1. New in TwD 1.7

Warning

Please note that this version of the book is still in the draft phase. Though it should all pretty much work (however some links and screenshots need to be updated). Please report any bugs, problems, etc, or submit change requests via GitHub: https://github.com/leifos/tango_with_django_book/tree/master/17

In this version of the online tutorial / book, we have updated and added a number of things:

- The code has been ported to work with Django 1.7
    - The database interaction has been updated from `syncdb` to `migratesql` and `migrate`
    - Rendering responses has been updated from `render_to_response` to `render`. So now there is no need to request the context in every view.
    - The `url` template tag is now being used in templates, which provides a relative reference to urls rather than an absolute reference..
    - Loading static files in the templates is now done with {% load staticfiles %}
    - Using `slugify` to create well formed URL strings
- A new chapter on authentication has been added
    - Where the login and registration is done with Django-Registration-Redux (see Chapter 12)
- The Bootstrap chapter has been updated to use Bootstrap 3.2.0 (see Chapter 13)
    - Also, includes some notes on how to use Django-Bootstrap-Toolkit
- A new chapter has been added on using template tags (see Chapter 14)
- Add a chapter on using JQuery with Django (see Chapter 18)

- The chapter on testing has been expanded - but is still a work in progress (see Chapter 20)
    - Includes how to use the package `coverage` to measure test coverage (see http://nedbatchelder.com/code/coverage/)

Navigation

How To Tango With Django 1.7 » How To Tango With Django 1.7 next

2. Overview

The aim of this book is to provide you with a practical guide to web development using Django 1.7. The book is designed primarily for students, providing a walkthrough of the steps involved in getting your first web applications up and running, as well as deploying them to a web server.

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.

2.1. 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, and not have to sit there scratching your head.

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 showing you how to put together a web application with all the bells and whistle from the onset, the book shortens the learning curve.

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 think it is worth mentioning, please get in touch with us so that we can improve the book - we already have a number of contributors and we will happily acknowledge your contribution!

2.2. What You will Learn

In this book, we will be taking an exampled-based approach (or inquiry-based learning). The book will show you how to design a web application called Rango (see the Design Brief in Section 2.4.1 below). Along the way, we'll show you how to perform the following tasks.

- Setup a development environment - including how to use the terminal, the Pip installer, how to work with Git, etc.
- 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 functionality provided by Django.
- Create forms that can utilise your database models to create dynamically generated webpages.
- Use the User Authentication services provided by Django.
- Incorporate external services into the application.
- Include Cascading Styling Sheets (CSS) and JavaScript within a web application.
- Design and apply CSS to improve the look and feel the web application.

- 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. Finally, all the code is available from GitHub at https://github.com/leifos/tango_with_django.

To see a fully-functional version of the application, you can also visit the How to Tango with Django website at http://www.tangowithdjango.com/rango/.

2.3. Technologies and Services

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

- Python, http://www.python.org
- Pip, http://www.pip-installer.org
- Django, https://www.djangoproject.com
- Git, http://git-scm.com
- GitHub, https://github.com
- HTML, http://www.w3.org/html/
- CSS, http://www.w3.org/Style/CSS/
- Javascript
- JQuery, http://jquery.com
- Twitter Bootstrap, http://getbootstrap.com/
- Bing Search API via Azure Datamarket, http://datamarket.azure.com
- PythonAnywhere, https://www.pythonanywhere.com

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 Microsoft Azure) and deploy your application quickly and easily (with PythonAnywhere).

2.4. Rango: Initial Design and Specification

As previously mentioned, 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.

2.4.1. 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" (see https://www.vocabulary.com/dictionary/es/rango).

- For the main page of the site, they would like visitors to be able to see:
    - the 5 most viewed pages;
    - the five most rango'ed categories; and
    - some way for visitors to browse or search through categories.
- When a user views a category page, they would like it 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 Bing's 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. And so, visitors to the site should be able to register for an account.

At first glance, the application to develop seems reasonably straightforward. In essence, it is just a list of categories which link to pages, right? 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.

2.5. 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 Page and the Category Page.
- The URL Mappings.
- An Entity-Relationship diagram to describe the data model that we'll be implementing.

2.6. 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.

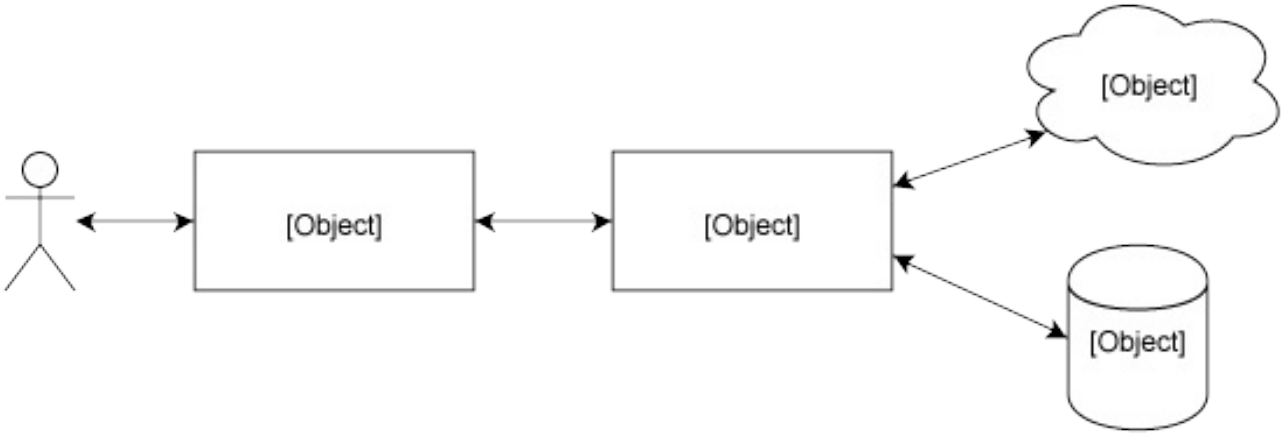


Figure 1: Overview of the system architecture for Rango. Note the inclusion of an external Search Application Programming Interface (API).

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 (i.e Chrome, Firefox, Safari, etc.) 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 Bing Search API.

For the most part, this book will focus on developing the middleware, though it should be quite evident from Figure 1 that we will have to interface with all the other components.

2.7. 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 available. For Rango, we'd like to make the index page of the site look like the screen shot shown in Figure 2. Our category page is shown in Figure 3.

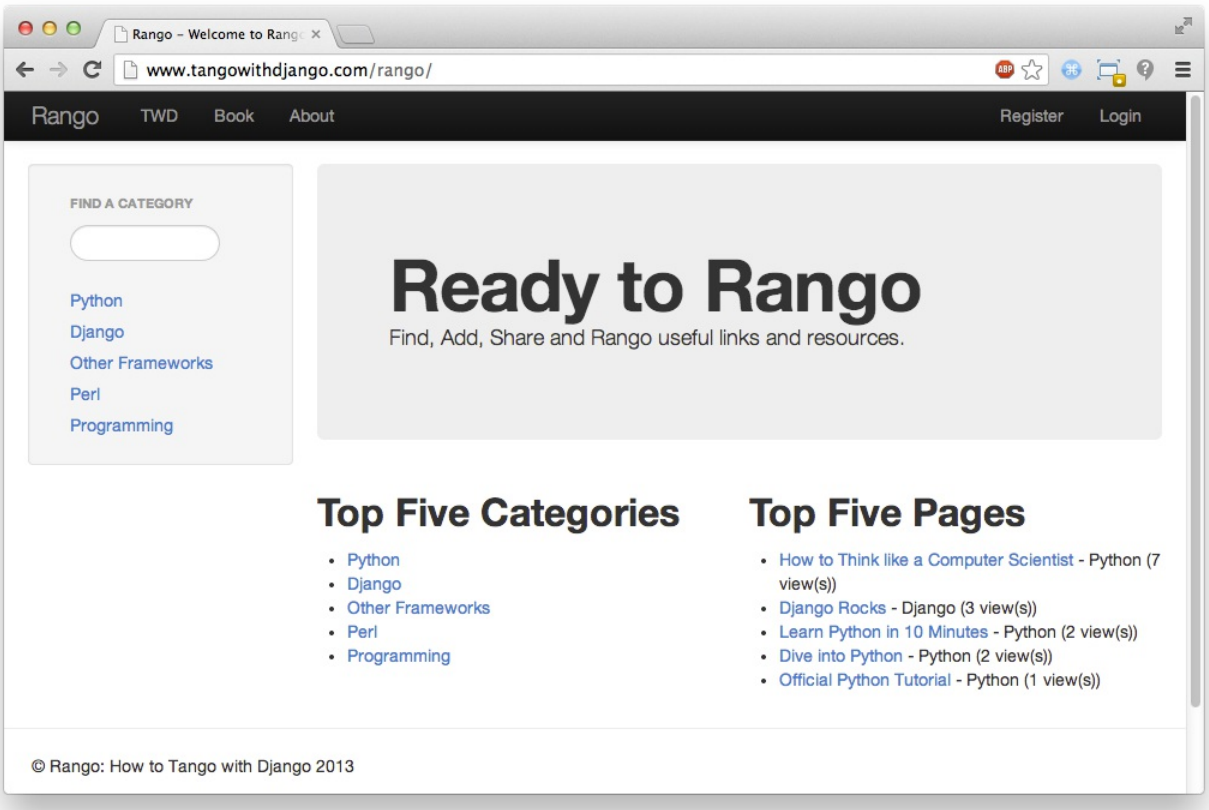


Figure 2: The index page with the categories bar on the left, also showing the top five pages and top five categories.

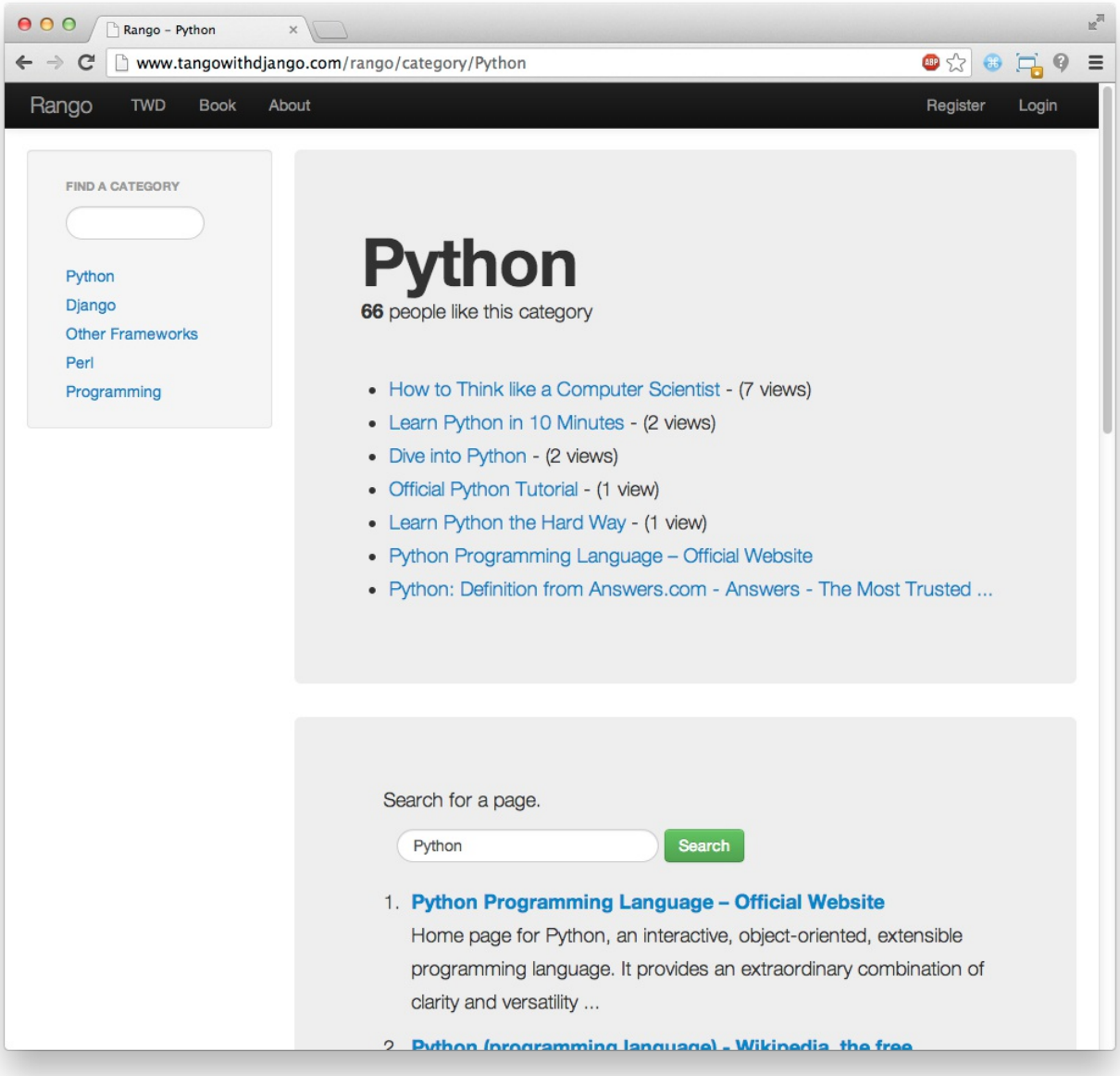


Figure 3: The category page showing the pages in the category (along with the number of views). Below, a search for Python has been conducted, with the results shown underneath.

2.8. 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 of these pages we will need to describe in some fashion the 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.

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

- `/rango/about/` will point to an about page view.
- `/rango/category/<category_name>/` will point to the category page view for `<category_name>`, where the category might be:
    - games;
    - python recipes; or
    - code and compilers.
- `/rango/etc/`, where `etc` could be replaced with a URL for any later function we wish to implement.

As we build our application, we will probably need to create other URL mappings. However, the ones listed above will get us started. We will also at some point have to transform category names in a valid URL string, as well as handle scenarios where the supplied category name does not exist.

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.

2.9. 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.

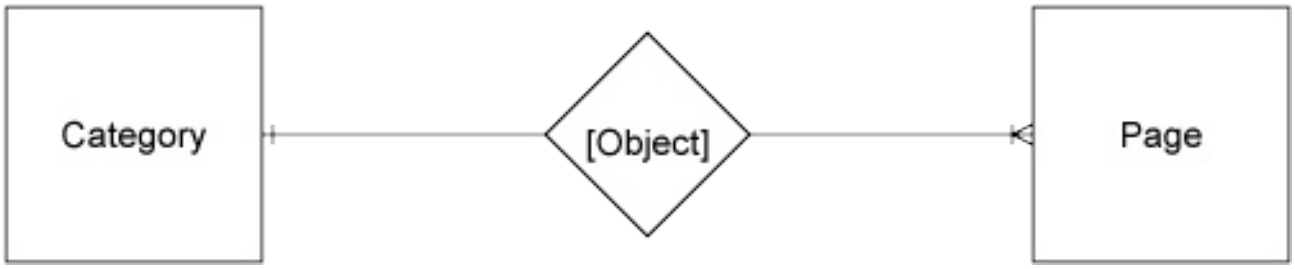


Figure 4: The Entity Relationship Diagram of Rango's two main entities.

Note that this specification is vague. One page may be in one or many categories. So we could model the relationship as a many-to-many. 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 may not be ideal.

It's good practice to note down any working assumptions like this. You never know when they may come back to haunt you! 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.

The resulting tables are shown below, where `Str` denotes a `string` or `char` field, `Int` denotes an `integer` field, `URL` denotes a URL field and `FK` denotes a Foreign Key.

Category Table

Field	Type
name	Str
views	Int
likes	Int

Page Table

Field	Type
category	FK
title	Str
url	URL
views	Int

We will also have a `User` table - which we have not shown here, but shall introduce later in the book. In the following chapters will we see how to instantiate these data models in Django and how to use Django's Object Relational Mapping to connect to the database.

2.10. 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 web sites. It's a good idea to become familiar and comfortable with producing such specifications and designs.

If you already have Python 2.7 and Django 1.7 installed, you have a good working knowledge of the command line, configured your paths, then you can skip straight to

the Django Basics chapter. Otherwise, get started with Chapter 3.

2.10.1. Working with The Official Django Tutorials

We suggest undertaking the Official Django Tutorials as part of the exercises associated with each of this book's chapters. You can find a mapping between the tutorial exercises and book chapters below. The tutorial exercises will help reinforce your understanding of the Django framework, and also help you build up your skills.

Tango with Django	Django Tutorial
Chapter 3	Part 1 - Models
Chapter 5	Part 2 - The Admin Interface
Chapter 6	Part 3 - URLs and Views
Chapter 7	Part 4 - Templates
Chapter 18	Part 5 - Testing
Chapter 11	Part 6 - CSS

Navigation

How To Tango With Django 1.7 »

previous next

3. Getting Ready to Tango

Let's get set up! To tango with Django, you'll need to ensure that you have everything you need installed on your computer and that you have a sound understanding of your development environment. This chapter walks you through what you need and what you need to know.

For this tutorial, you'll require the following key pieces of software.

- Python version 2.7.5+
- Django version 1.7

As Django is a web application framework written in the Python programming language, you will be required to have a working knowledge of 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.

- **A quick tutorial** - Learn Python in 10 Minutes by Stavros, http://www.korokithakis.net/tutorials/python/.
- **The Official Python Tutorial** at http://docs.python.org/2/tutorial/.
- **A brilliant book**: Think Python: How to Think like a Computer Scientist by Allen B. Downey, available online at http://www.greenteapress.com/thinkpython/.
- **An amazing online course**: Learn to Program, by Jennifer Campbell and Paul Gries at https://www.coursera.org/course/programming1.

3.1. Using the Terminal

In order to set up your environment learning how to use the Command Line Interpreter (CLI) provided by your Operating System is really important. Through the course of this tutorial, you will be interacting with the CLI routinely. If you are already familiar with using the command line interface you can skip directly to Installing the Software section.

UNIX-based operating systems all use a similar-looking terminal. Descendants, derivatives and clones of UNIX include Apple's OS X and the many available Linux

distributions available today. All of these operating systems contain a core set of commands which help you navigate through your filesystem and launch programs, all without the need of any graphical interface. This section provides the key commands you should familiarise yourself with.

Note

This tutorial is focused towards users of UNIX-based or UNIX-derived operating systems. While Python and Django can run in a Windows-based environment, many of the commands that we use in this book are for UNIX-based terminals. These commands can however be replicated in Windows by using the graphical user interface, using the relevant command in a Windows Command Prompt, or using Windows PowerShell which provides an CLI like similar to a UNIX terminal.

Upon launching a new terminal instance, you'll typically be presented with something like:

```
sibu:~ leif$
```

This is called 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 `leif` and computer name of `sibu`);
- your current working directory (the tilde, or `~`); and
- the privilege of your user account (the dollar sign, or `$`).

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.

Open up a terminal window and see what your prompt looks like.

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. For example, check the example terminal interactions below.

```
Last login: Mon Sep 23 11:35:44 on ttys003
sibu:~ leif$ pwd
/Users/leif
sibu:~ leif$
```

You can see that the present working directory in this example is: `/Users/leif`.

You'll also note that the prompt indicates that my present working directory is ~. This is because the tilde (`~`) 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 (`/`).

If you are not in your home directory you can change directory (`cd`) to your home directory by issuing the following command.

```
$ cd ~
```

Let's create a directory called `code`. To do thus, use the make directory command (`mkdir`), as shown below.

```
$ mkdir code
```

To move to the newly-created `code` directory, enter `cd code`. If you now check your current working directory, you'll notice that you will be in `~/code/`. This may also be reflected by your prompt. Note in the example below that the current working directory is printed after the `sibu` computer name.

Note

Whenever we refer to `<workspace>`, we'll be referring to your `code` directory.

```
sibu:~ leif$ mkdir code
sibu:~ leif$ cd code
sibu:code leif$
```

```
sibu:code leif$ pwd
/Users/leif/code
```

To list the files that are in a directory, you can issue the command `ls`. You can also see hidden files or directories - if you have any - you can issue the command `ls -a`, where `a` stands for all. If you `cd` back to your home directory (`cd ~`) and then issue `ls`, you'll see that you have something called `code` in your home directory.

To find out a bit more about what is in your directory, issue `ls -l`. This will provide a more detailed listing of your files and whether it is a directory or not (denoted by a `d` at the start of the line).

```
sibu:~ leif$ cd ~
sibu:~ leif$ ls -l

drwxr-xr-x   36 leif  staff    1224 23 Sep 10:42 code
```

The output also contains information on the permissions associated to the directory, who created it (`leif`), the group (`staff`), the size, the date/time the file was modified at, and, of course, the name.

You may also find it useful to be able to edit files within your terminal. There are many editors which you can use - some of which may already be installed on your computer. The nano editor for example is a straightforward editor - unlike vi which can take some time to learn. Below are a list of commonly-used UNIX commands that you will find useful.

3.1.1. Core Commands

All UNIX-based operating systems come with a series of built-in commands - with most focusing exclusively on file management. The commands you will use most frequently are listed below, each with a short explanation on what they do and how to use them.

- `pwd`: Prints your current working directory to the terminal. The full path of where you are presently is displayed.
- `ls`: Prints a list of files in the current working directory to the terminal. By

default, you do not see the sizes of files - this can be achieved by appending `-lh` to `ls`, giving the command `ls -lh`.

- `cd`: In conjunction with a path, allows you to change your current working directory. For example, the command `cd /home/leif/` changes the current working directory to `/home/leif/`. 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.

Note

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 reference guide.

3.2. Installing the Software

Now that you have a decent understanding of how to interact with the terminal, you can begin to install the software required for this tutorial.

3.2.1. Installing Python

So, how do you go about installing Python 2.7.5 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.

Warning

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!

3.2.1.1. Apple 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/.

Warning

Ensure that you download the `.dmg` file that is relevant to your particular 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?

3.2.1.2. 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. Hurrah!

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.

Note

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`.

3.2.1.3. 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 folder `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. Windows XP, has slightly different instructions, and so do Windows 8 installationsthis.

3.2.2. 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/python
```

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` folder 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-

>>> 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 in Section to bring up the environment variables settings dialog. Add a `PYTHONPATH` variable with the value being set to your `site-packages` folder, which is typically `C:\Python27\Lib\site-packages\`.

3.2.3. 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 will be using 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!

Note

With Windows-based computers, follow the same basic process. You won't need to enter the `sudo` command, however.

3.2.4. Installing Django

Once the Python package manager Pip is successfully installed on your computer, installing Django is easy. Open a Command Prompt or terminal window, and issue the following command.

```
$ pip install -U django==1.7
```

If you are using a UNIX-based operating system and receive complaints about insufficient permissions, you will need to run the command with elevated privileges using the `sudo` command. If this is the case, you must then run the following command instead.

```
$ sudo pip install -U django==1.7
```

The package manager will download Django and install it in the correct location for you. Upon completion, Django should be successfully installed. Note, if you didn't include the `==1.7`, then a different version of Django may be installed.

3.2.5. Installing the Python Imaging Library

During the course of building Rango, we will be uploading and handling images. This means we will need support from the Pillow (Python Imaging Library). To install this package issue the following command.

```
$ pip install pillow
```

Again, use `sudo` if required.

3.2.6. Installing Other Python Packages

It is worth noting that additional Python packages can be easily downloaded using the same manner. The Python Package Index provides a listing of all the packages available through Pip.

To get a list of the packages installed, you can run the following command.

```
$ pip list
```

3.2.7. Sharing your Package List

You can also get a list of the packages installed in a format that can be shared with other developers. To do this issue the following command.

```
$ pip freeze > requirements.txt
```

If you examine `requirements.txt` using either the command `more`, `less` or `cat`, you will see the same information but in a slightly different format. The `requirements.txt` can then use to install the same setup by issuing the following command. This is incredibly useful for setting up your environment on another computer, for example.

```
$ pip install -r requirements.txt
```

3.3. Integrated Development Environment

While not absolutely necessary, a good Python-based integrated development environment (IDE) can be very helpful to you during the development process. Several exist, with perhaps JetBrains' *PyCharm* 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. PyCharm and PyDev both support Django integration out of the box - though you will have to point the IDE to the version of Python that you are using.

3.3.1. 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 dont have to run it each and every time you want to use virtualenvironments.

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 interferring with your standard or other environements. Try ``pip list" to see you dont have Django or Pillow installed in your virtual environment. You can now install them with pip so that they exist in your virtual environment.

Later on when we go to deploy the application, we will go through a similar process see Chapter Deploying your Application and set up a virtual environment on PythonAnywhere.

3.3.2. 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 going through this right now so that we can get stuck into developing an application in Django. We have however provided a crash course on GIT. We highly recommend that you set up a GIT repository for your own projects. Doing so could save you from disaster.

3.4. Exercises

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

- Install Python 2.7.5+ 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 Environemnt (like PyCharm)

- 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.

Navigation

4. Django Basics

Let's get started with Django! In this chapter, we'll be giving you an overview of the how to get started with Django. 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-power webpage up and running!

4.1. Testing your Setup

Let's start by checking that your Python and Django installations are installed correctly, and are at the correct version for this tutorial. To do this, open a new terminal instance and issue the following command.

```
$ python --version
2.7.5
```

This starts your Python interpreter and executes the code within the string provided as part of the `-c` switch. You should see the version of your Python installation printed as the output to the process. If the version displayed is anything but `2.7.5`, you will need to go back to Section 3.2 and verify you have completed all the relevant steps for your operating system.

After verifying your Python installation, check your Django installation by issuing the following command.

```
$ python -c "import django; print(django.get_version())"
1.7
```

The command again executes the code within the string provided as part of the `-c` switch. After importing Django, you should see `1.7` printed underneath. If you see a different set of numbers or are greeted with a Python `ImportError`, go back to Section 3.2 or consult the Django Documentation on Installing Django for more information. If you find that you have got a different version of Django, it is possible that you will come across problems at some point. It's definitely worth making sure you have Django 1.7 installed.

4.2. Creating your Django Project

To create a new Django Project, go to your `code` directory (i.e. your `<workspace>` directory), and issue the following command:

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

Note

On Windows you may have to use the full path to the django-admin.py script. i.e. `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 the project configuration directory. Within this directory, you will find four Python scripts. We will discuss this 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.

Note

The project configuration directory has been created with new Django projects since version 1.4. Having two directories with the same name may seem quite a bit odd, but the change was made to separate out project-related components from its individual applications.

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, as 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 to test your work and run database commands. You'll be using this script a lot throughout the development cycle.

Note

See the Django documentation for more details about the Admin and Manage scripts.

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

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

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

```
$ python manage.py runserver

System check identified no issues (0 silenced).

You have unapplied migrations; your app may not work properly until th
Run 'python manage.py migrate' to apply them.

October 01, 2014 - 19:49:05
Django version 1.7c2, using settings 'tango_with_django_project.settin
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

```
$ python manage.py migrate

Operations to perform:
```

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

#TODO(leifos): add description of migrate command: from django tutorial: The migrate command looks at the INSTALLED_APPS setting and creates any necessary database tables according to the database settings in your mysite/settings.py file and the database migrations shipped with the app (we'll cover those later). You'll see a message for each migration it applies. If you're interested, run the command-line client for your database and type dt (PostgreSQL), SHOW TABLES; (MySQL), or .schema (SQLite) to display the tables Django created.

Now open up your favourite web browser and enter the URL http://127.0.0.1:8000/ [1]. You should see a webpage similar to the one shown in Figure 1.

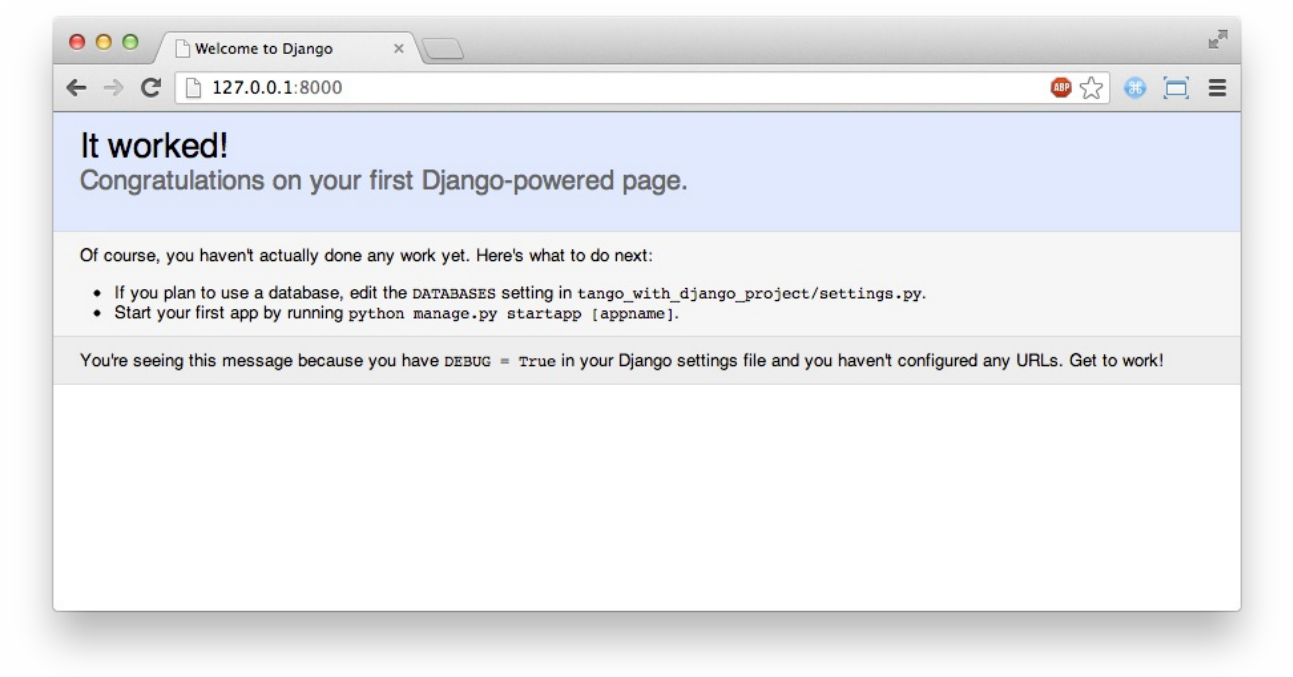


Figure 1: A screenshot of the initial Django page you will see when running the development server for the first time.

You can stop the development server at anytime by pushing `CTRL + C` in your terminal 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.

When setting ports, it is unlikely that you will be able to use TCP port 80 as this is 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, sometimes it is nice to be able to demo your application on a computer of a colleague. 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 which 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.

Note

The `django-admin.py` and `manage.py` scripts provides a lot of useful, time-saving functionality for you. `django-admin.py` allows you to start new projects and apps, along with other commands. Within your project directory, `manage.py` allows you to perform administrative tasks within the scope of your project only. Simply execute the relevant script name without any arguments to see what you can do with each. The official Django documentation provides a detailed list and explanation of each possible command you can supply for both scripts.

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

4.3. Creating a Django Application

A Django project is a collection of configurations and applications 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 small

series of applications, the idea is that you can theoretically drop an existing application into a different Django project and have it working with minimal effort. Why reinvent the wheel if it's already there? [2]

A Django application exists to perform a particular task. You need to create specific applications that are responsible for providing your site with particular kinds of functionality. For example, we could imagine that a project might consist of several applications 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 use them with to dispatch different content. There are many Django applications you can download and use in your projects. Since we are getting started, we'll kick off by walking through how to create your own application.

To start, create a new application called Rango. 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 another five Python scripts:

- another `__init__.py`, serving the exact same purpose as discussed previously;
- models.py, a place to store your application'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 application's code; and
- views.py, where you can store a series of functions that take a clients's requests and return responses.
- admin.py, where you can register your models so that you can benefit from some Django machinery which creates an admin interface for you (see #TODO(leifos):add link to admin chapter)

`views.py` and `models.py` are the two files you will use for any given application, 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 application'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` application 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 application by running the development server again. If you can start the server without errors, your application was picked up and you will be ready to proceed to the next step.

