development, we use *Cascading Style Sheets (CSS)* to describe the presentation of a HTML document (i.e. its look and feel).

Each element within a HTML document can be *styled*. The CSS for a given HTML element describes how it is to be rendered on screen. This is done by ascribing *values* to the different *properties* associated with an element. For example, the `font-size` property could be set to `24pt` to make any text contained within the specified HTML element to appear at 24pt. We could also set the `text-align` property to a value of `right` to make text appear within the HTML element on the right-hand side.

CSS Properties

There are many, many different CSS properties that you can use in your stylesheets. Each provides a different functionality. Check out the W3C website and HTML Dog for lists of available properties. pageresource.com also has a neat list of properties, with descriptions of what each one does. Check out Section css-course-reading-label for a more comprehensive set of links.

CSS works by following a *select and apply pattern* - for a specified element, a set of styling properties are applied. Take a look at the following example in the figure below, where we have some HTML containing `<h1>` tags. In the CSS code example, we specify that all `h1` be styled. We'll come back to selectors later on in this chapter. For now though, you can assume the CSS style defined will be applied to our `<h1>` tags. The style contains four properties:

- `font-size`, setting the size of the font to 16pt;
- `font-style`, which when set to `italic` italicises the contents of all `<h1>` tags within the document;
- `text-align` centres the text of the `<h1>` tags (when set to `center`); and
- `color`, which sets the colour of the text to red via hexadecimal code `#FF0000`.

With all of these properties applied, the resultant page render can be seen in the browser as shown in the figure below.

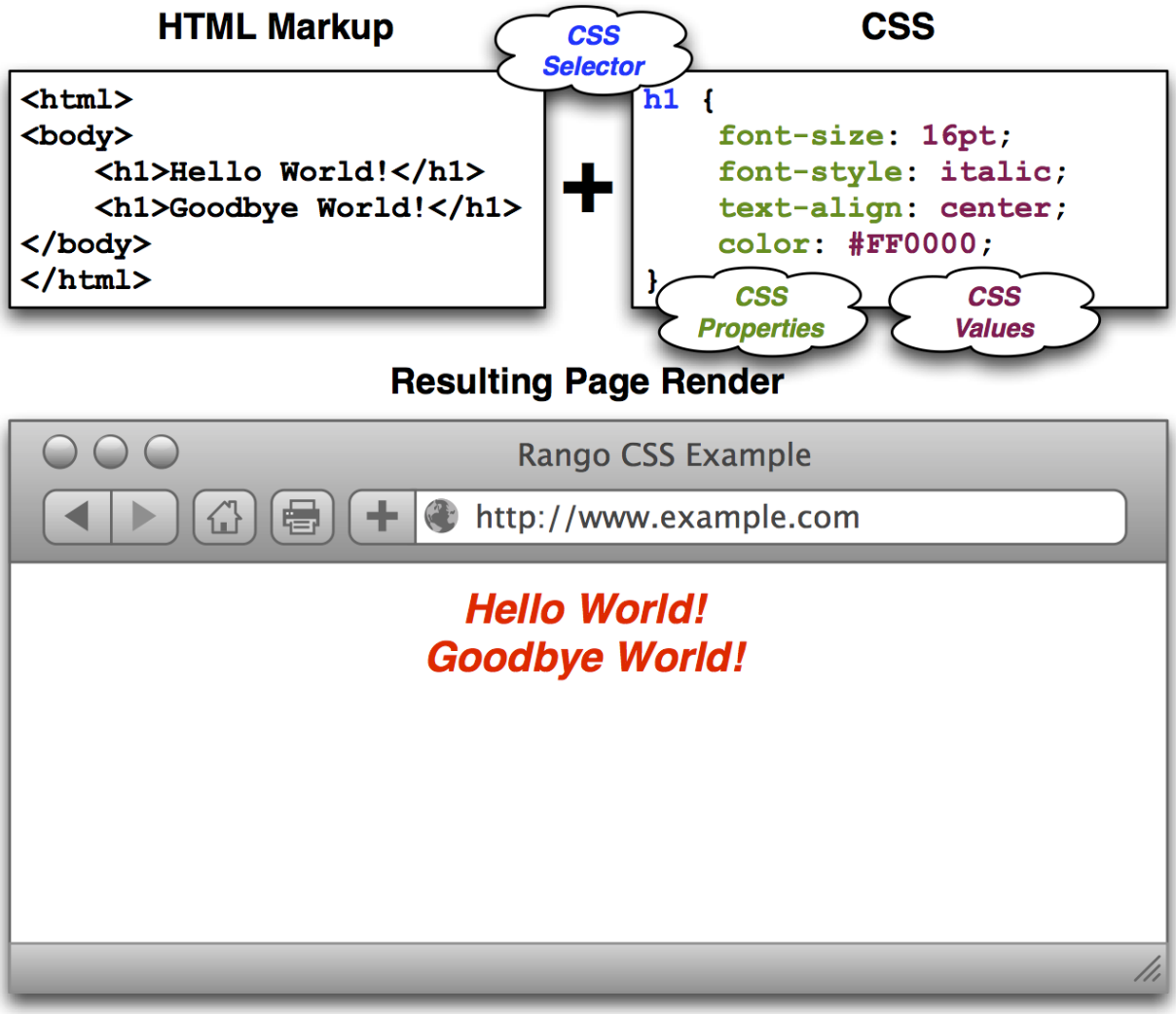


Illustration demonstrating the rendered output of the sample HTML markup and CSS stylesheet shown. Pay particular attention to the CSS example - the colours are used to demonstrate the syntax used to define styles and the property/value pairings associated with them.

What you see is what you *(may or may not)* get

Due to the nature of web development, *what you see isn't necessarily what you'll get.* This is because different browsers have their own way of interpreting web standards and so the pages may be rendered differently. This quirk can unfortunately lead to plenty of frustration, but today's modern browsers (or developers) are much more in agreement as to how different components of a page should be rendered.

Including Stylesheets

Including stylesheets in your webpages is a relatively straightforward process, and involves including a `<link>` tag within your HTML's `<head>`. Check out the minimal HTML markup sample below for the attributes required within a `<link>` tag.

```
<!DOCTYPE html>
<html>
    <head>
        <link rel="stylesheet" type="text/css" href="URL/TO/stylesheet.css" />
        <title>Sample Title</title>
    </head>

    <body>
        <h1>Hello world!</h1>
    </body>
</html>
```

As can be seen from above, there are at minimum three attributes that you must supply to the `<link>` tag:

- `rel`, which allows you to specify the relationship between the HTML document and the resource you're linking to (i.e., a stylesheet);
- `type`, in which you should specify the MIME type for CSS; and
- `href`, the attribute which you should point to the URL of the stylesheet you wish to include.

