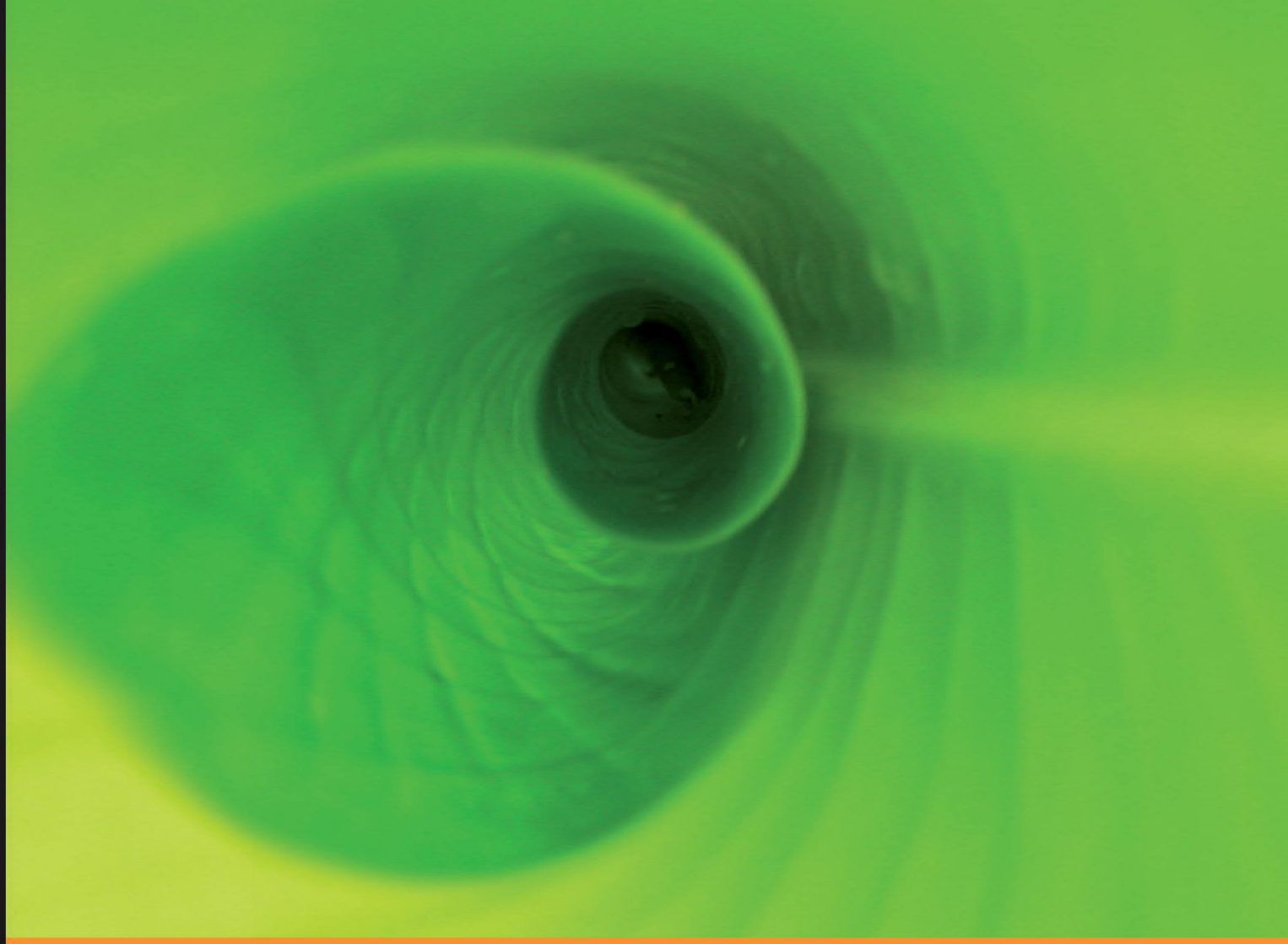


Community Experience Distilled

Django 1.2 e-commerce

Build powerful e-commerce applications using Django, a leading Python web framework

Jesse Legg

[PACKT] PUBLISHING open source* community experience distilled

Django 1.2 e-commerce

Build powerful e-commerce applications using Django, a leading Python web framework

Jesse Legg

[PACKT] open source
community experience distilled
PUBLISHING

BIRMINGHAM - MUMBAI

Django 1.2 e-commerce

First published: May 2010

Production Reference: 1120510

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

ISBN 978-1-847197-00-9

www.packtpub.com

Cover Image by Sujay Gawand (sujay0000@gmail.com)

Credits

Author

Jesse Legg

Reviewers

Clint Ecker

Josh Ourisman

Acquisition Editor

Steven Wilding

Development Editor

Dhiraj Chandiramani

Technical Editor

Bhavesh D. Bhasin

Indexer

Hemangini Bari

Editorial Team Leader

Gagandeep Singh

Project Team Leader

Priya Mukherji

Project Coordinator

Zainab Bagasrawala

Proofreader

Aaron Nash

Production Coordinator

Aparna Bhagat

Cover Work

Aparna Bhagat

About the Author

Jesse Legg is a web developer with experience in Python, PHP, JavaScript, and other programming languages. He has worked as a consultant computer programmer and as a web developer in new media and higher-education industries. He has nearly 10 years technology experience and has been building Django-based web projects since 2006. Jesse holds a degree in Computer Science from The College of Wooster and resides near Boston, Massachusetts. He writes an occasional blog post at jesselegg.com.

I would like to thank my wife and family for their support during the writing of this book. I'd also like to thank the technical reviewers and editing staff at Packt Publishing; writing a book is truly a team effort. And thanks to the Python development community for providing constant challenges, surprises, and friendship.

About the Reviewers

Clint Ecker has been working on the Web for the past 10 years in a myriad of languages and frameworks. He prefers to build applications in Django these days, but is exploring node.js, ruby, and finding a new appreciation for PHP. He began his web career at the CERIAS department at Purdue University, and more recently is a senior programmer and project manager at the premier technology publication, Ars Technica.

Josh Ourisman is a Python programmer and web developer with experience in numerous languages and frameworks, but with a love for both Python and Django. He currently works primarily on Django-based sites for Discovery Communications, the parent company of the Discovery Channel in Silver Spring, Maryland, and has previously worked on a great many sites for a large number of companies around the world.

Table of Contents

Preface

This book presents the implementation of web-based e-commerce applications using Django, the powerful Python-based web framework. It emphasizes common Django coding patterns, writing reusable apps, and leveraging community resources and open-source tools.

Django and Python can be used to build everything from quick application prototypes in an afternoon, to full-blown production applications with long-term lifecycles.

What this book covers

Chapter 1, *Django and E-commerce Development*, introduces Django, provides a quick overview of its history, and evaluates it as an e-commerce platform. We also introduce the concept of Django applications versus "apps", how to code for reusability, and why Django's project layout allows us to write more powerful, flexible software. Finally, we will begin configuring the sample project built-upon throughout the book.

Chapter 2, *Setting Up Shop in 30 Minutes*, shows us how to create a very simple, but complete, e-commerce store in 30 minutes. This includes the creation of a product catalog and categorization system using Django models, using built-in generic views to expose our catalog to the Web, and attaching a simple Google Checkout integration.

Chapter 3, Handling Customers and Their Orders, deals with customer and order management, including the use of Django's auth module, registration and profile forms. We'll also build a simple order-taking system and connect it to our customer data. Finally, we demonstrate a quick and easy way of handling customer product reviews.

Chapter 4, Building Payment Processors, starts to extend the simple system built thus far by creating a "pluggable" payment processing system, and updating our Google Checkout support to take advantage of it. Finally, this chapter discusses the Django pattern of class-based views and how to use them with our payment processing framework.

Chapter 5, From Payment to Porch: An Order Pipeline, adds additional information to our order-taking system, including tracking status and shipping, plus automatic calculation of shipping and handling charges. We integrate these new features into a simple CRM tool that would allow staff to look-up order details and monitor status.

Chapter 6, Searching the Product Catalog, explores the options for adding search capabilities to our catalog, including use of Django with, Sphinx, Solr, Whoosh, Haystack, and Xapian search. We integrate several of these search engines into our project and present the Haystack community-project that allows generic integration of a variety of search-engine backends.

Chapter 7, Data and Report Generation, covers report generation and working with our application's data. This includes serializing and exposing data via a web API, generating RSS and Atom feeds, and the basics of Salesforce integration. We also use Python charting modules to automatically generate PDF-based reports.

Chapter 8, Creating Rich, Interactive UIs, provides an overview of JavaScript and AJAX integration with our Django project. We discuss how to expose our Django model data as JSON and write clean JavaScript to enhance our user interfaces. We finish by demonstrating a simple AJAX rating tool.

Chapter 9, Selling Digital Goods, presents digital goods and the various tools and APIs to sell them. Digital goods include products, like music or video media, which are sold and distributed electronically. We cover using Amazon S3 for storage with Django and integrating with the Amazon Flexible Payment Services, which offers an API for handling micropayments.

Chapter 10, Deployment and Maintenance Strategies, offers us a variety of pointers for configuring, deploying, and maintaining our Django applications. This includes setting up Apache with `mod_wsgi`, automating a deployment process with fabric, handling virtual environments, and building distributable modules.

What you need for this book

This book requires Django 1.0 or higher and assumes a basic working knowledge of the Django framework and novice Python programming skills.

Who this book is for

This book is for anyone who is interested in learning more about application development with the Django framework. E-commerce applications contain a lot of general application design issues and make for a great example development project for anyone interested in Django applications generally.

We've assumed a fairly minimal amount of knowledge about the Django framework and Python language. But the book is geared at Django developers who have at least completed the Django tutorial and/or written some trivial apps.

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.