4.4. Creating a View

With our Rango application created, let's now create a simple view. For our first view, let's just send some simple 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` application 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 world!")
```

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 Resources Locator (URL) to the view.

4.5. Mapping URLs

Within the `rango` application directory, we now need to create a new file called `urls.py`. The contents of the file will allow you to map URLs for your application (e.g. `http://www.tangowithdjango.com/rango/`) to specific views. Check out the simple `urls.py` file below.

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

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

This code imports the relevant Django machinery that we use to create URL mappings. Importing the `views` module from `rango` also provides us with access to our simple view implemented previously, allowing us to reference the view in the

URL mapping we will create.

To create our mappings, we use a tuple. For Django to pick your mappings up, this tuple must be called `urlpatterns`. The `urlpatterns` tuple contains a series of calls to the `django.conf.urls.url()` function, with each call handling a unique mapping. In the code example above, we only use `url()` once, so we have therefore defined only one URL mapping. The first parameter we provide to the `django.conf.urls.url()` function is the regular expression `^$`, which matches to an empty string. Any URL supplied by the user that matches this pattern means that the view `views.index()` would be invoked by Django. The view would be passed a `HttpRequest` object as a parameter, containing information about the user's request to the server. We also make use of the optional parameter to the `url()` function, `name`, using the string `'index'` as the associated value.

Note

You might be thinking that matching a blank URL is pretty pointless - what use would it serve? When the URL pattern matching takes place, only a portion of the original URL string is considered. This is because our Django project will first process the original URL string (i.e. `http://www.tangowithdjango.com/rango/`). Once this has been processed, it is removed, with the remained being passed for pattern matching. In this instance, there would be nothing left - so an empty string would match!

Note

The `name` parameter is optional to the `django.conf.urls.url()` function. This is provided by Django to allow you to distinguish one mapping from another. It is entirely plausible that two separate URL mappings expressions could end calling the same view. `name` allows you to differentiate between them - something which is useful for reverse URL matching. Check out the Official Django documentation on this topic for more information.

You may have seen that within your project configuration directory a `urls.py` file already exists. Why make another? Technically, you can put all the URLs for your project's applications within this file. However, this is considered bad practice as it increases coupling on your individual applications. A separate `urls.py` file for each application allows you to set URLs for individual applications. With minimal coupling, you can then join them up to your project's master `urls.py` file later.

This means we need to configure the `urls.py` of our project `tango_with_django_project` and connect up our main project with our Rango application.

How do we do this? It's quite simple. 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` tuple as shown in the example below.

```
urlpatterns = patterns('',
    # Examples:
    # url(r'^$', 'tango_with_django_project_17.views.home', name='home
    # url(r'^blog/', include('blog.urls')),

    url(r'^admin/', include(admin.site.urls)),
    url(r'^rango/', include('rango.urls')), # ADD THIS NEW TUPLE!
)
```

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` (which we have already configured). This is done with the help of the `include()` function from within `django.conf.urls`. Think of this as a chain that processors the URL string - as illustrated in Figure 2. 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 and empty string to be passed on to the application rango. Rango now tries to match the empty string, which it does, and this then dispatches the `index()` view that we created.

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 hello world!`. It should look just like the screenshot shown in Figure 3.

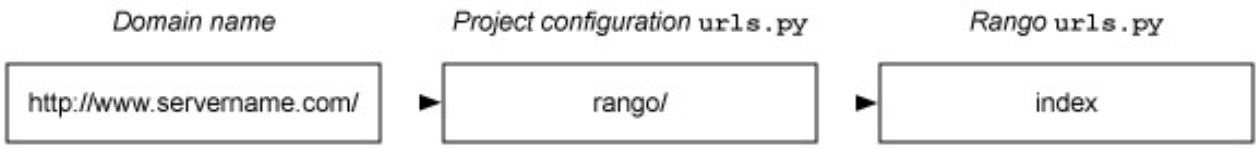


Figure 2: An illustration of a URL, showing how the different parts of the URL are the responsibility of different `url.py` files.

Figure 3: A screenshot of Google Chrome displaying our first Django-powered webpage. Hello, Rango!

Within each application, you will create a number of URL to view mappings. This initial mapping is quite simple. As we progress, we will create more sophisticated mappings that using allow the URLs to be parameterised.

It's important to have a good understanding of how URLs are handled in Django. If you are still bit confused or would like to know more check out the official Django documentation on URLs for further details and further examples.

Note

The 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.

4.6. 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.

4.6.1. 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.

4.6.2. Creating a new Django application

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

4.7. Exercises

Congratulations! You have got Rango up and running. This 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. Now try the following exercises to reinforce what you've learnt.

- Revise the procedure and make sure you follow how the URLs are mapped to views.
- Now create a new view called `about` which returns the following: `Rango says here is the about page.`
- Now map the this view to `/rango/about/`. For this step, you'll only need to edit the `urls.py` of the rango application.
- 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.
- If you haven't done so already, it is a good point to go off an complete part one of the official Django Tutorial.

4.7.1. Hints

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

- Your `index` view should be updated to include a link to the `about` view. Keep it simple for now - something like `Rango says: Hello world! <br/> <a href='/rango/about'>About</a>` will suffice. We'll be going back later to improve the presentation of these pages.
- The regular expression to match `about/` is `r'^about/'` - this will be handy when thinking about your URL pattern.
- The HTML to link back to the index page is `<a href="/rango/">Index</a>`. The link uses the same structure as the link to the `about` page shown above.

Footnotes

[1] This assumes that you are using the IP address 127.0.0.1 and port 8000 when running your Django development web server. If you do not explicitly provide a port to run the development server on, Django defaults to port 8000 for you.

[2] There are many applications available out there that you can use in your project. Take a look at PyPI and Django Packages to search for reusable apps which you can drop into your projects.

Navigation

How To Tango With Django 1.7 »

previous next

5. Templates and Static Media

In this chapter, we'll be extending your knowledge of Django by introducing you to the template engine as well as how to serve static media within your web pages.

5.1. Using Templates

Up to this point, you have plugged a few things together to create a Django-powered webpage. This is coupled a view, which is in turn coupled with a series of URL mappings. Here we will delve into how to combine templates into the mix.

Well-designed websites use a lot of repetition in their structure or layout. 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 from presentational concerns. In this chapter, you'll create a basic template which will be used to create a HTML page. This template will then be dispatched via a Django view. In Chapter 7, we will take this a step further by using templates in conjunction with models to dispatch dynamically generated data.

5.1.1. Configuring the Templates Directory

To get templates up and running, you will need to setup a directory 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`. Within the new templates directory, create another directory called `rango`. So the directory `<workspace>/tango_with_django_project/templates/rango/` will be the location in which we will be storing templates associated with our `rango` application.

To tell your Django project where the templates will be housed, open your project's `settings.py` file. Find the tuple `TEMPLATE_DIRS` and add in the path to your newly created `templates` directory, so it looks like the following example.

```
TEMPLATE_DIRS = ['<workspace>/tango_with_django_project/']
```

Note that you are required to use absolute paths to locate the `templates` directory. If you are part of a team or working on different computers, this may become a problem in the future. You'll have different usernames, meaning different paths to your `<workspace>` directory. The hard-coded path you entered above would not be the same on different computers. Of course, you could add in the template directory for each different setup, but that would be a pretty nasty way to tackle the problem. So, what can we do?

Warning

The road to hell is paved with hard-coded paths. Hard-coding paths is considered to be a software engineering anti-pattern, and will make your project less portable.

5.1.2. Dynamic Paths

The solution to the problem of hard-coding paths is to make use of built-in Python functions to work out the path of our `templates` directory automatically for us. This way, an absolute path can be obtained regardless of where you place your Django project's code on your filesystem. This in turn means that your project's code becomes more portable.

In Django 1.7 the `settings.py` file, now contains a variable called, `BASE_DIR`. This stores the path to the directory in which your project's `settings.py` module will be contained. This is obtained by using the special Python `__file__` attribute, which is set to the absolute path of your settings module. The `__file__` gives theabsolute path to the settings file, then the call to `os.path.dirname()` provides the reference to the absolute path of the directory. 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:

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

Let's make use of it now. Create a new variable in `settings.py` called `TEMPLATE_PATH` and store the path to the `templates` directory you created earlier. Using the `os.path.join()` function, your code should look like the following example.

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

Here we make use of the `os.path.join()` to mash together the `BASE_DIR` variable and `'templates'`, which would for example yield `<workspace>/tango_with_django_project/templates/`. We can then replace the hard-coded path we put in the `TEMPLATE_DIRS` with `TEMPLATE_PATH`, just like in the example below.

```
TEMPLATE_DIRS = [
    # Put strings here, like "/home/html/django_templates" or "C:/www/
    # Always use forward slashes, even on Windows.
    # Don't forget to use absolute paths, not relative paths.
    TEMPLATE_PATH,
]
```

We can keep the `TEMPLATE_PATH` variable at the top of our `settings.py` module to make it easy to access should it ever need to be changed. This is why we created an additional variable to store the template path.

Warning

When joining or concatenating system paths together, using `os.path.join()` is the preferred approach. Using this function ensures that the correct slashes are used depending on your operating system. On a POSIX-compatible operating system, 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.

5.1.3. Adding a Template

With your template directory and path 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>
        hello world! <strong>{{ boldmessage }}</strong><br />
        <a href="/rango/about/">About</a><br />
    </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, and we will be able to set a value for this variable to be displayed within the rendered output. We'll get to that in a moment.

To use this template, we need to re-configure the `index()` view that we created earlier. Instead of dispatching a simple message, we will change it to dispatch our template.

In `rango/views.py`, make sure the following import statement is at the top of the file.

```
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 con
    # Note the key boldmessage is the same as {{ boldmessage }} in the
    context_dict = {'boldmessage': "I am bold font from the context"}
```

```
    # 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_dict)
```

First we construct a dictionary of key-values 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 file name, and the context dictionary. The `render()` function will take this data and mash it together with the template to produce a complete HTML page. This is then returned and dispatched to the user's web browser.

When a template file is loaded with the Django templating system, a template context is created. In simple terms, a template context is essentially a Python dictionary that maps template variable names with Python variables. In the template we created earlier, we included a template variable name called `boldmessage`. In our `index(request)` view example, the string `I am bold font from the context` is mapped to template variable `boldmessage`. The string `I am bold font from the context` therefore replaces any instance of `{{ boldmessage }}` within the template.

Now that you have updated the view to employ the use of your template, run the Django development server and visit http://127.0.0.1:8000/rango/. You should see your template rendered in all its glory, just like the example shown in Figure 1.

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. Ensure that all the changes required have been 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_PATH`.

This example demonstrates how to use templates within your views. However, we have only touched upon some of the functionality provided by Django regarding templates. We will use templates in more sophisticated ways as we progress through this tutorial. In the meantime, you can find out more about templates from the

official Django documentation.

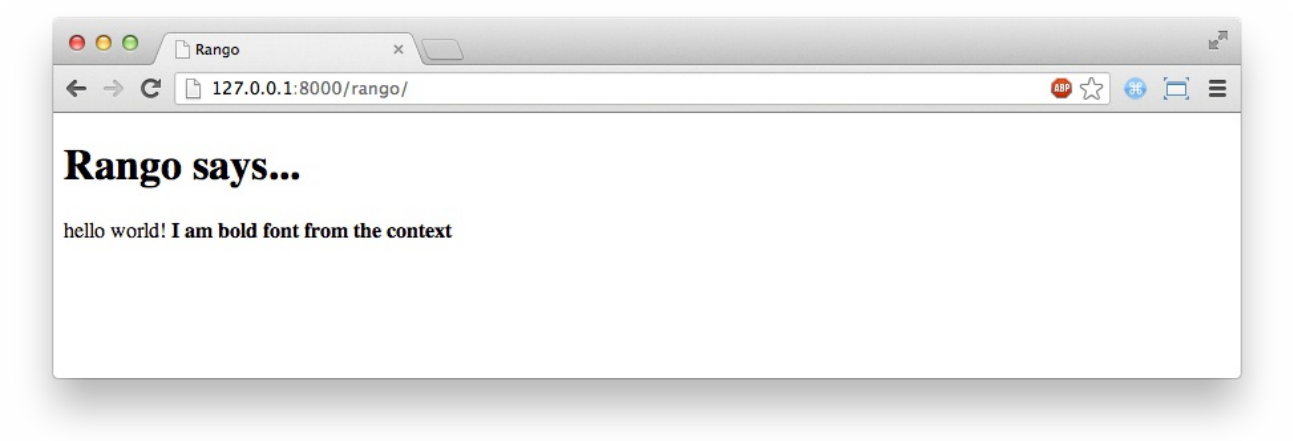


Figure 1: A screenshot of Google Chrome rendering the template used with this tutorial.

5.2. Serving Static Media

Admittedly, the Rango website is pretty plain as we have not included any styling or imagery. Cascading Style Sheets (CSS), JavaScript and images are essentially static media files which we can include in our webpages to add style and introduce dynamic behaviour. These files are served in a slightly different way from webpages. This is because they aren't generated on the fly like our HTML pages. This section shows you how to setup your Django project to serve static media to the client. We'll also modify our template to include some example static media.

5.2.1. Configuring the Static Media Directory

To get static media up and running, 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`

Now place an image within the `static/images` directory. As shown in Figure 2, we chose a picture of the chameleon, Rango - a fitting mascot, if ever there was one.

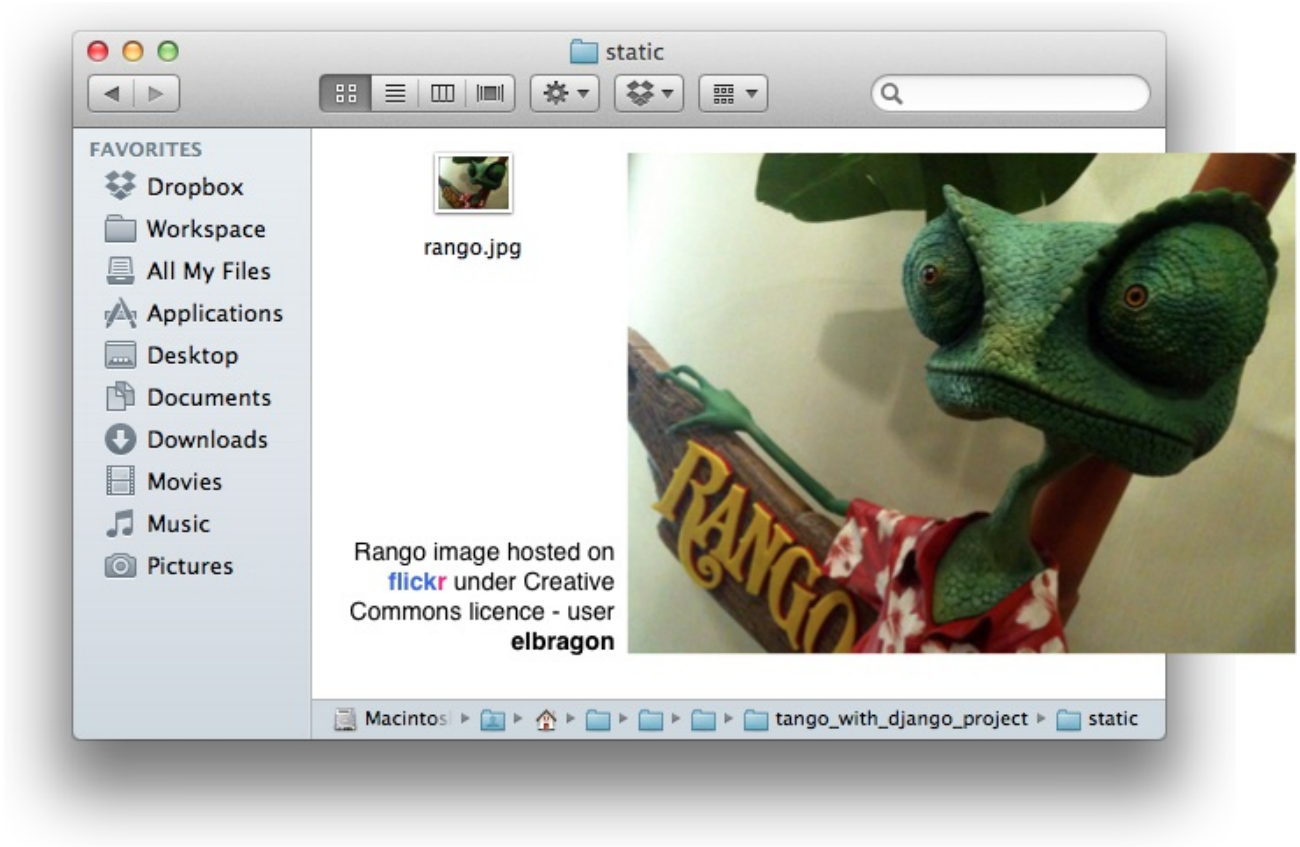


Figure 2: Rango the chameleon within our static media directory.

With our `static` directory created, we need to tell Django about it, just like we did with our `templates` directory earlier. In `settings.py` file, we need to update two variables: `STATIC_URL` and the `STATICFILES_DIRS` tuple. First, create a variable to store the path to the static directory (`STATIC_PATH`) as follows.

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

STATIC_URL = '/static/' # You may find this is already defined as such

STATICFILES_DIRS = (
    STATIC_PATH,
)
```

You've typed in some code, but what does it represent? The first variable `STATIC_URL` defines the base URL with which your Django applications will find static media files when the server is running. For example, when running the Django development server with `STATIC_URL` set to `/static/` like in the code example above, static media will be available at `http://127.0.0.1:8000/static/`. The official documentation on serving up static media warns that it is vitally important to

make sure that those slashes are there. Not configuring this problem can lead to a world of pain.

While `STATIC_URL` defines the URL to access media via the web server, `STATICFILES_DIRS` allows you to specify the location of the newly created `static` directory on your local disk. Just like the `TEMPLATE_DIRS` tuple, `STATICFILES_DIRS` requires an absolute path to the `static` directory. Here, we re-used the `BASE_DIR` defined in Section 5.1 to create the `STATIC_PATH`.

With those two settings updated, run your Django project's development server once more. If we want to view our image of Rango, visit the URL `http://127.0.0.1:8000/static/images/rango.jpg`. If it doesn't appear, you will want to check to see if everything has been correctly spelt and that you saved your `settings.py` file, and restart the development server. If it does appear, try putting in additional file types into the `static` directory and request them via your browser.

Caution

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 - or live - environment. The official Django documentation on Deployment provides further information about deploying static files in a production environment.

5.3. Static Media Files and Templates

Now that you have your Django project set up to handle static media, you can now access such media within your templates.

To demonstrate how to include static media, open up `index.html` 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>
        hello world! <strong>{{ boldmessage }}</strong><br />
        <a href="/rango/about/">About</a><br />
        <img src="{% static "images/rango.jpg" %}" alt="Picture of Ran
    </body>

</html>
```

First, we need to inform Django's template system that we will be using static media with the `{% load static %}` tag. This allows us to call the `static` template tag as done in `{% static "rango.jpg" %}`. As you can see, Django template tags are denoted by curly brackets `{ }`. In this example, the `static` tag will combine the `STATIC_URL` with `"rango.jpg"` so that the rendered HTML looks like the following.

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

If for some reason the image cannot be loaded, it is always nice to specify an alternative text tagline. This is what the `alt` attribute provides - the text here is used in the event the image fails to load.

With these minor changes in place, kick off the Django development server once more and visit `http://127.0.0.1:8000/rango`. Hopefully, you will see web page something like the one shown in Figure 3.

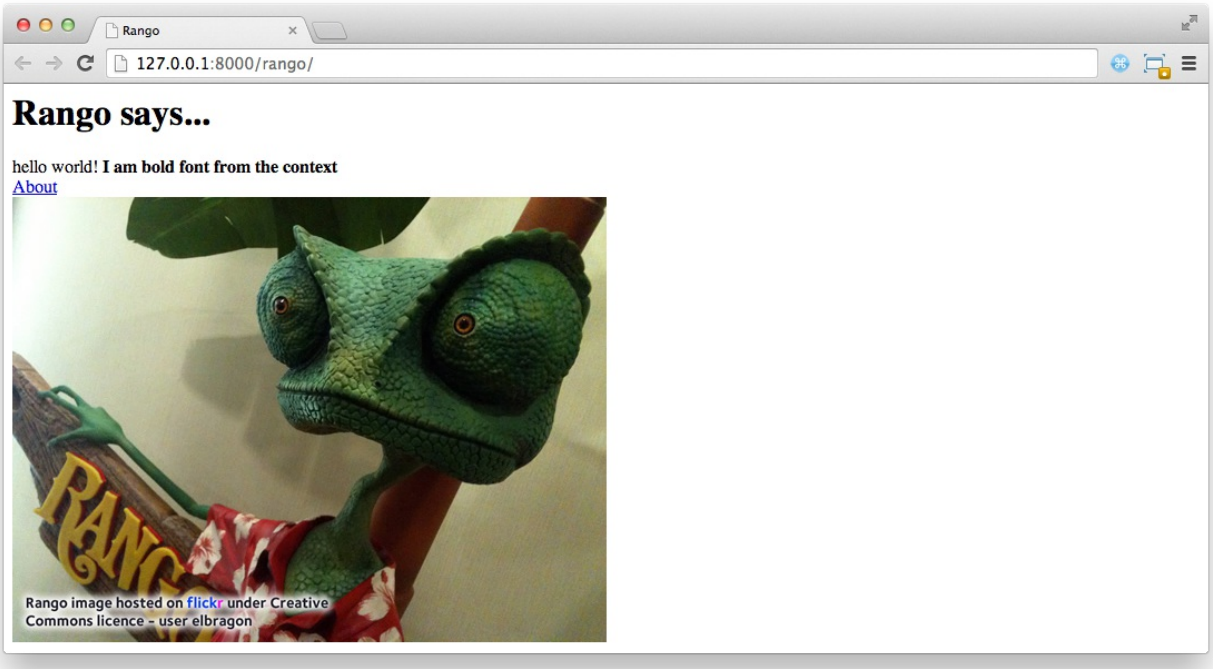


Figure 3: Our first Rango template, complete with a picture of Rango the chameleon.

The `{% static %}` function call should be used whenever you wish to reference static media within a template. The code example below demonstrates how you could include JavaScript, CSS and images into your templates - all with the correct HTML markup.

```
<!DOCTYPE html>

{% load static %}

<html>

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

    <body>
        <h1>Including Static Media</h1>
        <img src="{% static "images/rango.jpg" %}" alt="Picture of Ran
    </body>

</html>
```

Static files you reference will obviously need to be present within your `static` directory. If the file is not there or you have referenced it incorrectly, the console output provide by Django's lightweight development server will flag up any errors. Try referencing a non-existent file and see what happens.

For further information about including static media you can read through the official Django documentation on working with static files in templates.

Caution

Care should be taken in your templates to ensure that any document type declaration (e.g. `<!DOCTYPE html>`) you use in your webpages appears in the rendered output on the first line. This is why we put the Django template command `{% load static %}` on a line underneath the document type declaration, rather than at the very top. It is a requirement of HTML/XHTML variations that the document type declaration be declared on the very first line. Django commands placed before will obviously be removed in the final rendered output, but they may leave behind residual whitespace which means your output will fail validation on the W3C markup validation service.

#TODO(leifos): Note that this not the best practice when you go to deployment, and that they should see: https://docs.djangoproject.com/en/1.7/howto/static-files/deployment/ and that the following solution works when `DEBUG=True`

#TODO(leifos): the DEBUG variable in settings.py, lets you control the output when an error occurs, and is used for debugging. When the application is deployed it is not secure to leave DEBUG equal to True. When you set DEBUG to be False, then you will need to set the ALLOWED_HOSTS variable in settings.py, when running on your local machine this would be `127.0.0.1`. You will also need to update the project urls.py file:

```
from django.conf import settings # New Import
from django.conf.urls.static import static # New Import


if not settings.DEBUG:
        urlpatterns += static(settings.STATIC_URL, document_root=setti
```

#TODO(leifos): Maybe we should describe all this in the deployment chapter... probably makes the most sense

5.4. The Static Media Server

Now that you can dispatch static files, let's look at uploading media. Many websites provide their users with the ability to do this - for example, to upload a profile image. This section shows you how to add a simple development media server to your Django project. The development media server can be used in conjunction with file uploading forms which we will touch upon in Chapter 9.

So, how do we go about setting up a development media server? The first step is to create another new directory called `media` within our Django project's root (e.g. `<workspace>/tango_with_django_project/`). The new `media` directory should now be sitting alongside your `templates` and `static` directories. After you create the directory, you must then modify your Django project's `urls.py` file, located in the project configuration directory (e.g. `<workspace>/tango_with_django_project/tango_with_django_project/`). Add the following code to the `urls.py` file.

```
# At the top of your urls.py file, add the following line:
from django.conf import settings

# UNDERNEATH your urlpatterns definition, add the following two lines:
if settings.DEBUG:
    urlpatterns += patterns(
        'django.views.static',
        (r'media/(?P<path>.*)',
        'serve',
        {'document_root': settings.MEDIA_ROOT}), )
```

The `settings` module from `django.conf` allows us access to the variables defined within our project's `settings.py` file. The conditional statement then checks if the Django project is being run in DEBUG mode. If the project's `DEBUG` setting is set to `True`, then an additional URL matching pattern is appended to the `urlpatterns` tuple. The pattern states that for any file requested with a URL starting with `media/`, the request will be passed to the `django.views.static` view. This view handles the dispatching of uploaded media files for you.

With your `urls.py` file updated, we now need to modify our project's `settings.py` file. We now need to set the values of two variables. In your file, add `MEDIA_URL` and `MEDIA_ROOT`, setting them to the values as shown below.

```
MEDIA_URL = '/media/'
MEDIA_ROOT = os.path.join(BASE_DIR, 'media') # Absolute path to the me
```

The first variable `MEDIA_URL` defines the base URL from which all media files will be accessible on your development server. Setting the `MEDIA_URL` for example to `/media/` will mean that user uploaded files will be available from the URL `http://127.0.0.1:8000/media/`. `MEDIA_ROOT` is used to tell Django where uploaded files should be stored on your local disk. In the example above, we set this variable to the result of joining our `PROJECT_PATH` variable defined in Section 5.1 with `/media/`. This gives an absolute path of `<workspace>/tango_with_django_project/media/`.

Caution

As previously mentioned, the development media server supplied with Django is very useful for debugging purposes. However, it should **not** be used in a production environment. The official Django documentation on static files warns that such an approach is "grossly inefficient and insecure". If you do come to deploying your Django project, read the documentation to see an alternative solution for file uploading that can handle a high volume of requests in a much more secure manner.

You can test this setup works by placing an image file in your newly created `media` directory. Drop the file in, start the Django development server, and request the image in your browser. For example, if you added the file `rango.jpg` to `media`, the URL you should enter would look like `http://127.0.0.1:8000/media/rango.jpg`. The image should show in your browser. If it doesn't, you'll need to go back and check your setup.

#TODO(leifos): check that this still works (certainly you can access the images.. need to check the uploading)

5.5. Basic Workflow

With the chapter complete, you should now know how to setup and create templates, use templates within your views, setup and use Django to send static media files, include images within your templates and setup Django's static media server to allow for file uploads. We've actually 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 becomes 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` file. You may wish to use Django template variables (e.g. `{{ variable_name }}`) 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.
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 is another important process 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` tuple 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.

The next chapter will look at databases. We'll see how to make use of Django's excellent database layer to make your life easier and SQL free!

5.6. Exercises

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

- Convert the about page to use a template too from a template called `about.html`.
- Within the `about.html` template, add a picture stored within your project's static media.

Navigation

How To Tango With Django 1.7 »

previous next

6. Models and Databases

Working with databases often requires you to get your hands dirty messing about with SQL. In Django, a lot of this hassle is taken care of for you by Django's object relational mapping (ORM) functions, and how Django encapsulates databases tables through models. Essentially, a model is a Python object that describes your data model/table. Instead of directly working on the database table via SQL, all you have to do is manipulate the corresponding Python object. In this chapter, we'll walkthrough how to setup a database and the models required for Rango.

6.1. Rango's Requirements

First, let's go over the data requirements for Rango. The following list provides the key details of Rango's data requirements.

- Rango is a essentially a web page directory - a site containing links to other websites.
- There are a number of different webpage categories, and each category houses a number of links. We assumed in Chapter 2 that this is a one-to-many relationship. See the Entity Relationship Diagram below.
- A category has a name, number of visits, and number of likes.
- A page refers to a category, has a title, URL and a number of views.

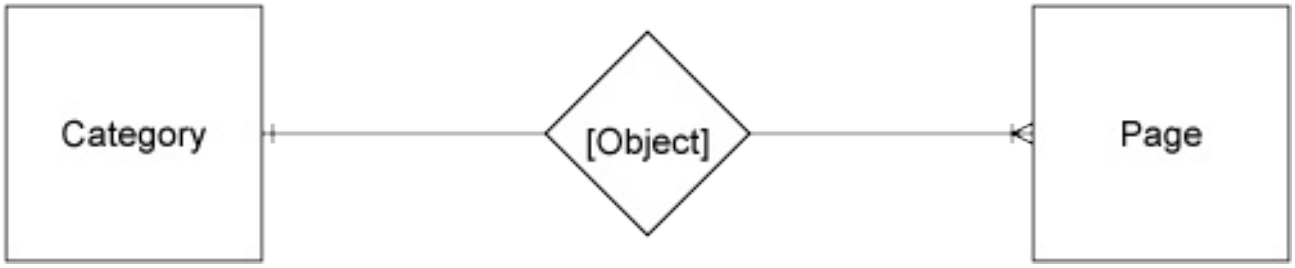


Figure 1: The Entity Relationship Diagram of Rango's two main entities.

6.2. Telling Django About Your Database

Before we can create any models, the database configuration needs to be setup. In Django 1.7, when you create a project, Django automatically populates the dictionary called `DATABASES`, which is located in your `settings.py`. It will contain

something like:

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

As you can see the default engine will be a SQLite3 backend. This provides us with access to the lightweight python database, SQLite, which is great for development purposes. The only other value we need to set is the `NAME` key/value pair, which we have set to `DATABASE_PATH`. For other database engines, other keys like `USER`, `PASSWORD`, `HOST` and `PORT` can also be added to the dictionary.

Note

While using an SQLite engine for this tutorial is fine, it may not perhaps be the best option when it comes to deploying your application. Instead, it may be better to use a more robust and scalable database engine. Django comes with out of the box support for several other popular database engines, such as PostgreSQL and MySQL. See the official Django documentation on Database Engines for more details. You can also check out this excellent article on the SQLite website which explains situation where you should and you shouldn't consider using the lightweight SQLite engine.

6.3. Creating Models

With your database configured in `settings.py`, let's create the two initial data models for the Rango application.

In `rango/models.py`, we will define two classes - both of which must inherit from `django.db.models.Model`. The two Python classes will be the definitions for models representing categories and pages. Define the `Category` and `Page` models as follows.

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

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

When you define a model, you need to specify the list of attributes and their associated types along with any optional parameters. Django provides a number of built-in fields. Some of the most commonly used are listed below.

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

Check out the Django documentation on model fields for a full listing.

For each field, you can specify the `unique` attribute. If set to `True`, only one instance of a particular value in that field may exist throughout the entire database model. For example, take a look at our `Category` model defined above. The field `name` has been set to unique - thus every category name must be unique.

This is useful if you wish to use a particular field as an additional database key. You can also specify additional attributes for each field such as specifying a default value (`default='value'`), and whether the value for a field can be `NULL` (`null=True`) or not.

Django also provides simple mechanisms that allows us to relate models/database tables together. These mechanisms are encapsulated in three further field types, and are listed below.

- `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.
- `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. You should be aware that Django creates an ID field for you automatically in each table relating to a model. You therefore do not need to explicitly define a primary key for each model - it's done for you!

Note

When creating a Django model, it's good practice to make sure you include the `__unicode__()` method - a method almost identical to the `__str__()` method. If you're unfamiliar with both of these, think of them as methods analogous to the `toString()` method in a Java class. The `__unicode()__` method is therefore used to provide a unicode representation of a model instance. Our `Category` model for example returns the name of the category in the `__unicode__()` method - something which will be incredibly handy to you when you begin to use the Django admin interface later on in this chapter.

Including a `__unicode__()` method in your classes is also useful when debugging your code. Issuing a `print` on a `Category` model instance without a `__unicode()__` method will return `<Category: Category object>`. We know it's a category, but which one? Including `__unicode()__` would then return `<Category: python>`, where `python` is the `name` of a given category. Much better!

6.4. Creating and Migrating the Database

With our models defined, we can now let Django work its magic and create the table representations in our database. In previous versions of Django this would be performed using the command:

```
$ python manage.py syncdb
```

However, Django 1.7 provides a migration tool to setup and update the database to reflect changes in the models. So the process has become a little more complicated -

but the idea is that if you make changes to the models, you will be able to update the database without having to delete it.

6.4.1. Setup Database and Create Superuser

If you have not done so already you first need to initial the database. This is done via the migrate command.

```
$ python manage.py migrate


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

If you rememnber in `settings.py` there was a list of INSTALLED_APPS, this initial call to migrate, creates the tables for the associated apps, i.e. auth, admin, etc. There should be a file called, `db.sqlite` in your project base directory.

Now you will want to 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 later on in this tutorial. 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.

6.4.2. Creating / Updating Models / Tables

Whenever you make changes to the models, then you need to register the changes, via the `makemigrations` command for the particular app. So for rango, we need to issue:

```
$ python manage.py makemigrations rango

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

If you inspect the `rango/migrations` folder, you will see that a python script have been created, called, `0001_initial.py''. To see the SQL that will be performed to make this migration, you can issue the command, ``python manage.py sqlmigrate <app_name> <migration_no>`. The migration number is show above as 0001, so we would issue the command, `python manage.py sqlmigrate rango 0001` for rango to see the SQL. Try it out.

Now, to apply these migrations (which will essentially create the database tables), then you need to issue:

```
$ python manage.py migrate

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

Warning

Whenever you add to existing models, you will have to repeat this processrunning `python manage.py makemigrations <app_name>`, and then `python manage.py migrate`

You may have also noticed that our `Category` model is currently lacking some fields that we defined in Rango's requirements. We will add these in later to remind you of the updating process.