With this tag added, your stylesheet should be included with your HTML page, and the styles within the stylesheet applied. It should be noted that CSS stylesheets are considered as a form of static media, meaning you should place them within your project's `static` directory.

Inline CSS

You can also add CSS to your HTML documents *inline*, meaning that the CSS is included as part of your HTML page. However, this isn't generally advised because it removes the abstraction between presentational semantics (CSS) and content (HTML).

Basic CSS Selectors

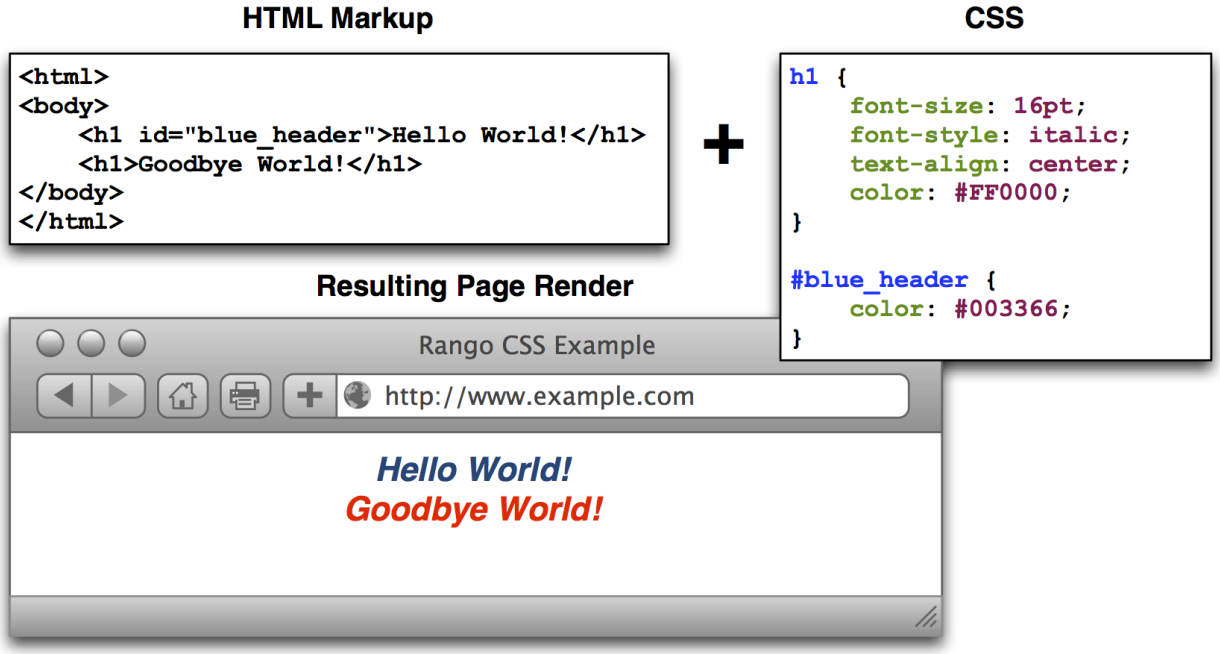
CSS selectors are used to map particular styles to particular HTML elements. In essence, a CSS selector is a *pattern*. Here, we cover three basic forms of CSS selector: *element selectors*, *id selectors* and *class selectors*. Later on in this chapter, we also touch on what are known as *pseudo-selectors*.

Element Selectors

Taking the CSS example from the rendering example shown above, we can see that the selector `h1` matches to any `<h1>` tag. Any selector referencing a tag like this can be called an *element selector*. We can apply element selectors to any HTML element such as `<body>`, `<h1>`, `<h2>`, `<h3>`, `<p>` and `<div>`. These can be all styled in a similar manner. However, using element selectors is pretty crude - styles are applied to *all* instances of a particular tag. We usually want a more fine-grained approach to selecting what elements we style, and this is where *id selectors* and *class selectors* come into play.

ID Selectors

The *id selector* is used to map to a unique element on your webpage. Each element on your webpage can be assigned a unique id via the `id` attribute, and it is this identifier that CSS uses to latch styles onto your element. This type of selector begins with a hash symbol (`#`), followed directly by the identifier of the element you wish to match to. Check out the figure below for an example

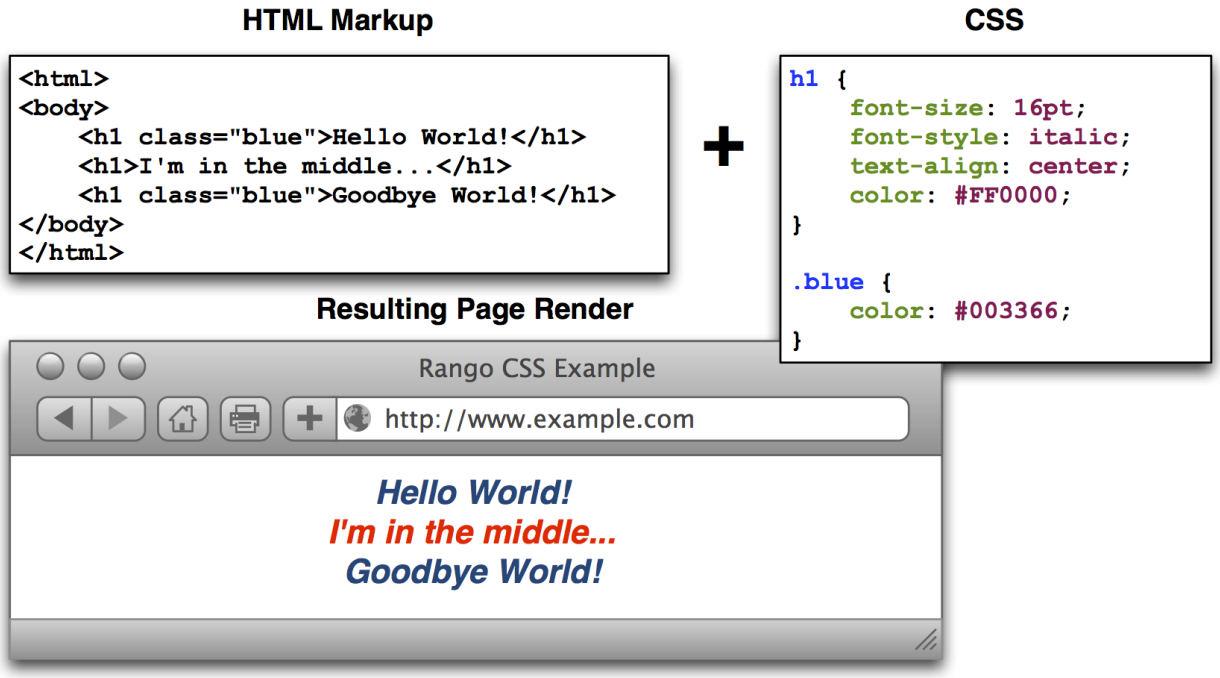


An illustration demonstrating the use of an *id selector* in CSS. Note the blue header has an identifier which matches the CSS attribute `#blue_header`.

