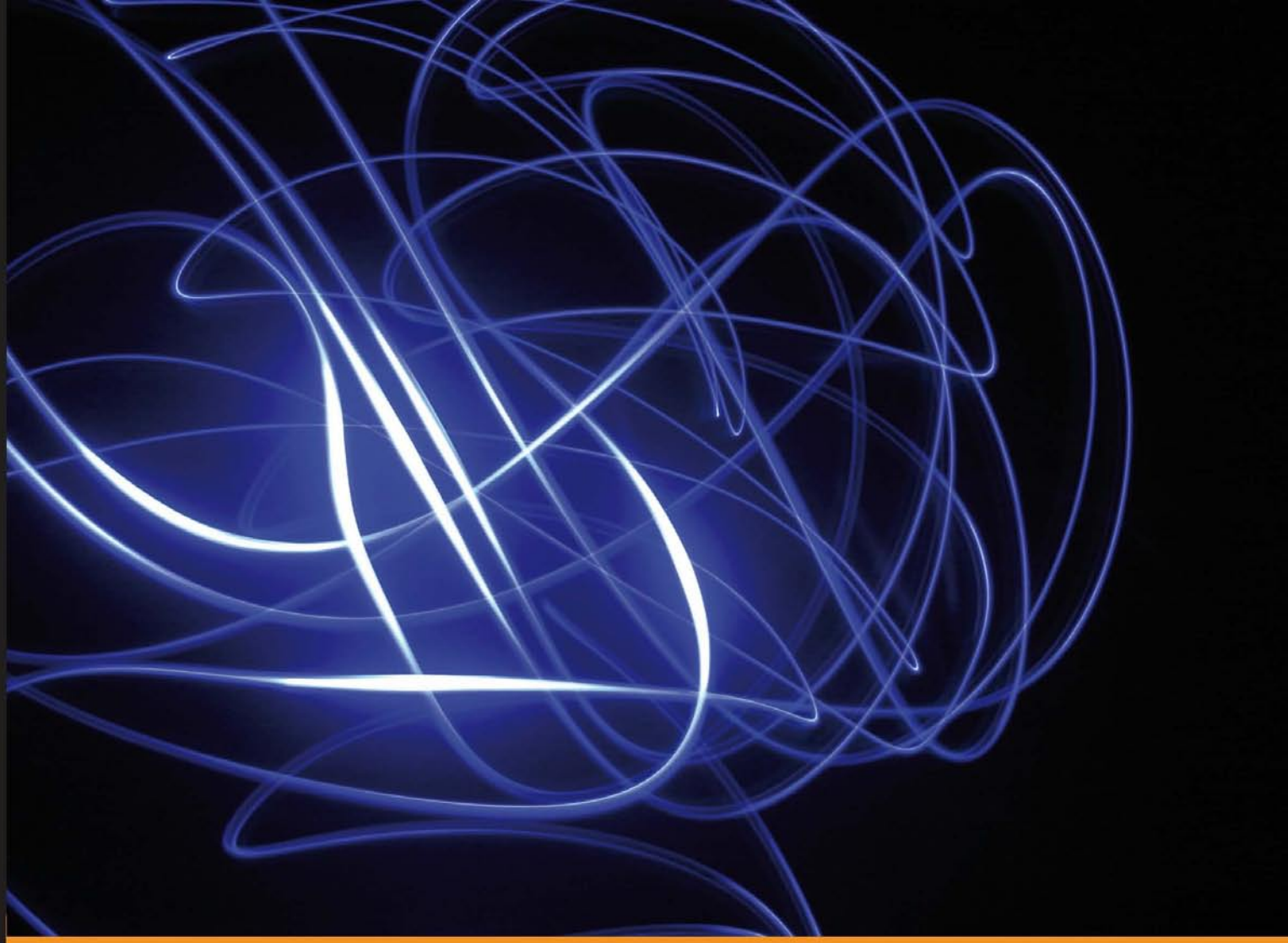


From Technologies to Solutions

Learning Website Development with Django

A beginner's tutorial to building web applications, quickly and cleanly, with the Django application framework

Ayman Hourieh

PACKT PUBLISHING

Learning Website Development with Django

A beginner's tutorial to building web applications, quickly and cleanly, with the Django application framework

Ayman Hourieh

PACKT
PUBLISHING
BIRMINGHAM - MUMBAI

Learning Website Development with Django

First published: April 2008.

Production Reference: 1040408

Published by Packt Publishing Ltd.
32 Lincoln Road
Olton
Birmingham, B27 6PA, UK.

ISBN 978-1-847193-35-3

www.packtpub.com

Cover Image by Raghuram Ashok (raghuram@iiitb.ac.in)

Credits

Author

Ayman Hourieh

Reviewers

Susmita Basu

Michael Cassidy

Wendy Langer

Jan Smith

Senior Acquisition Editor

Douglas Paterson

Development Editor

Ved Prakash Jha

Technical Editor

Sarah Cullington

Editorial Team Leader

Mithil Kulkarni

Project Manager

Abhijeet Deobhakta

Project Coordinator

Zenab Kapasi

Indexer

Monica Ajmera

Proofreaders

Martin Brooks

Chris Smith

Production Coordinator

Aparna Bhagat

Shantanu Zagade

Cover Designer

Aparna Bhagat

About the Author

Ayman Hourieh holds a bachelor degree in Computer Science. He joined the engineering team at Google in January 2008. Prior to that, he worked with web application development for more than two years. In addition, he has been contributing to several Open Source projects such as Mozilla Firefox. Ayman also worked as a teaching assistant in Computer Science courses for one year. Even after working with a variety of technologies, Python remains Ayman's favorite programming language. He found Django to be a powerful and flexible Python framework that helps developers to produce high-quality web applications in a short time.

I would like to thank my wife, Nadia, for all her help in writing this book. Not only did she provide support and motivation, but she also helped me greatly in proofreading and testing. I would also like to thank my mother and father for their continuous support and encouragement.

About the Reviewers

Michael Cassidy holds a bachelor degree in Computer Science. He currently works for Obsidian Consulting working on Python web applications. His primary focus is on automated testing of web applications.

Michael has been on a team using Django to update the database system of Computerbank, who recycle donated computers with quality, free software and distribute them to disadvantaged individuals and community groups.

Wendy Langer first learned to program in Microbee Basic. This all happened a long time ago, in a galaxy far, far, away. Later she learned Fortran and a little C++ while studying for a physics degree at University. Eventually she discovered the Python language, and thus began a love affair, which has not yet ended.

She has worked as a programmer in web development using technologies such as Python, Zope, Django, mySQL, and postgreSQL.

I would like to thank Jan Smith and Kerry Langer for their help during the review process.

Jan V. Smith has been working on open-source software since 2001. She is a Co-founder of OzZope, the Australian Zope Users Group. She contributed to 'Content Management mit Zope' by Stephan Richter and was a reviewer for 'Web Component Development with Zope 3' by Philipp von Weitershausen. Jan has written documentation for the open-source CMS Silva and a number of articles on issues relating to open source software.

Jan is Vice President of 'Computerbank Victoria' where donated computers are recycled with Linux and open source software and distributed to people on low incomes. She has built Computerbank's Plone and Silva websites and is currently building a Django database for Computerbank with Wendy Langer.

She lives in Melbourne Australia with her husband and son.

Table of Contents

Preface

Django is a high-level Python web application framework designed to support the development of dynamic websites, web applications, and web services. It is designed to promote rapid development and clean, pragmatic design and lets you build high-performing, elegant web applications quickly.

In this book, you will learn about employing this MVC web framework, which is written in Python, a powerful and popular programming language. The book emphasizes utilizing Django and Python to create a Web 2.0 bookmark-sharing application with many common features found in today's Web 2.0 sites. The book follows a tutorial style to introduce concepts and explain solutions to problems. It is not meant to be a reference manual for Python or Django. Django will be explained as we build features throughout the chapters, until we realize our goal of having a working Web 2.0 application for storing and sharing bookmarks.

I sincerely hope that you will enjoy reading the book as much as I enjoyed writing it. And I am sure that by its end, you will appreciate the benefits of using Python and Django for your next project. They are powerful, simple, and provide a robust environment for rapid development of your dynamic web applications.

What This Book Covers

Chapter 1 gives you an introduction to MVC web development frameworks, a history of Django, and explains why Python and Django are the best tools to use to achieve the aim of this book.

Chapter 2 provides a step-by-step guide to installing Python, Django and an appropriate database system so that you can create an empty project and set-up the development server.

Chapter 3 creates the main page so that we have the initial view and a URL. You will learn how to create templates for both the main page and the user page.

Chapter 4 is where the application really starts to take shape as user management is implemented. Learn how to log users in and out, create a registration form and allow users to manage their own accounts by changing email or password details.

Chapter 5 explores how to manage your growing bank of content. Create tags, tag clouds, and a bookmark submission form all of which interact with your database. Security features also come into play as you learn how to restrict access to certain pages and protect against malicious input.

Chapter 6 enables you to enhance your application with AJAX and jQuery as users can now edit entries in place and do live searching. Data entry is also made easier with the introduction of auto-completion.

Chapter 7 shows you how to enable users to vote and comment on their bookmark entries.

Chapter 8 focuses on the administration interface. You will learn how to create and customize the interface, which allows you to manage content and to set permissions for users and groups.

Chapter 9 will give your application a much more professional feel through the implementation of RSS feeds and pagination.

Chapter 10 tackles social networks providing the 'social' element of your application. Users will be able to build a friend network, browse the bookmarks of their friends, and invite their friends to join the website.

Chapter 11 covers extending and deploying your application. You will also learn about advanced features including offering the site in multiple languages, managing the site during high traffic, and configuring the site for a production environment.

Chapter 12 takes a brief look at extra Django features that have not been covered elsewhere in the book. You will gain the knowledge required to further your application and build on the basic skills that you have learned throughout the book.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

There are three styles for code. Code words in text are shown as follows: "We can include other contexts through the use of the `include` directive."