Code words in text are shown as follows: "To start with, create a new directory and place it on your `PYTHONPATH`".

A block of code is set as follows:

```
class Catalog(models.Model):
    name = models.CharField(max_length=255
    slug = models.SlugField(max_length=150)
    publisher = models.CharField(max_length=300)
    description = models.TextField()
    pub_date = models.DateTimeField(default=datetime.now)
```

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

```
$ django-admin.py index–rebuild
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "This code defines a two-column table with headings **Product Name** and **Product Description**, and then renders the product inventory into each row in the table".

Warnings or 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 send an e-mail to `feedback@packtpub.com`, and mention the book title via 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 e-mail `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 on, 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 `https://www.packtpub.com//sites/default/files/downloads/7009_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 content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please 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 will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

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

1
Django and e-commerce Development

The explosion of the Web in the late 1990s sparked a major evolution in the core operation of businesses everywhere. The impact initially seemed minor, as the Web was made up of very simple sites that were time consuming to create and offered limited functionality. But the eventual success of new companies such as Amazon.com and eBay demonstrated that buying and selling online was not only possible, but revolutionary.

This book will explore how the Django web framework, and its related technologies, can power the next leap forward for e-commerce and business on the Web. As a tool, Django is like a power drill: it's fast, efficient, and full of momentum. It represents a major competitive advantage over previous development platforms. Along with competing frameworks, such as Ruby on Rails, it heralds a new era for web development and the Internet as a whole.

This chapter will explore what makes Django so unique. Here we will:

- Discuss the pros and cons of Django as an e-commerce platform
- Learn to leverage Django's strengths to build powerful applications quickly
- Explore core concepts such as Django's approach to application and project layout
- Begin configuring our sample e-commerce project to be extended throughout the book

21st Century web development

An emerging trend in recent years has been the arrival of powerful new web development frameworks, of which Django is a premier example. These frameworks are designed to build content-rich websites quickly and easily. They have different individual philosophies, but all seek to solve one main problem—simplifying the repetitive and time-consuming development tasks that all websites and applications demand.

Solving this problem achieves two things:

- First, it radically alters the time and resources required to build web applications. Previous generations of development tools were either not designed specifically for the Web or were created during a different era with different needs. This new class of development tools is built specifically for today's applications and software schedules.
- The second benefit is quality. As the most basic and repetitive tasks are handled by the framework, developers can focus on adding value by building higher quality core application features and even advanced functionality that would previously be lost due to time or budget constraints. In many cases it also allows for a simplified division of labor; designers can focus on designing HTML templates, content producers can write content, and programmers can build software. When each team can focus exclusively on their task, with minimal interruption from the others, quality tends to increase.

Django is specifically designed to optimize for these characteristics. One unofficial motto is "don't repeat yourself". This means, do not reproduce the wheel, instead focus on the unique aspects of your application that provide new and profitable functionality.

Django as an e-commerce platform

Many of the problems that affect content-driven websites actually affect all businesses, even traditional brick-and-mortar ones. For instance, consider a restaurant that offers a menu of daily specials or whose menu changes frequently based on the selection of fresh food available in the city or region. To market this menu on the Web requires tools for quickly producing and updating content. We can imagine similar examples in the retail world, where product inventories ebb and flow depending on seasons, styles, and trends.

In the online world, this kind of content production is often taken for granted. The entire business of Amazon.com is built around a large, structured, and highly-detailed database of product information, along with customer ratings and reviews. Netflix's enormous selection of movie rentals would not be possible without an equally large content database.

Django provides an excellent basis to build e-commerce websites in part because of its ability to handle these content problems. Content is one of Django's specialties, having been created to serve the needs of the newspaper industry. It provides many out-of-the-box tools to handle these demands. For example, the built-in admin interface (as shown in the following screenshot) can automatically manage any application's data, in a way that anyone with a web browser can access. The admin tool can be attached to any Django project with a trivial amount of effort and just a few lines of code.

Handling content so effectively is possible because of Django's excellent **Object-relational Mapper** (**ORM**). Data-driven approaches to web development have been around for a decade. Originally this involved lots of tedious SQL code, but over a period of time much of this has been abstracted away in libraries or framework code. Django's ORM is arguably one of the best abstractions created thus far and its query interface is often cited as a killer feature. It's even possible to use the ORM portion of Django in standalone mode and take advantage of its expressive power. Django's ORM has even inspired other projects, most notably Google's App Engine project, which borrows features and syntax. We will highlight many features of Django's ORM throughout this book.

There are many other advantages to Django as an e-commerce platform. Rapid development tools, in general, mean cost reductions for initial development and long-term maintenance. Decreasing these costs is another specialty of Django. As an example, in *Chapter 2, Setting Up Shop in 30 Minutes*, we will create a fully functional web-based store, from scratch, in 30 minutes.

What about competing frameworks? What advantages does Django have over them? A big one is Python. Without stoking religious debate, Python, by its own merits, is an excellent programming language for the Web. Django is built in Python and features many "Pythonic" tactics at the core of its design. The language includes numerous modules that are useful to web developers, including a URL manipulation library, XML parsing tools, and web service tools. The Python community is massive, and provides additional modules and tools. Python developers can frequently avoid reinventing the wheel because either the language or the community already provides what they need.

Django is flexible and able to handle content as well as other e-commerce problems, such as shopping carts, user management, and payment processing. Built-in support for cookie-based user sessions and basic account authentication gives Django developers a major head-start over developers who must implement these from scratch. Submitting orders and processing payments through any of the Web's payment services is also very easy, as we will see in *Chapter 4, Building Payment Processors*.

All of these factors exhibit a common theme: competitive advantage. Cost savings, code reusability, and flexibility of use all allow businesses to do more with less. E-commerce sites built using Django can be deployed faster and respond quickly to change.

There are a few important criticisms of Django that might affect the decision to use it in your specific projects. First, it's relatively new and has yet to enter the mainstream software community. It's also open source, which unfortunately still prevents its use in some corporate environments, due to concerns over support, maintenance, and security. And finally, it is written in a different language and paradigm than many current mainstream e-commerce products. There are armies of Java developers because many e-commerce applications are built with it. By comparison there are fewer skilled Python developers.

The model-template-view pattern

The term "e-commerce" encompasses a wide array of applications. It includes everything from simple shopping-cart powered Internet stores to large-scale business-to-business systems. Fortunately, many of these applications begin with a similar set of technical needs. These include storing, updating, and retrieving important information from a data store (usually a database), rendering information in a common format such as HTML or XML, and interacting with users who will consume, manipulate, and process the information.

These basic needs happen to align with a design pattern known as **Model-View-Controller** (**MVC**). MVC is intended to simplify the construction of applications by dividing it up into three manageable parts: the model, which is focused on storing and structuring data, the view, which presents data exposed by the model, and finally the controller, which provides a means of interacting with and manipulating model data.

Django follows this design pattern, but prefers to call it **Model-Template-View** (**MTV**). It retains the model concept from MVC, but refers to the controller portion as "views" while replacing the MVC view with what it calls the "template". These can be standard HTML templates with added functionality provided by a Django specific template language. This language can be replaced with other template languages, such as Python's popular Jinja or Cheetah template systems. Django can also use templates to render other formats such as XML or JSON.

Django's MTV pattern is a powerful approach, especially for designing web-based applications. It allows components to be created in a reusable and modular way so that pieces of the system can be upgraded or replaced without disrupting the others. This simplifies development. By focusing on the abstraction each MTV component represents, developers need not worry about the implementation details of the other components, only how to interact with them. Django's implementation of the MTV pattern should be a comfort to e-commerce developers because it is a well known and solid foundation to build upon.

Payment processors and shopping carts

Another critical e-commerce component is the payment processor, sometimes called a payment gateway or a checkout system. There are hundreds of companies offering payment solutions today. Almost all process credit cards, but some allow for "e-checks" and other more obscure payment types. These payment processors generally compete on the fees they charge per transaction. The standard rates usually involve a small fee per transaction in addition to a percentage of the transaction (usually between 1.5 percent and percent).

PayPal has historically been one of the major web-based payment processing companies and continues to process a large majority of Internet payments. There are many alternatives, however, and two of the most formidable are powered by Google and Amazon. In my opinion the latter two processors are more developer-friendly and highly recommended. Google's processor is called "Checkout" and Amazon has built a payment processor called "Flexible Payment Services" as part of their suite of web-services tools. We will build Django apps for these payment processors in *Chapter 4*.

The traditional means of bundling products for sale and submitting them to a payment processor involves a shopping cart. Usually, this involves the customer who selects products from the catalog web pages by adding them to their shopping cart, and then checks out with their payment method via a payment processing service. We will begin implementing a Django-based shopping cart in *Chapter 3, Handling Customers and Their Orders*.

Django is able to interface with these payment processing systems easily, thanks in part to the availability of tools and modules in the Python community that simplify common tasks. Python code examples and in some cases entire wrapper modules are available for many payment processing solutions. In addition, Django's template system can be used to simplify rendering of XML and other data files required to submit payment requests.

Shopping carts are also relatively simple to implement using Django. The built-in session's framework provides an easy way to store cart and checkout information between pages in a user's session. We will build our sample project on this approach later in this book.

Exploring the Django framework

Django takes a disciplined approach to constructing websites. It has many conventions and design constructs that sometimes are not obvious to new developers. All frameworks employ their own philosophy in solving problems and the philosophical underpinnings tend to appear throughout. Some developers feel restricted by this effect, but it's a natural trade-off in the usage of frameworks (or any code library). If you leverage a framework in your development, it will steer you in certain directions. Much like a programming language can inform one's view of the world, so can frameworks.

There's an old proverb: *When in Rome, do as the Romans do*. The same can be said for development platforms. Some developers tend to apply their own philosophies in the context of a library or framework and end up frustrated. There is a great deal of grey-area here, of course, but it is important to spend some time understanding the Django-way of application development.

Much of Django's approach is informed by Python itself. Python is a programming language built on distinct philosophies. Its creator, Guido van Rossum, baked in many of these design decisions from the beginning.

What's in a Django app?

Django's introductory tutorials take a simplified approach to getting up and running on the platform. They introduce the concept of a Django project using `django-admin.py` and the `startproject` command. This is the easiest way to get going, but it hides a significant amount of activity that goes on under-the-hood. For the project that we will build throughout this book, we will take a more sophisticated approach, by thinking of our Django projects as a collection of normal, plain Python modules. Django calls these modules "apps."

In some ways the term "app" is unfortunate because it hints at something large, a full-blown application. But in Django, apps are usually small and simple. Properly designed apps can be plugged together to form powerful combinations, with each app solving its own portion of a larger problem.

Apps, like any good Python module, encapsulate a specific set of functionality; this is usually by focusing on a small problem and solving it. This is actually a well-established pattern of software design, originating in the UNIX operating system. Past and current versions of UNIX provided numerous tiny programs that solved a simple problem: pattern matching with `grep`, word counting with `wc`, and so on.

By emphasizing this approach in our projects, we can take full advantage of Django's rapid development philosophy. It will lead to better code and cleaner designs.

Solving small problems

Let's consider the role of Django apps in our e-commerce platform. The "solving small problems approach" fits well; many pieces of our e-commerce project will be common across multiple sites. By keeping our apps small and focused, we will be able to assemble the individual components in different ways for different projects.

For example, two e-commerce stores may share the same payment processor, but have entirely different needs for interacting with their customers. One site might need the ability to send an e-mail newsletter, while the other would not. If we were to build our Django project in large, monolithic sections, it would require more time and effort to satisfy the needs of these two projects. If, however, we use small, tiny pieces, we can simply plug-in the parts they have in common and upgrade or build separately the pieces that differ.

In larger development shops, this also allows for the internal reuse of apps across departments or functional groups. It can make code sharing and reuse much easier for in-house teams.

Keeping a modular design has other advantages. When a project decides to change payment processors, the upgrade is much simpler when the processing code lives alone in its own module. We can standardize the interface across all payment processors so other apps can interact with all of them the same way. In Python this is sometimes called "duck typing" and we will explore it more in *Chapter 4*.

Django's settings file has an important attribute known as `INSTALLED_APPS`. This is a Python sequence of module names that will be used for the project. In some ways this is Django's secret weapon. Ideally we can deploy dozens of entirely different sites by doing nothing more than creating a new settings file with an appropriate set of modules in `INSTALLED_APPS` and pointers to different databases and template locations.

Solving small problems with focused Django apps is the best way to achieve these goals. It is important to remember that we will be writing apps, or better yet, normal Python modules. These will be pieces of something larger, not full-blown applications themselves.

Reusable apps

Reusability is software engineering's Holy Grail. Unfortunately, over time it has often proven difficult to attain. It's almost impossible to build for reuse on the first try and other times it's just not practical. But it's still an important goal that often does work and leads to many efficiency gains.

In frameworks like Django that utilize an ORM to interact with and store information in a database, reusability faces additional challenges. The object-oriented programming model that is typically the heart of reusable code does not always translate into a relational database. Django's ORM does its best to accommodate this by offering a limited form of inheritance, for example, but it still has many challenges.

Another tendency is to build Django models that store data for overly specific problems. In *Chapter 2*, we will begin writing models for a product database. It would be very easy to apply model inheritance in an attempt to solve specific problems. For example, we may be tempted to extend a product model into a subclass specific for food: `FoodProduct`. We then may try and build a subclass specifically for noodles: `NoodleProduct`. Using inheritance this way often makes sense in other software projects, but when Django maps these models to the database, it can become a mess of entangled relationships and primary keys.

To avoid these issues with inheritance, some Django developers employ various model hacks. This includes things such as pickling attribute data into a text field or otherwise encoding extra attributes into a single field. These are often clever and sometimes very effective solutions, but they can also lead to bad designs and problems later on.

The best advice seems to be to keep things simple. Limit what your app does and the dependencies it needs to serve its core duty. Don't be afraid of developing a large app library.

Organizing Django projects

As mentioned earlier, Django projects are essentially collections of regular Python modules. These are managed using a settings file. We can have more than one settings file for any project—separate production and a development settings file, for example. Django settings files are also code, which adds even greater flexibility. Using import statements, for example, we can include a standard set of Django settings in every project and simply override the ones we need to change.

When using version control software, it's helpful to keep many different settings files. These files can be organized by developer, by server name, or by code branch. Keeping multiple settings files simplifies deployment and testing. A settings file for each test server or individual developer reduces confusion and prevents breakage as developers create and integrate new apps. However, it retains the convenience and safety of version control. Trying to maintain a large project with multiple developers and a single settings file is a recipe for disaster.

With multiple settings files floating around, how does Django know which one to use? In the command line environment for running interactive Python shells or scripts, this is controlled using an environment variable called `DJANGO_SETTINGS_MODULE`. Once your settings are in place, you can quickly switch back and forth between them by modifying this environment variable. For example:

```
# switch to our testing settings and run a Django interactive shell
$ export DJANGO_SETTINGS_MODULE=myproject.settings_testing
$ django-admin.py shell
# switch back to our production settings
$ export DJANGO_SETTINGS_MODULE=myproject.settings_production
```

A convenient alternative to the preceding manual process is to use the `settings` flag of `django-admin.py`, which will adjust the settings module appropriately. Switching to production settings looks like this:

```
$ django-admin.py--settings myproject.settings_production
```

A simple set of shell scripts can automate the use of several settings files. The following shell script wraps Django's `django-admin.py` and sets a specific settings module. This way we can run our project from the command line with a single simple command, like this.

```
#!/bin/sh
# myproject_dev - a shell script to run myproject in development mode
export DJANGO_SETTINGS_MODULE=myproject.settings_dev
django-admin.py $@
```

For server deployments, the settings file is specified as part of the server configuration. Sometimes this involves changing the environment variable from an Apache config file (`SetEnv DJANGO_SETTINGS_MODULEmyproject.settings`). On `mod_wsgi` setups, it usually means modifying the **Web Server Gateway Interface** (**WSGI**) script. We will explore these configuration techniques in *Chapter 10, Deployment and Maintenance Strategies*.

The other core piece of all Django projects is the root URLs file. Again, this is just a code that defines a set of URL patterns using regular expressions. We can include multiple versions (production vs. development) for our project and use normal Python flow-control to make adjustments such as adding special URL patterns when our project has the `DEBUG` setting turned on.

For larger Django projects, multiple settings and URL files can quickly get out of hand. As a result it is a smart practice to keep project files, such as settings and root URL configurations, completely separate from app code. Structuring your project and app libraries is often dependent on personal taste or company standards, but keeping a modular and loosely-coupled mindset is always beneficial.

Preparing the development environment

Arguably the most important part of preparing a new Django project is choosing a name. All kidding aside, the name is important and in that it has a few restrictions: it cannot be the same as a Django or Python module and it should conform to usual Python module naming conventions (no spaces, dashes, or other illegal characters). Django projects have a tradition of naming themselves after jazz musicians (Django itself refers to Django Reinhardt, a jazz guitarist). The project we will build in this book will be named Coleman.

To start with, create a new directory and place it on your `PYTHONPATH`. The easiest way to do this is to make a directory in the usual way and then create a corresponding `.pth` file to copy to your system's site-packages. For example, if you're working directory is `/home/jesse/ecommerce-projects`, then create a single line file called `ecommerce.pth` that looks like this:

```
/home/jesse/ecommerce-projects
```

Copy the file to your system-wide site-packages directory. This varies based on your operating system and how you installed Python. See your documentation or the value of `sys.path` from the Python interpreter.

Later on in this book we will examine some tools and techniques to drastically simplify Django and Python project dependencies and layouts. One of these tools is called **virtualenv**, which can help us to better manage `$PYTHONPATH`. For now, though, we'll do everything the old fashioned way.

Next we will create our project directory and subdirectories. These will correspond to Python modules in our code. From your working directory create two folders: one for our app library and one for our project. Call the first one `coleman` and the second `ecommerce_book`. We will refer to the `coleman` module as our app library and to `ecommerce_book` as our project module. Next we will create our first app module by creating a directory called `products inside our app library module`. We will begin building the `products` app in *Chapter 2*. Your final directory structure will look like this:

```
./coleman
./coleman/products
./ecommerce_book
```

One final piece of preparation: we must create a Python `__init__.py` file in each of these module locations. This will look like the following:

```
./coleman/__init__.py
./coleman/products/__init__.py
./ecommerce_book/__init__.py
```

Lastly, we will create our settings file and a root URLs file in our project module: `ecommerce_book/settings.py` and `ecommerce_book/urls.py`. It is recommended that you copy these files from the companion source code. We will then refer to `ecommerce_book.settings` in our `DJANGO_SETTINGS_MODULE`.

The project settings file in the companion code assumes that you have a working sqlite3 installation as the database backend (sqlite3 is included with Python 2.5 and later). You can change this by editing the `settings.py` file, which you will need to do in order to complete the full path to your sqlite3 database file. This is another file that fits nicely into the projects directory, and by placing it there you gain the ability to use Python's `os` module to locate it. This is more desirable than hard coding a path into `settings.py`. To take this approach with the database file, for example, you could define a setting to represent the project's location on the file system, and then append the database name:

```
PROJECT_HOME=os.path.dirname(os.path.realpath(__file__))
DATABASE_NAME=os.path.join(PROJECT_HOME, 'mydatabase.db')
```

Once you've made the appropriate changes, run `django-admin.pysyncdb` to create your initial database.

These instructions assume that you already have a working installation of Django and have the `django-admin.py` script installed on your system's path. For more information on configuring and installing Django, please see the documentation at `djangoproject.org/docs`.

Django 1.2

Django 1.2 includes a number of new features and very minimal backwards incompatible changes. As of Django 1.0, the Django development team adopted an **API stability** policy that alleviates the worry about backwards incompatible changes being introduced into the framework throughout the 1.x series of releases. In other words, with few exceptions, if your application runs on version 1.0, it should run on all 1.x versions.

Major new additions to the Django framework in version 1.2 include multiple-database support, improved if template tag, object-level permissions, e-mail backends, and much, much more.

Per the API stability policy, a minimal number of backwards incompatible changes are included in 1.2. Major changes to existing framework features, like the new `if` tag, necessitate some incompatible changes. In 1.2 these have been very minor and are well documented in the release notes.

The Django 1.2 release notes also include an overview and brief documentation of the new features and functionality available, and are an excellent starting point for projects that are upgrading.

Summary

This chapter has outlined several basic tenets of software development using Django. Planning is one of the most important steps for any project, and understanding Django's particular way of approaching web development helps in the planning phase. The Django way includes:

- Building apps that serve a single function and can be reused easily
- Keeping the design of models relatively focused
- Leveraging the power of Python when possible
- Organizing our projects, including settings and URL files

We also explored the reasons why Django makes a powerful platform for e-commerce applications and explained some of the basic needs these applications tend to have in common. In *Chapter 2* we will jump right in to starting our project by building a very basic web-based store in 30 minutes.

2

Setting Up Shop in 30 Minutes

In order to demonstrate Django's rapid development potential, we will begin by constructing a simple, but fully-featured, e-commerce store. The goal is to be up and running with a product catalog and products for sale, including a simple payment processing interface, in about half-an-hour. If this seems ambitious, remember that Django offers a lot of built-in shortcuts for the most common web-related development tasks. We will be taking full advantage of these and there will be side discussions of their general use.

Before we begin, let's take a moment to check our project setup. In the first chapter we planned a project layout that included two directories: one for files specific to our personal project (settings, URLs, and so on), and the other for our collection of e-commerce Python modules (`coleman`). This latter location is where the bulk of the code in the following chapters will live. If you have downloaded the source code from the Packt website, the contents of the archive download represents everything in this second location.

In addition to building our starter storefront, this chapter aims to demonstrate some other Django tools and techniques. In this chapter we will:

- Create our Django Product model to take advantage of the automatic admin tool
- Build a flexible but easy to use categorization system, to better organize our catalog of products
- Utilize Django's generic view framework to expose a quick set of views on our catalog data
- Take further advantage of generic views by building templates using Django's automatic naming conventions
- Finally, create a simple template for selling products through the Google Checkout API

Designing a product catalog

The starting point of our e-commerce application is the product catalog. In the real world, businesses may produce multiple catalogs for mutually exclusive or overlapping subsets of their products. Some examples are: fall and spring catalogs, catalogs based on a genre or sub-category of product such as catalogs for differing kinds of music (for example, rock versus classical), and many other possibilities. In some cases a single catalog may suffice, but allowing for multiple catalogs is a simple enhancement that will add flexibility and robustness to our application.

As an example, we will imagine a fictitious food and beverage company, CranStore.com, that specializes in cranberry products: cranberry drinks, food, and deserts. In addition, to promote tourism at their cranberry bog, they sell numerous gift items, including t-shirts, hats, mouse pads, and the like. We will consider this business to illustrate examples as they relate to the online store we are building throughout this book.

We will begin by defining a catalog model called `Catalog`. The basic model structure will look like this:

```
class Catalog(models.Model):
    name = models.CharField(max_length=255
    slug = models.SlugField(max_length=150)
    publisher = models.CharField(max_length=300)
    description = models.TextField()
    pub_date = models.DateTimeField(default=datetime.now)
```

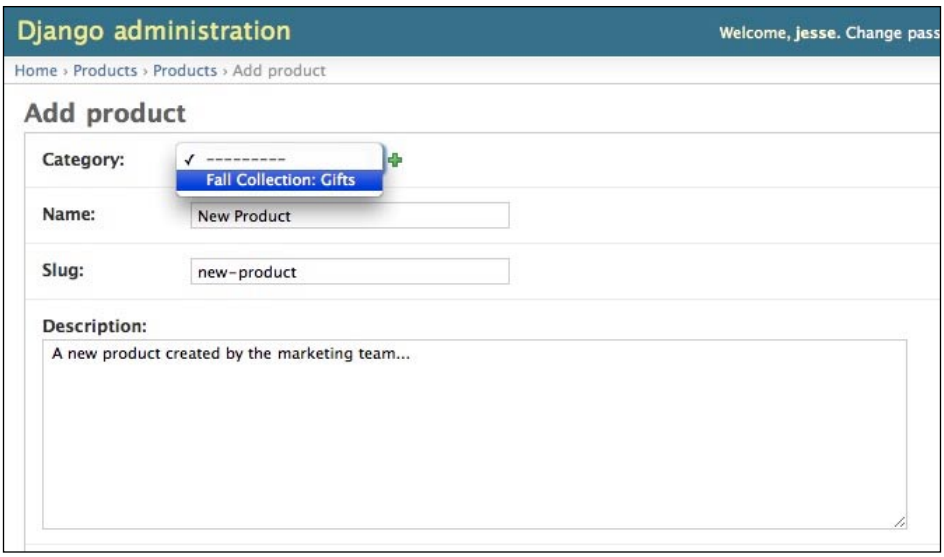
This is potentially the simplest model we will create in this book. It contains only five, very simple fields. But it is a good starting point for a short discussion about Django model design. Notice that we have not included any relationships to other models here. For example, there is no products `ManyToManyField`. New Django developers tend to overlook simple design decisions such as the one shown previously, but the ramifications are quite important.

The first reason for this design is a purely practical one. Using Django's built-in admin tool can be a pleasure or a burden, depending on the design of your models. If we were to include a products field in the `Catalog` design, it would be a `ManyToManyField` represented in the admin as an enormous multiple-select HTML widget. This is practically useless in cases where there could be thousands of possible selections.

If, instead, we attach a `ForeignKey` to `Catalog` on a `Product` model (which we will build shortly), we instantly increase the usability of Django's automatic admin tool. Instead of a select-box where we must *shift*-click to choose multiple products, we have a much simpler HTML drop-down interface with significantly fewer choices. This should ultimately increase the usability of the admin for our users.

For example, CranStore.com sells lots of t-shirts during the fall when cranberries are ready to harvest and tourism spikes. They may wish to run a special catalog of touristy products on their website during this time. For the rest of the year, they sell a smaller selection of items online. The developers at CranStore create two catalogs: one is named **Fall Collection** and the other is called **Standard Collection**.

When creating product information, the marketing team can decide which catalog an individual product belongs to by simply selecting them from the product editing page. This is more intuitive than selecting individual products out of a giant list of all products from the catalog admin page.



Secondly, designing the `Catalog` model this way prevents potential "bloat" from creeping into our models. Imagine that CranStore decides to start printing paper versions of their catalogs and mailing them to a subscriber list. This would be a second potential `ManyToManyField` on our `Catalog` model, a field called subscribers. As you can see, this pattern could repeat with each new feature CranStore decides to implement.

By keeping models as simple as possible, we prevent all kinds of needless complexity. In addition we also adhere to a basic Django design principle, that of "loose coupling". At the database level, the tables Django generates will be very similar regardless of where our `ManyToManyField` lives. Usually the only difference will be in the table name. Thus it generally makes more sense to focus on the practical aspects of Django model design. Django's excellent reverse relationship feature also allows a great deal of flexibility when it comes time to using the ORM to access our data.

Model design is difficult and planning up-front can pay great dividends later. Ideally, we want to take advantage of the automatic, built-in features that make Django so great. The admin tool is a huge part of this. Anyone who has had to build a CRUD interface by hand so that non-developers can manage content should recognize the power of this feature. In many ways it is Django's "killer app".

Creating the product model

Finally, let's implement our product model. We will start with a very basic set of fields that represent common and shared properties amongst all the products we're likely to sell. Things like a picture of the item, its name, a short description, and pricing information.