Class Selectors

The alternative option is to use *class selectors*. This approach is similar to that of *id selectors*, with the difference that you can legitimately target multiple elements with the same class. If you have a group of HTML elements that you wish to apply the same style to, use a class-based approach. The

selector for using this method is to precede the name of your class with a period (.) before opening up the style with curly braces ({ }). Check out the figure below for an example.



An illustration demonstrating the use of a *class selector* in CSS. The blue headers employ the use of the `.blue` CSS style to override the red text of the `h1` style.

Ensure `id`s are Unique

Try to use id selectors sparingly. Ask yourself: *do I absolutely need to apply an identifier to this element in order to target it?* If you need to apply a given set of styles to more than one element, the answer will always be **no**. In cases like this, you should use a class or element selector.

Fonts

Due to the huge number available, using fonts has historically been a pitfall when it comes to web development. Picture this scenario: a web developer has installed and uses a particular font on his or her webpage. The font is pretty arcane - so the probability of the font being present on other computers is relatively small. A user who visits the developer's webpage subsequently sees the page rendered incorrectly as the font is not present on their system. CSS tackles this particular issue with the `font-family` property.

The value you specify for `font-family` can be a *list* of possible fonts - and the first one your computer or other device has installed is the font that is used to render the webpage. Text within the specified

HTML element subsequently has the selected font applied. The example CSS shown below applies *Arial* if the font exists. If it doesn't, it looks for *Helvetica*. If that font doesn't exist, any available sans-serif font is applied.

```
h1 {
    font-family: 'Arial', 'Helvetica', sans-serif;
}
```

In 1996, Microsoft started the core fonts for the Web initiative with the aim of guaranteeing a particular set of fonts to be present on all computers. Today however, you can use pretty much any font you like - check out Google Fonts for examples of the fonts that you can use and this Web Designer Depot article on how to use such fonts.

Colours and Backgrounds

Colours are important in defining the look and feel of your website. You can change the colour of any element within your webpage, ranging from background colours to borders and text. In this book, we make use of words and *hexadecimal colour codes* to choose the colours we want. As you can see from the list of basic colours shown in the figure below, you can supply either a *hexadecimal* or *RGB (red-green-blue)* value for the colour you want to use. You can also specify words to describe your colours, such as `green`, `yellow` or `blue`.

Pick Colours Sensibly

Take great care when picking colours to use on your webpages. If you select colours that don't contrast well, people simply won't be to read your text! There are many websites available that can help you pick out a good colour scheme - try colorcombos.com for starters.

Applying colours to your elements is a straightforward process. The property that you use depends on the aspect of the element you wish to change! The following subsections explain the relevant properties and how to apply them.

Illustration of some basic colours with their corresponding hexadecimal and RGB values.

There are many different websites that you can use to aid you in picking the right hexadecimal codes to enter into your stylesheets. You aren't simply limited to the nine examples above! Try out html-color-codes.com for a simple grid of colours and their associated six character hexadecimal code. You can also try sites such as color-hex.com which gives you fine grained control over the colours you can choose.

Hexadecimal Colour Codes

For more information on how colours are coded with hexadecimal, check out this thorough tutorial.

Watch your English!

As you may have noticed, CSS uses American/International English to spell words. As such, there are a few words that are spelt slightly differently compared to their British counterparts, like `color` and `center`. If you have grown up in the United Kingdom, double check your spelling and be prepared to spell it the *wrong way!*

Text Colours

To change the colour of text within an element, you must apply the `color` property to the element containing the text you wish to change. The following CSS for example changes all the text within an element using class `red` to...red!

```
.red {
    color: #FF0000;
}
```

You can alter the presentation of a small portion of text within your webpage by wrapping the text within `<span>` tags. Assign a class or unique identifier to the element, and from there you can simply reference the `<span>` tag in your stylesheet while applying the `color` property.

Borders

You can change the colour of an element's *borders*, too. We'll discuss what borders are discussed as part of the CSS box model. For now, we'll show you how to apply colours to them to make everything look pretty.

Border colours can be specified with the `border-color` property. You can supply one colour for all four sides of your border, or specify a different colour for each side. To achieve this, you'll need to supply different colours, each separated by a space.

```
.some-element {
    border-color: #000000 #FF0000 #00FF00
}
```

In the example above, we use multiple colours to specify a different colour for three sides. Starting at the top, we rotate clockwise. Thus, the order of colours for each side would be `top right bottom left`.

Our example applies any element with class `some-element` with a black top border, a red right border and a green bottom border. No left border value is supplied, meaning that the left-hand border is left transparent. To specify a colour for only one side of an element's border, consider using the `border-top-color`, `border-right-color`, `border-bottom-color` and `border-left-color` properties where appropriate.

Background Colours

You can also change the colour of an element's background through use of the CSS `background-color` property. Like the `color` property described above, the `background-color` property can be easily applied by specifying a single colour as its value. Check out the example below which applies a bright green background to the entire webpage. Not very pretty!

```
body {
    background-color: #00FF00;
}
```

Background Images

Of course, a colour isn't the only way to change your backgrounds. You can also apply background images to your elements, too. We can achieve this through the `background-image` property.

```
#some-unique-element {
    background-image: url('../images/filename.png');
    background-color: #000000;
}
```

The example above makes use of `filename.png` as the background image for the element with identifier `some-unique-element`. The path to your image is specified *relative to the path of your CSS stylesheet*. Our example above uses the double dot notation to specify the relative path to the image. *Don't provide an absolute path here; it won't work as you expect!* We also apply a black background colour to fill the gaps left by our background image - it may not fill the entire size of the element.