6.5. Django Models and the Django Shell

Before we turn our attention to demonstrating the Django admin interface, it's worth noting that you can interact with Django models from the Django shell - a very

useful aid 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. The following terminal session demonstrates this functionality. Check out the inline commentary to see what each command does.

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

# Show all the current categories
>>> print Category.objects.all()
[] # Returns an empty list (no categories have been defined!)

# 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()
[<Category: test>] # We now have a category called 'test' saved in the

# 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, of which there are none because our table is empty. Then we create and save a Category, before printing out all the categories again. This second `print` should then show the `Category` just added.

Note

The example we provide 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 is good time to complete part one of the official Django

Tutorial to learn more about interacting with the models. Also check out the official Django documentation on the list of available commands for working with models.

6.6. Configuring the Admin Interface

One of the stand-out features of Django is that it provides a built in, web-based administrative interface that allows us to browse and edit data stored within our models and corresponding database tables. In the `settings.py` file, you will notice that one of the pre-installed apps is `django.contrib.admin`, and in your project's `urls.py` there is a urlpattern that matches `admin/`.

Start the development server:

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

and visit the url, `http://127.0.0.1:8000/admin/`. You should be able to log into the Django Admin interface using the username and password created for the superuser. The admin interface only contains tables relevant to the sites adminstration, `Groups` and `Users`. So we will need to instruct Django to also include the models from `rango`.

To do this, open the file `rango/admin.py` and add the following code:

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

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

This will register the models with the admin interface. If we were to have another model, it would be a trivial case of calling the `admin.site.register()` function, passing the model in as a parameter.

With all of these changes made, re-visit/refresh: `http://127.0.0.1:8000/admin/`. You should now be able to see the Category and Page models, like in Figure 2.

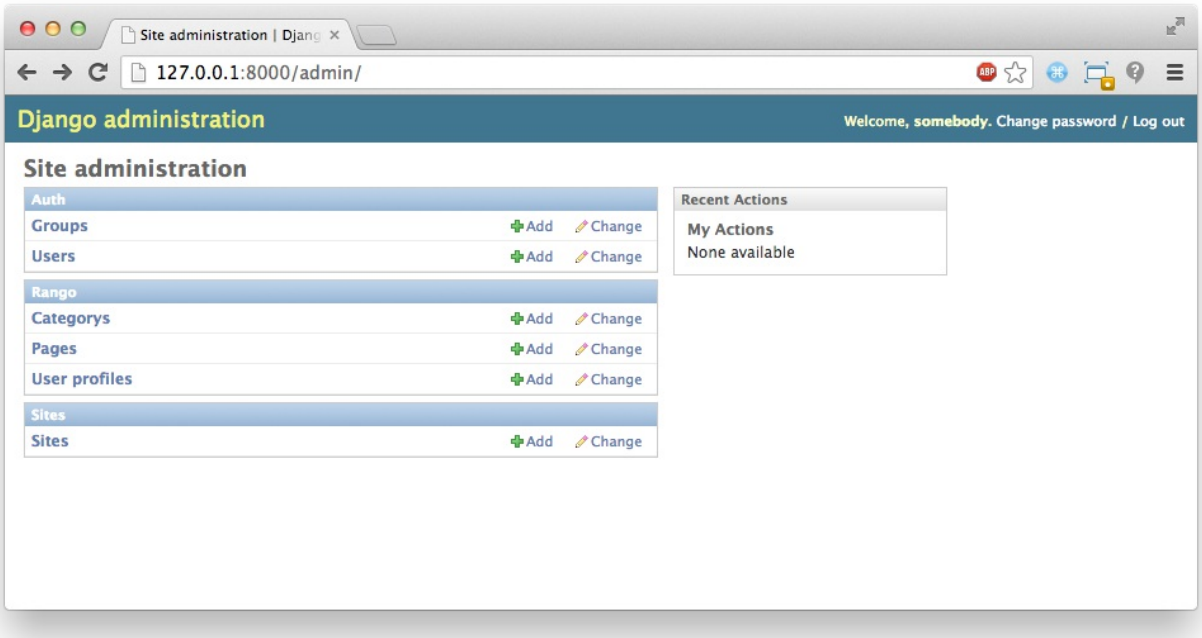


Figure 2: The Django admin interface. Note the Rango category, and the two models contained within.

Try clicking the `Categorys` link within the `Rango` section. From here, you should see the `test` category that we created via the Django shell. Try deleting the category as we'll be populating the database with a population script next. The interface is easy to use. Spend a few minutes creating, modifying and deleting both categories and pages. You can also add new users who can login to the Django admin interface for your project by adding a user to the `User` in the `Auth` application.

Note

Note the typo within the admin interface (categorys, not categories). This problem can be fixed by adding a nested `Meta` class into your model definitions with the `verbose_name_plural` attribute. Check out Django's official documentation on models for more information.

Note

The example `admin.py` file for our Rango application is the most simple, functional example available. There are many different features which you can use in the `admin.py` to perform all sorts of cool customisations, such as changing the way models appear in the admin interface. For this tutorial, we'll stick with the bare-bones admin interface, but you can check out the official Django documentation on the admin interface for more information if you're interested.

6.7. Creating a Population Script

It's highly likely that during the course of development, you'll come to a point where you will need to modify a Django model. When you do this, the easiest option - without external software - is to re-create your entire database and run `python manage.py syncdb` ...again! Since this slow and repetitive task can be such a pain, it's good practice to create what we call a population script for your database. This script is designed to automatically populate your database with test data for you, which can potentially save you lots of time.

To create a population script for Rango's database, we start by creating a new Python module within our 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_pro

import django
django.setup()

from rango.models import Category, Page


def populate():
    python_cat = add_cat('Python')

    add_page(cat=python_cat,
        title="Official Python Tutorial",
        url="http://docs.python.org/2/tutorial/")

    add_page(cat=python_cat,
        title="How to Think like a Computer Scientist",
        url="http://www.greenteapress.com/thinkpython/")

    add_page(cat=python_cat,
        title="Learn Python in 10 Minutes",
        url="http://www.korokithakis.net/tutorials/python/")

    django_cat = add_cat("Django")
```

```
    add_page(cat=django_cat,
        title="Official Django Tutorial",
        url="https://docs.djangoproject.com/en/1.5/intro/tutorial01/")

    add_page(cat=django_cat,
        title="Django Rocks",
        url="http://www.djangorocks.com/")

    add_page(cat=django_cat,
        title="How to Tango with Django",
        url="http://www.tangowithdjango.com/")

    frame_cat = add_cat("Other Frameworks")

    add_page(cat=frame_cat,
        title="Bottle",
        url="http://bottlepy.org/docs/dev/")

    add_page(cat=frame_cat,
        title="Flask",
        url="http://flask.pocoo.org")

    # Print out what we have added to the user.
    for c in Category.objects.all():
        for p in Page.objects.filter(category=c):
            print "- {0} - {1}".format(str(c), str(p))

def add_page(cat, title, url, views=0):
    p = Page.objects.get_or_create(category=cat, title=title, url=url,
    return p

def add_cat(name):
    c = Category.objects.get_or_create(name=name)[0]
    return c

# Start execution here!
if __name__ == '__main__':
    print "Starting Rango population script..."
    populate()
```

While this looks like a lot of code, what it does is relatively simple. As we define a

series of functions at the top of the file, code execution begins towards the bottom - look for the line `if __name__ == '__main__'`. We call the `populate()` function.

Warning

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 the project setting file. Then you can call `django.setup()` to import the django settings. If you don't, an exception will be raised. This is why we import `Category` and `Page` after the settings have been loaded.

The `populate()` function is responsible for the calling the `add_cat()` and `add_page()` functions, who are in turn responsible for the creation of new categories and pages respectively. `populate()` keeps tabs on category references for us as we create each individual `Page` model instance and store them within our database. Finally, we loop through our `Category` and `Page` models to print to the user all the `Page` instances and their corresponding categories.

Note

We make use of the convenience `get_or_create()` method for creating model instances. 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. This 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. As we mentioned previously, why reinvent the wheel if it's already there?

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.

The `[0]` at the end of our call to the method to retrieve the `object` portion of the tuple returned from `get_or_create()`. 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, we can run the script by changing the current working directory in a terminal to our Django project's root and executing the module with the command `$ python populate_rango.py`. You should then see output similar to that shown below.

```
$ 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
```

Now let's verify that the population script populated the database. Restart the Django development server, navigate to the admin interface, and check that you have some new categories and pages. Do you see all the pages if you click `Pages`, like in Figure 3?

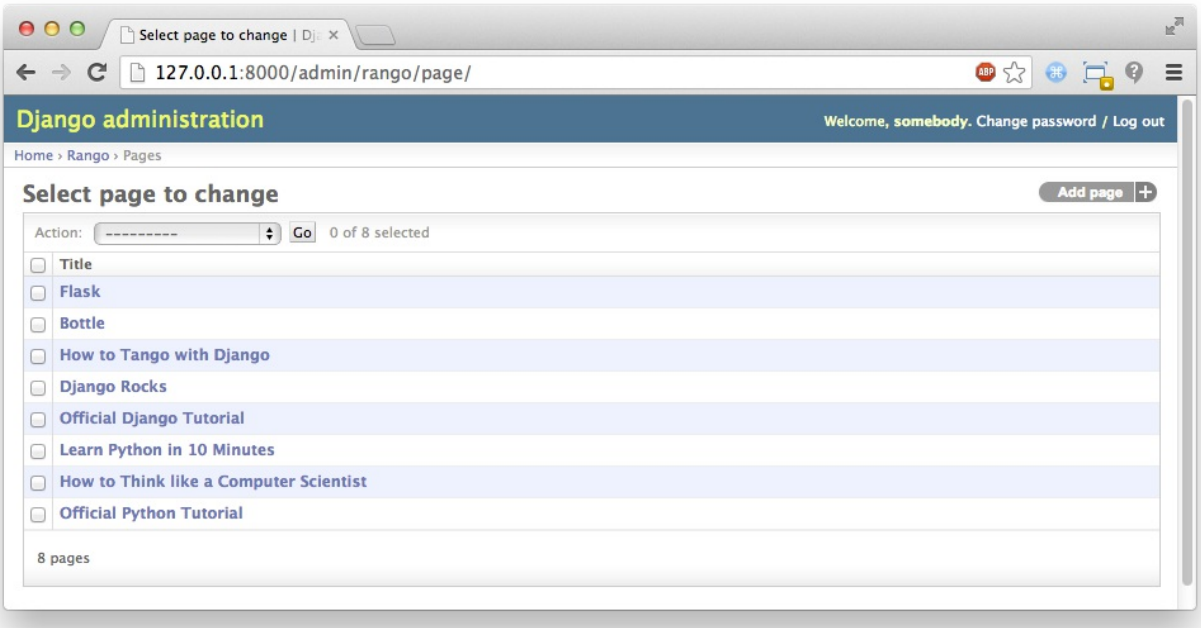


Figure 3: The Django admin interface, showing the Page table populated with sample data from our population script.

A population script takes a little bit of time to write but when you are working with a team, you will be able to share the population script so that everyone can create the database and have it populated. Also, for unit testing it will come in handy.

6.8. Basic Workflows

Now that we've covered the core principles of dealing with Django's models functionality, 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.

6.8.1. 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` file to make them accessible via the admin interface.

6.8.2. 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. Then perform the migration `$ python manage.py sqlmigrate <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).

Invariably there will be times when you will have to delete your database. In which case you will have to run the `migrate` command, then `createsuperuser` command, followed by the `sqlmigrate` commands for each app, then you can populate the database.

6.9. Exercises

Now that you've completed the chapter, try out these exercises to reinforce and practice what you have learnt.

- Update the Category model to include the additional attributes, `views` and `likes` where the default value is zero.
- Make the migrations for your app/model, then migrate your database
- 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.
- Undertake the part two of official Django tutorial if you have not done so. This will help to reinforce further what you have learnt here, and to learn more about customising the admin interface.
- Customise the Admin Interface - so that when you view the Page model it displays in a list the category, the name of the page and the url.

6.9.1. Hints

If you require some help or inspiration to get these exercises done, these hints will hopefully help you out.

- Modify the `Category` model by adding in the fields, `view` and `likes` as ``IntegerField``s.
- Modify the `add_cat` function in the `populate.py` script, to take the `views` and `likes`. Once you get the Category c, then you can update the number of views with `c.views`, and similarly with likes.
- To customise 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.

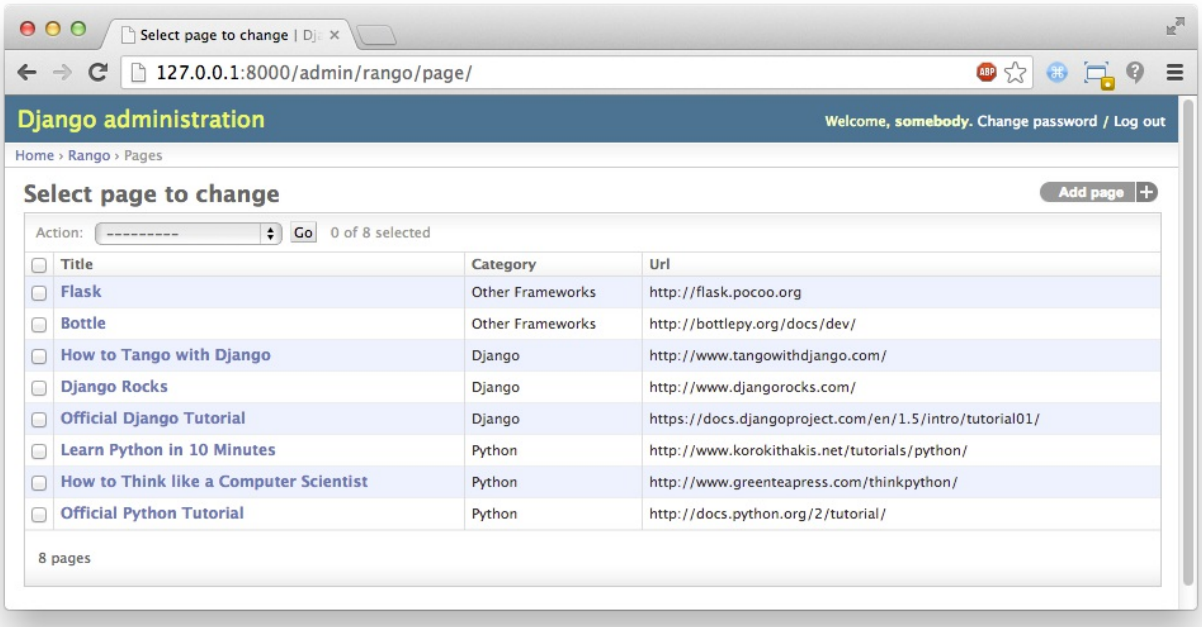


Figure 4: The updated admin interface page view, complete with columns for category and URL.

Navigation

How To Tango With Django 1.7 »

previous next

7. Models, Templates and Views

Now that we have the models set up and populated with some data, we can now start putting things together. We'll be figuring out how to access data from the models within the views, and how to present this data via the templates.

7.1. Basic Workflow: Data Driven Pages

There are five main steps that you must undertake to create a data driven webpage in Django.

1. First, import the models you wish to use into your application's `views.py` file.
2. Within the view you wish to use, query the model to get the data you want to present.
3. Pass the results from your model into the template's context.
4. Setup your template to present the data to the user in whatever way you wish.
5. If you have not done so already, map a URL to your view.

These steps highlight how Django's framework separates the concerns between models, views and templates.

7.2. Showing Categories on Rango's Homepage

One of the requirements regarding the main pages was to show the top five rango'ed categories.

7.2.1. Importing Required Models

To fulfil this requirement, we will go through each of the above steps. First, open `rango/views.py` and import the `Category` model from Rango's `models.py` file.

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

7.2.2. Modifying the Index View

With the first step out of the way, we then want to modify our `index()` function. If we cast our minds back, we should remember the `index()` function is responsible for the main page view. Modify the function to look like the example below.

```
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 which will be pass
    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 we have performed steps two and three in one go. First, we queried the `Category` model to retrieve the top five categories. Here we used the `order_by()` method to sort by the number of likes in descending order - hence the inclusion of the `-`. We then restricted this list to the first 5 `Category` objects in the list.

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

Note that the Category Model contains the field `likes`. So for this to work you need to have completed the exercises in the previous chapter, i.e. the Category Model needs to be updated to include the `likes` field.

7.2.3. Modifying the Index Template

With the view updated, all that is left for us to do is update the template `rango/index.html`, located within your project's `templates` directory. Change the HTML code of the file so that it looks like the example shown below.

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

```
    </head>

    <body>
        <h1>Rango says...hello world!</h1>

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

        <a href="/rango/about/">About</a>
    </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 (i.e. `{% 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, printing out 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 so.

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 three categories underneath the page title just like in Figure 1.

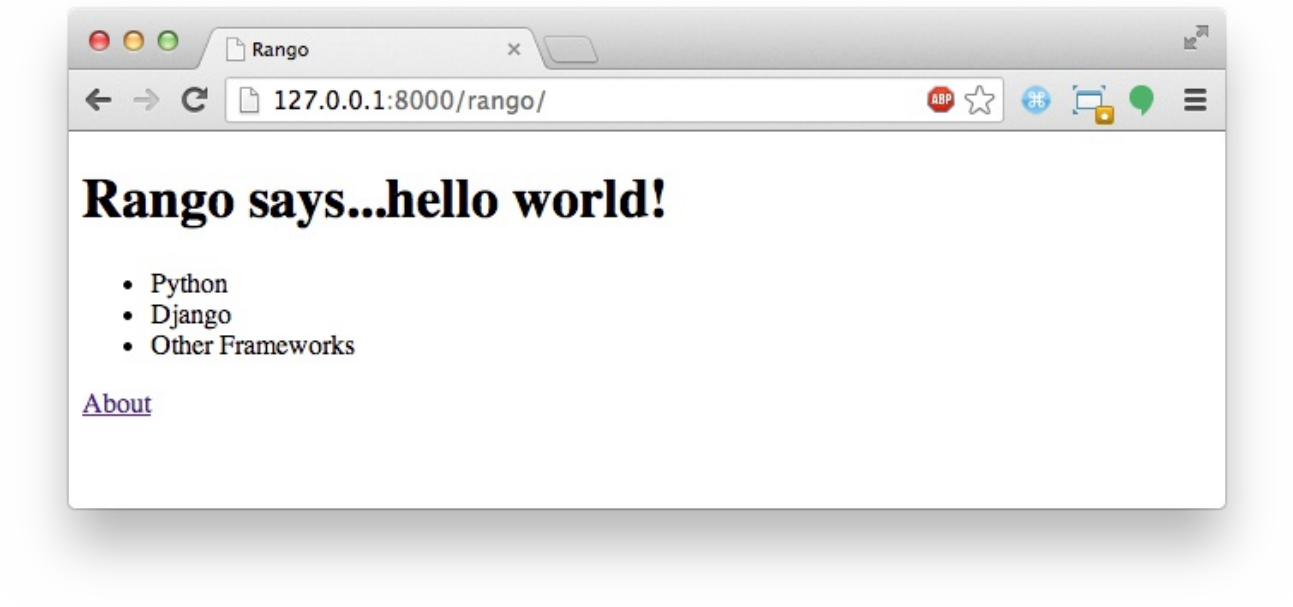


Figure 1: The Rango homepage - now dynamically generated - showing a list of categories. How exciting!

7.3. Creating a Details Page

According to Rango's specification, 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.

7.3.1. 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, these URLs are not easily understood by humans. Although we could probably infer that the number relates to a category, how would a user know what category relates to unique IDs 1 or 2? The user wouldn't know without trying.

Instead, we could just use the category name as part of the URL. `/rango/category/Python/` should give us a list of pages related to the Python category. This is a simple, readable and meaningful URL. If we go with this approach, we'll have to handle categories which have multiple words, like 'Other Frameworks', etc.

Note

Designing clean 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, based on the answers provided at: http://stackoverflow.com/questions/837828/how-do-i-create-a-slug-in-django

7.3.2. Update Category Table with Slug Field

To make clean urls we are going to include a slug field in the `Category` model. First we need to import the function `slugify` from django, which will replace whitespace with hyphens, i.e "how do i create a slug in django" turns into "how-do-i-create-a-slug-in-django".

Warning

While you can use spaces in URLs, it is considered to be unsafe to use them. Check out IETF Memo on URLs to read more.

Then 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 everytime 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(unique=True)

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

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

Now that you have performed this update to the Model, you will need to perform the migration.

Since we did not provide a default value for the slug, and we already have existing data in the model, then the migrate command will give you two options. Select the option to provide a default, and enter 'default'. Dont worry this will get updated shortly. Now re-run your population script. Since the `save` method is called for each Cateogry, the overrided `save` method will be executed, updating the slug field. Run the server, and inspect the data in the models via the admin interface.

In the admin interface you may want it to automatically pre-populate the slug field as your type in the category name. To do this you can update `rango/admin.py` with the following code:

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

# Add in this class to customized the Admin Interface
class CategoryAdmin(admin.ModelAdmin):
    prepopulated_fields = {'slug':('name',)}

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

Try out the admin interface and add in a new category. Pretty cool, hey! Now that we have added in slug fields we can now use them as clean urls :-).

7.3.3. Category Page Workflow

With our URLs design chosen, let's get started. We'll undertake the following steps.

1. Import the Page model into `rango/views.py`.
2. Create a new view in `rango/views.py` - called `category` - The `category` view will take an additional parameter, `category_name_url` which will stored the encoded category name.
    - We will need some help 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.

7.3.4. 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, `category()`.

```
def category(request, category_name_slug):

    # Create a context dictionary which we can pass to the template re
    context_dict = {}

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

        # Retrieve all of the associated pages.
        # Note that filter returns >= 1 model instance.
        pages = Page.objects.filter(category=category)

        # Adds our results list to the template context under name pag
        context_dict['pages'] = pages
        # We also add the category object from the database to the con
        # We'll use this in the template to verify that the category e
        context_dict['category'] = category
    except Category.DoesNotExist:
        # We get here if we didn't find the specified category.
        # Don't do anything - the template displays the "no category"
```

```
        pass

    # 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, then we attempt to extract the data from the models, and add in the relevant data to the context dictionary. We determine which category by using the value passed as parameter `category_name_slug` to the `category()` view function. If the category is found in the Category model, we can then pull out the associated Pages, and add this to the context dictionary, `context_dict`.

7.3.5. 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>
        <h1>{{ category_name }}</h1>
        {% if category %}
            {% 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 {{ category_name }} does not exist!
        {% endif %}
    </body>
```

```
</html>
```

The HTML code example again demonstrates how we utilise the data passed to the template via its context. We make use of the `category_name` variable and our `category` and `pages` objects. If `category` is not defined within our template context, the category was not found within the database, and a friendly error message is displayed stating this fact. If the opposite is true, we then proceed to check for `pages`. If `pages` is undefined or contains no elements, we display a message stating there are no pages present. Otherwise, the pages within the category are presented in a HTML list. For each page in the `pages` list, we present their `title` and `url` attributes.

Note

The Django template conditional tag - `{% if %}` - is a really neat way of determining the existence of an object within the template's context. Try getting into the habit of performing these checks to reduce the scope for potential exceptions that could be raised within your code.

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 - and presentational aspects of your Django applications should be encapsulated within templates.

7.3.6. Parameterised URL Mapping

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

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

As you can see, we have added in a rather complex entry that will invoke `view.category()` when the regular expression `r'^(?P<category_name_slug>\w+)/$'` is matched. We set up our regular expression to

look for any sequence of alphanumeric characters (e.g. a-z, A-Z, or 0-9) and the hyphen(-) before the trailing URL slash. This value is then passed to the view `views.category()` as parameter `category_name_slug`, the only argument after the mandatory `request` argument.

Note

When you wish to parameterise URLs, it's important to ensure that your URL pattern matches the parameters that the corresponding view takes in. To elaborate further, let's take the example we added above. The pattern added was as follows.

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

We can from here deduce that the characters (both alphanumeric and underscores) between `category/` and the trailing `/` at the end of a matching URL are to be passed to method `views.category()` as named parameter `category_name_slug`. For example, the URL `category/python-books/` would yield a `category_name_slug` of `python-books`.

As you should remember, all view functions defined as part of a Django project 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. Using the same example, our `category` view signature is altered such that it now looks like the following.

```
def category(request, category_name_slug):
    # ... code here ...
```

It's not the position of the additional parameters that matters, it's the name that must match anything defined within the URL pattern. Note how `category_name_slug` defined in the URL pattern matches the `category_name_slug` parameter defined for our view. Using `category_name_slug` in our view will give `python-books`, or whatever value was supplied as that part of the URL.

Note

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 regular expression problems.

7.3.7. 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 to provide users with a means to view 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>
<html>
    <head>
        <title>Rango</title>
    </head>

    <body>
        <h1>Rango says..hello world!</h1>

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

    </body>
</html>
```

Here we have updated each list element (`<li>`) adding a HTML hyperlink (`<a>`). The hyperlink has an `href` attribute, which we use to specify the target URL defined by `{{ category.slug }}`.

7.3.8. Demo

Let's try everything out now by visiting the Rango's homepage. You should see your homepage listing all the categories. The categories should now be clickable links. Clicking on `Python` should then take you to the `Python` detailed category view, as demonstrated in Figure 2. If you see a list of links like `Official Python Tutorial`, then you've successfully set up the new view. Try navigating a category which doesn't exist, like `/rango/category/computers`. You should see a message telling you that no pages exist in the category.

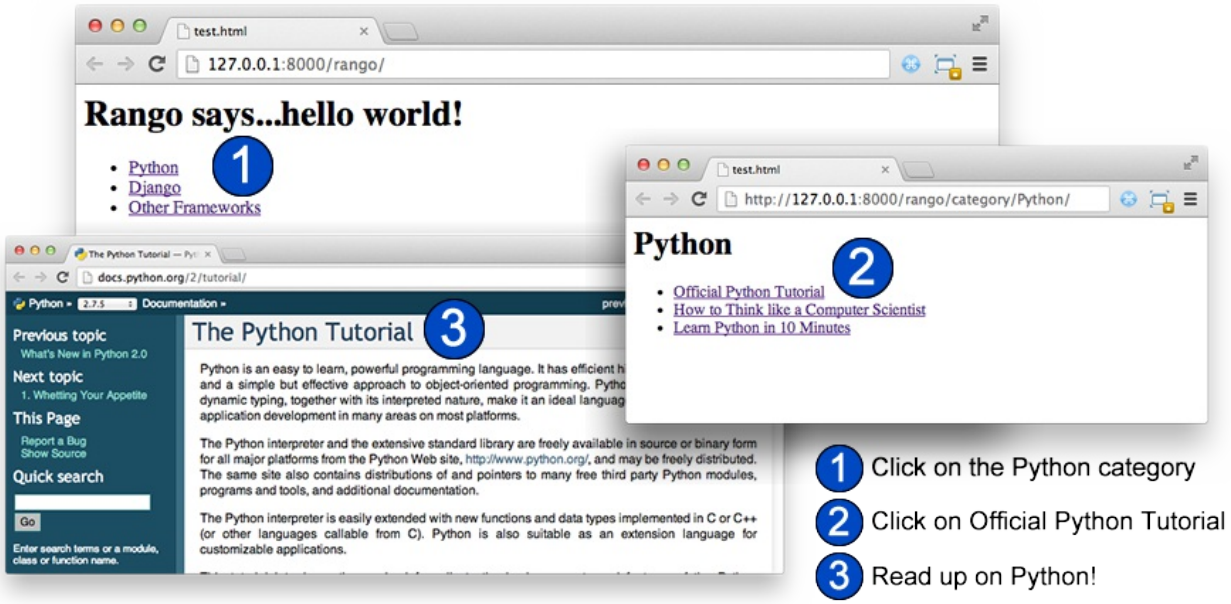


Figure 2: What your link structure should now look like. Starting with the Rango homepage, you are then presented with the category detail page. Clicking on a page link takes you to the linked website.

7.4. Exercises

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

- Modify the index page to also include the top 5 most viewed pages.
- Undertake the part three of official Django tutorial if you have not done so already to further what you've learnt here.

7.4.1. Hints

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

- Update the population script to add some value to the views count for each page.

Navigation

How To Tango With Django 1.7 »

previous next

8. Fun with Forms

So far we have only presented data through the views and templates that we have created. 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 gather information from users and send it back to your web application. 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 HTML hassle. This part of the tutorial will look at how to implement the necessary infrastructure that will allow users of Rango to add categories and pages to the database via forms.

8.1. Basic Workflow

The basic steps involved in creating a form and allowing users to enter data via the form 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 new one).

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.

8.2. Page and Category Forms

First, create a file called `forms.py` within the `rango` application directory. While this step is not absolutely necessary, as you could put the forms in the `models.py`, this makes the codebase a lot cleaner and clearer to understand.

8.2.1. 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.

```
from django import forms
from rango.models import Page, Category

class CategoryForm(forms.ModelForm):
    name = forms.CharField(max_length=128, help_text="Please enter the
    views = forms.IntegerField(widget=forms.HiddenInput(), initial=0)
    likes = forms.IntegerField(widget=forms.HiddenInput(), initial=0)
    slug = forms.CharField(widget=forms.HiddenInput(), required=False)

    # An inline class to provide additional information on the form.
    class Meta:
        # Provide an association between the ModelForm and a model
        model = Category
                fields = ('name',)


class PageForm(forms.ModelForm):
    title = forms.CharField(max_length=128, help_text="Please enter th
```

```
    url = forms.URLField(max_length=200, help_text="Please enter the U
    views = forms.IntegerField(widget=forms.HiddenInput(), initial=0)

    class Meta:
        # Provide an association between the ModelForm and a model
        model = Page

        # What fields do we want to include in our form?
        # This way we don't need every field in the model present.
        # Some fields may allow NULL values, so we may not want to inc
        # Here, we are hiding the foreign key.
                # we can either exclude the category field from the fo
                exclude = ('category',)
                #or specify the fields to include (i.e. not include th
        #fields = ('title', 'url', 'views')
```

#TODO(leifos): Note that in Django 1.7+ it is now required to specify the fields that are included, via `fields`, or specify the fields that 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 Chapter 6 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 without giving the control to the user as the field will be hidden, yet the form will provide the value to the model. 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 models 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 form, 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()` to populating this field. Essentially, you need to be careful when you define your models and forms to make sure that form is going to contain and pass on all the data that is required to populate your model correctly.

Besides the `CharField` and `IntegerField` widget, 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?). We highly recommend you have a look at the official Django documentation on widgets to see what components exist and the arguments you can provide to customise them.

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.

Note

We highly recommend you check out the official Django documentation on forms for further information about how to customise them.

8.2.2. 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`.

```
from rango.forms import CategoryForm

def add_category(request):
    # 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 call the index() view.
            # The user will be shown the homepage.
            return index(request)
        else:
            # The supplied form contained errors - just print them to
            print form.errors
    else:
        # If the request was not a POST, display the form to enter det
        form = CategoryForm()

    # Bad form (or form details), no form supplied...
    # 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 check the HTTP request method, to determine if it was a HTTP `GET` or `POST`. We can then handle different requests methods appropriately - i.e. whether we want to show a form (if it is a `GET`), or process form data (if it is a `POST`) - all from 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.

Note

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.

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 which 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` which will contain the relevant Django template code and HTML for the form and page.

8.2.3. 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>
```

```
        <form id="category_form" method="post" action="/rango/add_cate

            {% 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>
    </body>

</html>
```

Now, what does this code do? You can see that within the `<body>` of the HTML page that we place 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, the other visible form fields - with visible fields controlled by the `fields` attribute of 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 which can be used to explain to the user what he or she needs to enter.

Note

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 which 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.

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 CSRF token is required by the Django framework. 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 for more information about this.

8.2.4. 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 submit attribute. So we will need to follow suit in `rango/urls.py` and modify the `urlpatterns` as follows.

```
urlpatterns = patterns('',
    url(r'^$', views.index, name='index'),
    url(r'^about/$', views.about, name='about'),
    url(r'^add_category/$', views.add_category, name='add_category'),
    url(r'^category/(?P<category_name_url>\w+)$', views.category, name
```

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. Our new URL for adding a category is `/rango/add_category/`.

8.2.5. 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 just before the `</body>` closing tag.

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

8.2.6. Demo

Now let's try it out! Run your Django development server, and navigate to `http://127.0.0.1:8000/rango/`. Use your new link to jump to the add category page, and try adding a category. Figure 1 shows screenshots of the of the Add Category and Index Pages.