A block of code will be set as follows:

```
urlpatterns = patterns('',
  # Account management
  (r'^save/$', bookmark_save_page),
  (r'^vote/$', bookmark_vote_page),
)
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items will be made bold:

```
urlpatterns = patterns('',
  # Account management
  (r'^save/$', bookmark_save_page),
  (r'^vote/$', bookmark_vote_page),
)
```

Any command-line input and output is written as follows:

```
$ python manage.py sql bookmarks
```

New terms and **important words** are introduced in a bold-type font. Words that you see on the screen, in menus or dialog boxes for example, appear in our text like this: "clicking the **Next** button moves you to the next screen".

Important notes appear in a box like this.

Tips and tricks appear like this.

Reader Feedback

Feedback from our readers is always welcome. Let us know what you think about this book, what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply drop an email to `feedback@packtpub.com`, making sure to mention the book title in the subject of your message.

If there is a book that you need and would like to see us publish, please send us a note in the **SUGGEST A TITLE** form on `www.packtpub.com` or email `suggest@packtpub.com`. If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer Support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the Example Code for the Book

Visit `http://www.packtpub.com/files/code/3353_Code.zip` to directly download the example code.

The downloadable files contain instructions on how to use them.

Errata

Although we have taken every care to ensure the accuracy of our contents, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in text or code—we would be grateful if you would report this to us. By doing this you can save other readers from frustration, and help to improve subsequent versions of this book. If you find any errata, report them by visiting `http://www.packtpub.com/support`, selecting your book, clicking on the **let us know** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata added to the list of existing errata. The existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with some aspect of the book, and we will do our best to address it.

1
Introduction to Django

Welcome! In this book, you will learn about Django, an Open Source web framework that enables you to build clean and feature-rich web applications with minimal time and effort. Django is written in Python, a general purpose language that is well suited for developing web applications. Django loosely follows a model-view-controller design pattern, which greatly helps in building clean and maintainable web applications.

This chapter gives you an overview of the technologies used in this book. The following chapters will take you through a tutorial for building a social bookmarking application from the group using Django.

In this introduction, you will read about the following:

- The MVC pattern in web development.
- Why we should use Python.
- Why we should use Django.
- The history of Django.

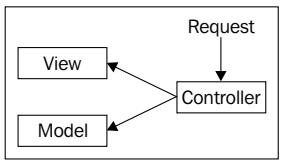
The MVC Pattern in Web Development

Web development has made great progress during the last few years. It began as a tedious task that involved using CGI for interfacing external programs with the web server. CGI applications used standard I/O facilities available to the C programming language in order to manually parse user input and produce page output. In addition to being difficult to work with, CGI required a separate copy of the program to be launched for each request, which used to quickly overwhelm servers.

Next, scripting languages were introduced to web development, and this inspired developers to create more efficient technologies. Languages such as Perl and PHP quickly made their way into the world of web development, and as a result, common web tasks such as cookie handling, session management, and text processing became much easier. Although scripting languages included libraries to deal with day-to-day web-related tasks, they lacked unified frameworks, as libraries were usually disparate in design, usage, and conventions. Therefore, the need for cohesive frameworks arose.

A few years ago, the model-view-controller pattern came for web-based applications was introduced. This software engineering pattern separates data (model), user interface (view), and data handling logic (controller), so that one can be changed without affecting the others. The benefits of this pattern are obvious. With it, designers can work on the interface without worrying about data storage or management. And developers are able to program the logic of data handling without getting into the details of presentation. As a result, the MVC pattern quickly found its way into web languages, and serious web developers started to embrace it in preference to previous techniques.

The diagram below shows how each of the components of the MVC pattern interact with each other to serve a user request:



Why Python?

Python is a general purpose programming language. Although it is used for a wide variety of applications, Python is very suitable for developing web applications. It has a clean and elegant syntax, and is supported by a large library of standard and contributed modules, which covers everything from multi-threading to the zipping of files. The language's object-oriented model is especially suited for MVC style development.

Sooner or later, performance will become a major concern with web projects, and Python's runtime environment shines here, as it is known to be fast and stable. Python supports a wide range of web servers through modules, including the infamous Apache. Furthermore, it is available for all the major platforms: UNIX/ Linux, Windows, and Mac. Python also supports a wide array of database servers, but you won't have to deal directly with them; Django provides a unified layer of access to all available database engines, as we will see later.

Python is free software; you can download and use it freely from `http://python.org/`. You are even allowed to distribute it without having to pay any fees. Access to the source code is available to those who want to add features or fix bugs. As a result, Python enjoys a large community of developers who quickly fix bugs and introduce new features.

Python is very easy to learn, and it is being adopted in many universities as the first programming language to be taught. Although this book assumes working knowledge of Python, advanced features will be explained as they are used. If you want to refresh your Python knowledge, you are recommended to read the official Python tutorial available at `http://python.org/doc/` before continuing with this book.

To sum up, Python was chosen over many other scripting languages for this book for the following reasons:

- Clean and elegant syntax.
- Large standard library of modules that covers a wide range of tasks.
- Extensive documentation.
- Mature runtime environment.
- Support for standard and proven technologies such as Linux and Apache.

If you want to learn more about Python and its features, the official Python website at `http://python.org/` and the Python book "Dive Into Python" (freely available at `http://www.diveintopython.org/`) are both excellent sources.

Why Django?

Since the spread of the MVC pattern into web development, Python has provided quite a few choices when it comes to web frameworks, such as Django, TurboGears and Zope. Although choosing one out of many can be confusing at first, having several competing frameworks can only be a good thing for the Python community, as it drives the development of all frameworks further and provides a rich set of options to choose from.

Django is one of the available frameworks for Python, so the question is: what sets it apart to become the topic of this book, and what makes it popular in the Python community? The next subsections will answer these questions by providing an overview of the main advantages of Django.

Tight Integration between Components

First of all, Django provides a set of tightly integrated components; all of these components have been developed by the Django team themselves. Django was originally developed as an in-house framework for managing a series of news-oriented websites. Later its code was released on the Internet and the Django team continued its development using the Open Source model. Because of its roots, Django's components were designed for integration, reusability and speed from the start.

Object-Relational Mapper

Django's database component, the Object-Relational Mapper (ORM), provides a bridge between the data model and the database engine. It supports a large set of database systems, and switching from one engine to another is a matter of changing a configuration file. This gives the developer great flexibility if a decision is made to change from one database engine to another.

Clean URL Design

The URL system in Django is very flexible and powerful; it lets you define patterns for the URLs in your application, and define Python functions to handle each pattern. This enables developers to create URLs that are both user and search engine friendly.

Automatic Administration Interface

Django comes with an administration interface that is ready to be used. This interface makes the management of your application's data a breeze. It is also highly flexible and customizable.

Advanced Development Environment

In addition, Django provides a very nice development environment. It comes with a lightweight web server for development and testing. When the debugging mode is enabled, Django provides very thorough and detailed error messages with a lot of debugging information. All of this makes isolating and fixing bugs very easy.

Multi-Lingual Support

Django supports multi-lingual websites through its built-in internationalization system. This can be very valuable for those working on websites with more than one language. The system makes translating the interface a very simple task.

The standard features expected of a web framework are all available in Django. These include the following:

- A template and text filtering engine with simple but extensible syntax.
- A form generation and validation API.
- An extensible authentication system.
- A caching system for speeding up the performance of applications.
- A feed framework for generating RSS feeds.

Even though Django does not provide a JavaScript library to simplify working with Ajax, choosing one and integrating it with Django is a straightforward matter, as we will see in later chapters.

So to conclude, Django provides a set of integrated and mature components, with excellent documentation, at `http://www.djangoproject.com/documentation/`, thanks to its large community of developers and users. With Django available, there has never been a better time to start learning a web development framework!

History of Django

Django started as an internal project at the Lawrence Journal-World newspaper in 2003. The web development team there often had to implement new features or even entire applications within hours. Therefore, Django was created to meet the fast deadlines of journalism websites, whilst at the same time keeping the development process clean and maintainable. By the summer of 2005, Django became mature enough to handle several high traffic sites, and the developers decided to release it to the public as an Open Source project. The project was named after the jazz guitarist Django Reinhardt.

Now that Django is an Open Source project, it has gathered developers and users from all over the world. Bug fixes and new features are introduced on a daily basis, while the original development team keeps an eye on the whole process to make sure that Django remains what it is meant to be—a web framework for building clean, maintainable and reusable web applications.

Summary

Web development has achieved large leaps of progress over the last few years. The advent of scripting languages, web frameworks, and Ajax made rapid development of web applications possible and easier than ever. This book takes you through a tutorial for building a Web 2.0 application using two hot technologies—Python and Django. The application allows users to store and share bookmarks. Many of the exciting Web 2.0 applications will be explained and developed throughout this book.

In the next chapter, we will set up our development environment by installing the necessary software, and get a feel for Django by creating our first application.

2
Getting Started

Python and Django are available for multiple platforms. In this chapter, we will see how to set up our development environment on UNIX/Linux, Windows and Mac OS X. We will also see how to create our first project and how to connect it to a database.

We will learn about the following topics in this chapter:

- We will learn the following topics in this chapter:
- Installing Python.
- Installing Django.
- Installing a database system.
- Creating your first project.
- Setting up the database.
- Launching the development server.

Installing the Required Software

Our development environment consists of Python, Django, and a database system. There are many different database systems available, but for the following examples, we will be using Sqlite3 which is included in the Python download. In this section, we will see how to install the necessary software packages.

Installing Python

Django is written in Python, so the the first step in setting up our development environment is to install Python. Python is available for a variety of operating systems, and installing Python is not very different from installing any other software package. The procedure depends on your operating system.

You will need a recent version of Python. Django requires Python 2.3 or higher. The latest version of Python at the time of writing is 2.5.

We will now describe the installation process for each operating system.

Installing Python on Windows

Python has a standard installer for Windows users. You will need to go to `http://www.python.org/download/` and download the latest version. Next, double-click the `.exe` file and follow the installation instructions. The graphical installer will guide you through the installation process and create shortcuts to Python executables in the Start menu.

Once the installation has been done, we need to add the Python directory to the system path so that we can access Python while using the command prompt. To do this, open the Control Panel, double-click the System icon, go to the Advanced tab and click the Environment Variables button. A new dialog box will open. Select the Path system variable, and append the path where you installed Python. (The default path is usually `c:\PythonXX`, where `XX` is your Python version, but the folder is actually named Python25, so if your have Python version 2.5, you should name the command `c:\Python25`.) Don't forget to separate the new path from the one before it with a semicolon.

If you want to test your installation, open the Run dialog, type `python` and hit *Enter*. The Python interactive shell should open.

Installing Python on UNIX/Linux

If you use Linux or UNIX, chances are that you already have Python installed. To check, open a terminal, type `python` and hit *Enter*. If you see the Python interactive shell, then you already have Python installed:

```
Python 2.5.1 (r251:54863, May 2 2007, 16:56:35)
[GCC 4.1.2 (Ubuntu 4.1.2-0ubuntu4)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

The first line of the output indicates the version installed on your system (2.5.1 here).

If you receive an error message instead of seeing the above output, or you have an old version of Python, you should read on.

UNIX users and Linux users are recommended to install and update Python through the system package manager. Although the actual details vary from system to system, it won't be any different from installing any other package.

For APT-based Linux distributions (such as Debian and Ubuntu), open a terminal and type:

```
$ sudo apt-get update
$ sudo apt-get install python
```

Or if you have the Synaptic package manager, simply search for Python, mark its package for installation, and click on Apply.

Users of other Linux distributions are recommended to check their system documentation for information on how to use the package manager to install packages.

Installing Python on Mac OS X

Mac OS X comes with Python pre-installed. However, due to Apple's release cycle, it's often an old version. If you start the Python interactive shell and find a version older than 2.3, you should visit this URL: `http://www.python.org/download/mac/` and download the most recent installer for your version of Mac OS X.

Now that Python is up and running, we are almost ready. Next, we will install Django and make sure that we have a database system.