Background Image Positioning

By default, background images default to the top-left corner of the relevant element and are repeated on both the horizontal and vertical axes. You can customise this functionality by altering how the image is repeated with the `background-image` property. You can also specify where the image is placed by default with the `background-position` property.

Containers, Block-Level and Inline Elements

Throughout the crash course thus far, we've introduced you to the `<span>` element but have neglected to tell you what it is. All will become clear in this section as we explain *inline* and *block-level* elements.

A `<span>` is considered to be a so-called *container element*. Along with a `<div>` tag, these elements are themselves meaningless and are provided only for you to *contain* and *separate* your page's content in a logical manner. For example, you may use a `<div>` to contain markup related to a navigation bar, with another `<div>` to contain markup related to the footer of your webpage. As containers themselves are meaningless, styles are usually applied to help control the presentational semantics of your webpage.

Containers come in two flavours: *block-level elements* and *inline elements*. Check out the figure below for an illustration of the two kinds in action, and read on for a short description of each.

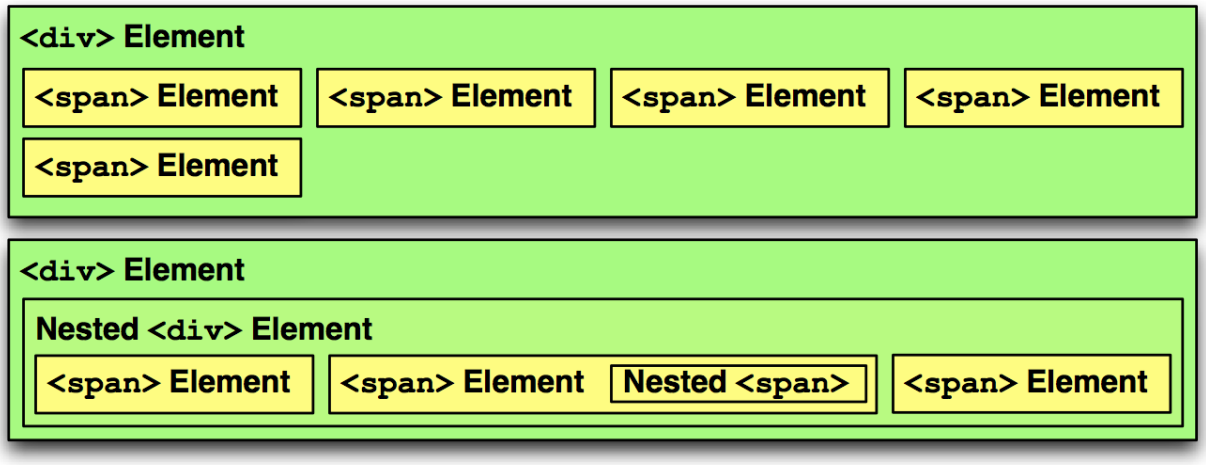


Diagram demonstrating how block-level elements and inline elements are rendered by default. With block-level elements as green, note how a line break is taken between each element. Conversely, inline elements can appear on the same line beside each other. You can also nest block-level and inline elements within each other, but block-level elements cannot be nested within an inline element.

Block-Level Elements

In simple terms, *block-level elements* are by default rectangular in shape and spread across the entire width of the containing element. Block-level elements therefore by default appear underneath each other. The rectangular structure of each block-level element is commonly referred to as the *box model*, which we discuss later on in this chapter. A typical block-level element you will use is the `<div>` tag, short for *division.*

Block-level elements can be nested within other block-level elements to create a hierarchy of elements. You can also nest *inline elements* within block-level elements, but not vice-versa! Read on to find out why.

Inline Elements

An *inline element* does exactly what it says on the tin. These elements appear *inline* to block-level elements on your webpage, and are commonly found to be wrapped around text. You'll find that `<span>` tags are commonly used for this purpose.

This text-wrapping application was explained in the text colours section, where a portion of text could be wrapped in `<span>` tags to change its colour. The corresponding HTML markup would look similar to the example below.

```
<div>
    This is some text wrapped within a block-level element. <span class="red">Th\
is text is wrapped within an inline element!</span> But this text isn't.
</div>
```

Refer back to the nested blocks figure above to refresh your mind about what you can and cannot nest before you move on.

Basic Positioning

An important concept that we have not yet covered in this CSS crash course regards the positioning of elements within your webpage. Most of the time, you'll be satisfied with inline elements appearing alongside each other, and block-level elements appearing underneath each other. These elements are said to be *positioned statically*.

However, there will be scenarios where you require a little bit more control on where everything goes. In this section, we'll briefly cover three important techniques for positioning elements within your webpage: *floats*, *relative positioning* and *absolute positioning*.

Floats

CSS *floats* are one of the most straightforward techniques for positioning elements within your webpage. Using floats allows us to position elements to the left or right of a particular container - or page.

Let's work through an example. Consider the following HTML markup and CSS code.

```
<div class="container">
    <span class="yellow">Span 1</span>
    <span class="red">Span 2</span>
</div>
```

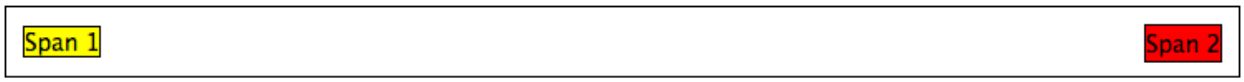
```
.container {
    border: 1px solid black;
}

.yellow {
    background-color: yellow;
    border: 1px solid black;
}

.red {
    background-color: red;
    border: 1px solid black;
}
```

This produces the output shown below.

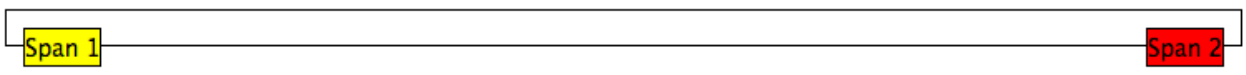
We can see that each element follows its natural flow: the container element with class `container` spans the entire width of its parent container, while each of the `<span>` elements are enclosed inline within the parent. Now suppose that we wish to then move the red element with text `Span 2` to the right of its container. We can achieve this by modifying our CSS `.red` class to look like the following example.

```
.red {
    background-color: red;
    border: 1px solid black;
    float: right;
}
```

By applying the `float: right;` property and value pairing, we should then see something similar to the example shown below.