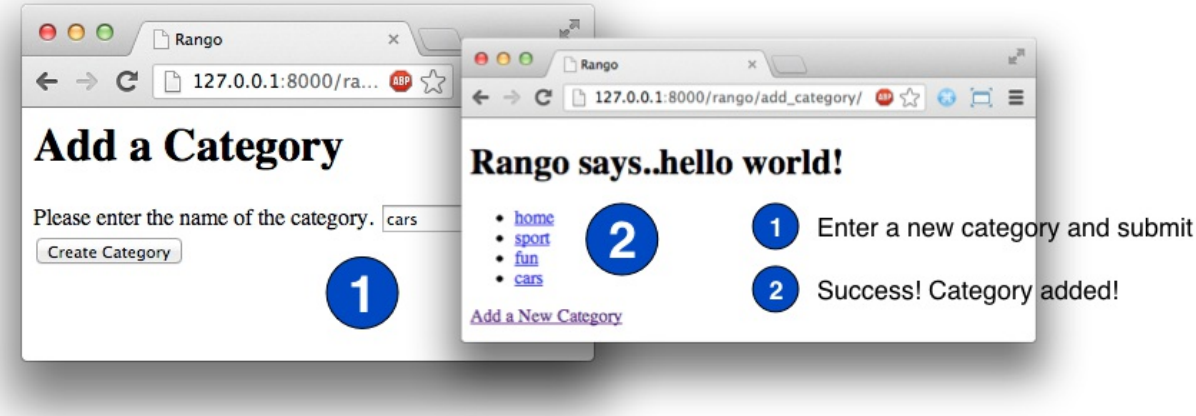


Figure 1: Adding a new category to Rango with our new form. The diagram illustrates the steps involved.

Note

If you add a number of categories, they will not always appear on the index page, that is because we are only showing the top 5 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. To see what is happening as you entered them in `rango/views.py` in `add_category()`, you can get the reference to the category model object created from `form.save()`, with `cat = form.save(commit=True)` and then print to console the category and slug, with `print cat, cat.slug` to see what is being created.

8.2.7. 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 well-formed, 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!

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. In our example above, 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://', prepen
        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 for the given field.
4. You must always end the `clean()` method by returning the reference to the `cleaned_data` dictionary. If you don't, you'll get some really frustrating errors!

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.

Note

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.

8.3. Exercises

Now that you've worked through the chapter, try these exercises to solidify your knowledge on Django's form functionality.

- What happens when 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?
- How could you gracefully handle when a user visits a category that does not exist?
- Undertake the part four of the official Django Tutorial if you have not done so already to reinforce what you have learnt here.

8.3.1. 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 for Pages - create a new view (`add_page()`), a new template (`rango/add_page.html`), URL mapping and then add a link from the category page. To get you started, here's the view logic for you.

```
from rango.forms import PageForm

def add_page(request, category_name_slug):

try:
        cat = Category.objects.get(slug=category_name_slug)
except Category.DoesNotExist:
        cat = None

        if request.method == 'POST':
```

```
        form = PageForm(request.POST)
        if form.is_valid():
        if cat:
                page = form.save(commit=False)
                page.category = cat
                page.views = 0
                page.save()
                # probably better to use a redirect here.
                return category(request, category_name_slug)
                else:
                print form.errors
        else:
                form = PageForm()

        context_dict = {'form':form, 'category': cat}

return render(request, 'rango/add_page.html', context_dict)
```

8.3.2. Hints

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

- Update the `category()` view to pass `category_name_slug` by inserting it to the view's `context_dict` dictionary.
- Update the `category.html` with a link to `/rango/category/<category_name_url>/add_page/`.
- Ensure that the link only appears when the requested category exists - with or without pages. i.e. in the template check with `{% if category %} .... {% else %} A category by this name does not exist {% endif %}`.
- Update `rango/urls.py` with a URL mapping to handle the above link.

Navigation

How To Tango With Django 1.7 »

previous next

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` application provided as part of a standard Django installation in package `django.contrib.auth`. According to Django's official documentation on Authentication, the application consists of the following aspects.

- Users.
- Permissions: a series of binary flags (e.g. yes/no) determining 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 in the area of user authentication. 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` tuple and check that `django.contrib.auth` and `django.contrib.contenttypes` are listed, so that it looks like the code below:

```
INSTALLED_APPS = (
        'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'rango',
```

```
)
```

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

Note

Remember, if you had to add the `auth` applications to your `INSTALLED_APPS` tuple, you will need to update your database with the `$ python manage.py migrate` command.

Passwords are stored by default in Django using the PBKDF2 algorithm, providing a good level of security for your user's data. You can read more about this as part of the official Django documentation on how django stores passwords. The documentation also provides an explanation of how to use different password hashers if you require a greater level of security.

If you want more control over how the passwords are hashed, then in the `settings.py` add in tuple to specify the `PASSWORD_HASHERS`:

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

Django will use the first hasher in `PASSWORD_HASHERS` i.e. settings.PASSWORD_HASHERS[0]. If you wanted to use a more secure hasher, you can install Bcrypt (see https://pypi.python.org/pypi/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',
)
```

However, to get up and running you don't need to explicitly specify the

`PASSWORD_HASHERS`, in which case Django defaults to, `django.contrib.auth.hashers.PBKDF2PasswordHasher`.

9.2. The `User` Model

The core of Django's authentication system is the `User` object, located at `django.contrib.auth.models.User`. A `User` object represents each of the people 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 comes complete with five primary 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 model also comes with other attributes such as `is_active` (which determines whether a particular account is active or not). Check the official Django documentation on the user model for a full list of attributes provided by the base `User` model.

9.3. Additional User Attributes

If you would like to include other attributes than what is provided by the `User` model, then you will needed to create a model that is associated with the the `User` model. For our Rango application, 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 instanc
    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 meanin
    def __unicode__(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
```

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

For Rango, we've added two fields to complete our user profile, and provided a `__unicode__()` method to return a meaningful value when a unicode representation of a `UserProfile` model instance is requested.

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

Note 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/`.

Warning

The Django `ImageField` field makes use of the Python Imaging Library (PIL). Back in Chapter 3, we discussed installing PIL along with Django to your setup. If you haven't got PIL installed, you'll need to install it now. If you don't, you'll be greeted with exceptions stating that the module `pil` cannot be found!

With our `UserProfile` model defined, we now edit Rango's `admin.py` file to include the new `UserProfile` model in the Django administration web interface. In the `admin.py` file, add the following line.

```
from rango.models import UserProfile

admin.site.register(UserProfile)
```

Note

Remember that your database must be updated with the creation of a new model. Run `$ python manage.py makemigrations rango` from your terminal to create the migration scripts for the new `UserProfile` model. Then run `$ python manage.py migrate`

9.4. Creating a User Registration View and Template

With our authentication infrastructure laid out, we can now begin to build onto it by providing users of our application with the opportunity to create new user accounts. We will achieve this via the creation of a new view and template combination.

Note

It is important to note that there are several off-the-shelf user registration packages available which reduce a lot of the hassle of building your own registration and log in forms. However, it is good to get a feeling for the underlying mechanics, because using such applications. It will also re-enforce your understanding of working with forms, how to extend upon the user model, and how to upload media.

To provide the user registration functionality we will go through the following steps:

1. Create a `UserForm` and `UserProfileForm`.

2. Add a view to handle the creation of a new user.
3. Create a template that displays the `UserForm` and `UserProfileForm`.
4. Map a URL to the view created.
5. Link the index page to the register page

9.4.1. 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. Neat!

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

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

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

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

You'll notice that within both classes, we added a nested `Meta` class. As the name of the nested class suggests, anything within a nested `Meta` class describes additional properties about the particular `ModelForm` class it belongs to. Each `Meta` class must at a bare minimum supply a `model` field, which references back to the model the `ModelForm` inheriting class should relate to. Our `UserForm` class is therefore associated with the `User` model, for example. As of Django 1.7, you also need to specify, `fields` or `exclude` to indicate which fields associated with the model should be present on the form.

Here we only want to show the fields, `username`, `email` and `password`, associated

with the `User` model, and the `website` and `picture` associated with the `UserProfile` model. For the `user` field within `UserProfile` we will need to make this association when we register the user.

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 then 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!

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

9.4.2. 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` file, add the following view function:

```
from rango.forms import UserForm, UserProfileForm

def register(request):

    # A boolean value for telling the template whether the registratio
    # Set to False initially. Code changes value to True when registra
    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 se
            # This delays saving the model until we're ready to avoid
            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
            if 'picture' in request.FILES:
                profile.picture = request.FILES['picture']

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

            # Update our variable to tell the template registration wa
            registered = True

        # Invalid form or forms - mistakes or something else?
        # Print problems to the terminal.
        # They'll also be shown to the user.
        else:
            print user_form.errors, profile_form.errors

    # Not a HTTP POST, so we render our form using two ModelForm insta
    # These forms will be blank, ready for user input.
    else:
        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, 're
```

Is the view a lot more complex? It might look so at first, but it isn't really. The only added complexity from our previous `add_category()` view is the need to handle two distinct `ModelForm` instances - one for the `User` model, and one for the `UserProfile` model. We also need to handle a user's profile image, if he or she chooses to upload one.

We also establish a link between the two model instances that we create. 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 in Section 9.4.1.

9.4.3. Creating the Registration Template

Now create a new template file, `rango/register.html` and add the following code:

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

    <body>
        <h1>Register with Rango</h1>

        {% if registered %}
        Rango says: <strong>thank you for registering!</strong>
        <a href="/rango/">Return to the homepage.</a><br />
        {% else %}
        Rango says: <strong>register here!</strong><br />

        <form id="user_form" method="post" action="/rango/register/"
                enctype="multipart/form-data">

            {% csrf_token %}

            <!-- Display each form. The as_p method wraps each element
                 (<p>) element. This ensures each element appears on a
                 making everything look neater. -->
            {{ 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 %}
    </body>
</html>
```

This HTML 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, apart from the title, only a success message is displayed.

Warning

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. You should also should remember to include the CSRF token, too. Ensure that you include `{% csrf_token %}` within your `<form>` element.

9.4.4. The `register()` View URL Mapping

Now we can add a URL mapping to our new view. In `rango/urls.py` modify the `urlpatterns` tuple as shown below:

```
urlpatterns = patterns('',
    url(r'^$', views.index, name='index'),
    url(r'^about/$', views.about, name='about'),
    url(r'^category/(?P<category_name_url>\w+)$', views.category, name
    url(r'^add_category/$', views.add_category, name='add_category'),
    url(r'^category/(?P<category_name_url>\w+)/add_page/$', views.add_
    url(r'^register/$', views.register, name='register'), # ADD NEW PA
    )
```

The newly added pattern points the URL `/rango/register/` to the `register()` view.

9.4.5. Linking Together

Finally, we can add a link pointing to that URL in our homepage `index.html` template. Underneath the link to the category addition page, add the following hyperlink.

```
<a href="/rango/register/">Register Here</a>
```

9.4.6. Demo

Easy! Now you'll have a new hyperlink with the text `Register Here` that'll take you to the registration page. Try it out now! Start your Django development server and try to register a new user account. Upload a profile image if you wish. Your registration form should look like the one illustrated in Figure 1.

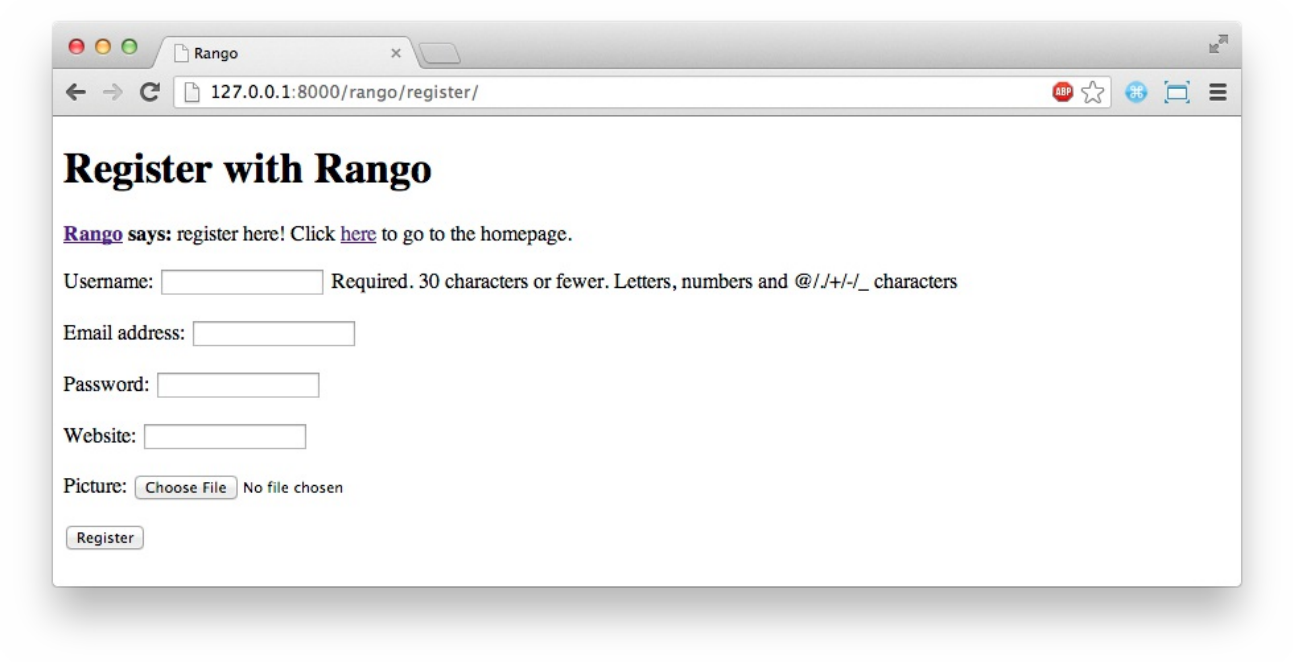


Figure 1: A screenshot illustrating the basic registration form you create as part of this tutorial.

Upon seeing the message indicating your details were successfully registered, the database should have two new entries in its tables corresponding to the `User` and `UserProfile` models.

9.5. Adding Login Functionality

With the ability to register accounts completed, we now need to add login in functionality. To achieve this we will need to undertake the workflow below:

- Create a login in view to handle 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

9.5.1. Creating the `login()` View

In `rango/views.py` create a new function called `user_login()` and add the following code:

```
def user_login(request):

    # If the request is a HTTP POST, try to pull out the relevant info
    if request.method == 'POST':
        # Gather the username and password provided by the user.
        # This information is obtained from the login form.
        username = request.POST['username']
        password = request.POST['password']

        # Use Django's machinery to attempt to see if the username/pas
        # 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
        # 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 u
                # We'll send the user back to the homepage.
                login(request, user)
                return HttpResponseRedirect('/rango/')
            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 use
            print "Invalid login details: {0}, {1}".format(username, p
            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 t
        # blank dictionary object...
        return render(request, 'rango/login.html', {})
```

This view may seem rather complicated as it has to handle a variety of situations. Like in previous examples, the `user_login()` view handles form rendering and processing.

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, the username and password are extracted from the form. These details are then used to attempt to authenticate the user (with Django's `authenticate()` function). `authenticate()` then returns a `User` object if the username/password combination exists within the database - or `None` if no match was found.

If we retrieve a `User` object, we can then check if the account is active or inactive - and return the appropriate response to the client's browser.

However, if an invalid form is sent, because the user did not add both a username and password the login form is presented back to the user with form error messages (i.e. username/password is missing).

Of particular interest in the code sample above is the use of the built-in Django machinery to help with the authentication process. Note the use of the `authenticate()` function to check whether the username and password provided match to a valid user account, and the `login()` function to signify to Django that the user is to be logged in.

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 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 i.e. OK. See the official

Django documentation on Redirection, to learn more.

All of these functions and classes are provided by Django, and as such you'll need to import them, so add the following imports to `rango/views.py`:

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

9.5.2. Creating a Login Template

With our new view created, we'll need to create a new template allowing users to login. 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. In your new template file, add the following code:

```
<!DOCTYPE html>
<html>
    <head>
        <!-- Is anyone getting tired of repeatedly entering the header
        <title>Rango</title>
    </head>

    <body>
        <h1>Login to Rango</h1>

        <form id="login_form" method="post" action="/rango/login/">
            {% csrf_token %}
            Username: <input type="text" name="username" value="" size
            <br />
            Password: <input type="password" name="password" value="" 
            <br />

            <input type="submit" value="submit" />
        </form>

    </body>
</html>
```

Ensure that you match up the input `name` attributes to those that you specified in the `user_login()` view - i.e. `username` for the username, and `password` for password.

Don't forget the `{% csrf_token %}`, either!

9.5.3. 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` file so that its `urlpatterns` tuple now looks like the code below:

```
urlpatterns = patterns('',
    url(r'^$', views.index, name='index'),
    url(r'^about/$', views.about, name='about'),
    url(r'^category/(?P<category_name_url>\w+)$', views.category, name
    url(r'^add_category/$', views.add_category, name='add_category'),
    url(r'^category/(?P<category_name_url>\w+)/add_page/$', views.add_
    url(r'^register/$', views.register, name='register'),
    url(r'^login/$', views.user_login, name='login'),
    )
```

9.5.4. 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 `index.html` template inside of the `templates/rango/` directory. Find the previously created category addition and registration links, and add the following hyperlink underneath. You may wish to include a line break (`<br />`) before the link.

```
<a href="/rango/login/">Login</a>
```

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 header, as shown in the code snippet below.

```
<h1>Rango says..hello world!</h1>
```

Replace this header with the following markup and Django template code. Note that we make use of the `user` object, which is available to Django's template system via the context. 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.

```
{% if user.is_authenticated %}
<h1>Rango says... hello {{ user.username }}!</h1>
{% else %}
<h1>Rango says... hello world!</h1>
{% endif %}
```

As you can see we have used Django's Template Language to check if the user is authenticated with `{% if user.is_authenticated %}`. The context variable which we pass through to the template will include a user variable if the user is logged in - so we can check whether they are authenticated or not. If so they will receive a personalised greeting in the header, i.e. `Rango says... hello leifos!`. Otherwise, the generic `Rango says... hello world!` header is displayed.

9.5.5. Demo

Check out Figure 2 for screenshots of what everything should look like.

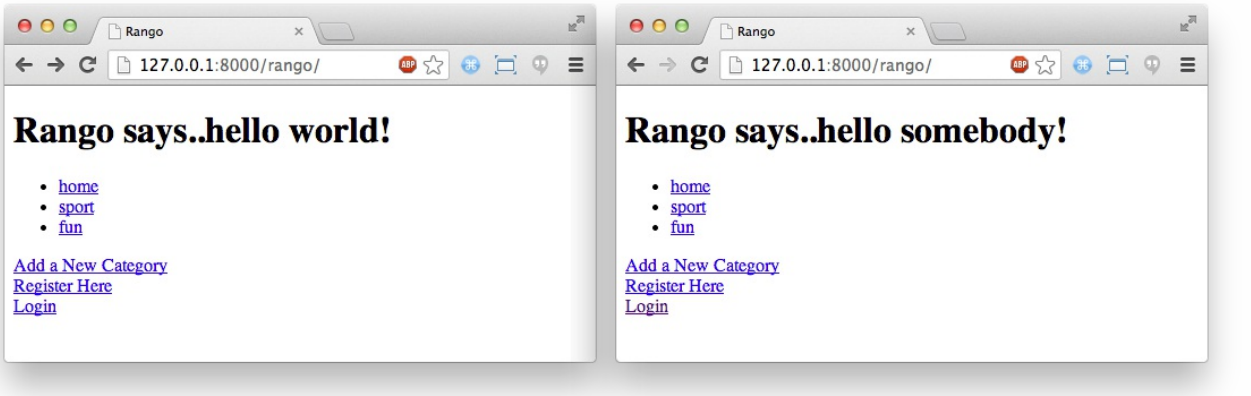


Figure 2: Screenshots illustrating the header users receive when not logged in, and logged in with username `somebody`.

With this completed, user logins should now be completed! 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.

9.6. 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 two ways in which we can achieve this goal:

- directly, by examining the `request` object and check if the user is authenticated, or,
- using a convenience decorator function that check 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 second 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 they try to access a page which calls that view, then the user is redirected to another page which you can set, typically the login page.

9.6.1. Restricting Access with a Decorator

To try this out, create a view in Rango's `views.py` file, 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. To use the decorator you will have to import it, so also add the following import:

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

We'll also add in another pattern to Rango's `urlpatterns` tuple in the `urls.py` file. Our tuple should then look something like the following example. Note the inclusion of mapping of the `views.restricted` view - this is the mapping you need to add.

```
urlpatterns = patterns('',
    url(r'^$', views.index, name='index'),
    url(r'^add_category/$', views.add_category, name='add_category'),
    url(r'^register/$', views.register, name='register'),
    url(r'^login/$', views.user_login, name='login'),
    url(r'^(?P<category_name_url>\w+)', views.category, name='category
    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 his or her browser. Django allows us to specify this in our project's `settings.py` file, 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/`.

9.7. 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 user is logged out, that their session is ended, and that if they subsequently

try to access a view, it will deny them access.

To provide log out functionality in `rango/views.py` add the a view called `user_logout()` with the following code:

```
from django.contrib.auth import logout

# Use the login_required() decorator to ensure only those logged in ca
@login_required
def user_logout(request):
    # Since we know the user is logged in, we can now just log them ou
    logout(request)

    # Take the user back to the homepage.
    return HttpResponseRedirect('/rango/')
```

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

```
urlpatterns = patterns('',
    url(r'^$', views.index, name='index'),
    url(r'^about/$', views.about, name='about'),
    url(r'^category/(?P<category_name_url>\w+)$', views.category, name
    url(r'^add_category/$', views.add_category, name='add_category'),
    url(r'^category/(?P<category_name_url>\w+)/add_page/$', views.add_
    url(r'^register/$', views.register, name='register'),
    url(r'^login/$', views.user_login, name='login'),
    url(r'^restricted/$', views.restricted, name='restricted'),
    url(r'^logout/$', views.user_logout, name='logout'),
    )
```

Now that all the machinery for logging a user out has been completed, 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 `index.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 HTML markup and Django template code. Note we also add a link to our restricted page at `/rango/restricted/`.

```
{% if user.is_authenticated %}
<a href="/rango/restricted/">Restricted Page</a><br />
<a href="/rango/logout/">Logout</a><br />
{% else %}
<a href="/rango/register/">Register Here</a><br />
<a href="/rango/login/">Login</a><br />
{% endif %}

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

Simple - 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.

9.8. Exercises

This chapter has 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.

- Customise the application so that only registered users can add/edit, while non-registered can only view/use the categories/pages. You'll also have ensure links to add/edit pages appear only if the user browsing the website is logged in.
- Provide informative error messages when users incorrectly enter their username or password.

In most applications you are going to require different levels of security when registering and managing users - for example, making sure the user enters an email address that they have access to, or sending users passwords that they have forgotten. While we could extend the current approach and build all the necessary infrastructure to support such functionality a `django-registration-redux` application has been developed which greatly simplifies the process - visit https://django-registration-redux.readthedocs.org to find out more about using this package. Templates can be found at: https://github.com/macdhuibh/django-registration-templates

Navigation

How To Tango With Django 1.7 »

previous next

10. Working with Templates

So far we've created several Django HTML templates for different pages in the application. You've probably already noticed that there is a lot of repeated HTML code in these templates.

While most sites will have lots of repeated structure (i.e. headers, sidebars, footers, etc) repeating the HTML in each template is a not good way to handle this. So instead of doing the same cut and paste hack job, we can minimize the amount of repetition in our code base by employing template inheritance provided by Django's Template Language.

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

1. Identify the re-occurring parts of each page that are repeated across your application (i.e. header bar, sidebar, footer, content pane)
2. In a base template, provide the skeleton structure of a standard 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), and then define a number of blocks which are subject to change depending on which page the user is viewing.
3. Create specific templates - all of which inherit from the base template - and specify the contents of each block.

10.1. 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>

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

```
    <body>
        <!-- Page specific content goes here -->
    </body>
</html>
```

Let's make this our base template, for the time being, and save it as `base.html` in the `templates` directory (e.g. `templates/base.html`).

Note

You should always aim to extract as much reoccurring content for your base templates. While it may be a bit more of a challenge for you to do initially, the time you will save in maintenance of your templates in the future far outweighs the initial overhead. Think about it: would you rather maintain one copy of your markup or multiple copies?

Warning

Remember that your page `<!DOCTYPE html>` declaration absolutely must be placed on the first line for your page! Not doing so will mean your markup will not comply with the W3C HTML5 guidelines.

10.2. Template Blocks

Now that we've identified our base template, we can prepare it for our inheriting templates. To do this, we need to include a Template Tag to indicate what can be overridden in the base template - this is done through the use of blocks.

Add a `body_block` to the base template as follows:

```
<!DOCTYPE html>

<html>
    <head lang="en">
        <meta charset="UTF-8">
        <title>Rango</title>
    </head>

    <body>
        {% block body_block %}{% endblock %}
    </body>
```

```
</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, for example:

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

When we create templates for each page we will inherit from `base.html` and override the contents of the `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 footer, a sidebar, etc. Blocks are a really powerful feature of Django's template system to learn more about them check out the official Django documentation on templates.

10.2.1. Abstracting Further

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

```
<!DOCTYPE html>

<html>
    <head>
        <title>Rango - {% block title %}How to Tango with Django!{% en
    </head>

    <body>
        <div>
            {% block body_block %}{% endblock %}
        </div>

        <hr />

        <div>
```

```
            <ul>
            {% if user.is_authenticated %}
                <li><a href="/rango/restricted/">Restricted Page</a></
                <li><a href="/rango/logout/">Logout</a></li>
                <li><a href="/rango/add_category/">Add a New Category<
            {% else %}
                <li><a href="/rango/register/">Register Here</a></li>
                <li><a href="/rango/login/">Login</a></li>
            {% endif %}

                <li><a href="/rango/about/">About</a></li>
            </ul>
        </div>
    </body>
</html>
```

We have introduced two new features into the template.

- The first is a new Django template block, `title`. This will allow us to specify a custom page title for each page inheriting from our base template. If an inheriting page does not make use of this feature, the title is defaulted to `Rango - How to Tango with Django!`
- We also bring across the list of links from our current `index.html` template and place 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 between the `body_block` content and the links.

Also note that we enclose the `body_block` within a HTML `<div>` tag - we'll be explaining the meaning of the `<div>` tag in Chapter 24. Our links are also converted to an unordered HTML list through use of the `<ul>` and `<li>` tags.

10.3. Template Inheritance

Now that we've created a base template with a block, we can now update the templates we have created to inherit from the base template. For example, let's refactor 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 'base.html' %}
```

The `extends` command takes one parameter, the template which is to be extended/inherited from (i.e. `rango/base.html`). We can then modify the `category.html` template so it looks like the following complete example.

Note

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`.

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

{% load staticfiles %}

{% block title %}{{ category_name }}{% endblock %}

{% block body_block %}
    <h1>{{ category_name }}</h1>
    {% if category %}
        {% 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 %}

        {% if user.is_authenticated %}
                <a href="/rango/category/{{category.slug}}/add_page/">
                {% endif %}
        {% else %}
                 The specified category {{ category_name }} does not e
    {% endif %}
```

```
{% endblock %}
```

Now that we inherit from `base.html`, all that exists within the `category.html` template is the `extends` command, the `title` block and the `body_block` block. 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 HTML document which is sent to the client's browser.

Note

Templates are very powerful and you can even create your own template tags. Here we have shown how we can minimise the repetition of structure HTML in our templates.

However, 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 webpages. This then saves you from having to call the Django ORM functions which gather the required data for the template in every view that renders it.

To learn more about the extensive functionality offered by Django's template language, check out the official Django documentation on templates.

10.4. Referring to URLs in Templates

So far we have been directly coding the URL of the page/view we want to show within the template, i.e. `<a href="/rango/about/"> About  </a>`. However, the preferred way is to use the template tag `url` to look up the url in the `urls.py` files. To do this we can change the way we reference the URL as follows:

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

The Django template engine will look up the `urls.py` files for a url with the `name='about'` (and then reverse match the actual url). This means if we change the url mappings in `urls.py` then we do not have to go through all the templates and update them. If we had not given our urlpattern a name, we could directly reference

it as follows:

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

Here we need to specify the application, and the view about.

You can now update the base template with the `url` template tag so that links in base template are rendered using the following code:

```
<div>
        <ul>
    {% if user.is_authenticated %}
        <li><a href="{% url 'restricted' %}">Restricted Page</a></li>
        <li><a href="{% url 'logout' %}">Logout</a></li>
        <li><a href="{% url 'add_category' %}">Add a New Category</a></
    {% else %}
        <li><a href="{% url 'register' %}">Register Here</a></li>
        <li><a href="{% url 'login' %}">Login</a></li>
    {% endif %}

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

In your `index.html` template you will notice that you have a parameterized url pattern, i.e. the `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, i.e. {% url 'category' category.slug %} within the template, as follows:

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

#TODO(leifos): The official tutorial provides an overview of how to use the url template tag, http://django.readthedocs.org/en/latest/intro/tutorial03.html and the answer at stackoverflow was helpful too: http://stackoverflow.com/questions/4599423/using-url-in-django-templates

#TODO(leifos): Also point out how the urls can be placed in a namespace and

referenced accordingly, see
http://django.readthedocs.org/en/latest/intro/tutorial03.html

10.5. Exercises

Now that you've worked through this chapter, we've got several exercises for you to work through. After completing them, you'll be a Django templating pro.

- Update all other existing templates within Rango's repertoire to extend from the `base.html` template. Follow the same process as we demonstrated above. Once completed, your templates should all inherit from `base.html`, as demonstrated in Figure 1. 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.
- Convert the restricted page to use a template. Call the template `restricted.html`, and ensure that it too extends from our `base.html` template.
- Change all the references to rango urls to use the url template tag.
- Add another link to our growing link collection that allows users to navigate back to Rango's homepage from anywhere on the website.

Warning

Remember to add `{% load static %}` 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 - you can't make use of modules included in templates you extend from!

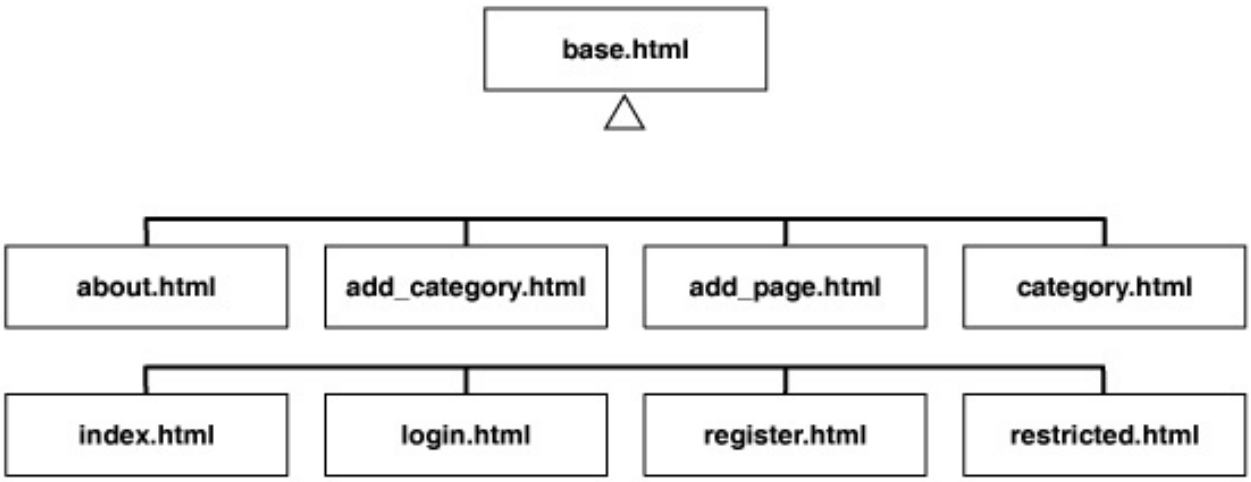


Figure 1: A class diagram demonstrating how your templates should inherit from *base.html*.

Note

Upon completion of these exercises, all of Rango's templates should inherit from `base.html`. Looking back at the contents of `base.html`, the `user` object - found within the context of a given Django request - is used to determine if the current user of Rango is logged in (through use of `user.is_authenticated`). As all of Rango's templates should inherit from this base template, we can say that all of Rango's templates now depend on having access to the context of a given request.

Due to this new dependency, you must check each of Rango's Django views. For each view, ensure that the context for each request is made available to the Django template engine. Throughout this tutorial, we've been using `render_to_response()` to achieve this. If you don't ensure this happens, your views may be rendered incorrectly - users may appear to be not logged in, even though Django thinks that they are!

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 access to objects such as `user`, which will allow the template engine to determine if the user is logged in (ie. authenticated).