Installing Django

Installing Django is very easy, but it depends on your operating system. Since Python is a platform-independent language, Django has one package that works everywhere regardless of your operating system.

To download Django, head to `http://www.djangoproject.com/download/`, and grab the latest official version. The code in this book was developed on Django 0.96 (the latest version at the time of writing), but most of the code should run on later official releases. Next, follow the instructions related to your platform.

Installing Django on Windows

After you download the Django archive, extract it to the C drive, and open a command prompt (by clicking on Start then Accessories). Change the current directory to where you extracted Django by issuing the following command, where x.xx is your Django version:

```
c:\>cd c:\Django-x.xx
```

Next, install Django by running the following command (for which you will need administrative privileges):

```
c:\Django-x.xx>python setup.py install
```

If the above instructions do not work, you can manually copy the `django` folder in the archive to the `Lib\site-packages` folder located in the Python installation directory. This will do the job of running the `setup.py` install command.

If you do not have a program to handle .tar.gz files on your system, I recommend using 7-Zip, which is free and available at `http://www.7-zip.org/`.

The last step is copying the `django-admin.py` file from `Django-x.xx\django\bin` to somewhere in your system path, such as `c:\windows` or the folder where you installed Python.

Once this has been done, you can safely remove the `c:\Django-x.xx` folder, because it is no longer needed.

That's it. To test your installation, open a command prompt and type the following command:

```
c:\>django-admin.py --version
```

If you see the current version of Django printed on screen, then everything is set.

Installing Django on UNIX/Linux and Mac OS X

Installation instructions for all UNIX and Linux systems are the same. You need to run the following commands in the directory where the `Django-x.xx.tar.gz` archive is located. These commands will extract the archive and install Django for you:

```
$ tar xfz Django-x.xx.tar.gz
$ cd Django-x.xx
$ sudo python setup.py install
```

The above instructions should work on any UNIX or Linux system as well as on Mac OS X. However, it may be easier to install Django through your system's package manager if it has a package for Django. Ubuntu has one, so to install Django on Ubuntu, simply look for a package called `python-django` in Synaptic, or run the following command:

```
$ sudo apt-get install python-django
```

You can test your installation by running this command:

```
$ django-admin.py --version
```

If you see the current version of Django printed on screen, then everything is set.

Installing a Database System

While Django does not require a database for it to function, the application that we are going to develop does. So in the last step of software installation, we are going to make sure that we have a database system for handling our data.

It is worth noting that Django supports several database engines: MySQL, PostgreSQL, MS SQL Server, Oracle, and SQLite. Interestingly however, you only need to learn one API in order to use any of these database systems. This is possibly because of Django's database layer, which abstracts access to the database system. We will learn about this later, but for now you only need to know that, regardless of which database system you choose, you will be able to run the Django applications developed in this book (or elsewhere) without modification.

If you have Python 2.5 or higher, you won't need to install anything, since Python 2.5 comes with the SQLite database management system contained in a module named `sqlite3`. Unlike client-server database systems, SQLite does not require a resident process in memory, and it stores the database in a single file, which makes it ideal for our development environment. Therefore, throughout this book, we will be using SQLite in our examples. Of course you are free to use your preferred database management system. We can tell Django what database system to use by editing a configuration file, as we will see in later sections. It is also worth noting that if you want to use MySQL, you will need to install MySQLdb, the MySQL driver for Python.

If you don't have Python 2.5, you can install the python module for SQLite manually by downloading it from `http://www.pysqlite.org/` (for Windows users) or through your package manager (for UNIX and Linux users).

Do I need Apache (or some other web server)?

Django comes with its own web server, and we are going to use it during the development phase, because it is lightweight and comes pre-configured for Django. However, Django does support Apache and other popular web servers such as Lighttpd. We will see how to configure Django for Apache when we prepare our application for deployment later in this book.

The same applies to the database manager. During the development phase, we will use SQLite because it is easy to set up, but when we deploy the application, we will switch to a database server such as MySQL.

As I said earlier, regardless of what components we use, our code will stay the same; Django handles all the communication with the web and database servers for us.

Creating Your First Project

Now that the software we need is in place, the time has come for the fun part - creating our first Django project!

As you may recall from the Django installation section, we used a command called `django-admin.py` to test our installation. This utility is at the heart of Django's project management facilities, as it enables the user to do a range of project management tasks, including the following:

- Creating a new project.
- Creating and managing the project's database.
- Validating the current project and testing for errors.
- Starting the development web server.

We will see how to use some of these tasks in the rest of this chapter, creating a basis for our bookmark-sharing application in the process.

Creating an Empty Project

To create your first Django project, open a terminal (or command prompt for Windows users), type the following command, and hit enter:

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

This command will make a folder named `django_bookmarks` in the current directory, and create the initial directory structure inside it. Let's see what kinds of files are created:

```
django_bookmarks/
    __init__.py
    manage.py
    settings.py
    urls.py
```

Here is a quick explanation of what these files are:

File Name	File Description
`__init__.py`	Django projects are Python packages, and this file is required to tell Python that the folder is to be treated as a package. A package in Python's terminology is a collection of modules, and they are used to group similar files together and prevent naming conflicts.
`manage.py`	This is another utility script used to manage your project. You can think of it as your project's version of `django-admin.py`. Actually, both `django-admin.py` and `manage.py` share the same back-end code.
`settings.py`	This is the main configuration file for your Django project. In this file you can specify a variety of options, including the database settings, site languages, which Django features are to be enabled, and so on. Various sections of this file will be explained as we build our application during the next chapters, but in this chapter, we will only see how to enter the database settings.
`url.py`	This is another configuration file. You can think of it as a mapping between URLs and Python functions that handle them. This file is one of Django's powerful features, and we will see how to utilize it in the next chapter.

When we start writing code for our application, we will create new files inside the project's folder. So the folder also serves as a container for our code.

Now that you have a general idea of the structure of a Django project, let's configure our database system.

Setting up the Database

In this section, we will work with code for the first time. Therefore, we will have to choose a source code editor to enter and edit code. There are many options on the market when it comes to source code editors. Some people prefer fully-fledged IDEs, whereas others like simple text editors. The choice is totally up to you; pick whichever you feel more comfortable with. If you already use a certain program to work with Python source files, then I suggest that you stick to it, as it will work just fine with Django. Otherwise, I can make a few recommendations:

- Scite (also known as Scintilla): This editor is lightweight yet very powerful. It is available for all major platforms, supports syntax highlighting and code completion, and works well with Python. The editor is Open Source and you can find it at `http://www.scintilla.org/SciTE.html`.
- EditPlus: This is another powerful editor for the Windows platform, and it supports syntax highlighting for Python through an extension. You can find this editor and the Python extension at `http://www.editplus.com/`. EditPlus note that EditPlus is shareware, and the free version can only be used for thirty days.
- TextMate: This popular text editor for Mac OS X also provides a rich set of features for Django developers, while being user-friendly at the same time. TextMate is not free but there is a thirty days trial version that you can download from `http://macromates.com/`.
- Eclipse + PyDev: This combination is an integrated development environment for Python. It supports all the standard features of IDEs from source version management to integrated debugging. It takes a while to learn all of its features, but for those who prefer a complete IDE (and especially those familiar with Eclipse), it is an excellent choice. More information on installation is available at `http://pydev.sourceforge.net/`.

Now that you have a source code editor ready, let's open `settings.py` in the project folder and see what it contains:

```
# Django settings for django_bookmarks project.
DEBUG = True
TEMPLATE_DEBUG = DEBUG
ADMINS = (
    # ('Your Name', 'your_email@domain.com'),
)
MANAGERS = ADMINS
DATABASE_ENGINE = ''   # 'postgresql_psycopg2', 'postgresql',
                       # 'mysql', 'sqlite3' or 'ado_mssql'.
DATABASE_NAME = ''     # Or path to database file
```

```
                        # if using sqlite3.
DATABASE_USER = ''      # Not used with sqlite3.
DATABASE_PASSWORD = ''  # Not used with sqlite3.
DATABASE_HOST = ''      # Set to empty string for localhost.
                        # Not used with sqlite3.
DATABASE_PORT = ''      # Set to empty string for default.
                        # Not used with sqlite3.

# The rest of the file was trimmed.
```

As you may have already noticed, the file contains a number of variables that control various aspects of the application. Entering a new value for a variable is as simple as doing a Python assignment statement. In addition, the file is extensively commented. These comments explain what each variable controls.

What concerns us now is configuring the database. As mentioned before, Django supports several database systems, so first of all we have to specify the database system that we are going to use. This is controlled by the `DATABASE_ENGINE` variable. As we are using SQLite, set this variable to `'sqlite3'`.

Next is the database name. We will choose a descriptive name for your database; edit `DATABASE_NAME` and set it to `'bookmarksdb'`. When using SQLite, this is all that you need to do. (If you are using a database server, you will need to enter information into the rest of the fields shown above and create the actual database inside the database server.)

After those simple edits, the database section in `settings.py` now looks like this:

```
DATABASE_ENGINE = 'sqlite3'
DATABASE_NAME = 'bookmarksdb'
DATABASE_USER = ''
DATABASE_PASSWORD = ''
DATABASE_HOST = ''
DATABASE_PORT = ''
```

Finally, we will tell Django to populate the configured database with tables. Although we haven't created any tables for our data yet (and we won't do so until the next chapter), Django requires several tables in the database for some of its features to function properly. Creating these tables is easy as it is only a matter of issuing the following command:

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

If you have entered the above command and everything is correct, status messages will scroll on the screen indicating that the tables are being created. When prompted for the superuser account, enter your preferred username, email address and password. It is important to create a superuser account otherwise you won't be able to gain access to your initial webpage once you have created it. If, on the other hand, the database is mis-configured, an error message will be printed to help you troubleshoot the issue.

With this done we are ready to launch our application.

Using python manage.py

When running a command that starts with `python manage.py`, make sure that you are currently in the project's directory where `manage.py` is located.

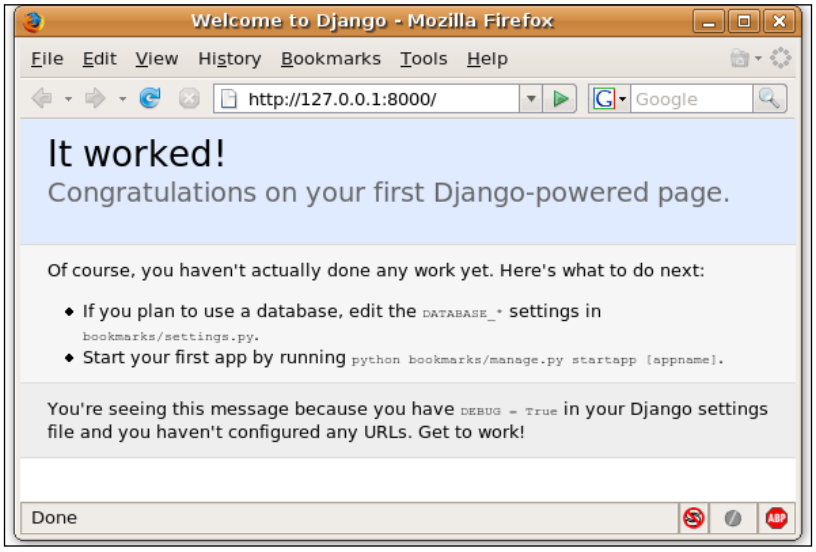
Launching the Development Server

As discussed before, Django comes with a lightweight web server for developing and testing applications. This server is pre-configured to work with Django, and more importantly, it restarts whenever you modify the code.