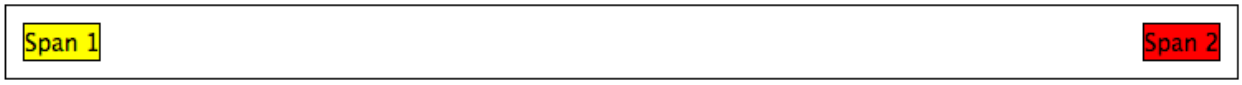


Note how the `.red` element now appears at the right of its parent container, `.container`. We have in effect disturbed the natural flow of our webpage by artificially moving an element! What if we then also applied `float: left;` to the `.yellow` `<span>`?



This would float the `.yellow` element, removing it from the natural flow of the webpage. In effect, it is not sitting on top of the `.container` container. This explains why the parent container does not now fill down with the `<span>` elements like you would expect. You can apply the `overflow: hidden;` property to the parent container as shown below to fix this problem. For more information on how this trick works, have a look at this QuirksMode.org online article.

```
.container {
    border: 1px solid black;
    overflow: hidden;
}
```



Applying `overflow: hidden` ensures that our `.container` pushes down to the appropriate height.

Relative Positioning

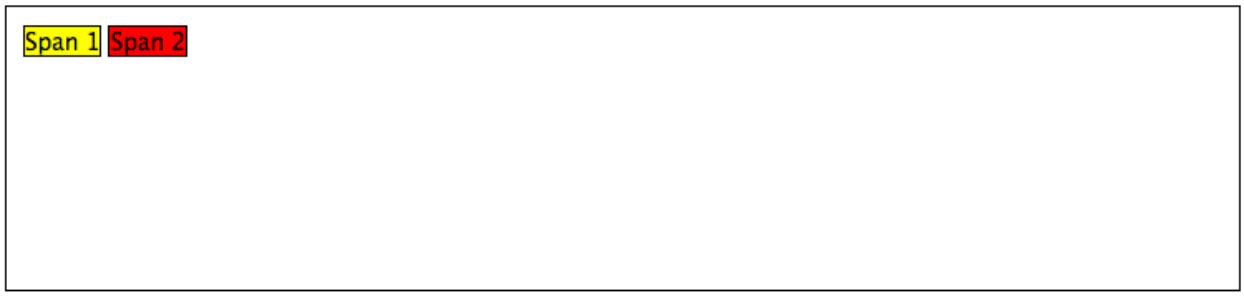
Relative positioning can be used if you require a greater degree of control over where elements are positioned on your webpage. As the name may suggest to you, relative positioning allows you to position an element *relative to where it would otherwise be located.* We make use of relative positioning with the `position: relative;` property and value pairing. However, that's only part of the story.

Let's explain how this works. Consider our previous example where two `<span>` elements are sitting within their container.

```
<div class="container">
    <span class="yellow">Span 1</span>
    <span class="red">Span 2</span>
</div>
```

```
.container {
    border: 1px solid black;
    height: 200px;
}

.yellow {
    background-color: yellow;
    border: 1px solid black;
}

.red {
    background-color: red;
    border: 1px solid black;
}
```

This produces the following result - just as we would expect. Note that we have artificially increased the `height` of our `container` element to 150 pixels. This will allow us more room with which to play with.



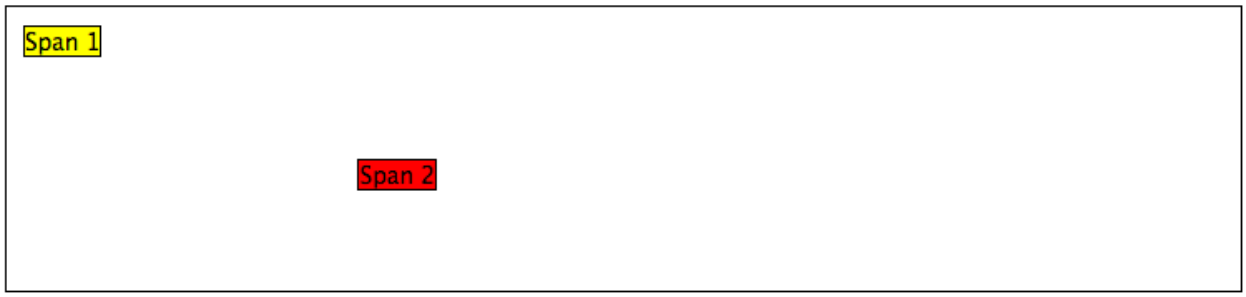
Now let's attempt to position our `.red` `<span>` element relatively. First, we apply the `position: relative` property and value pairing to our `.red` class, like so.

```
.red {
    background-color: red;
    border: 1px solid black;
    position: relative;
}
```

This has no effect on the positioning of our `.red` element. What it does do however is change the positioning of `.red` from `static` to `relative`. This paves the way for us to specify where - from the original position of our element - we now wish the element to be located.

```
.red {
    background-color: red;
    border: 1px solid black;
    position: relative;
    left: 150px;
    top: 80px;
}
```

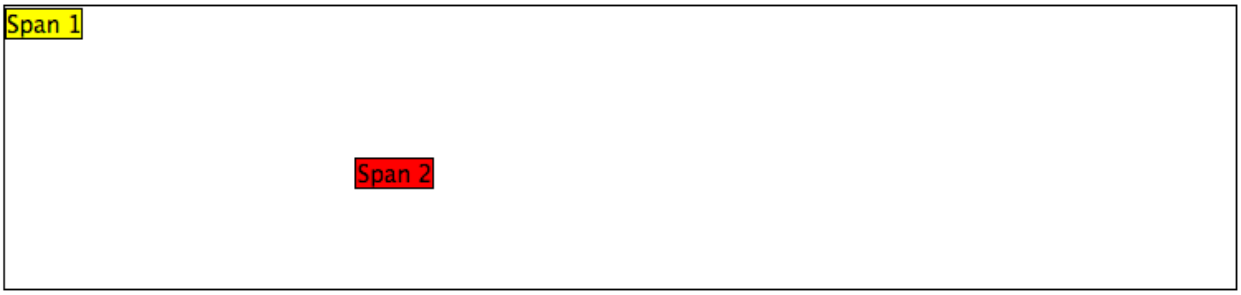
By applying the `left` and `top` properties as shown in the example above, we are wanting the `.red` element to be *pushed* 150 pixels *from the left*. In other words, we move the element 150 pixels to the right. Think about that carefully! The `top` property indicates that the element should be pushed 80 pixels from the *top* of the element. The result of our experimentation can be seen below.