```
def about(request):

    return render(request, 'rango/about.html', {})
```

Remember, the last parameter of `render()` is a dictionary with which you can use to pass additional data to the Django template engine. As we have no additional data to pass through we pass through an empty dictionary. Have a look at Section 5.1.3 to refresh your memory on `render()`.

Navigation

How To Tango With Django 1.7 »

previous next

11. Cookies and Sessions

In this chapter, we will be going through sessions and cookies, both of which go hand in hand, 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 (all behind the scenes). Here we will explore how to explicitly use cookies for other purposes.

11.1. Cookies, Cookies Everywhere!

If you are already comfortable with the ideas and concepts behind cookies, then skip straight to Section 11.2, if not read on.

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 to the client, which are in turn stored in a persistent browser cache. When a user requests a new page from the same web server, any cookies that are matched to that server are sent with the request. The server can then interpret the cookies as part of the request's context and generate a response to suit.

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.

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.

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 - does the information you want to store as a cookie really need to be sent? In many cases, there are alternate - and 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 unwise. What if the user's computer is compromised? The cookie could be taken by a malicious program. From there, hackers would have his or her credit card number - all because your web application's design is fundamentally flawed. You'll however be glad to know that a majority of websites use cookies for application specific functionality.

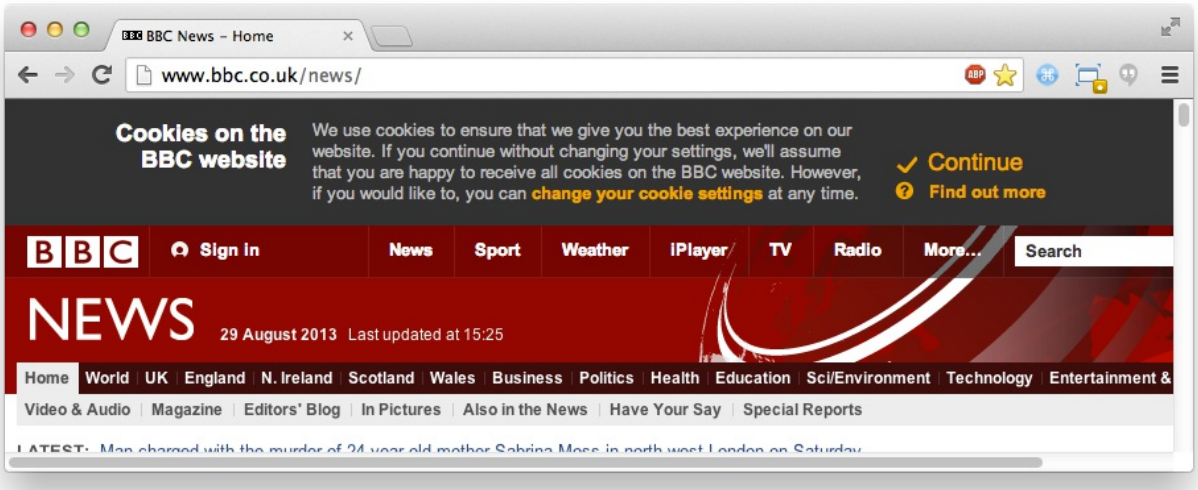


Figure 1: A screenshot of the BBC News website (hosted in the United Kingdom) with the cookie warning message presented at the top of the page.

Note

Note 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. Check out Figure 1, demonstrating such a warning on the BBC News website. You can read about the law here.

11.2. Sessions and the Stateless Protocol

All correspondence between web browsers (clients) and servers is achieved through the HTTP protocol. As we very briefly touched upon in Chapter 8, HTTP is a stateless protocol. This therefore 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 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 (and frustrating) - 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) 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, 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 your browser supports cookies, pretty much all websites create a new session for you when you visit. You can see this for yourself now - check out Figure 2. In Google Chrome's developer tools, you can view cookies which are sent by the web server you've accessed. In Figure 2, you can observe the selected cookie `sessionid`. The cookie contains a series of letters and numbers which Django uses to uniquely identify your session. From there, all your session details can be accessed - but only on the server side.

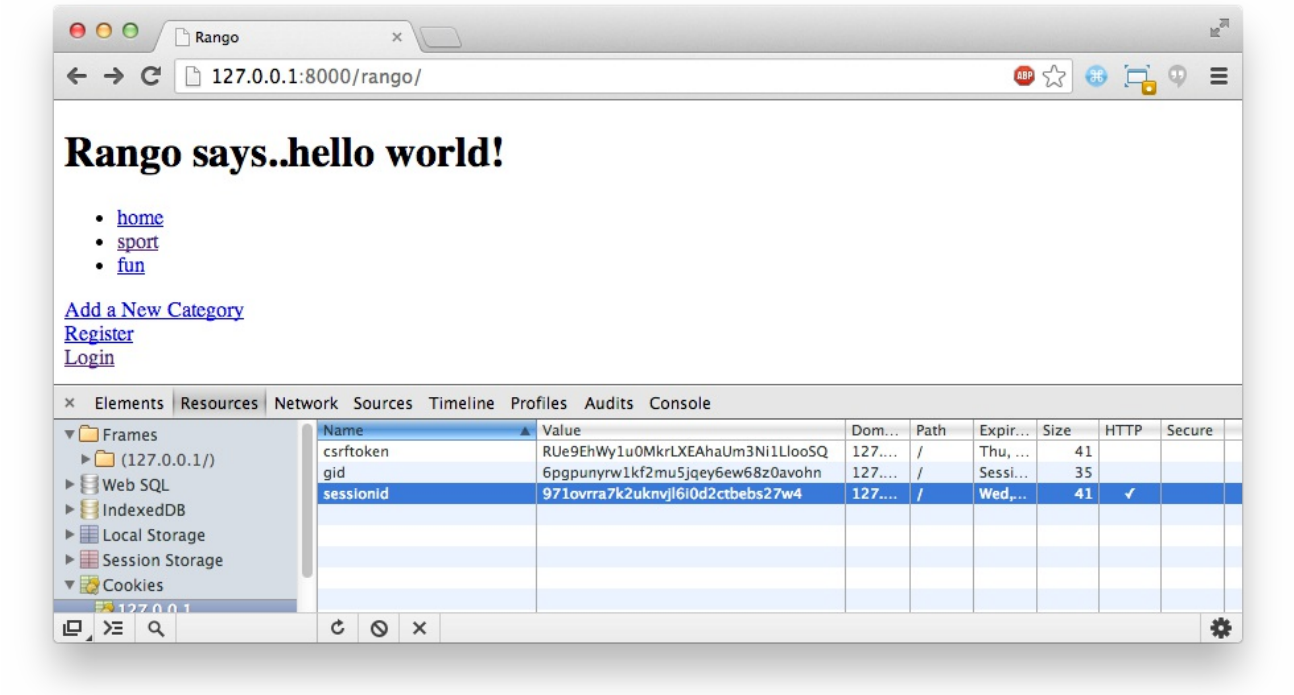


Figure 2: A screenshot of Google Chrome with the Developer Tools opened - check out the cookie `sessionid`...

Session IDs don't have to be stored with cookies, either. Legacy PHP applications typically include them as a querystring, or part of the URL to a given resource. If you've ever come across a URL like `http://www.site.com/index.php?sessid=omgPhPwtfIsThisIdDoingHere332i942394`, that's probably uniquely identifying you to the server. Interesting stuff!

Note

Have a closer look at Figure 2. Do you notice the token `csrftoken`? This cookie is to help prevent any cross-site forgery.

11.3. 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_CLASSES` tuple. You should find the `django.contrib.sessions.middleware.SessionMiddleware` module listed as a

string in the tuple - if you don't, add it to the tuple now. It is the `SessionMiddleware` middleware which 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 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. If you add the application now, you'll need to update your database with the migration commands.

Note

If you are looking for lightning fast 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.

11.4. A Cookie Tasting Session

We can now test out whether your browser supports cookies. While all modern web browsers do support cookies it is worthwhile checking your browser's settings regarding cookies. If you have your browser's security level set to a high level, certain cookies may get blocked. Look up your browser's documentation for more information, and enable cookies.

11.4.1. 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 a 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 `register()`.

You'll need to make sure that you are logged out of Rango if you've implemented the user authentication functionality. 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. After we successfully determine that cookies are indeed working, we can remove the code we add to restore the two views to their previous state.

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, outside any conditional blocks.

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

In the `register()` view, add the following three lines to the top of the function - again, to ensure that they are executed.

```
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/`. Once the page is loaded, navigate to the registration page. When the registration page is loaded, you should see `>>>> TEST COOKIE WORKED!` appear in your Django development server's console, like in Figure 3. If you do, everything works as intended!

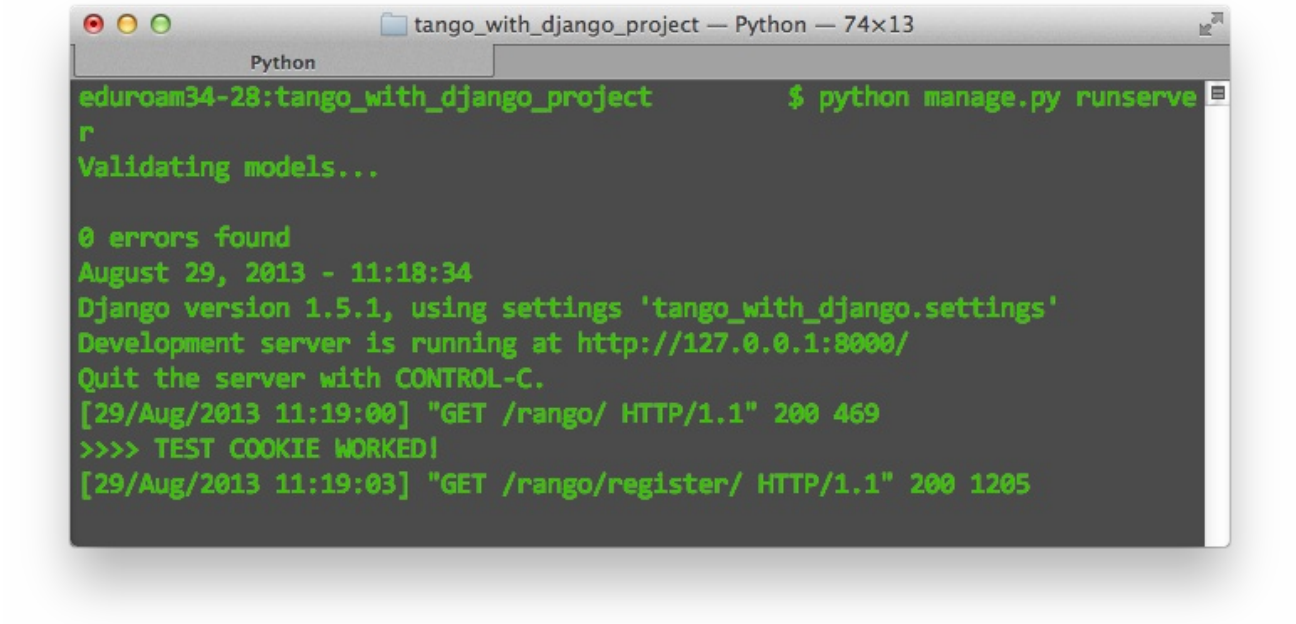


Figure 3: A screenshot of the Django development server's console output with the `>>>> TEST COOKIE WORKED!` message.

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.

Note

You can delete the code you added in this section - we only used it to demonstrate cookies in action.

11.5. Client Side Cookies: A Site Counter Example

Now we know 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 website, 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.

The sensible place to assume a user enters the Rango site is at the index page. Open `rango/views.py` and edit the `index()` view as follows:

```
def index(request):

    category_list = Category.objects.all()
    page_list = Page.objects.order_by('-views')[:5]
```

```
context_dict = {'categories': category_list, 'pages': page_list}

    # 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, we default to zero and cast that.
    visits = int(request.COOKIES.get('visits', '0'))

        reset_last_visit_time = False

    # Does the cookie last_visit exist?
    if 'last_visit' in request.COOKIES:
        # Yes it does! Get the cookie's value.
        last_visit = request.COOKIES['last_visit']
        # Cast the value to a Python date/time object.
        last_visit_time = datetime.strptime(last_visit[:-7], "%Y-%m-%d

        # If it's been more than a day since the last visit...
        if (datetime.now() - last_visit_time).days > 0:
            visits = visits + 1
            # ...and flag that the cookie last visit needs to be updat
                        reset_last_visit_time = True
    else:
        # Cookie last_visit doesn't exist, so flag that it should be s
        reset_last_visit_time = True


        context_dict['visits'] = visits

    # Obtain our Response object early so we can add cookie informatio
        response = render(request, 'rango/index.html', context_dict)
        if reset_last_visit_time:
                response.set_cookie('last_visit', datetime.now())
        response.set_cookie('visits', visits)

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

For reading through the code, you will see that a majority of the code deals with checking the current date and time. For this, you'll need to include Python's `datetime` module by adding the following import statement at the top of the `views.py` file.

```
from datetime import datetime
```

Make sure you also import the `datetime` object within the `datetime` module.

In the added code we check to see if the cookie `last_visit` exists. If it does, we can take the value from the cookie using the syntax `request.COOKIES['cookie_name']`, where `request` is the name of the `request` object, and `'cookie_name'` is the name of the cookie you wish to retrieve. **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. 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.

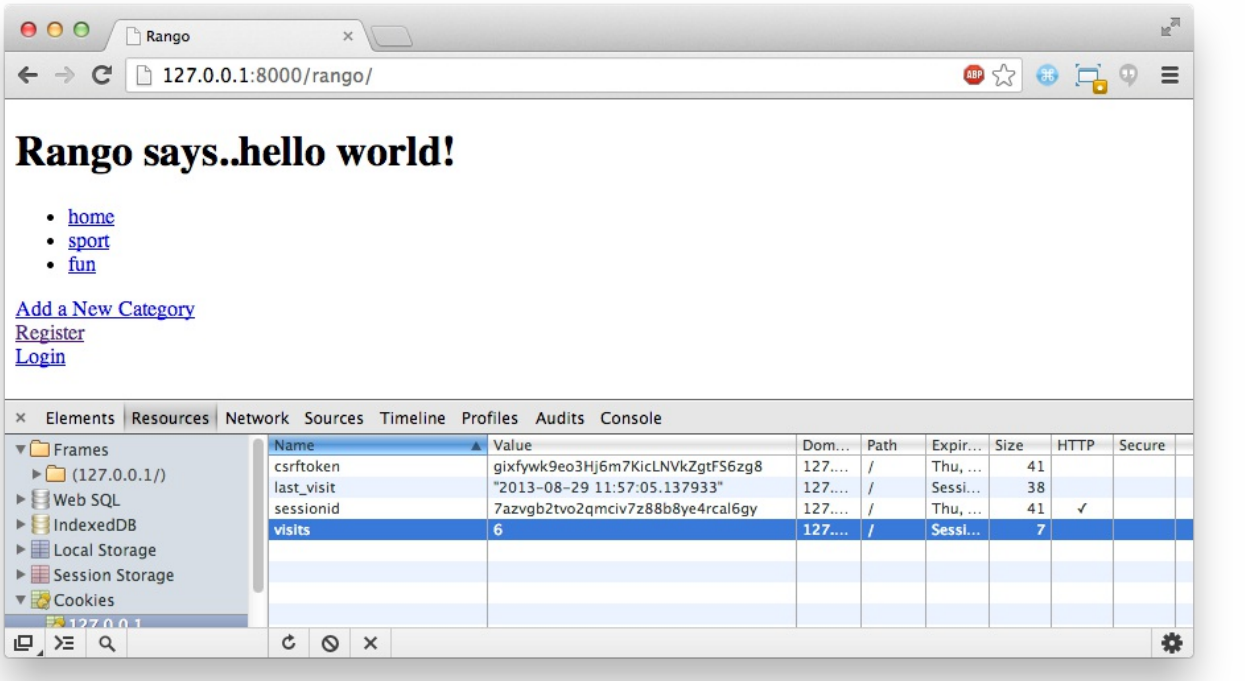


Figure 4: A screenshot of Google Chrome with the Developer Tools open showing the cookies for Rango. Note the *`visits`* cookie - the user has visited a total of six times, with each visit at least one day apart.

Now if you visit the Rango homepage, and inspect the developer tools provided by your browser, you should be able to see the cookies `visits` and `last_visit`. Figure

4 demonstrates the cookies in action.

Note

You may notice that the `visits` cookie doesn't increment when you refresh your web browser. Why? The sample code we provide above only increments the counter at least one whole day after a user revisits the Rango homepage. This is an unacceptable time to wait when testing - so why not temporarily change the delay to a shorter time period? In the updated `index` view, find the following line.

```
if (datetime.now() - last_visit_time).days > 0:
```

We can easily change this line to compare the number of seconds between visits. In the example below, we check if the user visited at least five seconds prior.

```
if (datetime.now() - last_visit_time).seconds > 5:
```

This means you need only wait five seconds to see your `visits` cookie increment, rather than a whole day. When you're happy your code works, you can revert the comparison back to the original per-day timespan.

Being able to find the difference between times using the `-` operator is one of the many awesome features that Python provides. When times are subtracted, a `timedelta` object is returned, which provides the `days` and `seconds` attributes we use in the code snippets above. You can check out the official Python documentation for more information on this type of object, and what other attributes it provides.

Instead of using the developer tools you can update the `index.html` and add, `<p>visits: {{ visits }}</p>` to show the number of visits.

11.6. Session Data

In the previous example, we used client side cookies. 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 which is stored on the client side (but is effectively anonymous) as the key to unlock the data.

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

1. Make sure that `MIDDLEWARE_CLASSES` in `settings.py` 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 another setup (i.e. a cache). See the official Django Documentation on Sessions for other backend configurations.

Now instead of storing the cookies directly in the request (and thus on the client's machine), you can access the server side cookies via the method call `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 data is stored serve side. Below we have updated the `index()` function with the session based cookies:

```
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}

    visits = request.session.get('visits)
        if not visits:
                visits = 0
    reset_last_visit_time = False

    last_visit = request.session.get('last_visit')
    if last_visit:
            last_visit_time = datetime.strptime(last_visit[:-7], "%Y-%
        if (datetime.now() - last_visit_time).seconds > 0:
            # ...reassign the value of the cookie to +1 of what it was
            visits = visits + 1
            # ...and update the last visit cookie, too.
            reset_last_visit_time = True
    else:
        # Cookie last_visit doesn't exist, so create it to the current
        reset_last_visit_time = True

    context_dict['visits'] = visits
        request.session['visits'] = visits
        if reset_last_visit_time:
                request.session['last_visit'] = str(datetime.now())
```

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

    return response
```

Warning

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 developer tools by deleting each cookie individually, or simply clear your browser's cache entirely - ensuring that cookies are deleted in the process.

Note

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.

11.7. 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` file. 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 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.

11.8. Clearing the Sessions Database

Session cookies accumulate. So if you are using the database backend you will have to periodically clear the database that stores the cookies. This can be done using `python manage.py clearsessions`. The Django documentations suggests running this daily as a cron job. See https://docs.djangoproject.com/en/1.7/topics/http/sessions/#clearing-the-session-store

11.9. Basic Considerations and Workflow

When using cookies within your Django application, there's 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 which 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. This can be done by checking the `request` parameter. 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.
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. Make sure that `MIDDLEWARE_CLASSES` in `settings.py` contains 'django.contrib.sessions.middleware.SessionMiddleware'.
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>']`

11.10. Exercises

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

- Change your cookies from client side to server side to make your application more secure. 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.
- Update the About page view and template telling the visitors how many times they have visited the site.

11.10.1. Hint

To aid you in your quest to complete the above exercises, the following hint may help you.

You'll have to pass the value from the cookie to the template context for it to be rendered as part of the page, as shown in the example below.

```
# If the visits session varible exists, take it and use it.
# If it doesn't, we haven't visited the site so set the count to zero.
if request.session.get('visits'):
    count = request.session.get('visits')
else:
    count = 0

# remember to include the visit data
return render(request, 'rango/about.html', {'visits': count})
```

Footnotes

[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.

Navigation

How To Tango With Django 1.7 »

previous next

12. User Authentication with Django-Registration-Redux

There are numerous add-on applications that have been developed that provide login, registration and authentication mechanisms. Since most applications will provide such facilitaties there is little point re-writing / re-inventing the urls, views, and templates. In this chapter, we are going to use the package `django-registration-redux` to provide these facilities. This will mean we will need re-factor our code - however, it will provide with some experience of using external applications, how easily they can be plugged into your Django project, along with login facilities with all the bells and whistles. It will also make our application much cleaner.

Note

This chapter is not neccessary. You can skip it, but we will be assuming that you have upgraded the authentication mechanisms, in subsequent chapters.

12.1. Setting up Django Registration Redux

To start we need to first install `django-registration-redux`:

```
$ pip install django-registration-redux
```

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` tuple:

```
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:

```
REGISTRATION_OPEN = True                # If True, users can register
ACCOUNT_ACTIVATION_DAYS = 7     # One-week activation window; you may,
REGISTRATION_AUTO_LOGIN = True  # If True, the user will be automatica
LOGIN_REDIRECT_URL = '/rango/'  # The page you want users to arrive at
LOGIN_URL = '/accounts/login/'  # The page users are directed to if th
                                                                # and
```

These settings should be self explanatory. Now, in `tango_with_django_project/urls.py` you can update the `urlpatterns` so it includes a reference to the registration package:

```
urlpatterns = patterns('',

url(r'^admin/', include(admin.site.urls)),
url(r'^rango/', include('rango.urls')),
(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 are 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.

12.2. Functionality and URL mapping

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

- 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:

- activiation complete (used in the two-step registration) -> `activitate/complete/`
- activite (used if the account actiona fails) -> `activite/<activation_key>/`
- activation email (notifies the user an activation email has been sent out)
    - activation email body (a text file, that contains the activiation email text)
    - activation email subject (a text file, that contains the subject line of the activiation email)

Now the catch. While Django Registration Redux provides all this functionality, it does not provide the templates. So we need to provide the templates associated with each view.

12.3. Setting up the Templates

In the quickstart guide, see https://django-registration-redux.readthedocs.org/en/latest/quickstart.html, it provides an overveiw of what templates are required, but it is not immediately clear what goes within each template.

However, it is possible to download a set of tempaltes from Anders Hofstee's GitHub account, see https://github.com/macdhuibh/django-registration-templates, and from here you can see what goes into the templates. We will use these templates as our guide here.

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.

12.3.1. Login Template

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

```
{% extends "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' %}">
{% 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 assocaited with each url.

12.3.2. Registration Template

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

```
{% extends "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 %}
```

12.3.3. Registration Complete Template

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

```
{% extends "base.html" %}


{% block body_block %}
<h1>Registration Complete</h1>
        <p>You are now registered</p>
{% endblock %}
```

12.3.4. Logout Template

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

```
{% extends "base.html" %}


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

12.3.5. Try out the Registration Process

Run the runserver 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, and we still have some legacy code.

12.3.6. Refactoring your project

Now you will need to update the `base.html` so that the new registration url/views are used:

- Update register to point to `<a href="{% url 'registration_register' %}">`
- login to point to `<a href="{% url 'auth_login' %}">`, and
- 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 de-commission the `register`, `login`, `logout` functionality from the `rango` application, i.e. remove the urls, views, and templates (or comment them out).

12.3.7. 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. This can be done by overriding the `RegistrationView` provided by `registration.backends.simple.views`. To do this, the `tango_with_django_project/urls.py`, import `RegistrationView`, add in a new registration class and then update the urlpatterns as follows:

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

# Create a new class that redirects the user to the index page, if suc
class MyRegistrationView(RegistrationView):
    def get_success_url(selfself,request, user):
        return '/rango/'


urlpatterns = patterns('',
    url(r'^admin/', include(admin.site.urls)),
    url(r'^rango/', include('rango.urls')),
        #Add in this url pattern to override the default pattern in ac
    url(r'^accounts/register/$', MyRegistrationView.as_view(), name='r
    (r'^accounts/', include('registration.backends.simple.urls')),
)
```

#TODO(leifos): Add in a customized registration form..

12.4. Exercises

- Provide users with password reset functionality

Navigation

How To Tango With Django 1.7 »

previous next

13. Bootstrapping Rango

In this chapter, we will be styling Rango using the Twitter Bootstrap 3.2 toolkit. We won't go into the details about how Bootstrap works, and we will be assuming you have some familiarity with CSS. If you don't, check out the CSS chapter so that you understand the basics and then check out some Bootstrap tutorials. However, you should be able to go through this section and piece things together.

To get started take a look at the Bootstrap 3.2.0 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. Given the html from the bootstrap website we need to do a bit of work on it before we can use it. This is what we did, the resulting html, which you can copy is below:

- Replaced all references of `../../` to be `http://getbootstrap.com`
- Replaced `dashboard.css` with the absolute reference, `http://getbootstrap.com/examples/dashboard/dashboard.css`
- Removed the search form from the top nav 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 the 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 %}`

13.1. The New Base Template

```
<!DOCTYPE html>
```

```
<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
    <meta name="description" content="">
    <meta name="author" content="">
    <link rel="icon" href="http://getbootstrap.com/favicon.ico">

    <title>Rango - {% block title %}How to Tango with Django!{% endblo

    <link href="http://getbootstrap.com/dist/css/bootstrap.min.css" re
    <link href="http://getbootstrap.com/examples/dashboard/dashboard.c

    <!--[if lt IE 9]>
      <script src="https://oss.maxcdn.com/html5shiv/3.7.2/html5shiv.mi
      <script src="https://oss.maxcdn.com/respond/1.4.2/respond.min.js
    <![endif]-->
  </head>

  <body>

    <div class="navbar navbar-inverse navbar-fixed-top" role="navigati
      <div class="container-fluid">
        <div class="navbar-header">
          <button type="button" class="navbar-toggle collapsed" data-t
            <span class="sr-only">Toggle navigation</span>
            <span class="icon-bar"></span>
            <span class="icon-bar"></span>
            <span class="icon-bar"></span>
          </button>
          <a class="navbar-brand" href="/rango/">Rango</a>
        </div>
        <div class="navbar-collapse collapse">
          <ul class="nav navbar-nav navbar-right">
                <li><a href="{% url 'index' %}">Home</a></li>
                    {% if user.is_authenticated %}
                        <li><a href="{% url 'restricted' %}">Restricte
                        <li><a href="{% url 'auth_logout' %}?next=/ran
                        <li><a href="{% url 'add_category' %}">Add a N
                    {% else %}
                        <li><a href="{% url 'registration_register' %}
                        <li><a href="{% url 'auth_login' %}">Login</a>
```

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

            </ul>
        </div>
      </div>
    </div>

    <div class="container-fluid">
      <div class="row">
        <div class="col-sm-3 col-md-2 sidebar">
                {% block side_block %}{% endblock %}

        </div>
        <div class="col-sm-9 col-sm-offset-3 col-md-10 col-md-offset-2
           <div>
                {% block body_block %}{% endblock %}
                </div>
        </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/1.11.1/j
    <script src="http://getbootstrap.com/dist/js/bootstrap.min.js"></s
    <script src="http://getbootstrap.com/assets/js/docs.min.js"></scri
    <!-- IE10 viewport hack for Surface/desktop Windows 8 bug -->
    <script src="http://getbootstrap.com/assets/js/ie10-viewport-bug-w
  </body>
</html>
```

If you take a close look at the 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 and the main pane which are denoted by the two `<div class="container-fluid">``s. In the nav bar section, we have injected all the links to the different parts of our website. Inside the main pane, there are two columns, one for the ``side_block`, and the other for the `body_block`.

13.2. Quick Style Change

Update your `base.html` with the html code above (assumes you are using the django-registration-redux package, if not you will need to update those url template tags). Reload your application. Obviously you will need a connection to the internet in order to download the css, js, and other related files. You should notice that your application looks heaps better with this one changes. Flip through the different pages. Since they all inherit from base, they will all be looking pretty good. Not perfect, but pretty good.

Note

You could download all the associated files and stored them in your static folder. If you do this, simply update the base template to reference the static files stored locally.

Now that we have the `base.html` all set up and ready to go, we can do a really quick face lift to Rango by going through the Bootstrap components and selecting the ones that suit the pages.

Lets update the `about.html` template, by putting a page header on the page (http://getbootstrap.com/components/#page-header). From the example, all we need to do is provide an encapsulating `<div>` with the `class="page-header"`:

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

{% load staticfiles %}

{% block title %}About{% endblock %}

{% block body_block %}
    <div class="page-header">
                <h1>About</h1>
            </div>
            <div>
            <p></strong>.</p>