To start the server, run the following command:

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

Next, open your browser, and navigate to `http://localhost:8000/`. You should see a welcome message as in the image below:



As you may have noticed, the web server runs on port 8000 by default. If you want to change the port, you can specify it in the command line:

```
$ python manage.py runserver <port number>
```

Congratulations! You have created and configured your first Django project. This project will be the basis on which we will build our bookmarking application. During the next chapter, we will start developing our application, and the page displayed by the web server will be replaced by something that we will have written ourselves!

Summary

In this chapter, we have prepared our development environment, created our first project, and learned how to launch the Django development server. We are now ready to start building our social bookmarking application!

Here is a quick summary of the Django features covered in this chapter:

- Here is a quick summary of the Django features covered in this chapter:
- Django can be downloaded from the official Django website at `http://www.djangoproject.com/`. Given that it is written in Python, the same package works on all major operating systems.
- To start a new Django project, issue the following command:

  ```
  $ django-admin.py startproject <project-name>
  ```

- To create database tables, issue the following command:

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

- To start the development server, issue the following command:

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

- Django project settings are stored in `settings.py`. This file is a regular Python source file that can be edited using any source code editor. To change a variable, simply assign the desired value to it.

The next chapter takes you through a tour of the main Django components and develops a working prototype for our bookmark sharing application. It's going to be a fun chapter with many new things to learn, so keep reading!

3
Building a Social Bookmarking Application

In the previous chapter we learned how to create an empty project, enter the database settings, and run the development server. Now we will start writing our bookmark-sharing application, and learn about views, models and templates in the process.

You can think of this chapter as a prolonged tour of the main Django components. You will learn how to create dynamic pages using views, how to store and manage data in the database using models, and how to simplify page generation using templates. While learning about these features, you will form a solid idea of how Django components work and interact with each other. Later chapters will explore these components deeper, as we develop more features and add them to our application.

The following topics are covered in this chapter:

- URLs and Views: Creating the main page.
- Models: Designing an initial database schema.
- Templates: Creating a template for the main page.
- Putting it all together: Generating user pages.

A Word about Django Terminology

Django is an MVC framework. However, the controller is called the "view", and the view is called the "template". The view in Django is the component which retrieves and manipulates data, whereas the template is the component that presents data to the user. For this reason, Django is sometimes called an MTV framework (where MTV stands for model template view). This different terminology neither changes the fact that Django is an MVC framework, nor affects how applications are developed. But keep the terminology in mind to avoid possible confusion if you have worked with other MVC frameworks in the past.

URLs and Views: Creating the Main Page

The first thing that comes to mind after seeing the welcome page of the development server is how can we change it? To create our own welcome page, we need to define an entry point to our application in the form of a URL, and tell Django to call a particular Python function when a visitor accesses this URL. We will write this Python function ourselves, and make it display our own welcome message.

Creating the Main Page View

A **view** in Django terminology is a regular Python function that responds to a page request by generating the corresponding page. To write our first Django view for the main page, we first need to create a Django **application** inside our project. You can think of an application as a container for views and data models. To create it, issue the following command within our `django_bookmarks` folder:

```
$ python manage.py startapp bookmarks
```

The syntax of application creation is very similar to that of project creation. We used `startapp` as the first parameter to `python manage.py`, and provided `bookmarks` as the name of our application.

After running this command, Django will create a folder named `bookmarks` inside the project folder with these three files:

- `__init__.py`: This file tells Python that `bookmarks` is a Python package.
- `views.py`: This file will contain our views.
- `models.py`: This file will contain our data models.

Now, let's create the main page view. Open the file `bookmarks/views.py` in your code editor and enter the following:

```
from django.http import HttpResponse
def main_page(request):
  output = '''
    <html>
      <head><title>%s</title></head>
      <body>
        <h1>%s</h1><p>%s</p>
      </body>
    </html>
  ''' % (
    'Django Bookmarks',
    'Welcome to Django Bookmarks',
    'Where you can store and share bookmarks!'
  )
  return HttpResponse(output)
```

The code is short and pretty straightforward. Let's go through it line by line:

- We import the class `HttpResponse` from `django.http`. We need this class in order to generate our response page.
- We define a Python function that takes one parameter named `request`; this parameter contains user input and other information. For example, `request.GET`, `request.POST` and `request.COOKIES` are dictionaries that contain get, post and cookie data respectively.
- We build the HTML code of the response page, wrap it within an `HttpResponse` object and return it.

A Django view is just a regular Python function. It takes user input as a parameter, and returns page output. But before we can see the output of this view, we need to connect it to a URL.

Creating the Main Page URL

As you may recall from the previous chapter, a file named `urls.py` was created when we started our project. This file contains valid URLs for our application, and maps each URL to a view that is a Python function. Let's examine the contents of this file and see how to edit it:

```
from django.conf.urls.defaults import *
urlpatterns = patterns('',
  # Example:
  # (r'^django_bookmarks/', include('django_bookmarks.foo.urls')),
  # Uncomment this for admin:
  #(r'^admin/', include('django.contrib.admin.urls')),
)
```

As you can probably tell, the file contains a table of URLs and their corresponding Python functions (or views). The table is called `urlpatterns`, and it initially contains example entries that are commented out. Each entry is a Python tuple that consists of a URL and its view.

The URL syntax may look familiar to you, because it uses regular expressions. Django gives you a lot of flexibility by letting you specify URL patterns using this powerful string matching technique. We will gradually learn about this syntax and how to utilize it. Let's start by removing the comments and adding an entry for the main page:

```
from django.conf.urls.defaults import *
from bookmarks.views import *
urlpatterns = patterns('',
  (r'^$', main_page),
)
```

Again, let's see the breakdown of this code:

- The file imports everything from the module `django.conf.urls.defaults`. This module provides the necessary functions to define URLs.
- We import everything from `bookmarks.views`. This is necessary to access our views, and connect them to URLs.
- The patterns function is used to define the URL table. It contains only one mapping for now — from `r'^$'` to our view `main_page`.

One last thing needs explaining before we see the view in action. The regular expression that we used will look a bit strange if you haven't used regular expressions before. It is a raw string that contains two characters, `^` and `$`. `r''`, which is the Python syntax for defining **raw** strings. If Python encounters such a raw string, then backslashes and other escape sequences are retained in the string, rather than interpreted in any way. In this syntax, backslashes are left in the string without change, and escape sequences are not interpreted. This is useful when working with regular expressions, because they often contain backslashes.

In regular expressions, `^` means the beginning of the string, and `$` means the end of the string. So `^$` basically means a string that doesn't contain anything; that is an empty string. Given that we are writing the view of the main page, the URL of the page is the root URL, and indeed it should be empty.

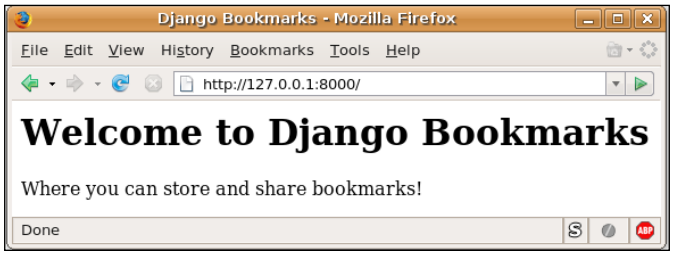
Python documentation of the `re` module covers regular expressions in detail. I recommend reading it if you want a thorough treatment of regular expressions. You can find the documentation online at:

`http://docs.python.org/lib/module-re.html`

Below is a table that summarizes regular expression syntax for those who want a quick refresher:

Symbol / Expression	Matched String
. (Dot)	Any character.
^ (Caret)	Start of string.
$	End of string.
*	0 or more repetitions.
+	1 or more repetitions.
?	0 or 1 repetitions.
|	A | B means A or B.
[a-z]	Any lowercase character.
\w	Any alphanumeric character or _.
\d	Any digit.

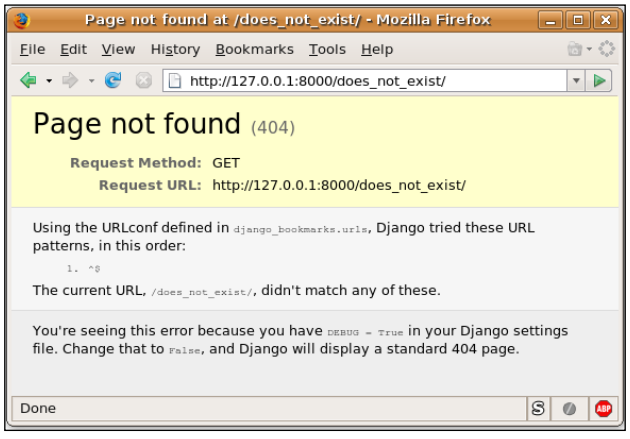
Now that everything is clear, we can test our first view. Launch the development server and go to `http://127.0.0.1:8000/` to see the page generated by the view.



Congratulations! Your first Django view is up and running.

Before we move to the next section, it is a good idea to understand what's going on behind the scenes:

- When a user requests the root URL at `http://127.0.0.1:8000/`, Django searches the URL table in `urls.py` for a URL that matches the request. Matching is done using regular expressions.
- If Django finds a matching URL, it calls its corresponding view. The view, which is a regular Python function, receives data generated by the user's browser as a parameter called the `request` object and returns the generated page wrapped in an `HttpResponse` object.
- If Django doesn't find a URL that matches the request, it displays a 404 "Page Not Found" error. You can test this by requesting `http://127.0.0.1:8000/does_not_exist/` as illustrated in the image below. Notice that Django displays helpful debugging information to assist you in figuring out what's wrong. Of course, these debugging messages can be turned off when the site goes live.



This way of mapping URLs to views gives the developer a lot of flexibility. URLs are not restricted to filenames as in PHP, and are not automatically mapped to function names as in `mod_python`. You are given total control over the mapping between URLs and functions. This is especially good for large projects, where URLs and function names often change during phases of the development.

Our main page looks a little basic without CSS. Therefore, we will learn to use templates, which will make it easy to style our pages using stylesheets. Before doing this, we will learn about database models and how to store and manage our data.

Models: Designing an Initial Database Schema

Almost every Web 2.0 application requires a database to store and manage its data. The database engine is a fundamental component of web development nowadays. Web applications offer the user a UI to enter and manage their data, and use a database engine behind the scenes to manage this data.

In Django, you can think of the view as the component responsible for collecting and displaying data, and the model as the component responsible for storing and managing it.