From this behaviour, we can see that the properties `right` and `bottom` *push* elements from the right and bottom respectively. We can test this out by applying the properties to our `.yellow` class as shown below.

```
.yellow {
    background-color: yellow;
    border: 1px solid black;
    float: right;
    position: relative;
    right: 10px;
    bottom: 10px;
}
```

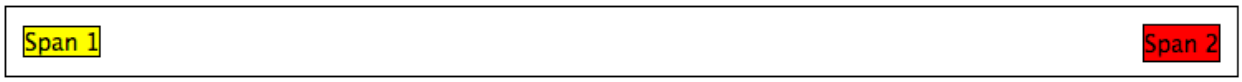
This produces the following output. The `.yellow` container is pushed into the top left-hand corner of our container by pushing up and to the right.



Order Matters

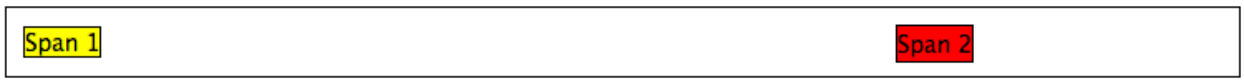
What happens if you apply both a `top` and `bottom` property, or a `left` and `right` property? Interestingly, the *first* property for the relevant axis is applied. For example, if `bottom` is specified before `top`, the `bottom` property is used.

We can even apply relative positioning to elements that are floated. Consider our earlier example where the two `<span>` elements were positioned on either side of the container by floating `.red` to the right.



We can then alter the `.red` class to the following.

```
.red {
    background-color: red;
    border: 1px solid black;
    float: right;
    position: relative;
    right: 100px;
}
```



This means that relative positioning works from the position at which the element would have otherwise been at - regardless of any other position changing properties being applied.

Absolute Positioning

Our final positioning technique is *absolute positioning*. While we still modify the `position` parameter of a style, we use `absolute` as the value instead of `relative`. In contrast to relative

positioning, absolute positioning places an element *relative to its first parent element that has a position value other than static*. This may sound a little bit confusing, but let's go through it step by step to figure out what exactly happens.

First, we can again take our earlier example of the two coloured `<span>` elements within a `<div>` container. The two `<span>` elements are placed side-by-side as they would naturally.

```
<div class="container">
    <span class="yellow">Span 1</span>
    <span class="red">Span 2</span>
</div>
```

```
.container {
    border: 1px solid black;
    height: 70px;
}

.yellow {
    background-color: yellow;
    border: 1px solid black;
}

.red {
    background-color: red;
    border: 1px solid black;
}
```

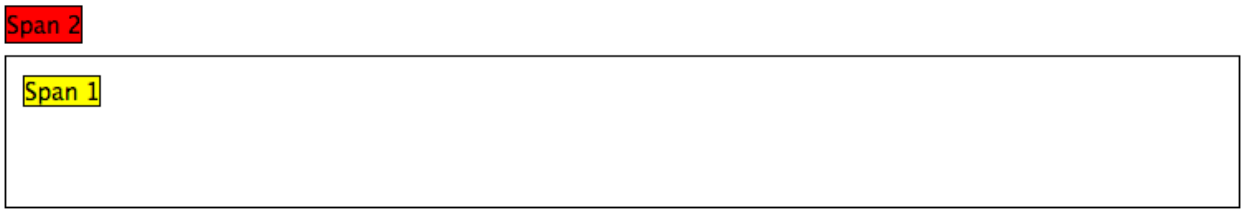
This produces the output shown below. Note that we again set our `.container` height to an artificial value of 70 pixels to give us more room.

We now apply absolute positioning to our `.red` element.

```
.red {
    background-color: red;
    border: 1px solid black;
    position: absolute;
}
```

Like with relative positioning, this has no overall effect on the positioning of our red element in the webpage. We must apply one or more of `top`, `bottom`, `left` or `right` in order for a new position to take effect. As a demonstration, we can apply `top` and `left` properties to our red element like in the example below.

```
.red {
    background-color: red;
    border: 1px solid black;
    position: absolute;
    top: 0;
    left: 0;
}
```

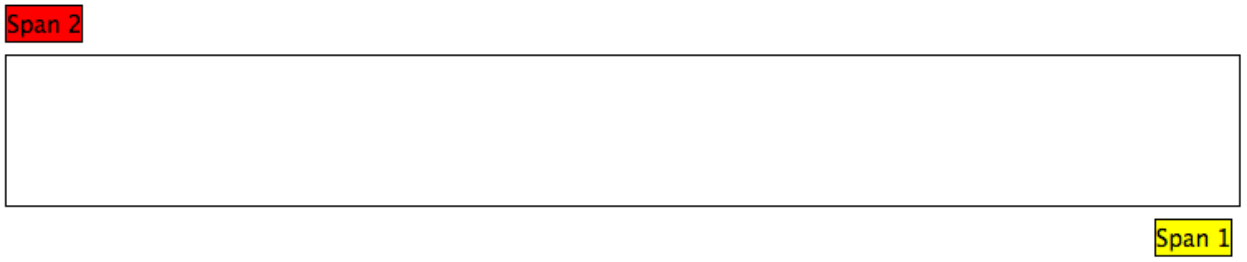


Wow, what happened here? Our red element is now positioned outside of our container! You'll note that if you run this code within your own web browser window, the red element appears in the top left-hand corner of the viewport. This therefore means that our `top`, `bottom`, `left` and `right` properties take on a slightly different meaning when absolute positioning is concerned.

As our container element's position is by default set to `position: static`, the red and yellow elements are moving to the top left and bottom right of our screen respectively. Let's now modify our `.yellow` class to move the yellow `<span>` to 5 pixels from the bottom right-hand corner of our page. The `.yellow` class now looks like the example below.

```
.yellow {
    background-color: yellow;
    border: 1px solid black;
    position: absolute;
    bottom: 5px;
    right: 5px;
}
```

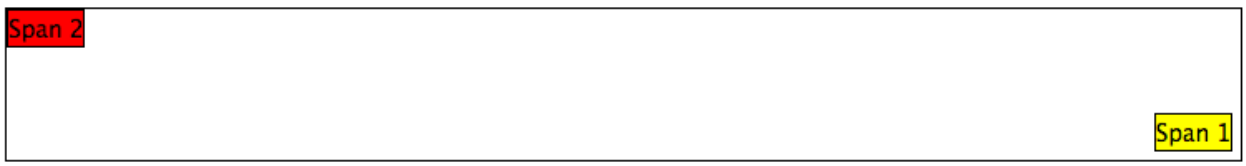
This produces the following result.