            <img  width="90" height="100" src="{% static "images/rango
            </div>
{% endblock %}
```

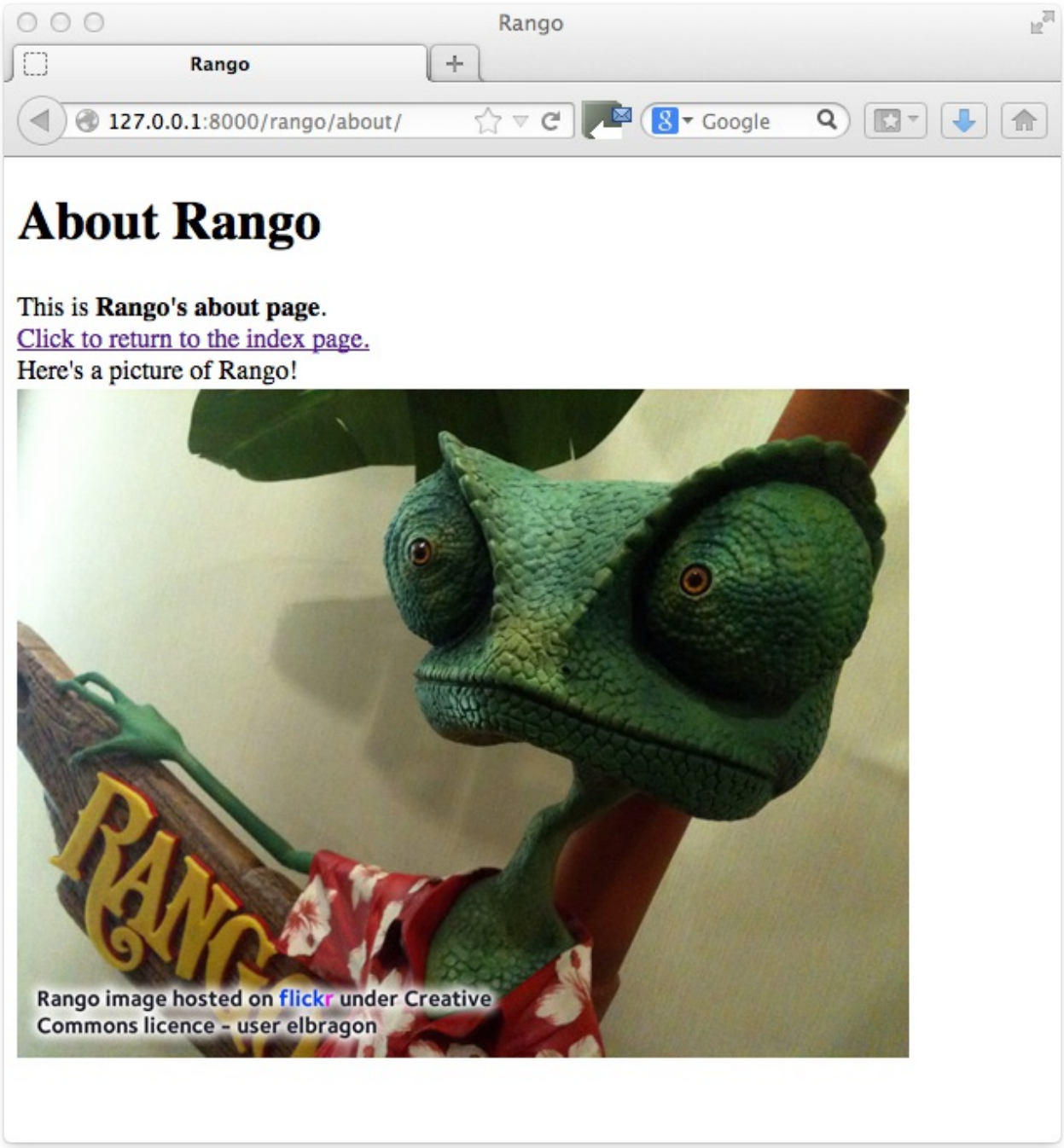


Figure 1: A screenshot of the About page without style.

#TODO(leifos):update this screen shot.

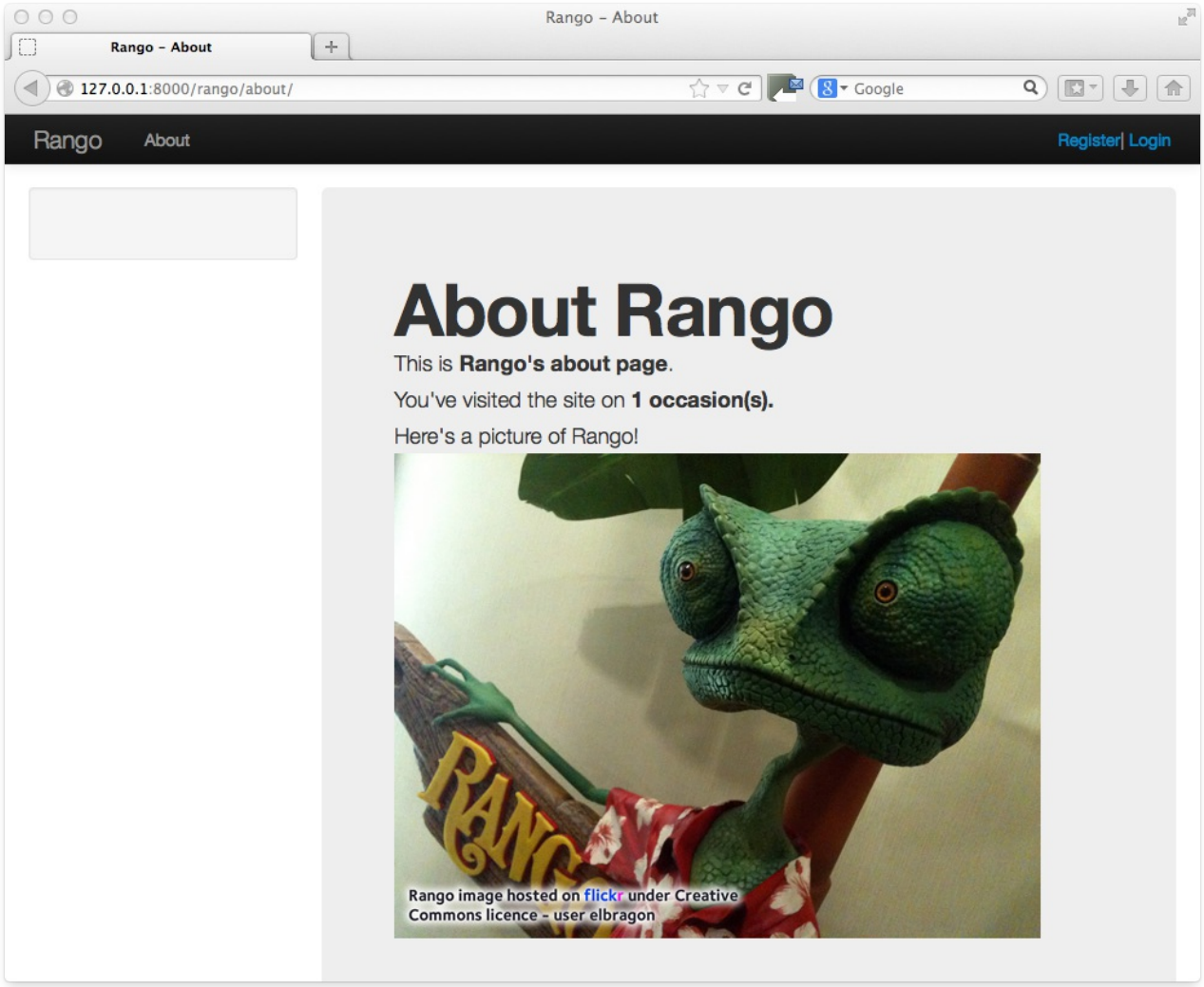


Figure 2: A screenshot of the About page with Bootstrap Styling applied.

#TODO(leifos):update this screen shot.

To each template, add in a page-header. Remember to update all the templates in both `rango` and `registration`.

> While the application looks much better, somethings look particularly out of place. For example on the registration page, the fields are not lined up, and the button looks like it is from the 20th century.

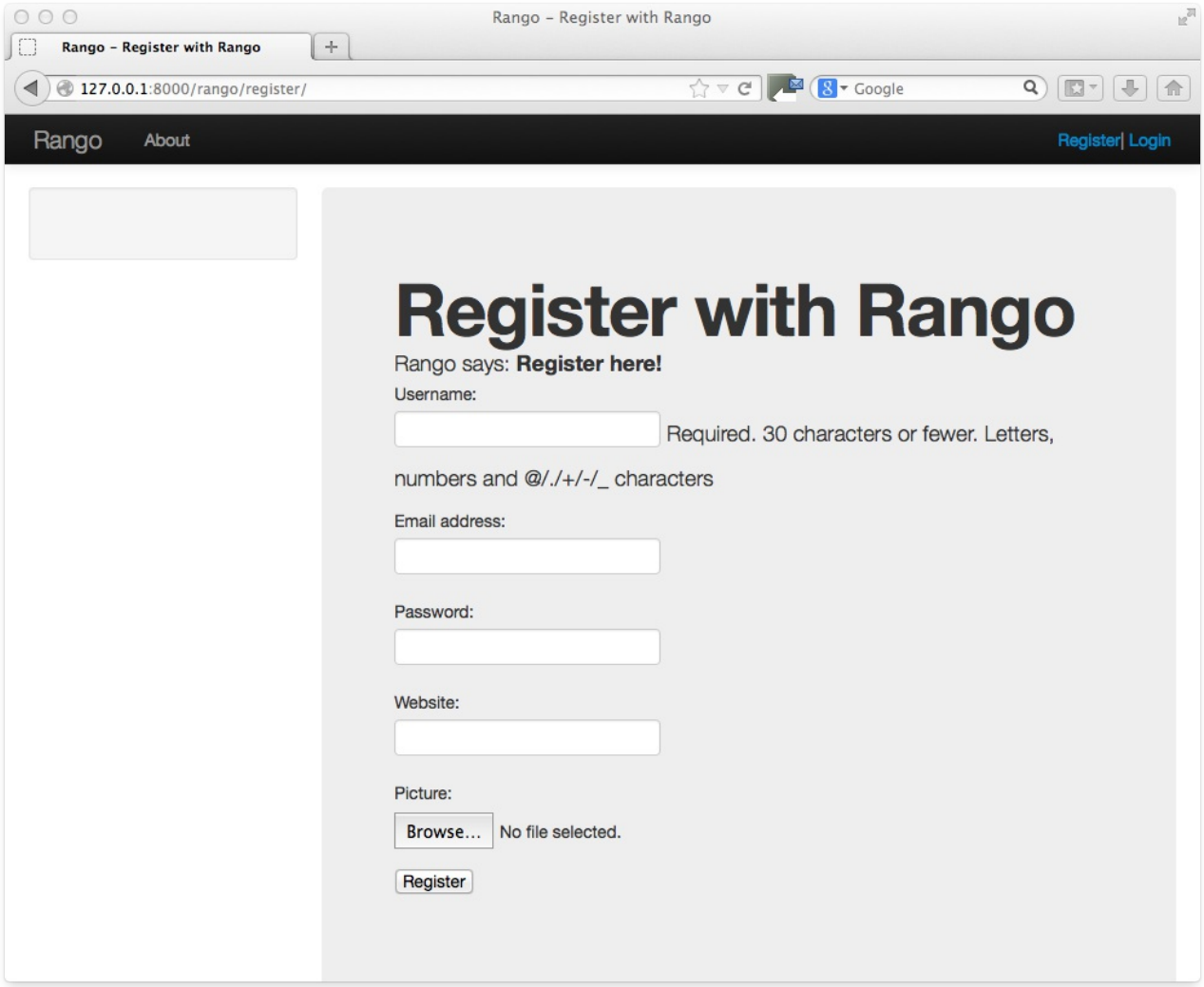


Figure 3: A screenshot of the Registration page with Bootstrap Styling applied but not customised.

#TODO(leifos):update this screen shot.

Also, you'll probably have noticed the sidebar is empty. In the next chapter we will sort that out with some handy navigation links. But first, lets sort out the Index page.

13.2.1. The Index Page

Since we have only encapsulated the title with a page header i.e. `<div class="page-header">`, we haven't really capitalised on the classes and styling that Bootstrap gives us. So here we have taken the columns from the fluid page and used them to house the top categories and top pages. Since the original page had four columns, we have taken two and made them bigger by adjusting the column sizes. Updatet the `index.html` template to look like the following:

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

```
{% load staticfiles %}

{% block title %}Index{% endblock %}

        {% block body_block %}
{% if user.is_authenticated %}
    <div class="page-header">

                <h1>Rango says... hello {{ user.username }}!</h1>
            {% else %}
                <h1>Rango says... hello world!</h1>
            {% endif %}
</div>

         <div class="row placeholders">
            <div class="col-xs-12 col-sm-6 placeholder">
               <h4>Categories</h4>

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

            </div>
            <div class="col-xs-12 col-sm-6 placeholder">
              <h4>Pages</h4>

                {% if pages %}
                    <ul>
                        {% for page in pages %}
                         <li><a href="{{page.url}}">{{ page.title }}</
                        {% endfor %}
                    </ul>
                {% else %}
                    <strong>There are no categories present.</strong>
                {% endif %}
            </div>
```

```
        </div>


       <p> visits: {{ visits }}</p>
        {% endblock %}
```

The page should look a lot better now. But the way the list items are presented is pretty horrible. Lets use the list-group style provided by Bootstrap, http://getbootstrap.com/components/#list-group. Change the `<ul>` elements to `<ul class="list-group">` and the `<li>` elements to `<li class="list-group-item">` then update the headings using a panel style:

```
    <div class="panel panel-primary">
    <div class="panel-heading">
            <h3 class="panel-title">Categories</h3>
    </div>
</div>


    <div class="panel panel-primary">
            <div class="panel-heading">
                    <h3 class="panel-title">Pages</h3>
            </div>
    </div>
```

Replacing `<h4>Categories</h4>` and `<h4>Pages</h4>` respectively. Now the page should look pretty neat.

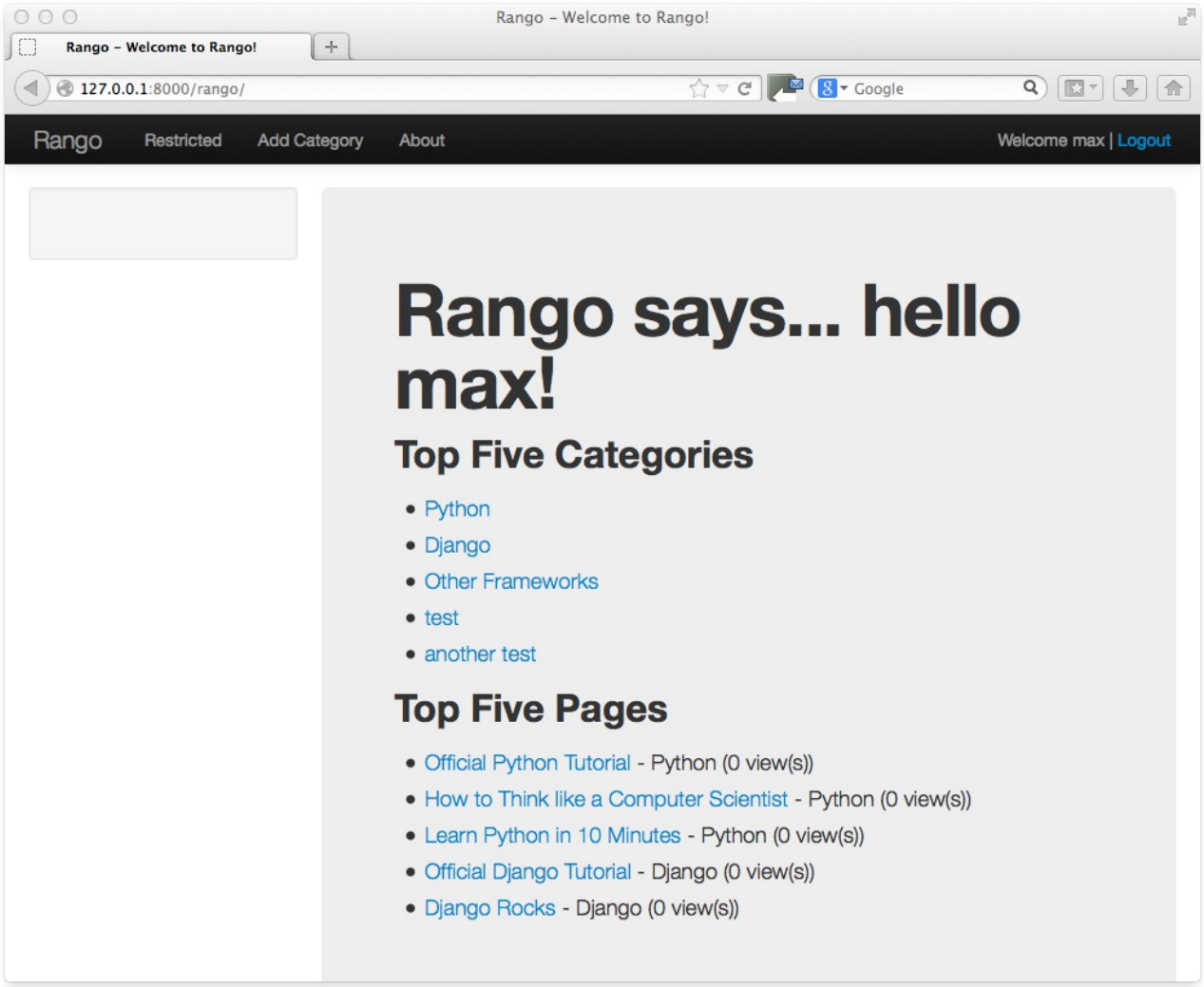


Figure 4: A screenshot of the Index page with a Hero Unit.

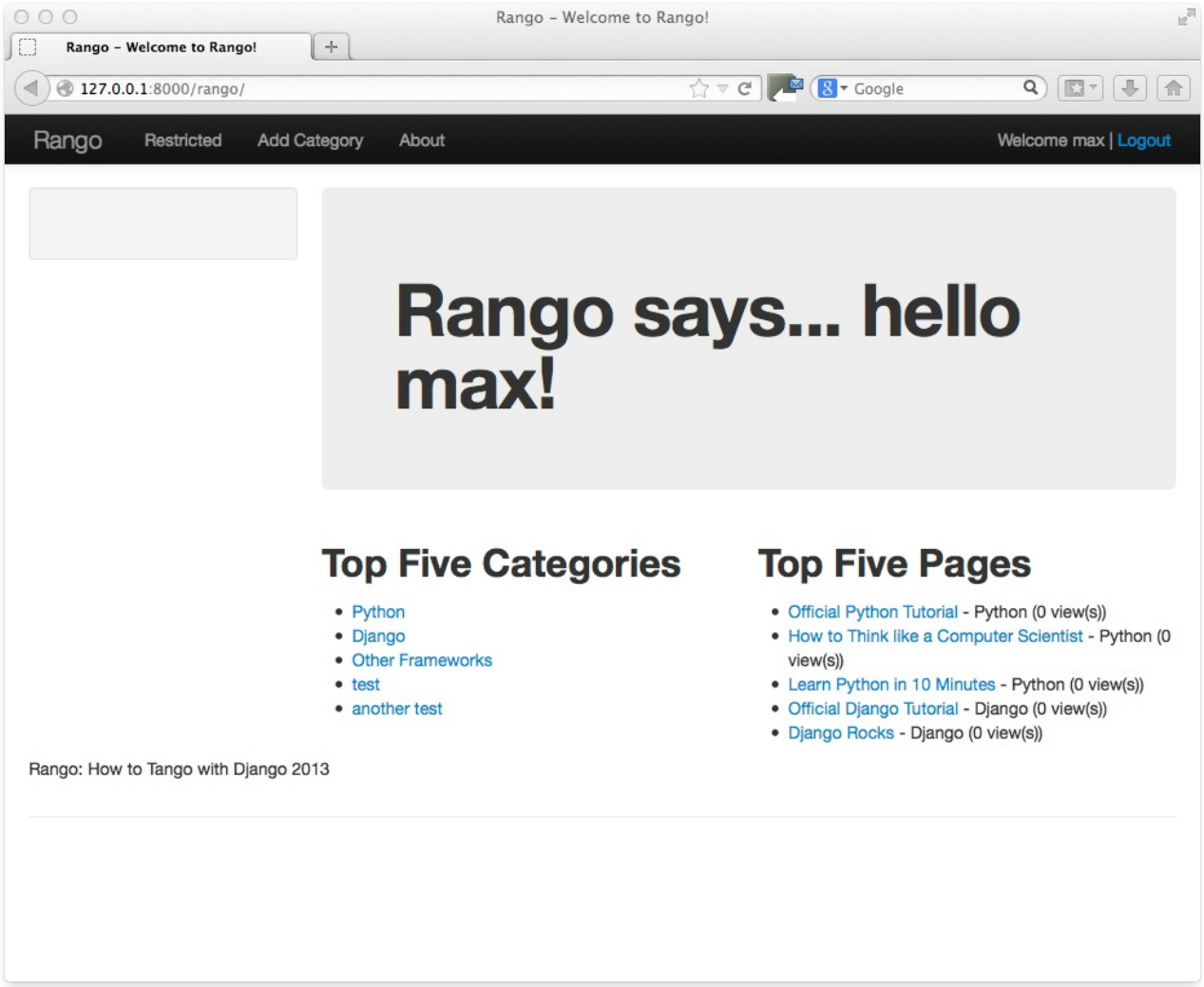


Figure 5: A screenshot of the Index page with customised Bootstrap Styling.

13.3. 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, see http://getbootstrap.com/examples/signin/. If you take a look at the source, you'll notice that there are a number of classes that we need to include to pimp out the basic login form. Update the `login.html` template as follows:

```
{% block body_block %}

<link href="http://getbootstrap.com/examples/signin/signin.css" rel="s

<form class="form-signin" role="form" method="post" action=".">
{% csrf_token %}

<h2 class="form-signin-heading">Please sign in</h2>
```

```
<input class="form-control" placeholder="Username" id="id_username" ma
<input type="password" class="form-control" placeholder="Password" id=

        <button class="btn btn-lg btn-primary btn-block" type="submit"
        </form>

{% endblock %}
```

Besided 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.

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 that can be assigned to buttons, see http://getbootstrap.com/css/#buttons.

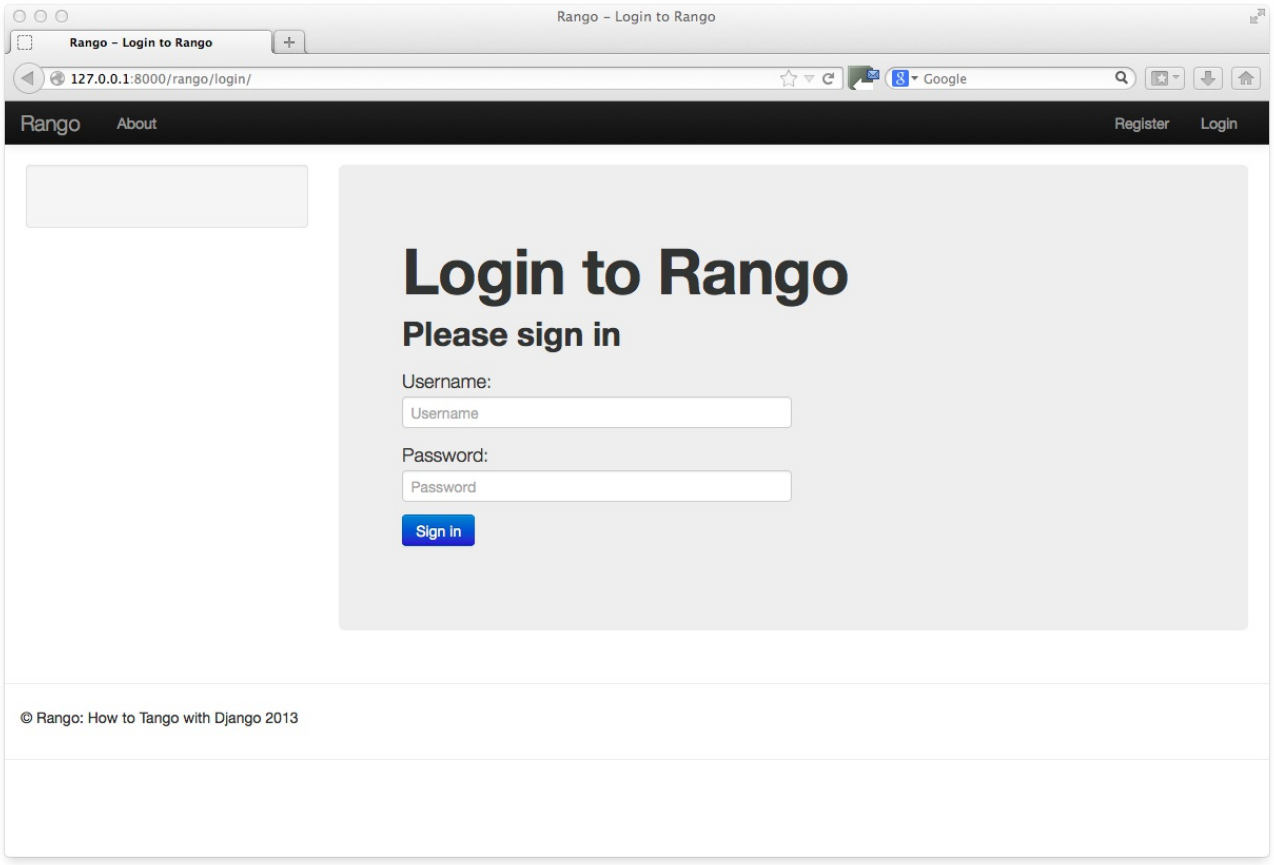


Figure 6: A screenshot of the login page with customised Bootstrap Styling.

#TODO(Leifos): update the screen shot

13.3.1. 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 'base.html' %}

{% block title %}Add Page{% endblock %}


{% block body_block %}
{% if category %}

                <form role="form"  id="page_form" method="post" action=
            <h2 class="form-signin-heading">Add a Page to <a href="/ra
                    {% 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
                </form>
            {%  else %}
            <p>This is category does not exist.</p>
        {%  endif %}


        {% endblock %}
```

And similarly for the `add_category.html` template (not shown).

13.4. The Registration Template

For the `registration_form.html`, we can update the form as follows:

```
{% extends "base.html" %}


{% block body_block %}
<form role="form"  method="post" action=".">
                {% csrf_token %}

<h2 class="form-signin-heading">Sign Up Here</h2>

<div class="form-group" >
 <p class="required"> <label for="id_username">Username:</label>
     <input class="form-control"  id="id_username" maxlength="30" name
</div>
 <div class="form-group">
    <p class="required"><label for="id_email">E-mail:</label>
        <input class="form-control" id="id_email" name="email" type="e
 </div>
<div class="form-group">
    <p class="required"><label for="id_password1">Password:</label>
        <input class="form-control" id="id_password1" name="password1"
</div>
<div class="form-group">
    <p class="required"><label for="id_password2">Password (again):</l
 <input class="form-control" id="id_password2" name="password2" type="
</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.

Note

I am not happy with this solution. I would prefer this to be automated. It does give you an idea of the class from bootstrap which you need to augment your html templates though.

13.5. Using Django-Bootstrap-Toolkit

A simpel alternative would be to use `django-bootstrap-toolkit` see https://github.com/dyve/django-bootstrap-toolkit. Note that there are other packages like this. To install the `django-bootstrap-toolkit` run, `pip install django-bootstrap-toolkit`. Add, `bootstrap_toolkit` to the `INSTALLED_APPS` tuple in `settings.py`. Then modify the template like that shown below:

In which case the `category.html` template would become:

This solutin is much cleaner, and automated. However, it does not render as nicely :-(. Probably requires some tweaking to improve how it renders.

13.6. The End Result

Now that Rango is starting to look better we can go back and add in the extra functionality that will really pull the application together.

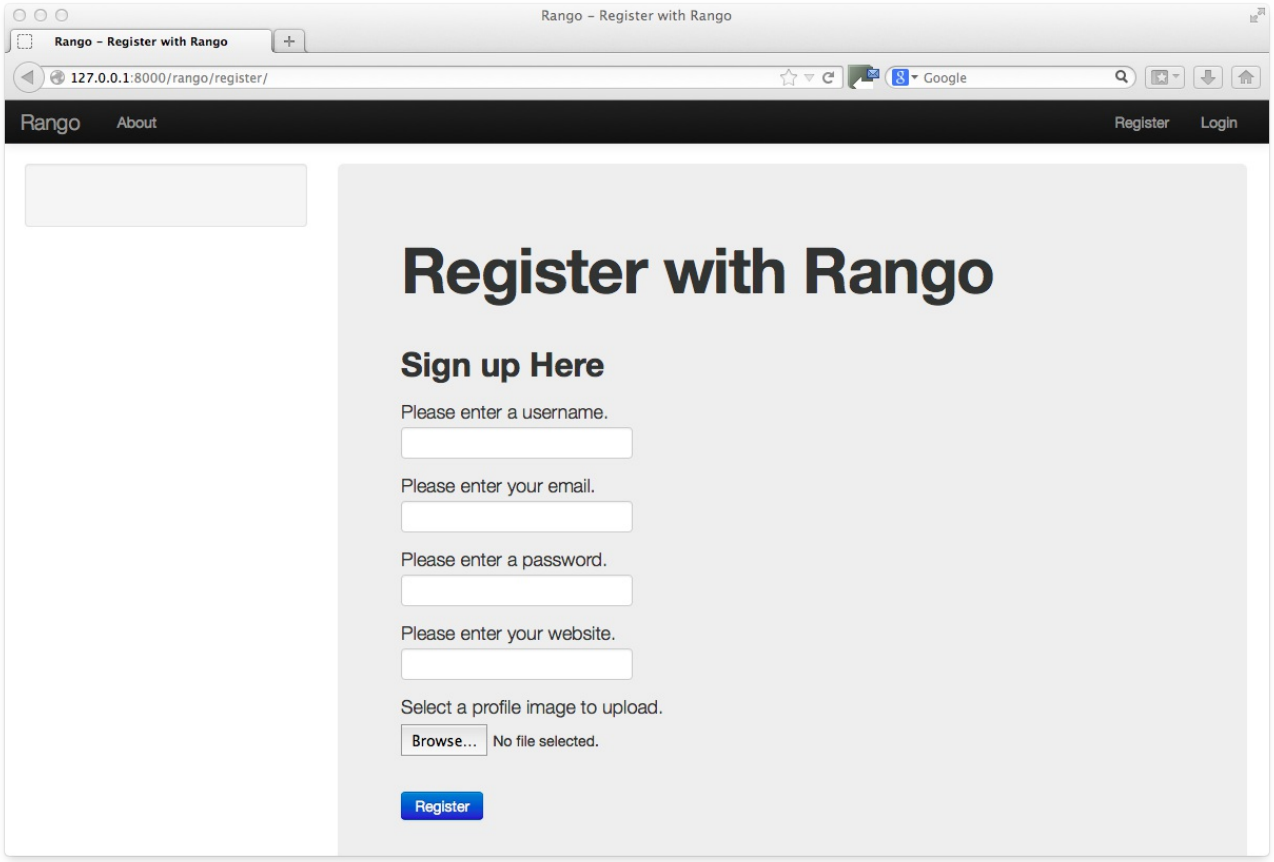


Figure 7: A screenshot of the Registration page with customised Bootstrap Styling.

14. Template Tags

14.1. Providing Categories on Every Page

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, if the category list has been passed through.
- Then in 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. It requires a lot of cutting and pasting of code. Also, we will run into problems, when we want to show the categories on pages serviced by the django-registration-redux package. So we need a different approach, by using `templatetags` that are included in the template that request the data required.

14.2. Using Template Tags

Create a directory `rango/templatetags`, and create two files, one called `__init__.py`, which will be empty, and another called, `rango_extras.py`, where you can add the following code:

```
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()}
```

As you can see we have made a method called, `get_category_list()` which

returns the list of categories, and that is assocaited with a template called `rango/cats.html`. Now create a template called "rango/cats.html`` in the templates directory with the following code:

```
{% if cats %}
    <ul class="nav nav-sidebar">
    {% for c in cats %}
        <li><a href="{% url 'category'  c.slug %}">{{ c.name }}</a></l
    {% endfor %}

{% else %}
    <li> <strong >There are no category present.</strong></li>

    </ul>
{% endif %}
```

Now in your `base.html` you can access the templatetag by first loading it up with `{% load rango_extras %}` and then slotting it into the page with `{% get_category_list %}`, i.e.:

```
<div class="col-sm-3 col-md-2 sidebar">

    {% block side_block %}
    {% get_category_list %}
    {% endblock %}

</div>
```

Note

You will need to restart your server every time you modify the templatetags so that they are registered.

14.3. Parameterised Template Tags

Now lets extend this so that when we visit a category page, it highlights which category we are in. To do this we need to paramterise the templatetag. So update the method in `rango_extras.py` to be:

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

This lets us pass through the category we are on. We can now update the `base.html` to pass through the category, if it exists.

```
<div class="col-sm-3 col-md-2 sidebar">

    {% block side_block %}
    {% get_category_list category %}
    {% endblock %}

</div>
```

Now update the `cats.html` template:

```
{% for c in cats %}
    {% if c == act_cat %} <li  class="active" > {% else  %} <li>{% end
            <a href="{% url 'category'  c.slug %}">{{ c.name }}</a></l
{% endfor %}
```

Here we check to see if the category being displayed is the same as the category being passed through (i.e. `act_cat`), if so, we assign the `active` class to it from Bootstrap (http://getbootstrap.com/components/#nav).

Restart the development web server, and now visit the pages. We have passed through the `category` variable. When you view a category page, the template has access to the `category` variable, and so provides a value to the templatetag `get_category_list()`. This is then used in the `cats.html` template to select which category to highlight as active.

Navigation

How To Tango With Django 1.7 »

previous next

15. Adding External Search Functionality

At this stage, our Rango application is looking pretty good - a majority of our required functionality is implemented. In this chapter, we will connect Rango up to Bing's Search API so that users can also search for pages, rather than just use the categories. First let's get started by setting up an account to use Bing's Search API, then construct a wrapper to call Bing's web search functionality before integrating the search into Rango.

15.1. The Bing Search API

The Bing Search API provides you with the ability to embed search results from the Bing search engine within your own applications. Through a straightforward interface, you can request results from Bing's servers to be returned in either XML or JSON. The data returned can then be interpreted by a XML or JSON parser, with the results then rendered as part of a template within your application.

Although the Bing API can handle requests for different kinds of content, we'll be focusing on web search only for this tutorial - as well as handling JSON responses. To use the Bing Search API, you will need to sign up for an API key. The key currently provides subscribers with access to 5000 queries per month, which should be more than enough for our purposes.

15.1.1. Registering for a Bing API Key

To register for a Bing API key, you must first register for a free Microsoft account. The account provides you with access to a wide range of Microsoft services. If you already have a Hotmail account, you already have one! You can create your free account and login at https://account.windowsazure.com.

When you account has been created, jump to the Windows Azure Marketplace Bing Search API page. At the top of the screen, you may first need to click the Sign In button - if you have already signed into your Microsoft account, you won't need to provide your account details again. If the text says Sign Out, you're already logged in.

Down the right hand side of the page is a list of transactions per month. At the bottom of the list is 5,000 Transactions/month. Click the sign up button to the right - you should be subscribing for a free service. You should then read the Publisher Offer Terms, and if you agree with them click Sign Up to continue. You will then be presented with a page confirming that you have successfully signed up.

Once you've signed up, click the Data link at the top of the page. From there, you should be presented with a list of data sources available through the Windows Azure Marketplace. At the top of the list should be Bing Search API - it should also say that you are subscribed to the data source. Click the use link associated with the Bing Search API located on the right of the page. You will then be presented with a screen similar to that shown in Figure 1.

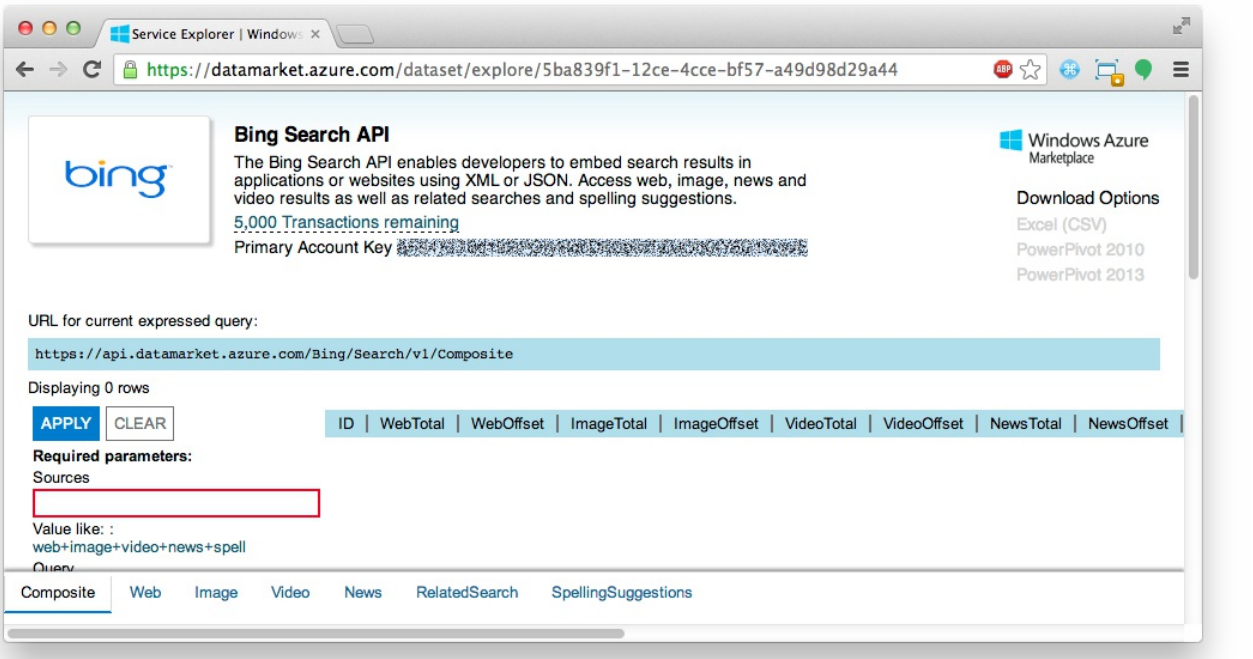


Figure 1: The Bing Search API service explorer page. In this screenshot, the Primary Account Key is deliberately obscured. You should make sure you keep your key secret, too!

This page allows you to try out the Bing Search API by filling out the boxes to the left. For example, the Query box allows you to specify a query to send to the API. Ensure that at the bottom of the screen you select Web for web search results only. Note the URL provided in the blue box at the top of the page changes as you alter the settings within the webpage. Take a note of the Web search URL. We'll be using part of this URL within our code later on. The following example is a URL to perform

a web search using the query rango.