We will chose the database engine that configured our database settings in the previous chapter. In this section, we will make use of the database to store and manage user accounts and bookmarks.

If you are used to dealing with the database directly through SQL queries, then you may find Django's approach to database access a bit different. Loosely speaking, Django abstracts access to database tables through Python classes. To store, manipulate and retrieve objects from the database, the developer uses a Python-based API. In order to do this, SQL knowledge is useful but not required.

This technique is best explained by example. For our bookmarking application, we need to store three types of data in the database:

- Users (ID, username, password, email)
- Links (ID, URL)
- Bookmarks (ID, title, user_id, link_id)

Each user will have their own entry in the Users table. This entry stores the username, password and email. Similarly, each link will have a corresponding entry in the links table. We will only store the link's URL for now.

As for the Bookmarks table, you can think of it as the joining table between Users and Links. When a user adds a bookmark, an entry for the bookmark's URL is added to the links table if it doesn't already exist, and then a joining entry is added to the Bookmarks table. This entry connects the user with the link, and stores the title that the user entered for their bookmark.

To convert this table design into Python code, we need to edit `models.py` in `bookmarks` and enter the details of each object type. `models.py` is the file where database models are stored, and it only contains an `import` line when it's created by `manage.py startapp`.

The Link Data Model

Let's start by creating the data model for the Links table, because it's the simplest one. Open `bookmarks/models.py` in your editor and type the following code:

```
from django.db import models
class Link(models.Model):
  url = models.URLField(unique=True)
```

Going through the code line by line, we learn the following:

- The `models` package contains classes that are required to define models, so it is imported first.
- We define a class named `Link`. This class inherits from `models.Model`, which is the base class for all models. The class contains one field named `url`, and it's of the type `models.URLField`. This field must be unique.

`models.URLField` is one of many field types provided by Django. Below is a partial table of these types:

Field Type	Description
IntegerField	An integer.
TextField	A large text field.
DateTimeField	A date and time field.
EmailField	An email field with 75 chars max.
URLField	A URL field with 200 chars max.
FileField	A file-upload field.

To use this model, we first need to activate it in our Django project. This is done by editing `settings.py`, looking for the `INSTALLED_APPS` variable, and adding our application name (`django_bookmarks.bookmarks`) to it:

```
INSTALLED_APPS = (
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.sites',
    'django_bookmarks.bookmarks',
)
```

Now, issue the following command to create a table for the Link data model in the database:

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

You may remember that we used this command in the previous chapter to create Django's own administrative tables. Whenever you add a data model, you need to issue this command in order to create its table in the database.

If you are familiar with SQL, you can examine the SQL query generated by Django by running the following command:

```
$ python manage.py sql bookmarks
```

The output from this command may differ slightly depending on your database engine. For SQLite, it should look similar to the following:

```
BEGIN;
CREATE TABLE "bookmarks_link" (
    "id" integer NOT NULL PRIMARY KEY,
    "url" varchar(200) NOT NULL UNIQUE
);
COMMIT;
```

What has happened here? Django analyzed our Link model, which is a regular Python class, and generated an SQL `CREATE` statement for a database table named `bookmarks_link`; this table will store instances of the Link class. Notice that Django automatically added an `id` field to the table. This field is the primary key of the table and can be used to identify links and connect them to bookmarks.

The power of Django's Python database API does not stop at creating tables. We can use it to create entries in the table, to modify entries, to search and browse the table contents, among many other things. To explore the API, we will use the Python interactive console. To launch the console use this command:

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

This shell differs from the standard Python shell in two ways. Firstly, the project path is added to `sys.path`, which simplifies importing modules from our project. Secondly, a special environment variable is created to hold the path to our `settings.py`. So whenever you need a Python shell to interact with your project, use the above command.

Now import the contents of the models module:

```
>>> from bookmarks.models import *
```

To add a new link, create a new instance of the `Link` class and call the `save` method on it:

```
>>> link1 = Link(url='http://www.packtpub.com/')
>>> link1.save()
>>> link2 = Link(url='http://www.example.com/')
>>> link2.save()
```

Calling the `save` method is required to store the object in the database. Before that, the object exists in memory only, and it will be lost if you close the console. However, when you call `save`, the object is stored in the database.

Table fields become attributes of objects. To examine and change a link's URL, type the following:

```
>>> link2.url
          'http://www.example.com/'
>>> link2.url = 'http://www.google.com/'
>>> link2.save()
```

To get a list of all available Link objects, type the following:

```
>>> links = Link.objects.all()
>>> for link in links:
...   print link.url
...
http://www.packtpub.com/
http://www.google.com/
```

To get an object by ID, type the following:

```
>>> Link.objects.get(id=1)
        <Link: Link object>
```

Finally, to delete a link, use the following:

```
>>> link2.delete()
>>> Link.objects.count()
        1
```

And output of 1 indicates that there is now only one remaining link. Notice that we were able to do all of the above without needing a single SQL statement. Django's model API can do a lot more; it covers all the tasks commonly done through SQL. Actually, the above method calls are converted to SQL statements, and the results of the statements are returned. The benefits of this approach are numerous:

- You don't have to learn another language to access the database. You already know Python and how use it to write views, so it's obviously a benefit if you can also use it to access the database.
- Django transparently handles the conversion between Python objects and table rows. You only work with Python objects, and Django automatically stores and retrieves them from the database for you.
- You don't have to worry about any special SQL syntax for different database engines, especially if you have to switch from one engine to another. The Django model API is the same no matter what engine you use, and it takes care of the differences for you.

The User Data Model

Now that you are comfortable with the concept of data models, let's move to the `User` object. Fortunately for us, management of user accounts is so common that Django comes with a user model ready for us to use. This model contains fields that are often associated with user accounts, such as username, password and email and so on. The model is called `User` and it's located in the `django.contrib.auth.models` package.

To explore the contents of this model, open the interactive console and type the following:

```
>>> from django.contrib.auth.models import User
>>> User.objects.all()
          [<User: ayman>]
```

We can see that the table represented by this model already contains a user. Remember when we configured the database in the previous chapter? We had to create the superuser account, and Django stored this user in the table of the User model.

You can examine the fields of `User` objects by using the `dir` function:

```
>>> user = User.objects.get(id=1)
>>> dir(user)
```

You will get a very long list of attributes. Among them you will find the username, email address and password. Good news! This model fulfills our needs. Django provides all the necessary attributes for our user objects, and we can use directly use the `User` model directly without any extra code.

The Bookmark Data Model

Only one model remains, the `Bookmark` model. When we examined it earlier, we realized that a bookmark connects a user to a link. A bookmark belongs to one user and one link. However, one user may have many bookmarks, and one link may be bookmarked by many users. In database language we say there is a many-to-many relationship between users and links. However, there is no way to actually represent a many-to-many relationship such as this one using a standard database system. In our particular case, we will invent the concept of a bookmark to break up this many-to-many relationship into its constituent one-to-many relationships.

The first of these is the one-to-many relationship between the user and their bookmarks. One user can have many bookmarks, but each bookmark is associated with only one user. That is to say, each user can bookmark a particular link once.

The second of these is the one-to-many relationship between a link and its bookmarks. One link can have many bookmarks associated with it if multiple users have bookmarked it, but each bookmark is associated with only one link.

Now that we have two separate one-to-many relationships, it is possible to represent all of this in a database system. To do so, we create a third table, the bookmarks table, which connects the user table and the links table. Each row in the bookmarks table has a reference to a row in the users table (that is to a particular user) and a reference to the links table (that is to a particular link). In SQL, these references to rows in "foreign" tables are known as foreign keys, but instead of working with SQL to define tables and foreign keys, we will use the Django model API to write a data model for the bookmarks.

The following code listing contains the Bookmark data model. You should insert it into `bookmarks/models.py`. For those who are new to relational databases and SQL, this section may seem a little obscure at first. However, it will all make more sense when you see it in action. Here is the code for the Bookmark data model:

```
from django.contrib.auth.models import User
class Bookmark(models.Model):
  title = models.CharField(maxlength=200)
  user = models.ForeignKey(User)
  link = models.ForeignKey(Link)
```

We first import the `User` class in order to refer to it in the Bookmark model. Next, we define a class for the Bookmark model as we did with `Link`. The new model contains a text field called `title`, and two foreign keys that refer back to the `User` and `Link` models.

Code Snippets and import statements

Python conventions suggest putting `import` statements at the beginning of the source file. When adding code to an existing file, there will be new `import` statements at the beginning of the new code snippet, but you are recommended to move these statements to the beginning of the file.

After you enter the model's code into `models.py`, you need to run `manage.py syncdb` in order to create its corresponding table.

Let's examine the SQL query generated by Django to see how it automatically handles foreign keys. Again, issue the following command:

```
$ python manage.py sql bookmarks
```

And you will see the `CREATE` statement for the `Bookmark` data model:

```
BEGIN;
CREATE TABLE "bookmarks_bookmark" (
    "id" integer NOT NULL PRIMARY KEY,
    "title" varchar(200) NOT NULL,
    "user_id" integer NOT NULL REFERENCES
      "auth_user" ("id"),
    "link_id" integer NOT NULL REFERENCES
      "bookmarks_link" ("id"),
);
CREATE TABLE "bookmarks_link" (
    "id" integer NOT NULL PRIMARY KEY,
    "url" varchar(200) NOT NULL UNIQUE
);
COMMIT;
```

Notice how Django appended `_id` to the table name to create foreign key fields and generated the necessary SQL for expressing one-to-many relations.

Now that the data models are ready, we have the facilities to store and manage our data. Django offers an elegant and straightforward Python API to store Python objects in the database, thus sparing the developer the burden of working with SQL and converting between SQL and Python types and idioms.

Next, we will learn about another major Django component, the template system. We will use it to simplify working with page creation, and then make use of all the information we learned in this chapter to create bookmark listing pages for users.

Templates: Creating a Template for the Main Page

In the first section of this chapter, we created a very simple view for our application's main page. We had to embed the HTML code of the page into the view's code. This approach has many disadvantages even for a basic view:

- Good software engineering practices always emphasize the separation between UI and business logic, because it enhances reusability. However, embedding HTML within Python code clearly violates this rule.
- Editing HTML embedded within Python requires Python knowledge, but this is impractical for many development teams whose web designers do not know Python.
- Handling HTML code within Python code is a tedious and error-prone task. For example, quotation marks in HTML may need to be escaped in Python string literals, and the overall result may be unclean and unreadable code.

Therefore, we'd better separate Django views from HTML code generation before continuing with our application. Fortunately for us, Django provides a component that facilitates this task; it is called the template system.

The idea of this system is simple, instead of embedding HTML code in the view, you store it in a separate file called a template. This template may contain placeholders for dynamic sections that are generated in the view. When generating a page, the view loads the template and passes dynamic values to it. In turn, the template replaces the placeholders with these values and generates the page.