But what if we don't want our elements to be positioned absolutely in relation to the entire page? More often than not, we'll be looking to adjusting the positioning of our elements in relation to a container. If we recall our definition for absolute positioning, we will note that absolute positions are calculated *relative to the first parent element that has a position value other than static.* As our container is the only parent for our two `<span>` elements, the container to which the absolutely positioned elements is therefore the `<body>` of our HTML page. We can fix this by adding `position: relative;` to our `.container` class, just like in the example below.

```
.container {
    border: 1px solid black;
    height: 70px;
    position: relative;
}
```

This produces the following result. `.container` becomes the first parent element with a position value of anything other than `relative`, meaning our `<span>` elements latch on!



Our elements are now absolutely positioned in relation to `.container`. Great! Now, let's adjust the positioning values of our two `<span>` elements to move them around.

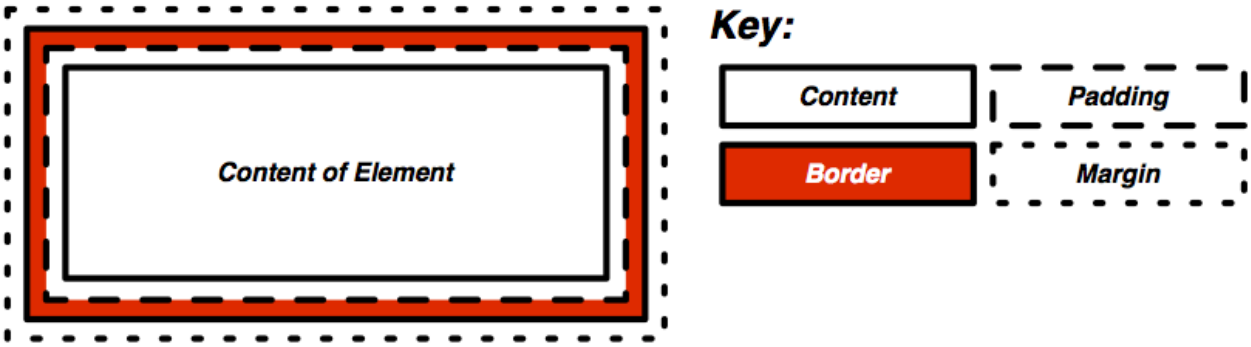
```
.yellow {
    background-color: yellow;
    border: 1px solid black;
    position: absolute;
    top: 20px;
    right: 100px;
}

.red {
    background-color: red;
    border: 1px solid black;
    position: absolute;
    float: right;
    bottom: 50px;
    left: 40px;
}
```

Note that we also apply `float: right;` to our `.red` element. This is to demonstrate that unlike relative positioning, absolute positioning *ignores any other positioning properties applied to an element*. `top: 10px` for example will always ensure that an element appears 10 pixels down from its parent (set with `position: relative;`), regardless of whether the element has been floated or not.

The Box Model

When using CSS, you're never too far away from using *padding*, *borders* and *margins*. These properties are some of the most fundamental styling techniques which you can apply to the elements within your webpages. They are incredibly important and are all related to what we call the *CSS box model*.

Each element that you create on a webpage can be considered as a box. The CSS box model is defined by the W3C as a formal means of describing the elements or boxes that you create, and how they are rendered in your web browser's viewport. Each element or box consists of *four separate areas*, all of which are illustrated in the figure below. The areas - listed from inside to outside - are the *content area*, the *padding area*, the *border area* and the *margin area*.



An illustration demonstrating the CSS box model, complete with key showing the four areas of the model.

For each element within a webpage, you can create a margin, apply some padding or a border with the respective properties `margin`, `padding` and `border`. Margins clear a transparent area around the border of your element; meaning margins are incredibly useful for creating a gap between elements. In contrast, padding creates a gap between the content of an element and its border. This therefore gives the impression that the element appears wider. If you supply a background colour for an element, the background colour is extended with the element's padding. Finally, borders are what you might expect them to be - they provide a border around your element's content and padding.

For more information on the CSS box model, check out addedbytes excellent explanation of the model. Why not even order a t-shirt with the box model on it?

Watch out for the `width`!

As you may gather from the box model illustration, the width of an element isn't defined simply by the value you enter as the element's `width`. Rather, you should always consider the width of the border and padding on both sides of your element. This can be represented mathematically as:

```
total_width = content_width + left padding + right padding + left border +
left margin + right margin
```

Don't forget this. You'll save yourself a lot of trouble if you don't!

Styling Lists

Lists are everywhere. Whether you're reading a list of learning outcomes for a course or a reading a list of times for the train, you know what a list looks like and appreciate its simplicity. If you have a list of items on a webpage, why not use a HTML list? Using lists within your webpages - according to Brainstorm and Raves - promotes good HTML document structure, allowing text-based browsers, screen readers and other browsers that do not support CSS to render your page in a sensible manner.

Lists however don't look particularly appealing to end-users. Take the following HTML list that we'll be styling as we go along trying out different things.

```
<ul class="sample-list">
    <li>Django</li>
    <li>How to Tango with Django</li>
    <li>Two Scoops of Django</li>
</ul>
```

Rendered without styling, the list looks pretty boring.

- Django
- How to Tango with Django
- Two Scoops of Django

Let's make some modifications. First, let's get rid of the ugly bullet points. With our `<ul>` element already (and conveniently) set with class `sample-list`, we can create the following style.

```
.sample-list {
    list-style-type: none;
}
```

This produces the following result. Note the lack of bullet points!

Django
How to Tango with Django
Two Scoops of Django

Let's now change the orientation of our list. We can do this by altering the `display` property of each of our list's elements (`<li>`). The following style maps to this for us.

```
.sample-list li {
    display: inline;
}
```

When applied, our list elements now appear on a single line, just like in the example below.

Django How to Tango with Django Two Scoops of Django

While we may have the correct orientation, our list now looks awful. Where does one element start and the other end? It's a complete mess! Let's adjust our list element style and add some contrast and padding to make things look nicer.

```
.example-list li {
    display: inline;
    background-color: #333333;
    color: #FFFFFF;
    padding: 10px;
}
```

When applied, our list looks so much better - and quite professional, too!

From the example, it is hopefully clear that lists can be easily customised to suit the requirements of your webpages. For more information and inspiration on how to style lists, you can check out some of the selected links below.