```
https://api.datamarket.azure.com/Bing/Search/v1/Web?Query=%27rango%27
```

We must also retrieve your API key so you can authenticate with the Bing servers when posing requests. To obtain your key, locate the text Primary Account Key at the top of the page and click the Show link next to it. Your key will then be exposed. We'll be using it later, so take a note of it - and keep it safe! If someone obtains your key, they'll be able to use your free query quota.

Note

The Bing API Service Explorer also keeps a tab of how many queries you have left of your monthly quota. Check out the top of the page to see!

15.2. Adding Search Functionality

To add search functionality to Rango, we first must write an additional function to query the Bing API. This code should take in the request from a particular user and return to the calling function a list of results. Any errors that occur during the API querying phase should also be handled gracefully within the function. Spinning off search functionality into an additional function also provides a nice abstraction between Django-related code and search functionality code.

To start, let's create a new Python module called `bing_search.py` within our `rango` application directory. Add the following code into the file. Check out the inline commentary for a description of what's going on throughout the function.

```
import json
import urllib, urllib2

# Add your BING_API_KEY

BING_API_KEY = '<insert_bing_api_key>'

def run_query(search_terms):
    # Specify the base
    root_url = 'https://api.datamarket.azure.com/Bing/Search/'
```

```
    source = 'Web'

    # Specify how many results we wish to be returned per page.
    # Offset specifies where in the results list to start from.
    # With results_per_page = 10 and offset = 11, this would start fro
    results_per_page = 10
    offset = 0

    # Wrap quotes around our query terms as required by the Bing API.
    # The query we will then use is stored within variable query.
    query = "'{0}'".format(search_terms)
    query = urllib.quote(query)

    # Construct the latter part of our request's URL.
    # Sets the format of the response to JSON and sets other propertie
    search_url = "{0}{1}?$format=json&$top={2}&$skip={3}&Query={4}".fo
        root_url,
        source,
        results_per_page,
        offset,
        query)

    # Setup authentication with the Bing servers.
    # The username MUST be a blank string, and put in your API key!
    username = ''


    # Create a 'password manager' which handles authentication for us.
    password_mgr = urllib2.HTTPPasswordMgrWithDefaultRealm()
    password_mgr.add_password(None, search_url, username, BING_API_KEY

    # Create our results list which we'll populate.
    results = []

    try:
        # Prepare for connecting to Bing's servers.
        handler = urllib2.HTTPBasicAuthHandler(password_mgr)
        opener = urllib2.build_opener(handler)
        urllib2.install_opener(opener)

        # Connect to the server and read the response generated.
        response = urllib2.urlopen(search_url).read()
```

```
        # Convert the string response to a Python dictionary object.
        json_response = json.loads(response)

        # Loop through each page returned, populating out results list
        for result in json_response['d']['results']:
            results.append({
            'title': result['Title'],
            'link': result['Url'],
            'summary': result['Description']})

    # Catch a URLError exception - something went wrong when connecting
    except urllib2.URLError, e:
        print "Error when querying the Bing API: ", e

    # Return the list of results to the calling function.
    return results
```

The logic of the function above can be broadly split into six main tasks:

- First, the function prepares for connecting to Bing by preparing the URL that we'll be requesting.
- The function then prepares authentication, making use of your Bing API key. Make sure you replace `<api_key>` with your actual Bing API key, otherwise you'll be going nowhere!
- We then connect to the Bing API through the command `urllib2.urlopen(search_url)`. The results from the server are read and saved as a string.
- This string is then parsed into a Python dictionary object using the `json` Python package.
- We loop through each of the returned results, populating a `results` dictionary. For each result, we take the `title` of the page, the `link` or URL and a short `summary` of each returned result.
- The dictionary is returned by the function.

Notice that results are passed from Bing's servers as JSON. This is because we explicitly specify to use JSON in our initial request - check out the `search_url` variable which we define. If an error occurs when attempting to connect to Bing's servers, the error is printed to the terminal via the `print` statement within the

`except` block.

Note

There are many different parameters that the Bing Search API can handle which we don't cover here. If you're interested in seeing how to tailor your results, check out the Bing Search API Migration Guide and FAQ.

15.3. Storing your API KEY safely

If you are putting your code into a public repository on GitHub or the like, then you should take some pre-cautions about sharing your API Key. One solution is to create a new file call, `keys.py` which has a variable `BING_API_KEY`. Then import the `BING_API_KEY` into `bing_search.py`. Update your `.gitignore` file to include `keys.py`, so that `keys.py` is not added to the repository. This way the key will only be stored locally.

15.4. Exercises

Taking the basic Bing Search API function we added above as a baseline, try out the following exercises. * If using a public repository, refactor the code so that your API key is not publicly accessible * Add a main() function to the bing_search.py to test out the BING Search API

- Hint: add the following code, so that when you `python bing_search.py` it calls the `main()` function:

```
if __name__ == '__main__':
    main()
```

- The main function should ask a user for a query (from the command line), and then issue the query to the BING API via the run_query method and print out the top ten results returned.
- Print out the rank, title and URL for each result.

15.5. Putting Search into Rango

To add external search functionality, we will need to perform the following steps.

1. We must first create a `search.html` template which extends from our `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 view to handle the rendering of the `search.html` template for us, as well as calling the `run_query()` function we defined above.

15.5.1. Adding a Search Template

Let's first create our `search.html` template. Add the following HTML markup and Django template code.

```
{% extends "base.html" %}

{% load staticfiles %}

{% block title %}Search{% endblock %}

{% block body_block %}

    <div class="page-header">
        <h1>Search with Rango</h1>
    </div>

    <div class="row">

        <div class="panel panel-primary">
            <br/>

            <form class="form-inline" id="user_form" method="post" act
                {% csrf_token %}
                <!-- Display the search form elements here -->
                <input class="form-control" type="text" size="50" name
                <input class="btn btn-primary" type="submit" name="sub
                <br />
            </form>

            <div class="panel">
                {% if result_list %}
                    <div class="panel-heading">
```

```
                    <h3 class="panel-title">Results</h3>
                    <!-- Display search results in an ordered list -->
                    <div class="panel-body">
                        <div class="list-group">
                            {% for result in result_list %}
                                <div class="list-group-item">
                                    <h4 class="list-group-item-heading
                                    <p class="list-group-item-text">{{
                                </div>
                            {% endfor %}
                        </div>
                    </div>
                {% endif %}
                </div>
            </div>
 </div>

{% endblock %}
```

The template code above performs two key tasks:

1. In all scenarios, the template presents a search box and a search buttons within a HTML `<form>` for users to enter and submit their search queries.
2. If a `results_list` object is passed to the template's context when being rendered, the template then iterates through the object displaying the results contained within.

To style the html we have made use of Bootstrap: panels, http://getbootstrap.com/components/#panels, list groups, http://getbootstrap.com/components/#list-group, and inline forms, http://getbootstrap.com/css/#forms-inline.

As you will see from our corresponding view code shortly, a `results_list` will only be passed to the template engine when there are results to return. There won't be results for example when a user lands on the search page for the first time - they wouldn't have posed a query yet!

15.5.2. Adding the View

With our search template added, we can then add the view which 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 Bing function to get the results list!
            result_list = run_query(query)

    return render(request, 'rango/search.html', {'result_list': result
```

By now, the code should be pretty self explanatory to you. The only major addition is the calling of the `run_query()` function we defined earlier in this chapter. To call it, we are required to also import the `bing_search.py` module, too. Ensure that before you run the script that you add the following import statement at the top of the `views.py` module.

```
from rango.bing_search import run_query
```

You'll also need to ensure you do the following, too.

1. Add a mapping between your `search()` view and the `/rango/search/` URL calling it `name='search'`
2. 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.

Note

According to the relevant article on Wikipedia, an Application Programming Interface (API) 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.

Navigation

16. 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 a website that you are likely to make in your own projects. Rango however at the present moment is not very cohesive. In this chapter, we challenge you to improve the application and its user experience by bringing together functionality that we've already implemented alongside some awesome new additions.

To make Rango more coherent and integrated it would be nice to add the following functionality.

- Track the click throughs 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 Chapter 19).
- 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
    - let users filter the set of categories shown in the side bar (see Chapter 19); and
    - instead of refreshing the entire page, when users search, it only updates the results (see Chapter 19)
- 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 Chapter 19, while some

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 next chapter, we have included code snipets and elaborated on how to implement these features.

16.1. Track Page Click Throughs

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.
    - 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.
- Update the `category.html` so that it uses `/rango/goto/?page_id=XXX` instead of using the direct URL, remember to use the `url` templatetag (i.e. <a href="{% url 'goto' %}?pageid={{page.id}}">)

16.1.1. 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.

```
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']`.

Note

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.

16.2. 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 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 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 function.
- Take the search form and results template markup from `search.html` and place it into `category.html`.
- Update the search form so that that action refers back to the category page, i.e.: `<form class="form-inline" id="user_form" method="post" action="{% url '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 include `{% if user.authenticated %}` to the `category.html` template to restrict access.

16.3. 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 re-directed the user to the rango index page, you will need re-direct them to a new form, to collect the website and url details. To add the UserProfile registration functionality:

- Create a `profile_registration.html` which will display the `UserProfileForm`.
- Create a `register_profile()` view to capture the profile detials
- Map the view to a url, i.e. `rango/add_profile/`.
- In the `MyRegistrationView` update the `get_success_url()` to point to `rango/add_profile/`

Another useful feature 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 on the right-hand side 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 create a users page, that lists all the users. If you click on a user page, then you can see their profile (but the user can only edit their own page).

Navigation

17. Making Rango Tango! Code and 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 checkout snippets of code and use them within your version of Rango.

17.1. Track Page Click Throughs

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.

17.1.1. 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
```

17.1.2. Mapping URL

In `/rango/urls.py` add the following code to the `urlpatterns` tuple.

```
url(r'^goto/$', views.track_url, name='goto'),
```

17.1.3. 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.tit
        {% 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`.

17.1.4. Updating Category View

Since we are tracking the number of click throughs you can now update the `category()` view so that you order the pages by the number of views, i.e:

```
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.

17.2. 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.

17.2.1. 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`.

17.2.2. Creating a Search Form Template

Take the search form from `search.html` and put it into the `category.html`. Be sure to change the action to point to the `category()` view as shown below.

```
 <form class="form-inline" id="user_form" method="post" action="{% url
     {% csrf_token %}
     <!-- Display the search form elements here -->
     <input class="form-control" type="text" size="50" name="query" va
     <input class="btn btn-primary" type="submit" name="submit" value=
</form>
```

Also include a `<div>` to house the results underneath.

```
<div class="panel">
        {% if result_list %}
        <div class="panel-heading">
                <h3 class="panel-title">Results</h3>
                <!-- Display search results in an ordered list -->
                <div class="panel-body">
```

```
                <div class="list-group">
                        {% for result in result_list %}
                    <div class="list-group-item">
                        <h4 class="list-group-item-heading"><a href="{
                        <p class="list-group-item-text">{{ result.summ
                    </div>
                {% endfor %}
            </div>
        </div>
    {% endif %}
</div>
```

17.2.3. 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 category(request, category_name_slug):
    context_dict = {}
    context_dict['result_list'] = None
    context_dict['query'] = None
    if request.method == 'POST':
        query = request.POST['query'].strip()

        if query:
            # Run our Bing function to get the results list!
            result_list = run_query(query)

            context_dict['result_list'] = result_list
            context_dict['query'] = query

    try:
        category = Category.objects.get(slug=category_name_slug)
        context_dict['category_name'] = category.name
        pages = Page.objects.filter(category=category).order_by('-view
        context_dict['pages'] = pages
        context_dict['category'] = category
    except Category.DoesNotExist:
        pass

    if not context_dict['query']:
```

```
        context_dict['query'] = category.name

    return render(request, 'rango/category.html', context_dict)
```

Notice that in the `context_dict` that we pass through, we will include the `result_list` and `query`, and if there is no query, we provide a default query, i.e. the category name. The query box then displays this variable.

Navigation

How To Tango With Django 1.7 »

previous next

18. JQuery and Django

JQuery rocks! JQuery is a library written in Javascript that seriously pimps its power and utility. A few lines of JQuery often encapsulates hundreds of lines of javascript. 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 Application
- explain how to interpret JQuery code
- and provide a number of small examples

18.1. Including JQuery in your Django Project/Application

In your base template include a reference to:

```
{% load staticfiles %}

<script src="{% static "js/jquery-1.11.1.js" %}"></script>
<script src="{% static "js/rango-jquery.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/1.11.1/jquer
```

Make sure you have your static files set up (see Chapter 4)

In the static folder create a js folder and plonk the JQuery javascript file (`jquery.js`) here along with an file called `rango-jquery.js`, which 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 JQuery, 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, then 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 my try to run, but the HTML elements may not have been downloaded (see http://api.jquery.com/ready/).

18.1.1. Stylistic Note

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 act on the element. So it is good to keep this in mind. There are different selection operators, and various actions that can then be performed/applied. In the next, subsections we will take a few JQuery functions that you can use to manipulate the HTML elements.

18.2. 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 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 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 not 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()`, which selects the element in the page with id equal to `about-btn`, and then it programatically assigns to the on 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()` but for more complex functions it is much cleaner, as the JQuery/Javascript code is maintained in a separate file (completely!!). This is because we assign the event handler at run-time rather than statically within the code. We achieve separation of concerns between the jquery/javascript code and the html code.

Note

Remember when it comes to CSS, JAVASCRIPT and HTML, you gotta keep them separated!

18.3. Selectors

There are different ways to select elements in JQuery. From the above example, shows how the `#` can be used to find `id` elements in your html document. To find

classes, you can use `.`, so, for example, if we added the following code:

```
$(".ouch").click( function(event) {
            alert("You clicked me! ouch!");
});
```

Then all elements, that had a `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.

Also, 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');
    });
```

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 it color to red or blue, respectively. Note, the JQuery `hover()` function takes two functions (see http://api.jquery.com/hover/), the JQuery `click()` requires the event to passed through (see http://api.jquery.com/click/).

Try adding the above code your `rango-jquery.js` file, make sure it is within the `$(document).ready()` function. What happens if you change the `$(this)` to `$(p)`?

Hover, is an example of a mouse move event, for descriptions on other such events, see: http://api.jquery.com/category/events/mouse-events/

18.4. 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 color of the element. The `css` is one example of DOM manipulation, however, the stand JQuery library

provides many other ways 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 will mean the button will now appear in the bootstrap style (assuming you are using the Bootstrap toolkit).

It is also possible to access the html of a particular element. For example, lets put a `div` in the `about.html`:

```
<div id="msg">Hello</div>
```

Then add the following JQuery to `rango-jquery.js`:

```
$("#about-btn").click( function(event) {
msgstr = $("#msg").html()
        msgstr = msgstr + "o"
        $("#msg").html(msgstr)
 });
```

On click of the element with id `#about-btn`, we first get the html inside the element with id `msg` and appeand "o" to it. Then we 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 rudimentry guide to using JQuery and incroporating it within your Django Application. 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 (see http://jquery.com). In the next chapter we will be using the JQuery to help provide AJAX functionality within Rango.

Navigation

19. 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 resources at the Mozilla website: https://developer.mozilla.org/en-US/docs/AJAX

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. Otherwise, download the latest version of JQuery and include it within your application (see Chapter ..).

19.1. AJAX based Functionality

To make the interaction with the Rango application more seamless let's add in a number of features that use AJAX, such as:

- Add a "Like Button" to let registered users "like" a particular category
- Add inline category suggestions - so that when a user types they can quickly find a category
- Add an "Add Button" to let registered users quickly and easily add a Page to the Category

Create a new file, called `rango-ajax.js` and add it to your `js` directory. Then in your base template include:

```
<script src="{% static "js/jquery.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/1.11.1/jquer
```

Now that the pre-reqs for using JQuery are in place we can use it to pimp the rango application.

19.2. Add a "Like Button"

It would be nice to let user, 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", we'll be trusting them not to click the like button multiple times.

19.2.1. 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.

19.2.2. 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:

```
<p>

<strong id="like_count">{{ category.likes }}</strong> people like this

{% if user.is_authenticated %}
        <button id="likes" data-catid="{{category.id}}" class="btn btn
        <span class="glyphicon glyphicon-thumbs-up"></span>
        Like
        </button>
{% endif %}

</p>
```

19.2.3. Create a Like Category View

Create a 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 = Cat.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 denote that they like a category. The view assumes that a variable `category_id` has been passed through via a GET so that the 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_category/$', views.like_category, name='like_category'),
```

19.2.4. 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/', {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_category/` encoding the `category_id` in the request. If the request is successful, then the HTML element with id like_count (i.e. the) 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 here, when the button is clicked an AJAX request is made, given our url mapping, this invokes the `like_category` view which 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.

19.3. 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 which 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.

19.3.1. 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; lets

call it category_suggest

With the mapping, view, and template for this view in place, you will need to update the `base.html` template and add in some javascript so that the categories can be displayed as the user types.

- 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.
    - Above this 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 with id="cats" with the data received.
    - Here you can use the JQuery `.html()` function i.e. `$('#cats').html( data )`

19.3.2. Parameterise the Get Category List Function

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 upper-case or lower-case letters. If it on the other hand was important to take into account whether letters was upper-case 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=s

        if max_results > 0:
                if len(cat_list) > max_results:
                        cat_list = cat_list[:max_results]

        return cat_list
```

19.3.3. Create a Suggest Category View

Using the `get_category_list` function we can now create a view that returns the top 8 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/category_list.html', {'cat_list'
```

Note here we are re-using the `rango/cats.html` template :-).

19.3.4. Map View to URL

Add the following code to `urlpatterns` in `rango/urls.py`:

```
url(r'^suggest_category/$', views.suggest_category, name='suggest_cate
```

19.3.5. Update Base Template

In the base template in the sidebar div add in the following HTML code:

```
<ul class="nav nav-list">
        <li class="nav-header">Find a Category</li>
        <form>
```

```
        <label></label>
        <li><input  class="search-query span10" type="text" name="sugg
        </form>
</ul>

<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 which will send the suggestion request.

19.3.6. 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_category/', {suggestion: query}, functio
         $('#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.

19.4. 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 mini-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 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'^auto_add_page/$', views.auto_add_page, name='auto_add_page'),`
- Add an event handler to the button using JQuery - when added hide the button. The response could also update the pages listed on the category page, too.

19.4.1. Hints

HTML Template code:

```
{% if user.is_authenticated %}
        <button data-catid="{{category.id}}" data-title="{{ result.tit
{% endif %}
```

JQuery code:

Note here we are assigned the event handler to all the buttons with class `rango-add`.

View code:

```
@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=ti

            pages = Page.objects.filter(category=category).order_by('-

            # Adds our results list to the template context under name
            context_dict['pages'] = pages

    return render(request, 'rango/page_list.html', context_dict)
```

Navigation

How To Tango With Django 1.7 »

previous next

20. 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 dont 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 http://docs.python-guide.org/en/latest/writing/tests/, 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.

Note

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.

20.1. Running Tests

In built in Django is machinery to test the applications built. You can do this 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 application. At the moment, nothing much happens. That is because you may have noticed the file `rango/tests.py` only contains an import statement. Everytime 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.

20.2. Testing the models in Rango

Ok, lets 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 cate
        """
                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 check if the values are equal, with the `assertEqual` method, but other types of assertions are also possible. See the Python Documentation on unit tests, https://docs.python.org/2/library/unittest.html for other commands (i.e. `assertItemsEqual`, `assertListEqual`, `assertDictEqual`, etc). Django's testing machinery is derived from Python's but also provides an number of other asserts and specific test cases.

Now lets run 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_17/rango/tests.py",
    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 Catgegory models, `save()` method, that checks the value of views, and updates it accordingly.

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
            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?

20.3. Testing Views

So far we have writtent 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 displa
        """
        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: did the page load ok? Does the response, i.e. the html contain the phrase "There are no categories present.", and does the context dictionary contain 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):
    """
    If no questions exist, an appropriate message should be 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 if the page loads, if it contains the text `tmp test temp` and if the number of categories is equal to 4.

#TODO(leifos): add in some tests showing how to test different forms i.e. login etc.

20.4. Testing the Rendered Page

#TODO(leifos): add an example 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.

20.5. 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` which 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, ie. `rango/views`. For more details about using the package `coverage` visit: http://nedbatchelder.com/code/coverage/

20.6. Exercises

- Lets say that we want to extend the `Page` to include two additional fields, `last_visit` and `first_visit` which 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 help develop these tests.

- Check out the tutorial on test driven development by Harry Percival.

Navigation

How To Tango With Django 1.7 »

previous next

21. 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 which sets you up with an adequate amount of storage space and CPU time to get a Django application up and running.

Note

You can do this chapter independently (assuming you have some knowledge of git, if not refer to the chapter on using git).

21.1. 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 (like hosting specific domains and ssh abilities).

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 from.

21.2. The PythonAnywhere Web Interface

The PythonAnywhere web interface contains a dashboard which in turn provides a series of tabs allowing you to manage your application. The tabs as illustrated in Figure 1 include:

- a consoles tab, allowing you to create and interact with Python and Bash

console instances;
- a files tab, which allows you to upload to and organise files within your disk quota;
- a web tab, allowing you to configure settings for your hosted web application;
- a schedule tab, allowing you to setup tasks to be executed at particular times; and
- a databases tab, which allows you to configure a MySQL instance for your applications should you require it.

Of the the five tabs provided, we'll be working primarily with the consoles and web tabs. The PythonAnywhere wiki provides a series of detailed explanations on how to use the other tabs.

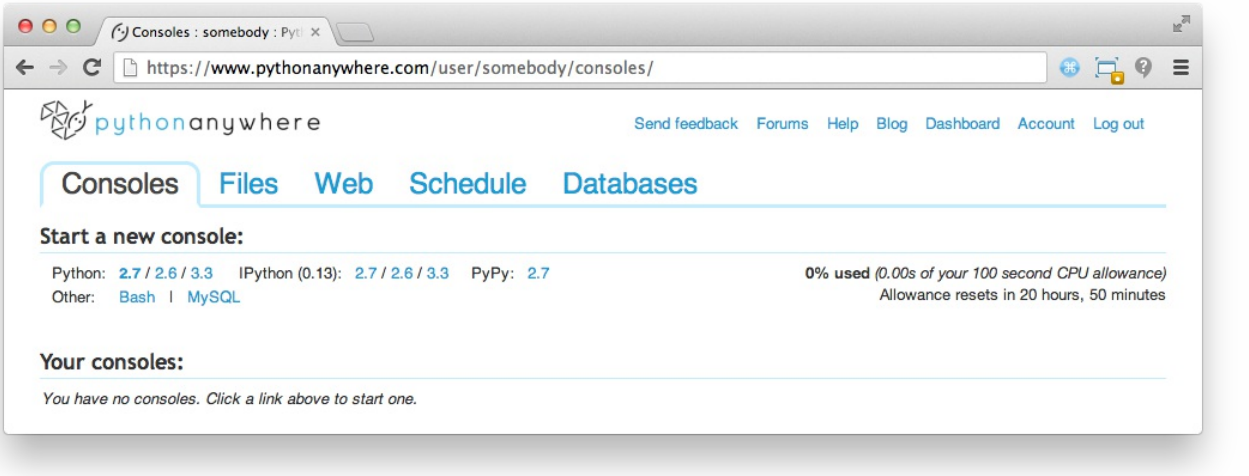


Figure 1: The PythonAnywhere dashboard, showing the Consoles tab.

21.3. Creating a Virtual Environment

As part of a its standard installation, PythonAnywhere comes with a number of packages pre-installed (i.e Python 2.7.4 and Django 1.3). However, since we are using a different setup, we need to setup a virtual environment and run our application inside that. Since our code base is compatible with Python 2.7.4 we can continue use that, but we will have to set up our virtual environment to use Django 1.5.4.

First, open a Bash console from the PythonAnywhere Consoles tab by clicking the Bash link. When the terminal is ready for you to interact, enter the following commands.

```
$ source virtualenvwrapper.sh
$ mkvirtualenv rango
```

The first command imports the virtual environment wrapper. The wrapper provides a series of extensions by Doug Hellman to the original `virtualenv` tool, making it easier for us to create, delete and use virtual environments. The second command creates a new virtual environment called `rango`. This process should take a short while to create, after which you will be presented with a slightly different prompt.

```
(rango)16:38 ~ $
```

Note the inclusion of `(rango)` compared to your previous command prompt. This signifies that we have activated the `rango` virtual environment, so any package installations will be done within that environment, leaving the wider system setup alone. If you then 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
```

Now we can customise our virtual environment by installing the required packages for our Rango application. Installing may take a considerable amount of time to install as your CPU time is limited - so have some patience. Install all the required packages:

```
$ pip install -U django==1.7
$ pip install pillow
$ pip install django-registration-redux
$ pip install django-bootstrap-toolkit
```

Aternatively, you could 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 a job lot.

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
```

Note

PythonAnywhere also provides instructions on how to setup virtual environments, see https://www.pythonanywhere.com/wiki/VirtualEnvForNewerDjango.

21.3.1. Virtual Environment Switching

Moving between virtual environments can be done pretty easily. First you need to make sure that `virtualenvwrapper.sh` has been loaded by running `source virtualenvwrapper.sh`.

Rather than doing this each time you open up a console, you can add it to your `.bashrc` profile which is located in your home directory. Doing so will ensure the command is executed automatically for you every time you start a new Bash console instance. Any Bash consoles active will need to be closed for the changes to take effect.

With this done, you can then launch into a pre-existing virtual environment with the `workon` command. 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.

```
(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 ~ $
```

21.4. 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_NAM
```

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.

If you haven't put your code in a Git repository, you can clone the version we have made, by issuing the following command:

```
16:54 ~ $ git clone https://github.com/leifos/tango_with_django17.git
```

#TODO(leifos): upload code to github

Note

It doesn't matter if you clone your Git repository within your new virtual environment or not. You're only creating files within your disk quota, which doesn't require your special Python setup.

21.4.1. 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. `workon rango`). From your home directory, move into the `tango_with_django` directory, and issue the following commands

```
(rango) 16:55 ~/tango_with_django $ python manage.py makemigrations ra
(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 application, then the migrate command creates the SQLlite3 database. Once the database is created, the database can be populated and a superuser created.

21.5. Setting up your Web Application

Now that the database is setup, we need to configure the PythonAnywhere NGINX webserver to serve up your application . Within PythonAnywhere's web interface, navigate to your dashboard and click on 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.

Then, navigate to the PythonAnywhere subdomain at `http://<username>.pythonanywhere.com` in a new browser tab. You should be presented with the default `Hello, World!` webpage. This is because the WSGI script is currently serving up this page and not your Django application.

21.5.1. 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 dashboard. From there, click the Web tab. Under the Code heading you can see a link to the WSGI configuration file: e.g. `/var/www/<username>_pythonanywhere_com_wsgi.py`

The good 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. The example below demonstrates a possible configuration for you application.

```
# TURN ON THE VIRTUAL ENVIRONMENT FOR YOUR APPLICATION
activate_this = '/home/<username>/.virtualenvs/rango/bin/activate_this
execfile(activate_this, dict(__file__=activate_this))
import os
import sys

# ADD YOUR PROJECT TO THE PYTHONPATH FOR THE PYTHON INSTANCE
path = '/home/<username>/tango_with_django_17/'
if path not in sys.path:
    sys.path.append(path)

# IMPORTANTLY GO TO THE PROJECT DIR
os.chdir(path)

# TELL DJANGO WHERE YOUR SETTINGS MODULE IS LOCATED
os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'tango_with_django_pro

# IMPORT THE DJANGO SETUP - NEW TO 1.7
import django
django.setup()

# IMPORT THE DJANGO WSGI HANDLER TO TAKE CARE OF REQUESTS
import django.core.handlers.wsgi
application = django.core.handlers.wsgi.WSGIHandler()
```

Ensure that you replace `<username>` with your 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 code sample above begins by activating your virtual environment `rango` as this has been configured with all the required packages. The script then 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

dashboard, and click the Reload button at the top of the page. When the application is reloaded, visiting `http://<username>.pythonanywhere.com` should present you with your Django application, all ready to go!

Note

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.

21.5.2. Assigning Static Paths

We're almost there. One issue which 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 Web tab and choose the subdomain hosting your application from the list on the left.

Underneath the Static files header, perform the following.

1. Click `Enter URL` and enter `/static/admin`, followed by return.
2. Click the corresponding `Enter path` text. Set this to `/home/<username>/.virtualenvs/rango/lib/python2.7/site-packages/django/contrib/admin/static/admin`, where `<username>` should be replaced with your PythonAnywhere username. 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.
3. Repeat the two steps above for the URL `/static/` and path `/home/<username>/tango_with_django/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 about potential `HTTP 502 - Bad Gateway` errors!

21.5.3. Bing API Key

Update `bing_search.py` or `keys.py` with your own BING API Key to use the search

functionality in Rango. Again, you will have to hit the Reload button for the changes to take effect.

21.5.4. 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.

Changing the value of `DEBUG` also means you should set the `ALLOWED_HOSTS` property. Failing to perform this step will make Django return `HTTP 400 Bad Request` errors. Alter `ALLOWED_HOSTS` so that it includes your PythonAnywhere subdomain like in the example below.

```
ALLOWED_HOSTS = ['<username>.pythonanywhere.com']
```

Again, ensure `<username>` is changed to your PythonAnywhere username. Once complete, save the file and reload the application via the PythonAnywhere web interface.

21.6. Log Files

Deploying your web application to an online environment introduces another layer of complexity. It is likely that you will encounter new and bizzare 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.

21.7. Exercises

Congratulations, you've successfully deployed Rango!

- Tweet a link of your application to @tangowithdjango.
- Or email us to let us know, and tell us your thoughts on the book.

Navigation

How To Tango With Django 1.7 »

previous next

22. Summary

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, Bing Search, PythonAnywhere, etc 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: https://github.com/leifos/tango_with_django_book/tree/master/17

Navigation

How To Tango With Django 1.7 »

previous next

23. A Git Crash Course

We strongly recommend that you spend some time familiarising yourself with a version control system. For your benefit, this section 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 very popular, 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.

23.1. Why Use Version Control?

As your software engineering skills develop, you will find that you are able to plan and implement ever more complex 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.

Most projects have many files and many people working on those files. This is a recipe for chaos. Have you ever encountered one or more of the following situations:

- Made a change 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)?
- Wanted to experiment with a new feature without interfering without working code?

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 get into the habit now.

23.2. 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 from here. Although we do not cover TortoiseGit in this crash course, many tutorials exist online for it. Check this TortoiseGit tutorial if you are interested in using it.

23.3. The Git System

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 part of Git stored outwith your computer. This is considered a huge advantage of Git - you can make changes to your local repository when you may not have Internet access, and then apply those changes to the remote repository at a later stage.

Note

We keep repeating the word repository, but what do we actually mean by that? In the context of version control systems, consider a repository as a form of data structure that contains a set of commit objects, and a set of references to commit objects, called heads. You can find out more about what these are on this Git tutorial - and we will be explaining what the terminology for head means later on.

For now though, the following bullet points provide an explanation of each part of the Git system.

- 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. If you're feeling adventurous, you can set up a Git server on a computer with Internet access and a properly configured firewall (check out this Git server tutorial, for example), or 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. The key difference however is that the local repository is stored on your own computer. It is to this repository 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, and we will explain that in a bit more detail later.
- The index is technically part of the local repository. The index stores a list of files that you want to be managed with version control. This is explained in more detail in the commands and workflow section. 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 - only ever edit files in your workspace. The local repository contains a load of files that Git uses to keep track of your version controlled content. If you start messing around with these files, you'll more than likely break something!

Next, we'll be looking at how to get your Git workspace set up. We'll also discuss the basic workflow you should use when using Git.

23.4. Setting up Git

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!

23.4.1. 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://<USERNAME>:
<PASSWORD>@github.com/<OWNER>/<REPO_NAME>.git <workspace>
```

where you replace

- `<USERNAME>` with your GitHub username;
- `<PASSWORD>` 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. Although optional, we will specify it here to create the `<workspace>` directory.

If all is successful, you should see some positive messages in your terminal or Command Prompt alerting you to the fact that the clone has been successful.

23.4.2. 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 files for your project.

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 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 directory therefore contains the workspace, local repository and index.

23.4.3. 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. This is really easy to do. Simply open a Command Prompt/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. We don't want random commits from some guy called John Doe! How unlucky would it be if you were actually called John Doe?

Anyway, moving on to the second feature. Git 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. This file which should reside in the root of your 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:

```
config/api_keys.py

*.pyc
```

In this example file, there are two entries. The first one prompts git to ignore the file `api_keys.py` residing within the `config` directory. The second entry prompts Git to ignore all instance of files with a `.pyc` extension. This is really cool: you can use wildcards to make generic entries if you need to!

23.5. 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.