To help you better understand the concept, let's apply it to our `main_page` view. First of all, to keep our directory structure clean, we will create a separate folder called `templates` in our project folder. Next, we need to inform Django of our newly-created `templates` folder. So, open `settings.py`, look for the `TEMPLATE_DIRS` variable, and add the absolute path of your templates folder to it. If you don't want to hard-code the path into `settings.py`, you can use the following little snippet that will also work:

```
import os.path
TEMPLATE_DIRS = (
  os.path.join(os.path.dirname(__file__), 'templates'),
)
```

Next, create a file called `main_page.html` in the templates folder with the following content:

```
<html>
  <head>
    <title>{{ head_title }}</title>
  </head>
  <body>
    <h1>{{ page_title }}</h1>
    <p>{{ page_body }}</p>
  </body>
</html>
```

The structure of the template is very similar to the HTML code that we embedded in the `main_page` view. There is one small difference however; we used a special syntax to indicate sections that we wanted to change in the view. For example, `{{ head_title }}` indicates a variable called `head_title` that can be changed inside the view. Template variables are always surrounded by double braces.

Now, let's see how to use this template in the view. Edit `bookmarks/views.py` and replace its contents with the following code:

```
from django.http import HttpResponse
from django.template import Context
from django.template.loader import get_template

def main_page(request):
  template = get_template('main_page.html')
  variables = Context({
    'head_title': 'Django Bookmarks',
    'page_title': 'Welcome to Django Bookmarks',
    'page_body': 'Where you can store and share bookmarks!'
  })
  output = template.render(variables)
  return HttpResponse(output)
```

As usual, we will go through the code line by line:

- To load a template, we used the `get_template` method, which is found in the `django.template.loader` module. This method takes the filename of a template and returns a template object.
- To set variable values in the template, we created an object called `variables` of type `Context`. The constructor for this type takes a Python dictionary whose keys are variable names (without double braces), and whose values are the values of these variables.

- To replace template variables and create HTML output from the template, we used the `render` method. This method takes a `Context` object as a parameter, so here we pass the `variables` object to it.
- Finally, we returned the HTML output wrapped in an `HttpResponse` object.

As you can see, the benefits of this approach over the old one are clear. We no longer have to deal with HTML within Python. Putting the HTML code into its own file is a lot cleaner. In addition, the template system provided by Django makes template management an easy and straightforward task.

The template system offers a lot in addition to variable substitution. It provides conditional statements to test whether a variable is empty or not, and a 'for' loop to iterate through a list and print its items, among many other features. We will see how to employ some of these features in the next section, in which we will use all the knowledge that we have learned previously to create user pages.

Putting It All Together: Generating User Pages

This chapter has covered a lot of material. It has introduced the concepts of views, models and templates. In the final section, we will write another view and make use of all the information that we have learned so far. This view will display a list of all the bookmarks that belong to a certain user.

Creating the URL

The URL of this view will have the form: `user/`*username*, where *username* is the owner of the bookmarks that we want to see. This URL is different from the first URL that we added because it contains a dynamic portion. So we will have to employ the power of regular expressions in order to express this URL. Open `urls.py` and edit it so that the URL table looks like this:

```
urlpatterns = patterns('',
  (r'^$', main_page),
  (r'^user/(\w+)/$', user_page),
)
```

The pattern here looks more complicated than the first one. `\w` means an alphanumeric character or the underscore. The `+` sign after it causes the regular expression to match one or more repetitions of what precedes the sign. So in effect, `\w+` means any string that consists of alphanumeric characters and possibly the underscore. We have surrounded this portion of the regular expression with parentheses; this will cause Django to capture the string that matches this portion, and pass it to the view, as we will see later.

Writing the View

Now that we've added an entry for the new URL to the URL table, let's write the actual view for it. Open `bookmarks/views.py` and enter the following code:

```
from django.http import HttpResponse, Http404
from django.contrib.auth.models import User

def user_page(request, username):
  try:
    user = User.objects.get(username=username)
  except:
    raise Http404('Requested user not found.')
  bookmarks = user.bookmark_set.all()
  template = get_template('user_page.html')
  variables = Context({
    'username': username,
    'bookmarks': bookmarks
  })
  output = template.render(variables)
  return HttpResponse(output)
```

Most of the view should already look familiar. Therefore, we will only examine what's new:

- Unlike our first view, `user_page` takes an extra parameter in addition to the familiar request object. Remember that the pattern for this URL contains capturing parentheses? Strings captured by URL patterns are passed as parameters to views. The captured string in this URL is passed as the `username` parameter.
- We used `User.objects.get` to obtain the user object whose username is requested. We can use a similar technique to query any table by a unique column. This method throws an exception if there are no records that match the query, or if the matched record is not unique.
- If the requested username is not available in the database, we generate a 404 "Page Not Found" error by raising an exception of the type `Http404`.
- To obtain the list of bookmarks for a particular user object, we can conveniently use the `bookmark_set` attribute available in the user object. Django detects relations between data models and automatically generates such attributes. There is no need to worry about constructing SQL `JOIN` queries ourselves to obtain user bookmarks for example.

Designing the Template

The previous view loads a template called `user_page.html` and passes the username and bookmarks to it. We will write this template now. Create a file called `user_page.html` in the `templates` directory and enter the following code into it:

```
<html>
  <head>
    <title>Django Bookmarks - User: {{ username }}</title>
  </head>
  <body>
    <h1>Bookmarks for {{ username }}</h1>
    {% if bookmarks %}
      <ul>
        {% for bookmark in bookmarks %}
          <li><a href="{{ bookmark.link.url }}">
            {{ bookmark.title }}</a></li>
        {% endfor %}
      </ul>
    {% else %}
      <p>No bookmarks found.</p>
    {% endif %}
  </body>
</html>
```

This template is more involved than our first one. In addition to variables, it uses an 'if' condition and a 'for' loop to display bookmarks. The `bookmarks` variable is a list object, so we can't output it directly in the template; we have to make sure that it's not empty and then iterate through its items.

Checking whether a variable is empty or not in a template is done using the following syntax:

```
{% if variable %}
  <p>variable contains data.</p>
{% else %}
  <p>variable is empty</p>
{% endif %}
```

This 'if' condition works as expected. If the variable contains data, only the first line is printed to the browser. On the other hand, if the variable is indeed empty, only the second line is printed.

To iterate through a list and print its items, we use the following syntax:

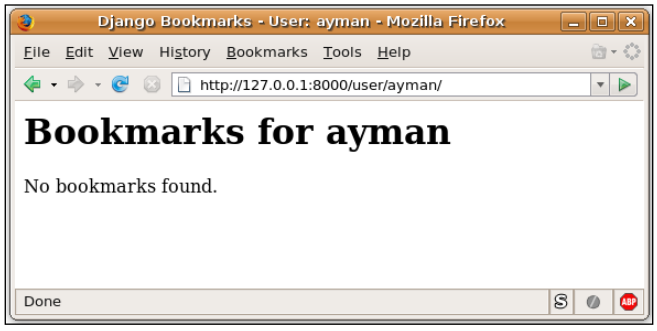
```
{% for item in list %}
  {{ item }}
{% endfor %}
```

Finally, if a variable has attributes, you can access them in a way similar to Python:

```
{{ variable.attribute }}
```

We utilized the constructs above to create the `user_page.html` template. First, it checks whether `bookmarks` is empty or not. If `bookmarks` does contain items, a 'for' loop iterates through them and creates links from them. If `bookmarks` is empty, a message is printed saying so.

Now, launch the development server and direct your browser to `http://127.0.0.1:8000/user/your_username` (replacing *your_username* with your actual username), you should see something similar to the following:



Our template worked, but the list of bookmarks is empty. This is a good opportunity to experiment with the data model API and add some bookmarks through the interactive console. As we saw earlier, you can start the console by running the following command:

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

Populating the Model with Data

First, obtain references to your user object and the link that we created in the data models section:

```
>>> from django.contrib.auth.models import User
>>> from bookmarks.models import *
>>> user = User.objects.get(id=1)
>>> link = Link.objects.get(id=1)
```

Notice that `user.bookmark_set` is empty:

```
>>> user.bookmark_set.all()
        []
```

Now create an object that connects the two:

```
>>> bookmark = Bookmark(
...   title='Packt Publishing',
...   user=user,
...   link=link
... )
>>> bookmark.save()
```

Examine the `user.bookmark_set` attribute again:

```
>>> user.bookmark_set.all()
        [<Bookmark: Bookmark object>]
```

Great, our user object now has a bookmark. Refresh the page in your browser to see the change:

Experiment with adding more bookmarks if you like. You can access a bookmark's owner by using `bookmark.user`. This is another attribute that is automatically generated by Django. Because the relation between users and bookmarks is one-to-many, each user has a set of bookmarks accessible through the `user.bookmark_set` attribute, whereas each bookmark has exactly one owner who is accessible through the `bookmark.user` attribute.

Summary

In this chapter, we learned about the three main components of Django: the view, model and template. We wrote data models to store the data of our application, and then created views and templates to display this data. We also learned how to map URLs to views, and how to use the interactive console to experiment with our Django project.

Below is a summary of the Django features covered in this chapter:

- To create an application within a project, run the following command:

  ```
  $ python manage.py startapp <app-name>
  ```

- After writing a data model, the following command should be run to create the corresponding tables in the database:

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

- To view the SQL queries generated by Django, issue the following command:

  ```
  $ python manage.py sql <app-name>
  ```

- Data models provide a variety of methods to interact with the database engine:
  - The `save` method saves an object into the database.
  - The `objects.get` method retrieves an object by a unique field.
  - The `objects.all` method retrieves a list of all objects.
  - The `delete` method deletes an object from the database.

- To generate a 404 "Page Not Found" error, raise an exception of type type `Http404`.

In the next chapter, we will continue developing our application, but we will focus mostly on user management features, such as registration and logging in. The next chapter provides a lot of useful information, so read on!

4

User Registration and Management

User registration and account management are universal features found in every web application. Users need to identify themselves to the application before they can post and share content with other users. User accounts are also required for online discussions and friend networks, among many other uses. Therefore, this chapter will focus on building features related to account registration and management, and taking advantage of the user authentication system that comes with Django.

In this chapter, you will learn about the following:

- Creating a login page.
- Enabling logout functionality.
- Creating a registration form.
- Enabling users to update their account information.

Whilst developing the above items, we will learn about two important Django features:

- Template inheritance.
- The forms library.

Session Authentication

In the previous chapter, we learned about the User data model and used it to store user information in the database. In fact, this data model is part of a larger Django application that provides a variety of features related to user authentication and management. The Django authentication system is available in the `django.contrib.auth` package. It is installed by default as part of Django, and projects created with the `django-admin.py` utility have it enabled by default.

You can double-check to make sure that you have the authentication system enabled by examining the `INSTALLED_APPS` variable in `settings.py`. This variable contains the names of the applications available for your project. You may remember that we had to edit this variable and add the `bookmarks` application that we created ourselves. The variable should look similar to the following:

```
INSTALLED_APPS = (
 'django.contrib.auth',
 'django.contrib.contenttypes',
 'django.contrib.sessions',
 'django.contrib.sites',
 'django_bookmarks.bookmarks',
)
```

In future, whenever you need to activate an application for your project, you can simply add its package name to the `INSTALLED_APPS` variable. Depending on the application, you may also need to run `python manage.py syncdb` to create the application's data models in the database. After that, the application will become available in your project.