- Have a look at this excellent tutorial on styling lists on A List Apart.
- Check out this advanced tutorial from Web Designer Wall that uses graphics to make awesome looking lists. In the tutorial, the author uses Photoshop - you could try using a simpler graphics package if you don't feel confident with Photoshop.
- This site provides some great inspiration and tips on how you can style lists.

The possibilities of styling lists are endless! You could say it's a never-ending list...

Styling Links

CSS provides you with the ability to easily style hyperlinks in any way you wish. You can change their colour, their font or any other aspect that you wish - and you can even change how they look when you hover over them!

Hyperlinks are represented within a HTML page through the `<a>` tag, which is short for *anchor*. We can apply styling to all hyperlinks within your webpage as shown in following example.

```
a {
    color: red;
    text-decoration: none;
}
```

Every hyperlink's text colour is changed to red, with the default underline of the text removed. If we then want to change the `color` and `text-decoration` properties again when a user hovers over a link, we can create another style using the so-called pseudo-selector `:hover`. Our two styles now look like the example below.

```
a {
    color: red;
    text-decoration: none;
}

a:hover {
    color: blue;
    text-decoration: underline;
}
```

This produces links as shown below. Notice the change in colour of the second link - it is being hovered over.

You may not however wish for the same link styles across the entire webpage. For example, your navigation bar may have a dark background while the rest of your page has a light background. This would necessitate having different link stylings for the two areas of your webpage. The example below demonstrates how you can apply different link styles by using a slightly more complex CSS style selector.

```
#dark {
    background-color: black;
}

#dark a {
    color: white;
    text-decoration: underline;
}

#dark a:hover {
    color: aqua;
```

```
}

.light {
    background-color: white;
}

.light a {
    color: black;
    text-decoration: none;
}

.light a:hover {
    color: olive;
    text-decoration: underline;
}
```

We can then construct some simple markup to demonstrate these classes.

```
<div id="dark">
    <a href="http://www.google.co.uk/">Google Search</a>
</div>

<div class="light">
    <a href="http://www.bing.co.uk/">Bing Search</a>
</div>
```

The resultant output looks similar to the example shown below. Code up the example above, and hover over the links in your browser to see the text colours change!

With a small amount of CSS, you can make some big changes in the way your webpages appear to users.

The Cascade

It's worth pointing out where the *Cascading* in *Cascading Style Sheets* comes into play. Looking back at the CSS rendering example way back at the start of this chapter, you will notice that the red

text shown is **bold**, yet no such property is defined in our `h1` style. This is a perfect example of what we mean by *cascading styles*. Most HTML elements have associated with them a *default style* which web browsers apply. For `<h1>` elements, the W3C website provides a typical style that is applied. If you check the typical style, you'll notice that it contains a `font-weight: bold;` property and value pairing, explaining where the **bold** text comes from. As we define a further style for `<h1>` elements, typical property/value pairings *cascade* down into our style. If we define a new value for an existing property/value pairing (such as we do for `font-size`), we *override* the existing value. This process can be repeated many times - and the property/value pairings at the end of the process are applied to the relevant element. Check out the figure below for a graphical representation of the cascading process.

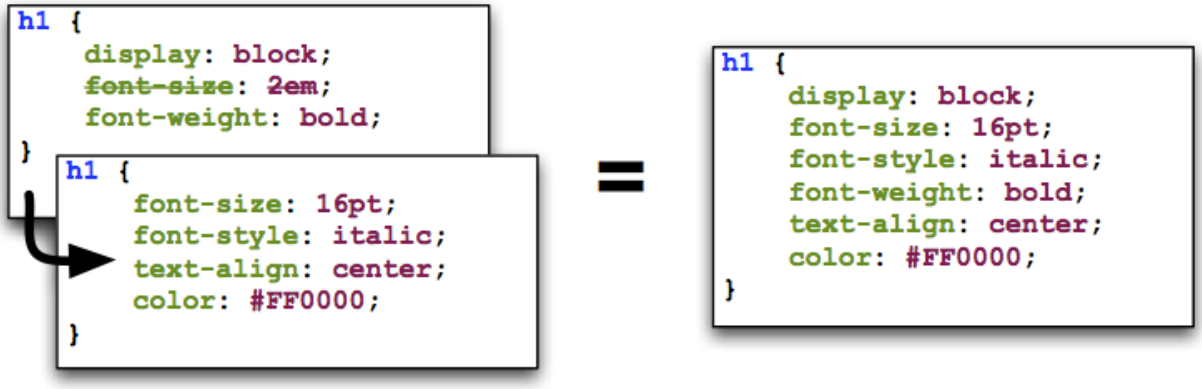


Illustration demonstrating the *cascading* in *Cascading Style Sheets* at work. Take note of the `font-size` property in our `h1` style - it is overridden from the default value. The cascading styles produce the resultant style, shown on the right of the illustration.

Additional Reading

What we've discussed in this section is by no means a definitive guide to CSS. There are 300-page books devoted to CSS alone! What we have provided you with here is a very brief introduction showing you the very basics of what CSS is and how you can use it.

As you develop your web applications, you'll undoubtedly run into issues and frustrating problems with styling web content. This is part of the learning experience, and you still have a bit to learn. We strongly recommend that you invest some time trying out several online tutorials about CSS - there isn't really any need to buy a book (unless you want to).

- The *W3C* provides a neat tutorial on CSS, taking you by the hand and guiding you through the different stages required. They also introduce you to several new HTML elements along the way, and show you how to style them accordingly.
- W3Schools also provides some cool CSS tutorials. Instead of guiding you through the process of creating a webpage with CSS, *W3Schools* has a series of mini-tutorials and code examples

to show you to achieve a particular feature, such as setting a background image. We highly recommend that you have a look here.
- html.net has a series of lessons on CSS which you can work through. Like W3Schools, the tutorials on *html.net* are split into different parts, allowing you to jump into a particular part you may be stuck with.

This list is by no means exhaustive, and a quick web search will indeed yield much more about CSS for you to chew on. Just remember: CSS can be tricky to learn, and there may be times where you feel you want to throw your computer through the window. We say this is pretty normal - but take a break if you get to that stage. We'll be tackling some more advanced CSS stuff as we progress through the tutorial in the next few sections.

CSS And Browser Compatibility

With an increasing array of devices equipped with more and more powerful processors, we can make our web-based content do more. To keep up, CSS has constantly evolved to provide new and intuitive ways to express the presentational semantics of our SGML-based markup. To this end, support for relatively new CSS properties may be limited on several browsers, which can be a source of frustration. The only way to reliably ensure that your website works across a wide range of different browsers and platforms is to test, test and test some more!