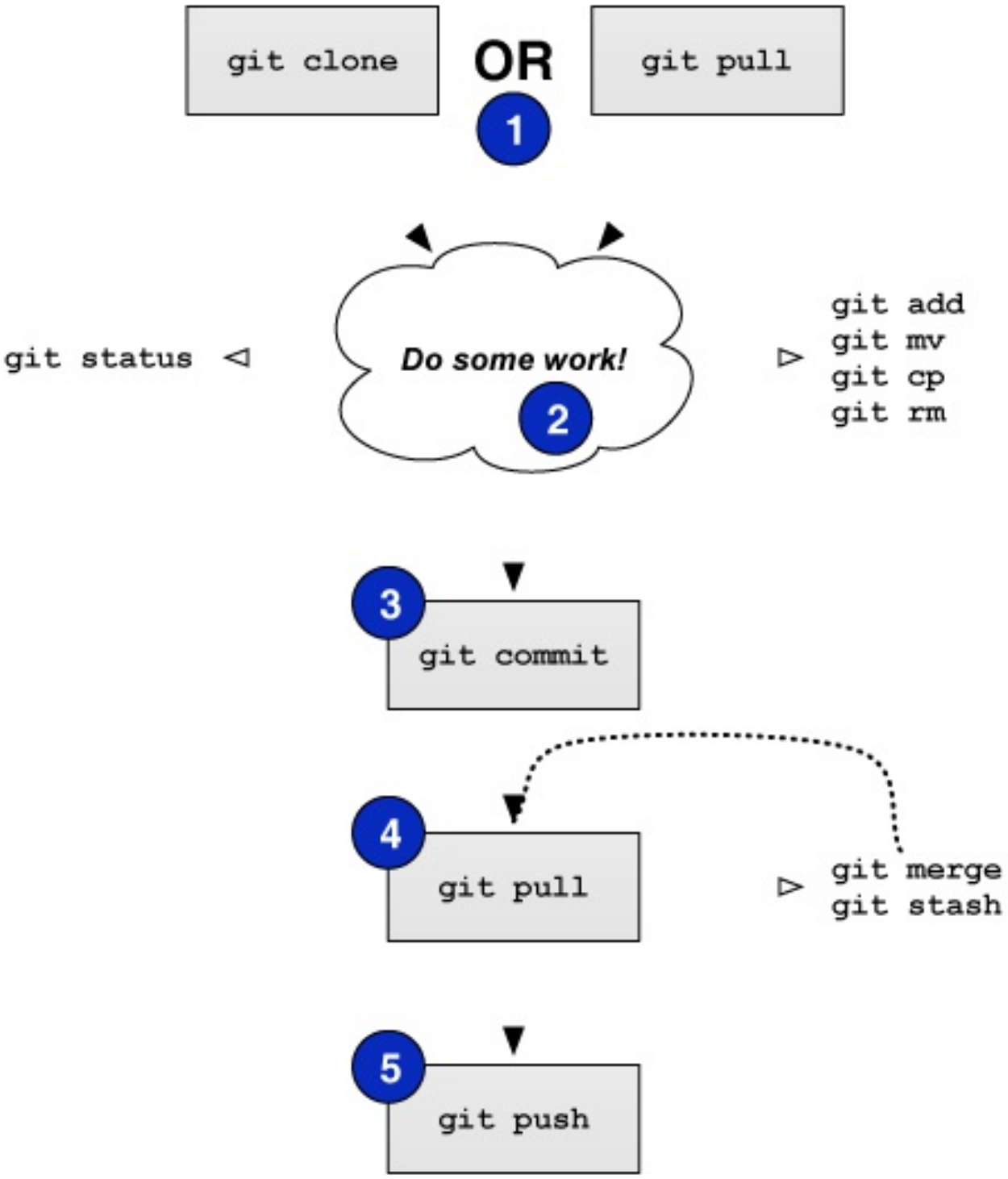


Figure 1: A diagram depicting the basic workflow and associated commands of interacting with a Git repository.

We have provided a pictorial representation of the basic Git workflow in Figure 1. Match each of the numbers in the blue circles to the numbered descriptions below.

23.5.1. 1. Starting Off

Before you can start work on your project, you must prepare Git for your forthcoming geek session. If you haven't yet sorted out your project's Git workspace, you'll need

to `clone` the repository to obtain a copy of all of its files. Check out Section 23.4.1 for more information on how to achieve this.

If you have previously made a clone of the remote repository, it's good practice to get into the habit of updating you local copy by using the `git pull` command. This 'pulls' changes from the remote repository. By doing this, you'll be working from the same page as your team members, which will help keep the issue of conflicting file contents from making your life a nightmare.

23.5.2. 2. Doing Some Work!

Once your workspace has been updated with the latest changes, the onus is on you to do some work! Within your workspace, you can take existing files and modify them. You can delete them too, or add new files to be version controlled.

It's not all plain sailing, however. You must be aware that as you work away, you need to keep Git up-to-date on the list of files you have added, removed or updated by modifying the 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 which let you update the local index. Three of the commands are near-identical to those that were discussed in Chapter 3, 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 newbie 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 can be used in the fashion `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 <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.
- `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 <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 the same as the above commands, 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 forgot 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.

```
$ git status

# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   chapters/requirements.rst
#
# Changes not staged for commit:
#   (use "git add/rm <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working dir
#
#       modified:   ../TODO.txt
#       modified:   chapters/deploy.rst
#       deleted:    chapters/index.rst
```

```
#       deleted:    images/css-font.png
#       modified:   images/git-sequence.pdf
#       modified:   omnigraffle/git-sequence.graffle
#
```

For further information on this useful command, check out the official Git documentation.

23.5.3. 3. Committing your Changes

We've mentioned committing several times in the previous step - but what on earth does it mean? In the world of Git, 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 disastrously 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 online discussion about when the ideal time to commit is - have a look on this Stack Overflow page for the opinions of several developers.

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. The HEAD is essentially the latest commit at the top of the pile - have a look at this Stack Overflow page for more information.

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 - though we highly recommend it. Instead of simply issuing `git commit`, run the following amended command.

```
$ git commit -m "Updated helpers.py to include a Unicode conversion function, str_to_unicode()."
```

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 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.

Note

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. 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!

23.5.4. 4. Synchronising your Repository

After you've committed your local repository and committed your changes, you're just about ready to send your commits 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, making any `git push` command very difficult to do!

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 in the remote repository to your local repository.

Note

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.

23.5.5. 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 desperately 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.

If what you are pushing is particularly important, you can also optionally 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.

23.6. 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, someone changed something, or everything is so terribly messed up you have no idea what happened. 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.

Warning

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 distinct steps:

- determining which commit to roll back to; and
- performing the rollback.

To determine what commit to rollback to, you can make use of the `git log` command. Issuing this command within your workspace directory will provide a list of recent commits that you made, your name and the date at which you made the commit. Additionally, the message that is stored with each commit is displayed. This is 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                    <
Author: John Doe <someaddress@domain.com>                          <
Date:   Mon Jul 8 19:56:21 2013 +0100                              <

    Nearly finished initial version of the requirements chapter    <

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 team members. From there, you can start to recover from the mistake by putting your head down and getting on with your project.

23.7. Exercises

If you haven't undertaken what we've been discussing in this chapter already, you should go through everything now to ensure your system and repository is ready to go.

First, ensure that you have setup your environment correctly. Install all of the prerequisites, including Python 2.7.5 and Django 1.5.4. Django should be installed by Pip, the package manager.

Once that is complete, create a new Git repository on Github for your project. To try

out the commands, you can create a new file `readme.md` in the root of your workspace. 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 file some more. Add, commit and push - and then try to revert to the initial version to see if it all works as expected.

Upon completion of these exercises, all that is left for us to discuss is the environment you just setup. While all may be good just now, what if you have another Python application that requires a different version to run? This is where the concept of virtual environments comes into play. Virtual environments allow multiple installations of Python and their relevant packages to exist in harmony, without disrupting one another. This is the generally accepted approach to configuring a Python setup nowadays. We don't go into much detail about them in this chapter now but you will be using a virtual environment when it comes to deploying your application. For now though, check out this article to read up on what they are, and how they can benefit you.

Note

There are many more advanced aspects of Git that we have not covered here, such as branching and merging. There are many fantastic tutorials available online if you are interested in taking your super-awesome version control skills a step further. For more details about such features take a look at this tutorial on getting started with Git, the Git Guide or Learning about Git Branching.

Navigation

How To Tango With Django 1.7 »

previous next

24. 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.

Note

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 24.12 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 Figure 1 where we have some HTML containing `<h1>` tags. In the CSS code example, we specify that all `h1` are styled. We'll come back to selectors in Section 24.2. For now though, you can assume the CSS style defined will be applied to our `<h1>` tags. The style contains four properties:

- the first property (`font-size`) sets the size of the font to 16pt;
- the second property (`font-style`) italicises the contents of all `<h1>` tags within the document;
- the third property (`text-align`) centres the text of the `<h1>` tags; and
- the final property (`color`) 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 in Figure 1.

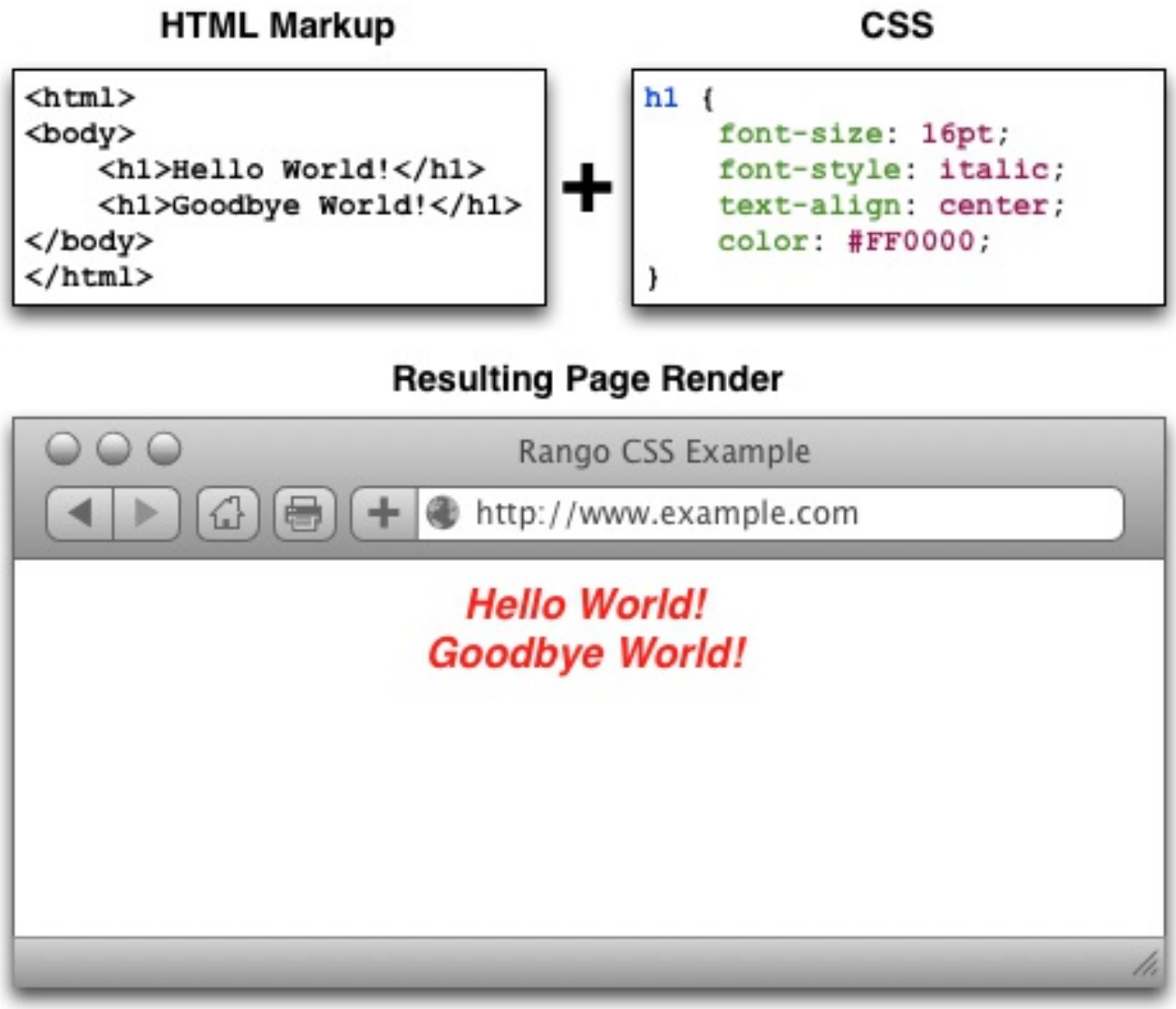


Figure 1: Illustration demonstrating the rendered output of the sample HTML markup and CSS stylesheet shown. Pay particular attention to the CSS example - the colours are used to demonstrate the syntax used to define styles and the property/value pairings associated with them.

Note

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.

24.1. 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
        <title>Sample Title</title>
    </head>

    <body>
        <h1>Hello world!</h1>
    </body>
</html>
```

As can be seen from above, there are at minimum three attributes which 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 in included with your HTML page, and the styles within the stylesheet applied. It should be noted that CSS stylesheets are considered by Django as static media, meaning you should place them within your project's `static` directory.

Note

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 nice abstraction between presentational semantics (CSS) and content (HTML).

24.2. 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. In Section 24.10, we also touch on what are known as pseudo-selectors.

24.3. Element Selectors

Taking the CSS example from Figure 1, 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.

24.3.1. 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 Figure 2 for an example.

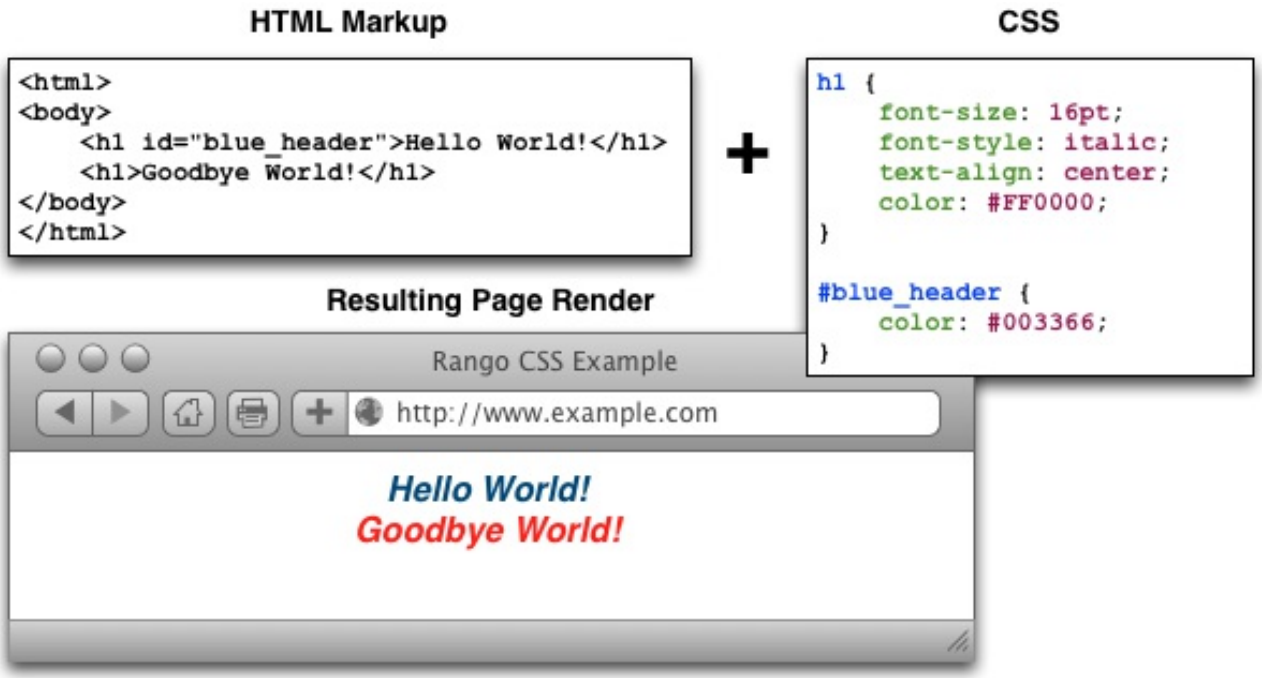


Figure 2: An illustration demonstrating the use of an id selector in CSS. Note the blue header has an identifier which matches the CSS attribute `#blue_header`.

24.3.2. 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 Figure 3 for an example.

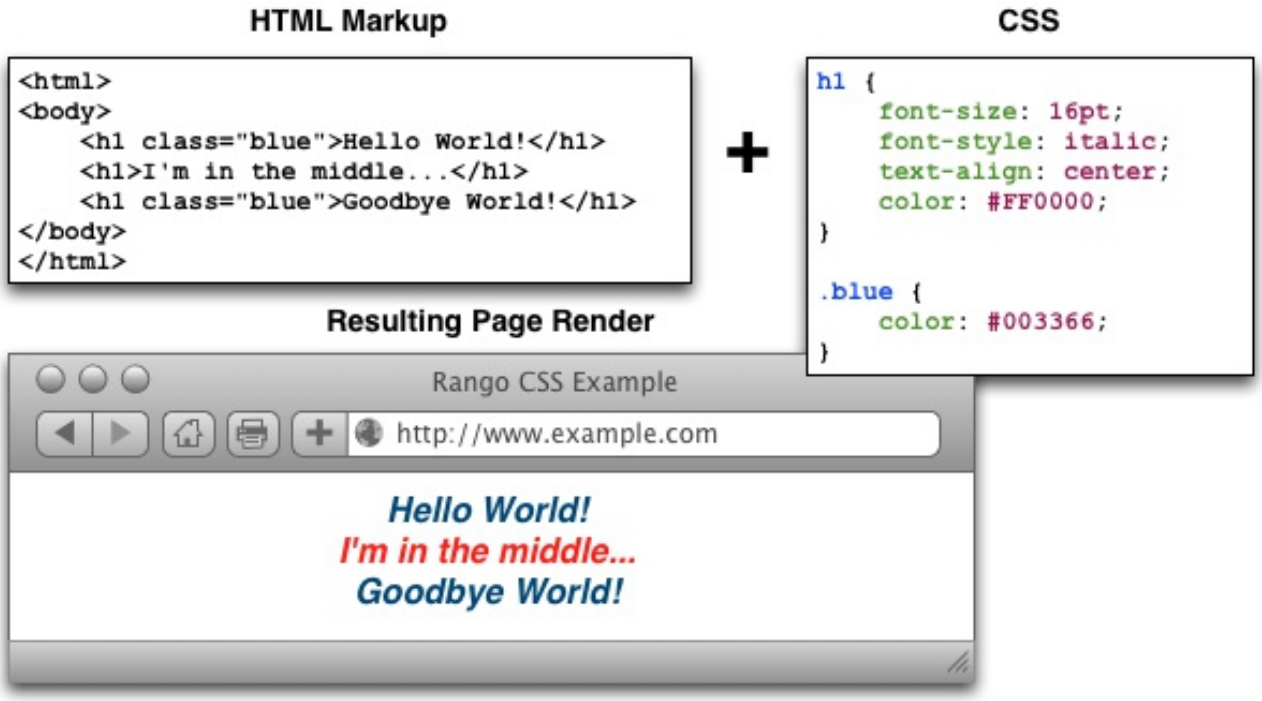


Figure 3: An illustration demonstrating the use of a class selector in CSS. The blue headers employ the use of the `.blue` CSS style to override the red text of the `h1` style.

Warning

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 it to more than one element, the answer will always be **no**. In cases like this, you should use a class or element selector.

24.4. 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 typesets you can use and this Web Designer Depot article on how to use such fonts.

24.5. 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 in Figure 4, 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`.

Warning

You must take great care when picking colours to use on your webpages. Don't select colours that don't contrast well - people simply won't be able to read them! 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.

Figure 4: Illustration of some basic colours with their corresponding hexadecimal and RGB values. Illustration adapted from W3Schools.

There are many different websites which 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-grain control over the colours you can choose.

Note

For more information on how colours are coded with hexadecimal, check out this thorough tutorial.

Warning

As you may have noticed, CSS uses American/International English to spell words. As such, there are a few

words which are spelt slightly differently compared to their British counterparts, like `color` and `center`. If you have grown up in Great Britain, double check your spelling and be prepared to spell it the wrong way! Hah!

24.5.1. 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.

24.5.2. Borders

You can change the colour of an element's borders, too. We'll discuss what borders are in Section 24.8 - but 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 color 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.

24.5.3. 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. Yuck!

```
body {
    background-color: #00FF00;
}
```

24.5.4. 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.

Note

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.

24.6. 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 Figure 5 for an illustration of the two kinds in action, and read on for a short description of each.

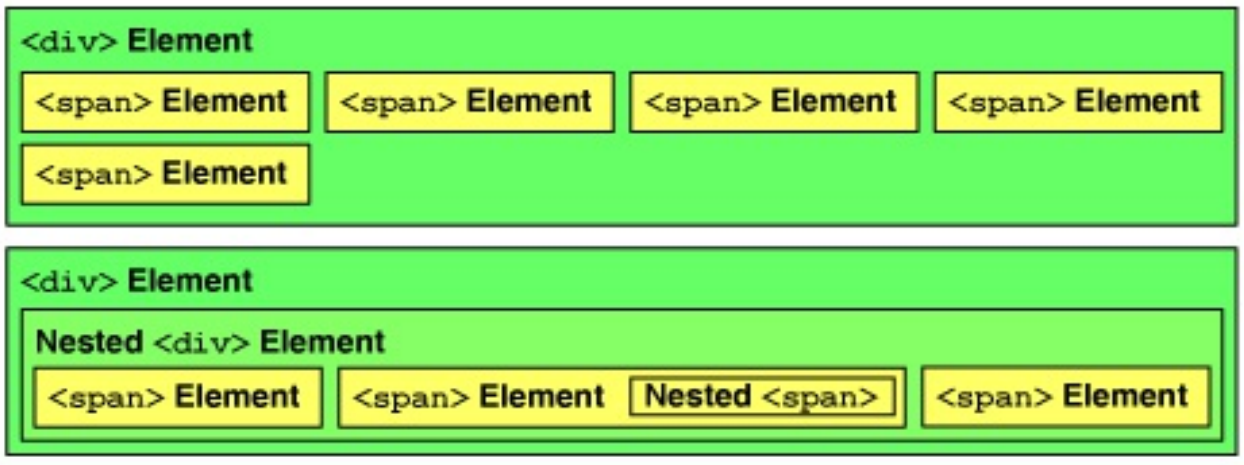


Figure 5: Diagram demonstrating how block-level elements and inline elements are rendered by default. With block-level elements as green, note how a line break is taken between each element. Conversely, inline elements can appear on the same line beside each other. You can also nest block-level and inline elements within each other, but block-level elements cannot be nested within an inline element.

24.6.1. Block-Level Elements

In simple terms, a block-level element 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 in Section 24.8. 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.

24.6.2. 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 Section 24.5.1, 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 clas
</div>
```

Refer back to Figure 5 to refresh your mind about what you can and cannot nest before you move on.

24.7. 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.

24.7.1. 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="blue">Span 2</span>
</div>
```

```
.container {
    border: 1px solid black;
}

.yellow {
    background-color: yellow;
    border: 1px solid black;
}

.blue {
    background-color: blue;
    border: 1px solid black;
}
```

This produces the output shown below.

Span 1 Span 2

We can see that each element follows its natural flow: the container element with class `container` spans the entire width of its parent container, while each of the `<span>` elements are enclosed inline within the parent. Now suppose that we wish to

then move the blue element with text `Span 2` to the right of its container. We can achieve this by modifying our CSS `.blue` class to look like the following example.

```
.blue {
    background-color: blue;
    border: 1px solid black;
    float: right;
}
```

By applying the `float: right;` property and value pairing, we should then see something similar to the example shown below.

Span 1 Span 2

Note how the `.blue` 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>`?

Span 1 Span 2

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;
}
```

Span 1 Span 2

Applying `overflow: hidden` ensures that that our `.container` pushes down to the appropriate height.

24.7.2. Relative Positioning

Relative positioning can be used if you required 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="blue">Span 2</span>
</div>
```

```
.container {
    border: 1px solid black;
    height: 200px;
}

.yellow {
    background-color: yellow;
    border: 1px solid black;
}

.blue {
    background-color: blue;
    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.

Span 1 Span 2

Now let's attempt to position our `.blue` `<span>` element relatively. First, we apply the `position: relative;` property and value pairing to our `.blue` class, like so.

```
.blue {
    background-color: blue;
    border: 1px solid black;
    position: relative;
}
```

This has no effect on the positioning of our `.blue` element. What it does do however is change the positioning of `.blue` 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 at.

```
.blue {
    background-color: blue;
    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 `.blue` 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 our experimentation can be seen below.

Span 1

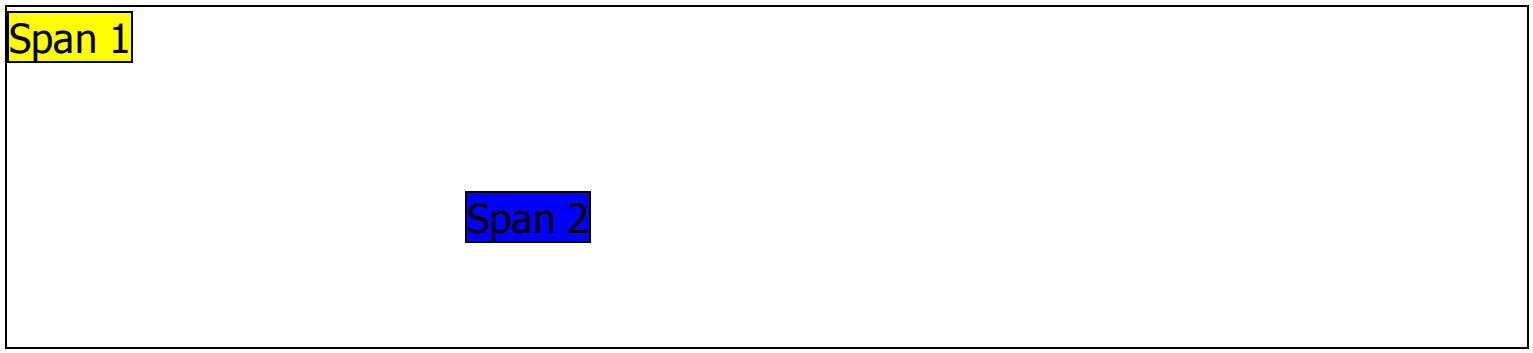
Span 2

From this behaviour, we can deduce 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: blue;
    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.



Note

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 which are floated. Consider our earlier example where the two `<span>` elements were positioned on either side of the container by floating `.blue` to the right.

We can then alter the `.blue` class to the following.

```
.blue {
    background-color: blue;
    border: 1px solid black;
```

```
    float: right;
    position: relative;
    right: 100px;
}
```

This therefore 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. Neat!

24.7.3. 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="blue">Span 2</span>
</div>
```

```
.container {
    border: 1px solid black;
    height: 70px;
}

.yellow {
    background-color: yellow;
    border: 1px solid black;
}
```

```
.blue {
    background-color: blue;
    border: 1px solid black;
}
```

This produces the output shown below. Note that we again set our `.container` height to an artificial value of 70 pixels to give us more room.

Span 1 Span 2

We now apply absolute positioning to our `.blue` element.

```
.blue {
    background-color: blue;
    border: 1px solid black;
    position: absolute;
}
```

Like with relative positioning, this has no overall effect on the positioning of our blue 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 blue element like in the example below.

```
.blue {
    background-color: blue;
    border: 1px solid black;
    position: absolute;
    top: 0;
    left: 0;
}
```

Span 2

Span 1

Wow, what happened here? Our blue element is now positioned outside of our

container! You'll note that if you run this code within your own web browser window, the blue 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 blue 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.

Span 2

Span 1

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!

Span 2

Span 1

Our elements are now absolutely positioned in relation to `.container`. Awesome! 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;
}

.blue {
    background-color: blue;
    border: 1px solid black;
    position: absolute;
    float: right;
    bottom: 50px;
    left: 40px;
}
```

Span 2

Span 1

Note that we also apply `float: right;` to our `.blue` 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.

24.8. 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 Figure 6. The areas - listed from inside to outside - are the content area, the padding area, the border area and the margin area.

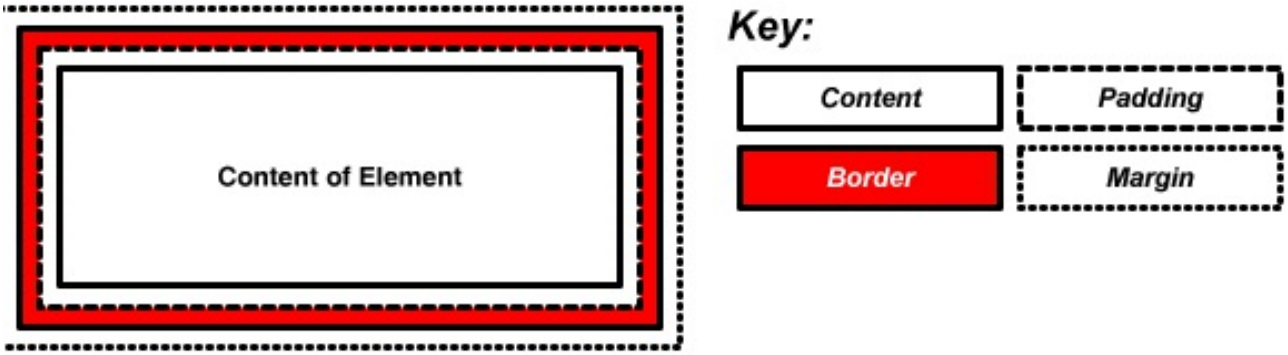


Figure 6: An illustration demonstrating the CSS box model, complete with key showing the four areas of the model.

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. Heck, why not even order a t-shirt with the box model on it?

Warning

As you may gather from examining Figure 6, 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!

24.9. 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 now lacking 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!

Django How to Tango with Django Two Scoops of Django

From the example, it is hopefully clear that lists can be easily customised to suit the requirements of your webpages. For more information and inspiration on how to style lists, you can check out some of the selected links below.

- Have a look at this excellent tutorial on styling lists on A List Apart.
- Have a look at this about.com article which demonstrates how to use your own bullets!
- Check out this advanced tutorial from Web Designer Wall which 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 awesome site compilation from devsnippets.com provides some great inspiration and tips on how you can style lists.

The possibilities of styling lists is endless! You could say it's a never-ending list...

24.10. 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. Hover over them to see them change!

Django How to Tango with Django Two Scoops of Django

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. Again, hover over the links to see them change!

Google Search

Bing Search

With a small amount of CSS, you can make some big changes in the way your webpages appear to end users.

24.11. The Cascade

It's worth pointing out where the Cascading in Cascading Style Sheets comes into play. You may have noticed in the example rendered output in Figure 1 that the red text 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 7 for a graphical representation of the cascading process.

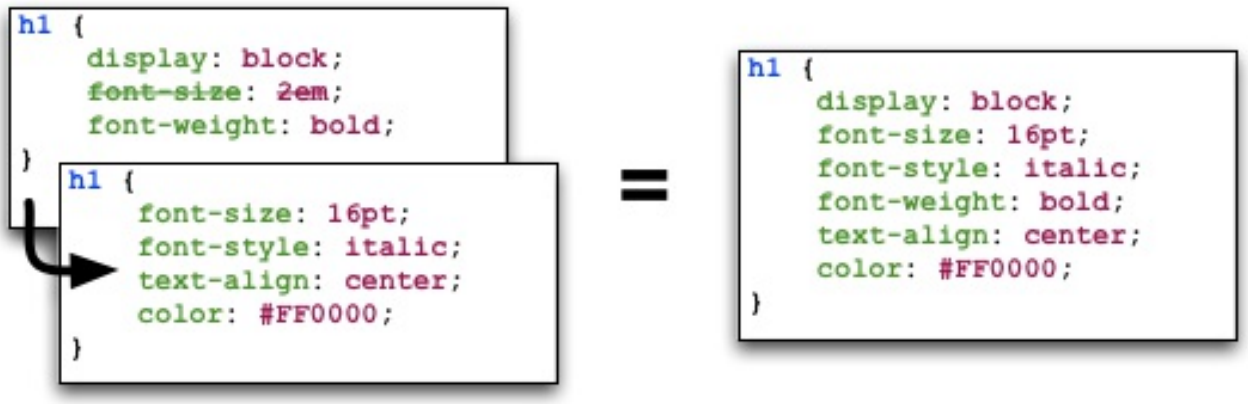


Figure 7: Illustration demonstrating the cascading in Cascading Style Sheets at work. Take note of the *`font-size`* property in our *`h1`* style - it is overridden from the default value. The cascading styles produce the resultant style, shown on the right of the illustration.

24.12. 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 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.
- It's also worth having a look at CSSeasy.com's collection of tutorials, providing you with the basics on how to develop different kinds of page layouts.

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.

Note

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!

Navigation

How To Tango With Django 1.7 »

previous