Before we start using the authentication system, let's have a quick look at the features that it provides:

- **Users**: A comprehensive User data model with fields commonly required by web applications.
- **Permissions**: Yes/No flags that indicate whether a user may access a certain feature or not.
- **Groups**: A data model for grouping more than one user together and applying the same set of permissions to them.
- **Messages**: Provides the functionality for displaying information and error messages to the user.

We will only use features related to user management in this chapter. Later chapters will explore other features in detail.

Creating the Login Page

When we examined the User data model in the previous chapter, we noticed that it already contained a user account. This account was created during the process of starting a new project. So the natural question to ask is "how do we log in to this account?"

Those who have worked on programming a session management system in a low-level web framework (such as PHP and its library) will know that this task is not straightforward. There are many things that could go wrong in such a task, and a little mistake may open the system to security problems. Fortunately however, Django developers have carefully implemented a session management system for us, and activating it requires exposing only certain views to the user. We don't have to worry about managing user sessions or checking passwords. All of these are already implemented and ready to be used.

To introduce the session system into our project, let's start with the login page. First of all, you need to add a new URL entry to `urls.py`. Open the file in your editor and change it so that the URL table looks like the following snippet:

```
urlpatterns = patterns('',
  (r'^$', main_page),
  (r'^user/(\w+)/$', user_page),
  (r'^login/$', 'django.contrib.auth.views.login'),
)
```

The new URL entry is slightly different from the previous ones. Instead of providing the view itself, we are passing its module path as a string. This is a convenient shortcut from Django, and it is often used with views that come from a package outside the current project. Django will automatically import the view for us. The path starts with `django.contrib` which is a package that contains various add-ons for Django. As you can tell, the authentication system lives in this package.

The module `django.contrib.auth.views` contains a number of views related to session management. We have only exposed the login view for now. As its name suggests, this view handles user login requests. But before we can see it in action, we need to write a template for it.

The `login` view requires the availability of a template called `registration/login.html`. It loads this template and passes an object that represents the login form to it. We will learn about form objects in detail when we create a user registration form, but for now, we only need to know that this object is called `form` and has the following attributes: `form.username`, `form.password` and `form.has_errors`. When printed, the first two attributes generate HTML code for the username and password text fields, whereas `form.has_errors` is a Boolean attribute that is set to true if logging-in fails after submitting the form.

The next step is creating a template for the view. Make a folder named `registration` within the `templates` folder, and create a file called `login.html` in it. Enter the following code into `login.html`:

```
<html>
  <head>
    <title>Django Bookmarks - User Login</title>
```

```
    </head>
    <body>
      <h1>User Login</h1>
      {% if form.has_errors %}
        <p>Your username and password didn't match.
          Please try again.</p>
      {% endif %}
      <form method="post" action=".">
        <p><label for="id_username">Username:</label>
          {{ form.username }}</p>
        <p><label for="id_password">Password:</label>
          {{ form.password }}</p>
        <input type="hidden" name="next" value="/" />
        <input type="submit" value="login" />
      </form>
    </body>
  </html>
```

The code in this template should look familiar by now. We first check to see if there is a login error from a previous login attempt. And then create an HTML form that contains username and password fields, as well as a submit button and a hidden field called `next`. This hidden variable contains a URL that tells the view where to redirect the user after they have successfully logged in. We will redirect the user to the main page for now.

We are ready to try the login view! Run the development server and navigate to `http://127.0.0.1:8000/login/`. You will be greeted by the following login page:

Remember when you created the database in the second chapter? You had to enter a username and password for the superuser account after running `python manage.py syncdb`. You can use this account to log in. So either go ahead and enter your credentials, or try a wrong password to see the error message. After successfully logging in, you will be redirected to the main page.

Now that we can log in, it is a good idea to make the main page indicate whether you are logged in or not. So let's rewrite its view and template. First open `templates/main_page.html` and replace its contents with the following:

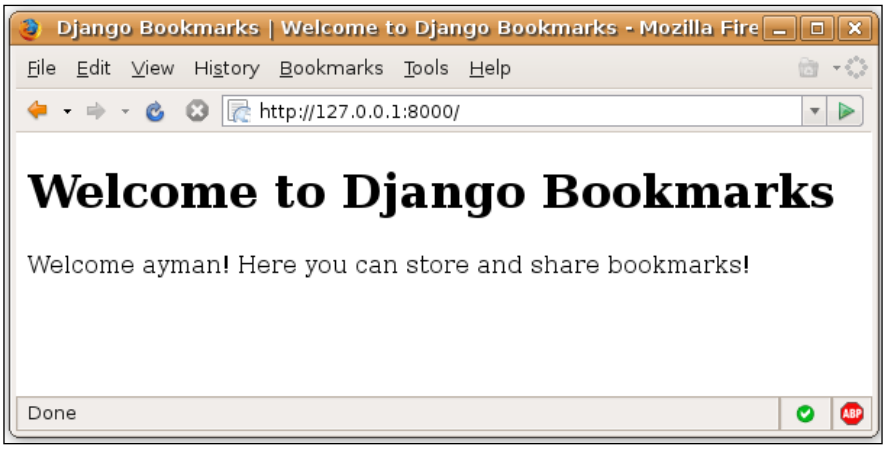
```
<html>
  <head>
    <title>Django Bookmarks</title>
  </head>
  <body>
    <h1>Welcome to Django Bookmarks</h1>
    {% if user.username %}
      <p>Welcome {{ user.username }}!
        Here you can store and share bookmarks!</p>
    {% else %}
      <p>Welcome anonymous user!
      You need to <a href="/login/">login</a>
      before you can store and share bookmarks.</p>
    {% endif %}
  </body>
</html>
```

The template now checks whether a variable called `user.username` is set or not. If it is, the logged-in user is greeted. Otherwise, a link to the login page is displayed. The template assumes that the `user` variable is passed to it (the user variable is a Django object that we will learn about shortly), so let's modify the main page view to reflect this. Open `bookmarks/views.py` and change the view as follows:

```
def main_page(request):
  template = get_template('main_page.html')
  variables = Context({ 'user': request.user })
  output = template.render(variables)
  return HttpResponse(output)
```

The code simply loads the main page template, passes `request.user` (which contains the current user object) to it, and renders the page. `request.user` is where you can access the current user object. (We will learn about this shortly.)

Reload the main page, and you will see a friendlier message that mentions the logged in username as in the figure below:



You may have noticed by now that loading a template, passing variables to it, and rendering the page is a very common task. Indeed, it is so common that Django provides a shortcut for it. Once again, let's rewrite the main page view in `bookmarks/views.py` to make use of this shortcut:

```
from django.shortcuts import render_to_response
def main_page(request):
  return render_to_response(
    'main_page.html',
    { 'user': request.user }
  )
```

Using the `render_to_response` method from the `django.shortcuts` package, we have reduced the view to one statement. This method takes the template name and a dictionary of template variables as parameters, and returns an `HttpResponse` object.

The user object available at `request.user` is the same type of `User` object as we have dealt with before. We are already familiar with its data fields, so let's learn about some of its methods:

- `is_authenticated()` returns a Boolean value indicating whether the user is logged in or not.
- `get_full_name()` returns the first name and the last name of the user, with a space between them.
- `email_user(subject, message, from_email=None)` sends an email to the user.

- `set_password(raw_password)` sets the user password to the passed value.
- `check_password(raw_password)` returns a Boolean value indicating whether the passed password matches the user password.

The names of these methods are self-explanatory. But one may wonder; why is there a `set_password` method when one can just as easily set the password attribute of the user object? To answer this question, we need to examine the contents of the password field. Open the interactive console and type the following statements:

```
>>> from django.contrib.auth.models import User
>>> user = User.objects.get(id=1)
>>> user.password
                'sha1$e1f02$bc3c0ef7d3e5e405cbaac0a44cb007c3d34c372c'
```

You will receive a different string depending on your password, but it will be a long series of random-looking characters that is totally different from your actual password. What happened here? For security reasons, Django does not store your password in plain text in the database. Instead, it stores a hash value of your password. This hash is very difficult to reverse back, but it can still be used to verify the password when processing a login request. Remember when I said that implementing a session management system carries many caveats? Password storage is one of them, and Django handles it out of the box. You only need to remember to call the `set_password` method instead of directly accessing the password attribute, and the method will hash the password for you.

Enabling Logout Functionality

Now that we have a login page, the next step is providing a way for the user to log out. When the user hits the URL `logout/`, we will log them out and redirect them back to the main page.

To do this, first create a new view in `bookmarks/views.py`:

```
from django.http import HttpResponseRedirect
from django.contrib.auth import logout
def logout_page(request):
  logout(request)
  return HttpResponseRedirect('/')
```

The method almost reads as plain English. We use the `logout` method provided by `django.contrib.auth` to invalidate the user's session. Next, we redirect the user to the main page by returning an `HttpResponseRedirect` object. The constructor of this object takes the destination URL as a parameter.

Now we need to add a URL entry for this view. Open `urls.py` and create an entry as follows:

```
urlpatterns = patterns('',
  (r'^$', main_page),
  (r'^user/(\w+)/$', user_page),
  (r'^login/$', 'django.contrib.auth.views.login'),
  (r'^logout/$', logout_page),
)
```

That's it. To test the new view, make sure that you are logged in, and then hit `http://127.0.0.1:8000/logout/`. You will be forwarded back to the main page as an anonymous user.

To make the logout link accessible to the user, we need to edit all the templates that we have created so far and add a link to them. Even for a few templates, this is impractical for many reasons. To overcome this difficulty, we will restructure our templates by utilizing a featured called template inheritance.

Improving Template Structure

We have created three templates so far. They all share the same general structure, and only differ in the title and main content. Wouldn't it be great if we could factor out the shared sections into a single file so that, if we want to modify all the pages in future, we need only edit one file?

Fortunately, the Django template system already provides such a feature-template inheritance. The idea is simple; we create a **base** template that contains the structure shared by all templates in the system. We also declare certain **blocks** of the base template to be modifiable by **child** templates. Next, we create a template that **extends** the base template and modifies its blocks. The idea is very similar to class inheritance in object-oriented programming.

Let's apply this feature to our project. Create a file called `base.html` in `templates` with the following content:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html>
<head>
  <title>Django Bookmarks | {% block title %}{% endblock %}</title>
</head>
<body>
  <h1>{% block head %}{% endblock %}</h1>
  {% block content %}{% endblock %}
</body>
</html>
```

The template utilizes a new template tag called `block`. This tag is used to define sections that are modifiable by child templates. Our base template contains three blocks, one for the title, one for the page heading and one for the body.

To see how to modify these blocks using a child template, edit `templates/main_page.html` and replace its content with the following:

```
{% extends "base.html" %}
{% block title %}Welcome to Django Bookmarks{% endblock %}
{% block head %}Welcome to Django Bookmarks{% endblock %}
{% block content %}
  {% if user.username %}
    <p>Welcome {{ user.username }}!
      Here you can store and share bookmarks!</p>
  {% else %}
    <p>Welcome anonymous user!
      You need to <a href="/login/">login</a>
      before you can store and share bookmarks.</p>
  {% endif %}
{% endblock %}
```

The new template of the main page starts by declaring that it extends `base.html`. This means that `main_page.html` is a child of `base.html`. It inherits its code and only changes blocks as necessary. Redefining a block in a child template isn't any different from declaring it for the first time. `main_page.html` doesn't contain the general HTML structure any longer; it only redefines what it wants to redefine from the base template.

Next, let's restructure `templates/user_page.html` to make use of the new base template:

```
{% extends "base.html" %}
{% block title %}{{ username }}{% endblock %}
{% block head %}Bookmarks for {{ username }}{% endblock %}
{% block content %}
  {% if bookmarks %}
    <ul>
      {% for bookmark in bookmarks %}
        <li><a href="{{ bookmark.link.url }}">
          {{ bookmark.title }}</a></li>
      {% endfor %}
    </ul>
  {% else %}
    <p>No bookmarks found.</p>
  {% endif %}
{% endblock %}
```

Again, we simply redefine dynamic sections of the base template. The code that generates bookmark listings is still exactly the same.

Finally, let's see how to convert `templates/registration/login.html`:

```
{% extends "base.html" %}
{% block title %}User Login{% endblock %}
{% block head %}User Login{% endblock %}
{% block content %}
  {% if form.has_errors %}
    <p>Your username and password didn't match.
       Please try again.</p>
  {% endif %}

  <form method="post" action=".">
    <p><label for="id_username">Username:</label>
      {{ form.username }}</p>
    <p><label for="id_password">Password:</label>
      {{ form.password }}</p>
    <input type="submit" value="login" />
    <input type="hidden" name="next" value="/" />
  </form>
{% endblock %}
```

Now that our templates have a common base, we can start improving our site usability and its appearance. Let's begin by adding a CSS stylesheet to the project. Stylesheets and images are static files; Django does not serve them. In a production environment, this task is left to the web server. But because we are currently using Django's development server, we are going to use a workaround in order to make it serve static content.

Open `urls.py`, and update it so that it becomes as follows (new code is highlighted):

```
import os.path
from django.conf.urls.defaults import *
from bookmarks.views import *
site_media = os.path.join(
  os.path.dirname(__file__), 'site_media'
)
urlpatterns = patterns('',
  (r'^$', main_page),
  (r'^user/(\w+)/$', user_page),
  (r'^login/$', 'django.contrib.auth.views.login'),
  (r'^logout/$', logout_page),
  (r'^site_media/(?P<path>.*)$', 'django.views.static.serve',
    { 'document_root': site_media }),
)
```

The new entry binds all URLs under the `site_media` directory to Django's static file serving view. Unlike previous URL entries, this one contains a third element. Some views have additional options that can be controlled by providing a dictionary as the third element in the view's URL entry. Here we are using this technique to tell the view where our static files are located.

Next, create a directory called `site_media` in your project directory. Inside it, create a blank file called `style.css`. Now, we will edit `templates/base.html` to link the stylesheet to the template and add a navigation menu:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html>
<head>
  <title>Django Bookmarks |
    {% block title %}{% endblock %}</title>
  <link rel="stylesheet" href="/site_media/style.css"
    type="text/css" />
</head>
<body>
  <div id="nav">
    <a href="/">home</a> |
    {% if user.is_authenticated %}
      welcome {{ user.username }}
      (<a href="/logout">logout</a>)
    {% else %}
      <a href="/login/">login</a>
    {% endif %}
  </div>
  <h1>{% block head %}{% endblock %}</h1>
  {% block content %}{% endblock %}
</body>
</html>
```

To correctly position the navigation menu on the page, edit the newly created stylesheet so as to add the following code:

```
#nav {
  float: right;
}
```

We are almost done. The navigation menu works well on the main page, but if you try the `user_page` view now, you will notice that the menu displays a login link no matter whether you are logged in or not. This happens because the 'if' condition in the navigation menu uses the `user` object to check the current user status but this object isn't passed to the `user_page.html` template via the `Context` object. To overcome this problem, we have two options:

- Edit our `user_page` view, and pass the `user` object to the template in it.
- Use a `RequestContext` object. This object is slightly different from normal `Context` objects. It automatically passes the `user` object to the template (along with several other variables). In order to do this, the `RequestContext` constructor takes the `request` object as its first parameter, and a dictionary of template variables as the second parameter.

We will use the second approach because it is cleaner, as we will need to pass the `user` object to every template that we write in the future. Also, it is always better to factor out common code and reduce the amount of written code.

We will modify both `main_page` and `user_page` to use `RequestContext` objects. Edit `bookmarks/views.py`, locate the `main_page` view and modify it as in the following snippet:

```
from django.template import RequestContext

def main_page(request):
  return render_to_response(
    'main_page.html', RequestContext(request)
  )
```

As you can see, we do not need to pass `request.user` through the Context object any more; `RequestContext` handles this for us. Now rewrite the `user_page` view to use `render_to_response` and `RequestContext`:

```
def user_page(request, username):
  try:
    user = User.objects.get(username=username)
  except:
    raise Http404('Requested user not found.')
  bookmarks = user.bookmark_set.all()
  variables = RequestContext(request, {
     'username': username,
     'bookmarks': bookmarks
  })
  return render_to_response('user_page.html', variables)
```

The code for the views looks much more compact now. Although the restructuring of the templates took some time, the new structure will serve us better in the long term.

Our application now supports logging in and out. In the next section we are going to add user registration functionality.

User Registration

The first user account was added to the database during the creation of our Django project. However, site visitors also need a method to create accounts on the site. User registration is a basic feature found in all social networking sites nowadays. We will create a user registration form in this section, and in the process we will also learn about the Django library that handles form generation and processing.

Django Forms

Creating, validating and processing forms is an all too common task. Web applications receive input and collect data from users by means of web forms. So naturally Django comes with its own library to handle these tasks. In Django 0.96, the library is called `newforms` but it will be renamed to `forms` in Django 1.0 (once it has been released). To avoid having to go through a lot of code updates when the name changes, it is recommended to import the package like this:

```
from django import newforms as forms
```

This way, when the library name changes in Django 1.0, all you have to do is change the above `import` statement to:

```
from django import forms
```

This package renaming issue exists because `newforms` replaces a previous form handling library that was called `forms`. To give developers time to update their code for the new forms library, Django developers decided to let the two libraries coexist under different names for a while. From now on, we will use this convention to import the `newforms` package, and when I mention the forms package, I will mean `newforms` imported as `forms`.

The Django forms library handles 3 common tasks:

- HTML form generation.
- Server-side validation of user input.
- HTML form redisplay in case of input errors.

The way in which this library works is similar to the way in which Django's data models work. You start by defining a class that represents your form. This class must be derived from the `forms.Form` base class. Attributes in this class represent form fields. The `forms` package provides many field types, in a way similar to how the `models` package provides many database types.

When you create an object from a class that is derived from the `forms.Form` base class, you can interact with it using a variety of methods. There are methods for HTML code generation, methods to access the input data and methods to validate the form.

We will learn about the forms library by creating a user registration form in the next section.

Designing the User Registration Form

Let's start by creating our first Django form. Create a new file in the `bookmarks` application folder and call it `forms.py`. Then open the file in your code editor and enter the following code:

```
from django import newforms as forms
class RegistrationForm(forms.Form):
  username = forms.CharField(label='Username', max_length=30)
  email = forms.EmailField(label='Email')
  password1 = forms.CharField(
     label='Password',
     widget=forms.PasswordInput()
  )
  password2 = forms.CharField(
     label='Password (Again)',
     widget=forms.PasswordInput()
  )
```

After examining the code, you will notice that the way in which we defined this class is similar to the way in which we defined the model classes. We derived the `RegistrationForm` class from `forms.Form`. All form classes need to inherit from this class. Next, we defined the fields that this form contains. There are many field types in the forms package. There are several parameters, listed below, which can be passed to the constructor of any field type. Some specialized field types can take other parameters in addition to these ones:

- `label`: The label of the field when HTML code is generated.
- `required`: Whether the user must enter a value or not. It is set to true by default. To change it, pass `required=False` to the constructor.
- `widget`: This parameter lets you control how the field is rendered in HTML. We used it above to make the `CharField` of the password become a password input field.
- `help_text`: A description of the field. Will be displayed when the form is rendered.

The following is a table of commonly used field types:

Field Type	Description
CharField	Returns a string.
IntegerField	Returns an integer.
DateField	Returns a Python `datetime.date` object.
DateTimeField	Returns a Python `datetime.datetime` object.
EmailField	Returns a valid email address as a string.
URLField	Returns a valid URL as a string.

And here is a partial list of available form widgets:

Widget Type	Description
PasswordInput	A password text field.
HiddenInput	A hidden input field.
Textarea	A text area that enables text entry on multiple lines.
FileInput	A file upload field.

We can learn more about the forms API by experimenting in the interactive console. Run the console and issue the following commands:

```
$ python manage.py shell
>>> from bookmarks.forms import *
>>> form = RegistrationForm()
```

Now we have an instance of the `RegistrationForm` class. Let's see how it is rendered in HTML:

```
>>> print form.as_table()
```

This command will give you a long output in which you will see the HTML rendering of the form using table tags. You can also render the form using `ul` and `p` tags by calling `form.as_ul()` and `form.as_p()` respectively.

In addition, you can render individual form fields with the following code:

```
>>> print form['username']
  <input id="id_username" type="text" name="username" maxlength="30" />
```

Now that we know how to render the form, let's move to input validation. We can pass input to a form using its constructor:

```
>>> form = RegistrationForm({
...   'username': 'test',
...   'email': 'test@example.com',
...   'password1': 'test',
...   'password2': 'test'
... })
>>> form.is_valid()
        True
```

`form.is_valid()` returned `True` because all the fields were provided and the email address is valid. Now try to pass an invalid form field:

```
>>> form = RegistrationForm({
...   'username': 'test',
...   'email': 'invalid email',
...   'password1': 'test',
...   'password2': 'test'
... })
>>> form.is_valid()
        False
>>> form.errors
        {'email': [u'Enter a valid e-mail address.']}
```

Django did the form-validation for us! You will get similar results if you do not pass a value for a field, because fields are required by default.

You can check whether a form has data or not using the `form.is_bound` attribute. If you try to validate an unbound form, you will get an exception. User input can be accessed through a dictionary at `form.data`, and if the form is valid, validated user input can be accessed at `form.clean_data`.

Now that you are comfortable with `forms.Form` instances, we need to improve data validation for our form. The form in its current state detects missing fields and invalid email addresses but we still need to do the following:

- Prevent the user from entering an invalid username or a username that's already in use.
- Make sure that the two password fields match.