```
class Product(models.Model):
    name = models.CharField(max_length=300)
    slug = models.SlugField(max_length=150)
    description = models.TextField()
    photo = models.ImageField(upload_to='product_photo',
                              blank=True)
    manufacturer = models.CharField(max_length=300,
                                    blank=True)
    price_in_dollars = models.DecimalField(max_digits=6,
                                           decimal_places=2)
```

Most e-commerce applications will need to capture many additional details about their products. We will add the ability to create arbitrary sets of attributes and add them as details to our products later in this chapter. For now, let's assume that these six fields are sufficient.

A few notes about this model: first, we have used a `DecimalField` to represent the product's price. Django makes it relatively simple to implement a custom field and such a field may be appropriate here. But for now we'll keep it simple and use a plain and built-in `DecimalField` to represent currency values.

Notice, too, the way we're storing the manufacturer information as a plain `CharField`. Depending on your application, it may be beneficial to build a Manufacturer model and convert this field to a `ForeignKey`. We will explore more issues like this later in the book, but for now we'll keep it simple.

Lastly, you may have realized by now that there is no connection to a `Catalog` model, either by a `ForeignKey` or `ManyToManyField`. Earlier we discussed the placement of this field in terms of whether it belonged to the `Catalog` or in the `Product` model and decided, for several reasons, that the `Product` was the better place. We will be adding a `ForeignKey` to our `Product` model, but not directly to the `Catalog`. In order to support categorization of products within a catalog, we will be creating a new model in the next section and using that as the connection point for our products.

Categorizing products

So far we've built a simple product model and simple catalog model. These models are excellent building blocks on which to begin adding new, more sophisticated functionality. As it stands, our catalog model design is unconnected to our products. We could add, as mentioned earlier, a `ForeignKey` from `Product` to `Catalog`. But this would allow for little in the way of organizing within the catalog, other than what we can do with the basic `filter()`, `order_by()`, and other ORM methods that Django provides.

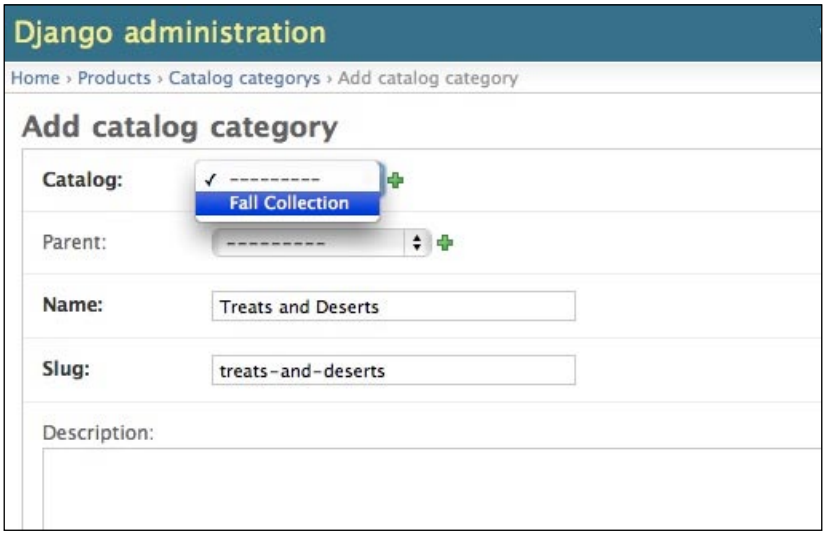
Product categories are an extremely common solution to organization problems in web-based stores. Almost all major Internet retailers organize their products this way. It helps them to provide a more structured interface for their users and can give search engine indexers more precise pages to crawl. We'll discuss more about these tactics later, but let's begin by adding categories to the simple model designs that we created earlier.

Even though our `Catalog` model is relatively simple, we must make some design decisions before adding our category information. First, we have designed `Catalog` so that our e-commerce site can produce different catalogs for different scenarios. Each scenario, however, may have different category requirements.

Suppose our fictitious cranberry business, CranStore.com, wants to create a special catalog for the holiday season, when it begins selling numerous gifts and decorative items such as cranberry garland, dried cranberry ornaments, cranberry scented candles and the like. The majority of these special holiday gifts are not available year-round. However, some products will be available in other catalogs at different times of the year—for example, the very popular cranberry-scented candles.

We need to be able to support this use-case. In order to do so, we have to structure our models in a way that may feel awkward at first, but ultimately allows for great flexibility. Let's write a category model called `CatalogCategory` and add it to our products application's `models.py`:

```
class CatalogCategory(models.Model):
   catalog = models.ForeignKey('Catalog',
                               related_name='categories')
   parent = models.ForeignKey('self', blank=True, null=True,
                              related_name='children')
   name = models.CharField(max_length=300)
   slug =  models.SlugField(max_length=150)
   description = models.TextField(blank=True)
```



In addition, we can now add a relationship between the `Product` model we wrote earlier and our new `CatalogCategory` model. The full, updated model appears as follows:

```
class Product(models.Model):
    category = models.ForeignKey('CatalogCategory',
                                 related_name='products')
    name = models.CharField(max_length=300)
    slug = models.SlugField(max_length=150)
    description = models.TextField()
    photo = models.ImageField(upload_to='product_photo',
                              blank=True)
```

```
    manufacturer = models.CharField(max_length=300,
                                    blank=True)
    price_in_dollars = models.DecimalField(max_digits=6,
                                           decimal_places=2)
```

There are a couple of big changes here, so let's tackle the `CatalogCategory` first. This model creates a relationship to our earlier `Catalog` model, but also includes a `ForeignKey` relationship to itself. This is called a recursive relationship and is designed as a simple way of creating category hierarchies. A category is a top-level category if it has no parent relationship to other category. Otherwise, it is a sub-category.

The second change is the addition of the `ForeignKey` to a category on our `Product` model. An initial design inclination here is to relate products directly to their catalog. We even discussed this design earlier in the chapter, and it would be great for simple catalogs where we didn't need the extra organization functionality of categories.

However, when categories and sub-categories get involved, the design has the potential to become seriously complex. This approach to our `Product` model allows us to manage a single relationship between products and their categories, thus keeping it relatively simple. This relationship is also implicitly a relationship between our product and a catalog. We still have access to the catalog our product lives in, via the category.

Another advantage of this design is that it remains extremely easy for non-developers to create products, catalogs, and complicated category structures, all within the built-in Django admin tool. As illustrated in the screenshots throughout this chapter, this model design gives us an extremely intuitive admin interface.

One final note about the above code: the default description when `CatalogCategory` model objects are printed or displayed in the admin is not particularly helpful. Let's add a custom `__unicode__` method to this model that prints a more informative display of our categories. This method will include the parent category information if we're working with a sub-category, as well as the catalog to which the category belongs.

```
def __unicode__(self):
    if self.parent:
        return u'%s: %s - %s' % (self.catalog.name,
                                 self.parent.name,
                                 self.name)
        return u'%s: %s' % (self.catalog.name, self.name)
```

Adding additional product details

The application we've built so far is actually quite powerful, but in this section we'll take our design to another level of flexibility. Products, as they currently exist, have a limited amount of information that can be stored about them. These are the six fields discussed earlier: name, description, photo, manufacturer, and price. These attributes will be common to all products in our catalog. There may be more attributes appropriate for this model, for example, size or weight, but we have left those unimplemented for now.

A lot of product information, though, is specific to only certain products or certain kinds of products. Capturing this information requires a more sophisticated design. Our goal, as always, will be to keep things as simple as possible and to take advantage of the built-in functionality that Django offers.

In an ideal world, we would like to allow an unlimited number of fields to capture all the potential descriptive information for any product. Unfortunately, this is not possible using the relational database systems that drive the majority of web-based applications. Instead, we must create a set of models with relationships in such a way that it effectively gives us what we need.

Database administrators are constantly implementing designs like this. But when it comes to frameworks like Django, with their own ORM systems, there's a great temptation to try and create fancy models with `ManyToManyFields` all over the place, which can capture everything and do so very cleverly. This is almost always more complex than is necessary, but because Django makes it so easy, it becomes a great temptation.

In our case we will build a fairly simple pair of models that allow us to store arbitrary information for any product. These models are the `ProductDetail` and `ProductAttribute` and they are as follows:

```
class ProductDetail(models.Model):
    '''
    The ``ProductDetail`` model represents information unique to a
    specific product. This is a generic design that can be used
    to extend the information contained in the ``Product`` model with
    specific, extra details.
    '''
    product = models.ForeignKey('Product',
                                related_name='details')
    attribute = models.ForeignKey('ProductAttribute')
```

```
    value = models.CharField(max_length=500)
    description = models.TextField(blank=True)

    def __unicode__(self):
        return u'%s: %s - %s' % (self.product,
                                 self.attribute,
                                 self.value)

class ProductAttribute(models.Model):
    '''
    The ``ProductAttribute`` model represents a class of feature found
    across a set of products. It does not store any data values
    related to the attribute, but only describes what kind of a
    product feature we are trying to capture. Possible attributes
    include things such as materials, colors, sizes, and many, many
    more.
    '''
    name = models.CharField(max_length=300)
    description = models.TextField(blank=True)

    def __unicode__(self):
        return u'%s' % self.name
```

As noted in the model docstrings, the `ProductDetail` design relates a kind of attribute to a specific product and stores the attribute value, as well as an optional extended description of the value.

Suppose our friends at CranStore.com offer a new selection of premium cranberries, created from special seedlings developed over years of careful breeding. Each variety of these premium cranberries features different characteristics. For example, some are extremely tart, while others are extra sweet, and so on.

To capture this additional information, we created a **Tartness** `ProductAttribute` and used it to add `ProductDetail` instances for the relevant product. You can see the end-result of adding this attribute in the previous screenshot.

Viewing the product catalog

We now have a relatively complete and sophisticated product catalog with products, categories, and additional product information. This acts as the core of our e-commerce application. Now we will write some quick views and get our catalog running and published to a web server.

In the Django design philosophy, views represent a specific interpretation of the data stored in our models. It is through views that templates, and ultimately the outside world, access our model data. Very often the model data we expose in our views are simply the model objects themselves. In other words, we provided direct access to a model object and all of its fields to the template.

Other times, we may be exposing smaller or larger portions of our model data, including `QuerySets` or lists of models or a subset of all model data that match a specific filter or other ORM expression.

Exposing a full model object or set of objects according to some filter parameter, is so common that Django provides automatic built-in assistance. This is accomplished using 'generic views', of which Django includes over a dozen. They are broken into four basic kinds: simple, date-based, list-based, and create/update/delete (CRUD).

Simple generic views include two functions: `direct_to_template` and `redirect_to`. These are by far the simplest possible views and writing by hand more than once would be overkill. The `direct_to_template` generic view simply renders a regular Django HTML template, with an optional set of variables added to the context. The `redirect_to` generic view is even simpler; it raises a standard **HTTP 302** status code, otherwise known as a "non-permanent redirection". It can also optionally raise a permanent redirection (**HTTP 301**).

The real power behind generic views becomes evident in the next set: the date-based views. These are designed to create automatic archives for all of your site content organized by date. This is ideal for newspaper or blog applications, where content is published frequently and finding content based on a date is an effective way to interact with the information.

There are seven different date-based views. The majority of these are for specific time intervals: by year, by month, by week, by day, and for the current day. The remaining two views are for content indexes, called `archive_index`, which can often be used for homepages and for details on a specific object, useful for permalink pages.

The next set of views is called list-based because they process and present lists of model data. There are only two kinds of list-based views: `object_list` and `object_detail`. The former provides a template with context variable that represents a list of objects. Usually the template will iterate over this list using a for-loop tag. The latter view, `object_detail`, provides a simple view of a single item in the list—for example, a single product out of a list of all products in the catalog.

And lastly we have CRUD views. CRUD stands for create, update, and delete, and is used to provide automatic interfaces for users to modify model data in templates. Sometimes users will need to edit information, but are not staff members and cannot use the Django admin tool. This often happens, when users are editing content they have created (user-generated content) or more simply when they need to edit their profile information, such as payment method or shipping address.

CRUD views are extremely useful for simplifying a very common pattern of gathering and editing data. When combined with Django's `forms` module, its power can be extended even further. We will utilize generic CRUD views in later chapters.

For now, let's build an initial set of views on our product catalog using the list-based generic views. We have two use-cases. The first use-case is where we enlist a set of products in our catalog or within a category in our catalog. The second use-case is a detail page for every product we are selling. Generic views make writing these pages easy. For our catalog homepage we will implement a special-case of the list-based `object_list` view.

When using generic views, it is rarely necessary to create a `views.py` file or write any view handling code at all. The bulk of the time, generic views are integrated directly into the URL pattern definitions in our `urls.py` file. Here is the basic set of URLs for our product catalog:

```
urlpatterns = patterns(
    'django.views.generic.list_detail',
    url(r'^product/$', 'object_list',
        {'queryset': Product.objects.all()}),
    url(r'^product/(?P<slug>[-\w]+)/$', 'object_detail',
        {'queryset': Product.objects.all()}))
```

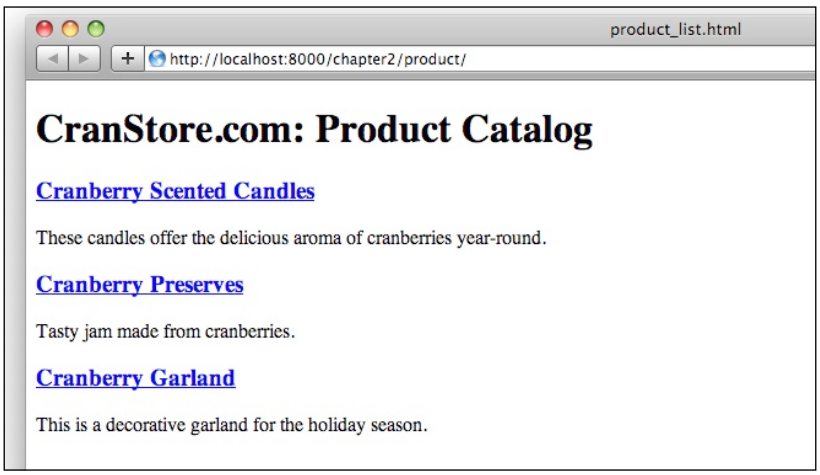
Designing simple product HTML templates

Now that we have exposed our product data via generic views and created corresponding URL patterns, we need to implement some templates to render the product information to our browser. Another advantage of using generic views is that templates are automatically specified for each view based upon the name of the application and model in use. Since our `urls.py` file is part of the `products` application and we are rendering generic views on our Product model, our application will automatically look for templates in the following locations within the template folder: `products/product_detail.html` and `products/product_list.html`.

These templates will get rather simple contexts. The `object_list` template will receive a variable whose default name is `object_list`. It is a `QuerySet` corresponding to the data exposed by your view. The `object_detail` template will receive a variable whose default name is `object`. It corresponds to the object you are viewing in more detail.

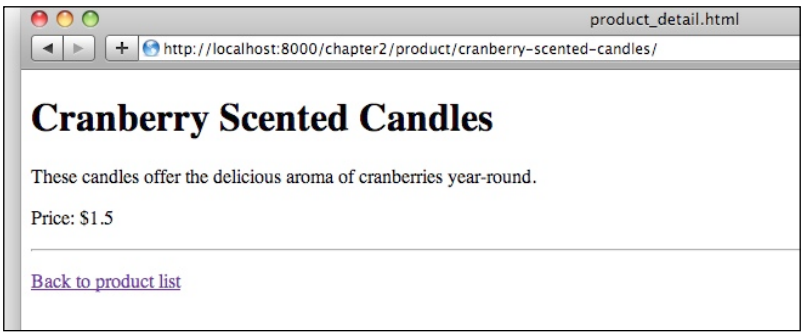
In our example, the list view will receive a `QuerySet` of all `Product` objects, while the detail view receives the object specified by the slug in the URL. In our list template we loop over the products like so (extra HTML portions are removed):

```
{% for object in object_list %}
 <h3><a href="{{ object.get_absolute_url }}">
 {{ object.name }}</a></h3>
 <p>{{ object.description }}</p>
{% endfor %}
```



The detail template works similarly, though the output in the template includes more details about the item:

```
<h1>{{ object.name }}</h1>
<p>{{ object.description }}</p>
<p>Price: ${{ object.price_in_dollars }}</p>
<hr/>
<p><a href="{% url product_list %}">Back to product list</a></p>
```



We can print the extra attributes, if any happen to exist, very easily in the template. The output from the snippet below is demonstrated in the following screenshot:

```
<h1>{{ object.name }}</h1>
<p>{{ object.description }}</p>
<p>Price: ${{ object.price_in_dollars }}</p>
{% for detail in object.details.all %}
{% if forloop.first %}<p>Additional details:</p>{%endif%}
<li>{{ detail.attribute.name }}: {{ detail.value }}</li>
{% endfor %}
```

We now have a very spartan, but functional product catalog rendering to HTML from our Django database. We can add, modify, and delete products, catalogs and additional attributes. Now let's get started selling. In the next section we will create a Google Checkout API account and use its sandbox to set up a simple order mechanism for our online store.

Getting paid: A quick payment integration

An online store is of little use if you can't sell products to customers. We will thoroughly explore the topic of payment processors in *Chapter 4, Building Payment Processors*. For now, however, to succeed in our promise to build a fully functional web store in 30 minutes, we will make use of a quick payment-integration with the Google Checkout API.

First, you will need to sign-up for a Google Checkout account. For the examples in this book, it is recommended that you use the Checkout sandbox, which is a test bed, non-live version of the Google Checkout system. You can get a seller account in the sandbox at the following URL: `http://sandbox.google.com/checkout/sell`.

Once you have an account, you can access the API document from this URL: `http://code.google.com/apis/checkout/developer/`.

We will create a very simple **Buy It Now** button using the Checkout API. After you create a sandbox seller account and log in for the first time, you can access several automatic checkout tools via the **Tools** tab at the top of the page. From here, select the **Buy It Now** button and you can generate HTML for an initial button. We will build on top of this HTML in our templates.

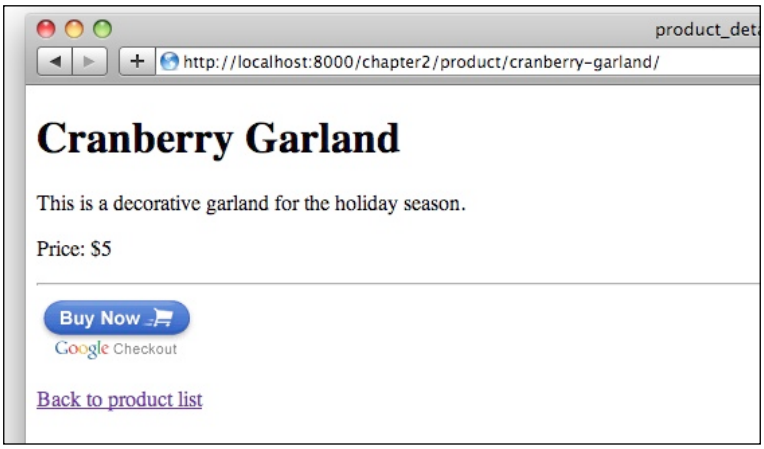
The buy it now HTML will looks like this (note that the actual merchant ID numbers, even though they're in a sandbox account, have been removed):

```
<form    action="https://sandbox.google.com/checkout/api/checkout/
v2/checkoutForm/Merchant/xxxxx" id="BB_BuyButtonForm" method="post"
name="BB_BuyButtonForm">
   <input name="item_name_1"
         type="hidden" value="Cranberry Preserves"/>
   <input name="item_description_1"
         type="hidden" value="Tasty jam made from cranberries."/>
   <input name="item_quantity_1" type="hidden" value="1"/>
   <input name="item_price_1" type="hidden" value="0.5"/>
   <input name="item_currency_1" type="hidden" value="USD"/>
   <input name="_charset_" type="hidden" value="utf-8"/>
```

```
    <input alt="" src="https://sandbox.google.com/checkout/buttons/
buy.gif?merchant_id=xxxxx&amp;w=117&amp;h=48&amp;style=white&amp;varia
nt=text&amp;loc=en_US" type="image"/>
</form>
```

Now we can copy and paste this HTML code into our product detail template. We will then modify the form code to use our Django template variables for the appropriate HTML form fields. Ultimately the form will look like this:

```
<form action="https://sandbox.google.com/checkout/api/checkout/v2/
checkoutForm/Merchant/xxxxx" id="BB_BuyButtonForm" method="post"
name="BB_BuyButtonForm">
    <input name="item_name_1" type="hidden" value="{{ object.name
                                                    }}"/>
    <input name="item_description_1"
        type="hidden" value="{{ object.description }}"/>
    <input name="item_quantity_1" type="hidden" value="1"/>
    <input name="item_price_1"
        type="hidden" value="{{ object.price_in_dollars }}"/>
    <input name="item_currency_1" type="hidden" value="USD"/>
    <input name="_charset_" type="hidden" value="utf-8"/>
    <input alt="" src="https://sandbox.google.com/checkout/buttons/
     buy.gif?merchant_id=xxxxx&amp;w=117&amp;h=48&amp;style=white&amp;
     variant=text&amp;loc=en_US" type="image" />
</form>
```



Our simple product catalog that started out as a couple of Django models in the beginning of this chapter is now fully capable of processing transactions and selling products. If these **Buy It Now** buttons were activated through the real Google Checkout web service and not the sandbox, they would be totally live and able to sell any item in our product database.

Unfortunately, the **Buy It Now** button only supports selling a single item at a time. The method we've covered above will work for some very simple applications, but more than likely we will need something more sophisticated for a larger site. The typical method of supporting complicated transactions is to implement a "shopping cart" tool, a pattern that can be seen on many of the largest e-commerce sites.

The buy it now button has other problems as well. You cannot specify a choice of sizes, for example: small, medium, or large. These may be stored on our Product model, but could not be specified in a single **Buy It Now** button without additional work. It is also a requirement of the "buy it now" tool that the customer checkout on Google's site with a minimal amount of custom branding. If we want to provide a full-fledged checkout experience, including our own branding and HTML design, we must look to the full Checkout API. In *Chapter 4* we will cover these topics and more.

Summary

Congratulations! You have built a complete, functional e-commerce site in 30 minutes. It is spartan and simple, but it will work for some applications. We have skipped over many aspects of an e-commerce application, but will build upon this skeleton framework throughout the remainder of this book. You should now have built a basic framework that:

- Represents Product information, such as price and description in a Django model
- Stores and organizes products using the database and a catalog Django model
- Employs generic views and their related templates to run a simple web-based storefront
- Sells products and accepts payment through the Google Checkout system

In the next chapter we will construct a new module to handle our customer's information, accounts, and their orders. We will also implement some basic promotional activities, such as product discounts. We will also create a simple shopping cart and further refine the checkout process.

3
Handling Customers and Their Orders

Now that we have a basic e-commerce framework constructed in Django, we will begin enhancing it with additional functionality. The first obvious need is a method of tracking our customer's information, especially their orders, and providing them with access to information about their accounts. Django will once again help us with these features. Utilizing some popular third-party modules, the Django community will be playing a part as well. In short, this chapter will explain:

- The Django authentication framework and User model
- Use of third-party modules, django-registration, and django-profiles to simplify the account creation process
- Creation of an ordering system, including a model to track customer orders
- A very simple customer product review tool

First, we will discuss Django's built-in module for handling authentication, users, and permissions. We will implement an account creation system using django-registration and django-profiles, two third-party modules that have a significant adoption rate in the Django community.

Once we've established accounts for our customers, we can begin associating orders with them. In fact, we will build a very general order framework that will not only support orders from registered users, but also anonymous ones. To assist in these goals, we will build a classic shopping cart tool and discuss how to implement it using Django's session framework. We will also survey the requirements of a more powerful payment processing system, which we will ultimately create in *Chapter 4, Building Payment Processors*.

Finally, we will end the chapter with a discussion of Django's comments framework and how to employ it as a rudimentary customer feedback tool. This feedback tool will allow our customer's to provide feedback for any product in our database.

Django's auth module

One of the most tedious and error-prone tasks of a web developer is managing account details. The reason this is so difficult is because it has to be perfect. If a user cannot log in, most sites are useless. It's also really important to maintain the security of user accounts, allow users to update their information, change their passwords, and many other administrative tasks.

This functionality must be present in almost all major websites and applications. This is another area where Django can save a lot of time and effort, while improving the quality of your software. By providing a standard user account system, developers can avoid the headaches inherent in writing their own user code or developing an in-house library.

All Django sites can use the same authentication interface, thus relieving developers not just of the burden of developing their own interface, but also of maintenance. Django's `auth` module is heavily used and rigorously tested, so when bugs are found, especially critical security problems, they are generally fixed very quickly. An in-house module or a one-off utility generally will not have this many eyes looking at it, monitoring it for changes, and testing any additions.

The `auth` module, like many of Django's built-in tools, is also very flexible. It supports a pluggable authentication back-end that allows developers to write custom authentication sources. This means if you have a system of authenticating users already, you can very easily replace the default authentication back-end with one that supports your system. Possible systems include LDAP, SQL, or almost anything that can be accessed via Python code. Django can even support multiple back-ends that are accessed depending on the user. More information on custom authentication back-ends is available in the Django documentation at: `http://docs.djangoproject.com/en/dev/topics/auth/#other-authentication-sources`.

Django's default authentication system uses a database table to store user information, including passwords. Passwords are not an ideal authentication mechanism, but it's the standard idiom for most networked software. As a result, one of the most important security concerns is to protect user passwords at all costs. Often users use the same password for multiple sites or applications and their password is only as secure as the weakest site they use.

To ensure that a user's password is kept secure, a website or application should never deal in plaintext passwords. This means it should never e-mail passwords in plaintext or display a password in plaintext on an account profile, and it definitely means passwords should never be stored in a database in plaintext. It may come as a surprise to many developers (and users!) how often passwords are passed around in plaintext.

It is unfortunate, but true, that many web-based login forms are accessed via standard HTTP connections and not a secure SSL connection. This means that passwords are sent across the Internet in plaintext. Though it requires a small degree of skill, the implication of this is that anyone listening on a connection can retrieve a user's login password.

It is highly recommended that any serious e-commerce system secure their login process, at the very least. If your site displays any kind of sensitive data, it is also recommended that you secure those connections using HTTPS. SSL Certificates are required to use secure HTTP, but for trivial or intranet applications, it is possible to generate your own certificates. However, for anything public-facing that will be used by a large number of people, it is recommended you purchase a certificate from one of the well-known, trusted root certificate authorities. Further discussion of this issue is beyond the scope of this book, but you can find much more information on the Web. A good place to start, for Apache users, is the Apache SSL/TLS Encryption page: `http://httpd.apache.org/docs/2.0/ssl/`.

The other cases of plaintext passwords are easily mitigated, either automatically by Django, or using some simple best practices. For example, Django's `auth` module automatically stores user passwords in the database using a salted hash technique. By default, Django uses the standard Python `sha1` hash algorithm. This ensures that no passwords are readable simply by browsing the database, by making sure they are not stored in plaintext. And, the salt ensures the stored hashed value is not decipherable without knowing the password.

The case of sending plaintext passwords in e-mail is easily avoided: just don't do it! If you need to implement a password reset function, there are many ways to do so using simple URLs that can be included in an e-mail. The third-party `django-registration` tool handles this automatically, as we will see later in this chapter.

Django users and profiles

Django's `auth` module provides all user accounts with a `User` object. The `User` model can be found in `django.contrib.auth.models`. It provides a number of basic user functionalities, including storage of a name, e-mail address, and access privileges—as in whether the user is a staff member, has an active account, or is a super user. It also records when the user signed-up for an account and when they last logged-in.

For an e-commerce application, the default set of information stored by Django for each user is useful, but likely not enough. For example, any web store that needs to ship its product to their customers will want to store the customer's mailing address. The `User` model does not include a field for mailing address, nor does it include phone numbers, birthdays, or a lot of other information that might be useful.

This is an intentionally minimalistic design on Django's part. Instead of trying to predict what fields Django developers require, the decision was made to provide a simple method of extending the Django `User` model. This way the application developers can create whatever representation of additional user data they need and attach it automatically to Django's built-in `User` model.

Extending this user information is very simple: write your own model with a `ForeignKey` to the `User` model and whatever extra data attributes you require. This is called a profile model. When you have built this model, you can tell Django about it using a special setting: `AUTH_PROFILE_MODULE`. This setting will point to your custom extension model using the `app.model` syntax. For example, if your app is called `myapp` and this app contains a model called `UserProfile`, you would refer to it as the string `'myapp.UserProfile'`.

By creating this setting, you automatically enable a built-in feature for each `User` object: the `get_profile()` method. This method automatically returns an instance of your profile model specific to the `User` object you're working with. You can even access profile fields using dot-notation like so: `user_object.get_profile().address1`.

Note that when a `User` object is created, by a new user registering an account, a corresponding profile model is not automatically created. You must create it yourself. The best way to accomplish this is to use Django's signals feature and write a handler to listen for `django.db.models.signals.post_save` on the `User` model. This would look something like this:

```
fromdjango.db.models.signals import post_save
fromdjango.contrib.auth.models import User
frommyapp.models import UserProfile

defprofile_creation_handler(sender, **kwargs):
instance = kwargs['instance']
profile, created = UserProfile.objects.get_or_create(user=instance)

post_save.connect(profile_creation_handler, sender=User)
```

Django uses signals to send messages from one piece of code to any other piece, which is called "listening". In the previous source code, we were setting up a listener, or handler, called `profile_creation_handler`. This handler function will run every time Django sends the `post_save` signal. But this particular handler will ignore any `post_save` signals that are not sent by a `User` object, as specified by the `sender` keyword argument in the `connect` call. The `connect` method is how we attach handlers to a specific signal. Django's documentation includes an explanation of the signal system and a full list of all built-in signals.

Creating accounts with django-registration

The `auth` module provides a set of functions to handle basic operations involving user accounts. There are built-in views that you can use to allow users to log in, log out, and reset passwords. There are also forms included in the `auth` module that you can use for similar functionality. One such form is the `UserCreationForm`, which provides a very simple mechanism to create new user accounts.

The built-in views and forms are very useful, but somewhat low-level. There is a lot of functionality you must provide yourself if you intend to use them. You could also write your own forms and views to handle these operations, especially if you need some custom functionality. Generally, though, it's best not to repeat ourselves. If code already exists, and looks like it will meet our needs, then we should try using it first.

The ultimate example of this is the community of third-party applications that exist for Django. Sometimes these exist as a small snippet of code on `djangosnippets.org`. Other times these are fully-fledged open source projects with multiple contributors and their own homepage.

One such larger project is `django-registration`, a reusable Django application designed to provide several convenient functions for handling new user creation. This registration app was created by James Bennett, the current Django release manager, and it is very popular and widely used. The current version of the project, with installation instructions, is available at this URL: `http://bitbucket.org/ubernostrum/django-registration/wiki/Home`.

After installing the registration app, as with any well-written third-party Django app, adding it to your project is very simple. Simply update the `INSTALLED_APPS` setting in your settings file by adding an entry for `'registration'`. Once added to your settings, you can run `syncdb` to create the necessary tables. Django-registration uses a few extra models to manage the current state of a user registration.

For example, when a user signs up for an account, it creates a new `User` object and then sends them an e-mail to verify their e-mail address. Until the link provided in this e-mail is clicked, the new `User` object is marked as inactive and cannot be used for logging in. This behavior is customizable, of course, but it is recommended that you verify e-mail addresses of new user accounts to prevent spam or other fraudulent account creations.

There are currently a few competing philosophies around how to build and distribute third-party Django applications. It doesn't make sense, often, to include things such as templates in a third-party utility, because every developer will require customization to these templates. Instead, projects are often designed with default template names in their included views. These default templates are not included with the app, but can quickly be built by the developer for their project and the app views will automatically locate them. This is very similar to the way generic views worked in the previous chapter.

If you prefer to get up and running quickly, you can simply add an entry for the default URLs file, included with the registration app, to your root URLs. This would look something like this (on registration v0.8):

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

This will automatically create URL patterns for the views exposed by `django-registration`. These include a register view and an account verification view. For example, if you included the above URL pattern in your root URLs, the registration view will be at `/accounts/register/`.

The register view is the main view in the registration app. This view and all the others are described in detail in the registration documentation. We will only discuss the register view here. By default the register view renders a template called `registration/registration_form.html` and provides this template with a context variable called form. The form variable is an instance of the `RegistrationForm` class, which is included in the app. This form provides all the necessary fields to create an account. These fields include: username, e-mail, and two fields to verify a password.

It is possible to substitute your own form class for the default `RegistrationForm` returned by the register view. To do this, you will need to add a custom URL pattern for the register view and provide a view parameter called `form_class`, which points to a Python class to use as the form. For example, to add a custom form to the register view, but keep all other registration views at their defaults, you could modify the above URL pattern like so:

```
(r'^accounts/register/$', 'registration.views.register',
 {'form_class': MyRegistrationForm}),
(r'^accounts/', include('registration.backends.default.urls')),
```

By placing the custom URL pattern ahead of the defaults, we can override registration's built-in register view. We've also instructed the register view to use our own registration form instead of the default. Note also that each view in the registration app has many parameters to override. Another useful example is the `template_name` parameter, which can be changed to use your own template instead of the default `registration/registration_form.html`.

Generally, when creating a custom registration form, you will want to inherit from registration's default `RegistrationForm` and add your own fields. The registration app also includes some additional registration forms with added functionality, which you can use as examples when writing your own.

Using `django-registration` to allow your users to create accounts is completely optional. We could spend time implementing our own approach to this problem, but as we've seen that the flexibility provided by this particular third-party tool is enough to allow many customizations. We have only scratched the surface of the available customizations in this app, so it is highly recommended you explore the documentation further to see all that it can do.

Extending the user model with django-profiles

Just as `django-registration` provides a basic set of services for users to create accounts, login, and change passwords, another third-party app provides similar services for profile creation. `Django-profiles` is also written by James Bennett and is available from the following URL: `http://bitbucket.org/ubernostrum/django-profiles/wiki/Home`.

The `django-profiles` app builds off of Django's built-in support for extending the User model using the `AUTH_PROFILE_MODULE` setting we discussed earlier. It essentially adds a set of generic views, URL patterns, and templates to allow your site to quickly and automatically handle profile creation and changes.

Like `django-registration`, the profiles app is installed by following the system installation instructions included in the app documentation, then adding `'profiles'` to your `INSTALLED_APPS` setting. You will also need to create a profile model, as we previously discussed, and set `AUTH_PROFILE_MODULE` appropriately. Once installed, you can add the profile URL patterns to your root URL patterns file like so:

```
(r'^profiles/', include('profiles.urls')),
```

As before, you can customize almost all of the profiles application using custom template names, custom forms, and even custom URL patterns.

In the next section we will build a profile class that will extend Django's User model. Using `django-profiles` is not necessary to proceed to the next section, nor is it required for the rest of this book. In the remaining chapters we will assume that the profile we built in the next section is available on all `User` objects, whether it was created manually, using `django-profiles`, or by some other means.

The customer profile

We will now build a very lightweight profile model to store additional information about our customers. The `Customer` model will initially include three fields: a `ForeignKey` to the `User` model, an address, and a phone number. Because the profile model is simply a regular Django model class, any additional information can be added using the usual fields.

To simplify this model design slightly, we will also be creating a `CustomerAddress` model that represents a specific address. It will include fields such as three lines of address information (apartment number, street name, and so on), city, postal code, and state information. The `CustomerAddress` model will be referenced as a `ForeignKey` from the `Customer` profile.

Here is the sample code for the profile model:

```
class Customer(models.Model):
'''
The Customer model represents a customer of the online store. It
extends Django's built-in auth.User model, which contains information
such as first and last name, and e-mail, and adds phone number and
address information.
'''
    user = models.ForeignKey(User)
    address = models.ForeignKey('CustomerAddress')
    phone_number = PhoneNumberField(blank=True)

classCustomerAddress(models.Model):
    '''
    The CustomerAddress model represents a customer's address. It is
    biased in favor of US addresses, but should contain enough fields
    to also represent addresses in other countries.
    '''
    line_1 = models.CharField(max_length=300)
    line_2 = models.CharField(max_length=300)
    line_3 = models.CharField(max_length=300)
    city = models.CharField(max_length=150)
    postalcode = models.CharField(max_length=10)
    state = USStateField(blank=True)
    country = models.CharField(max_length=150)
```

The idea here is that our `USStateField` is optional, with postal code and country information captured for non-US addresses. This could be modified to be even more flexible for international addresses by using a `CharField` instead of `USStateField`.

These models will live in an application called customers in the `coleman` framework. Once we've created them, we will need to do two things to activate the `Customer` class as an extension to the `User` model. First, add `'coleman.customers'` to `INSTALLED_APPS`, then, set `AUTH_PROFILE_MODULE='customers.Customer'`. These two steps will also allow `django-profiles`, if in use, to automatically detect the profile model.

Taking orders: Models

Now that we have a way for customers to create accounts and update their profiles with their address information, we can build a simple order taking system. Broadly speaking, there are three parts to the order taking process:

- A set of views to allow customers to manage their orders. This includes a shopping cart utility that they can view at any time. It also includes an order review/check-out page.
- The checkout process itself. This is where the order is translated from our internal representation to a format that can be processed by a payment processor. For example, for Google Checkout's API, an XML document is created and submitted via HTTP Post to a Checkout endpoint on Google's servers.
- An order model to store data about an order, what products are involved and who is doing the ordering. This will usually only be stored after an order is completed, so that customers and site staff can review or confirm the details.

We will begin by constructing a model to represent orders. Even though in the ordering process this is the last task we perform, we need to know exactly what information to gather during the earlier parts of the process. The best way to do this is to define the model in advance and build views, a shopping cart, and checkout processor around this data.

A simple order model has two main actors: the customer doing the ordering and the products they want. Additional data we will probably need include things such as:

- The date the order was placed
- The total price of the order
- The price of individual products in the order and a status of where the order currently stands

This model looks something like the following:

```
class Order(models.Model):
    '''
    The Order model represents a customer order. It includes a
    ManyToManyField of products the customer is ordering and stores
    the date and total price information.
    '''
    customer = models.ForeignKey(User, blank=True, null=True)
    status_code = models.ForeignKey('StatusCode')
    date_placed = models.DateTimeField()
    total_price = models.DecimalField(max_digits=7, decimal_places=2)
    comments = models.TextField(blank=True)
    products = models.ManyToManyField(Product,
                                      through='ProductInOrder')
```

Notice that we have built the order model around Django's built-in `User` object instead of our `Customer` profile model. This is partly because it's easier to work with `User` objects in views, but also to prevent changes to the profile model from affecting our orders. The `User` model is mostly guaranteed to never change, whereas if we decided to replace our `Customer` profile model, deleted a profile accidentally, or had some other change, it could damage or delete our `Order` information.

There are two relationship fields in the `Order` model, `status_code` and `products`. We will discuss status codes first. This field is intended to function as a tracking system. Using a `ForeignKey` and a `StatusCode` model lets us add new statuses to reflect future changes in our ordering process, but still require all `Order`s to have a standard set of statuses for filtering and other purposes. The `StatusCode` model looks like this:

```
classStatusCode(models.Model):
    '''
    The StatusCode model represents the status of an order in the
    system.
    '''
    short_name = models.CharField(max_length=10)
    name = models.CharField(max_length=300)
    description = models.TextField()
```

Returning to our fictitious business example from *Chapter 2, Setting Up Shop in 30 Minutes*, imagine that CranStore.com has a four-step ordering process. This includes orders that are newly placed, those that have been shipped, those that were shipped and received by the customer, and those that have a problem. These status conditions map to the `StatusCode` model using the following values for `short_name`: NEW, SHIPPED, CLOSED, EXCEPTION. Additional details as to what each state involves can be added to the name and description fields for use on customer-friendly status tracking pages.

The second relationship field in the `Order` model is significantly more complicated and probably the most important field for all orders. This is the `ManyToManyField` called products. This field relates each order to a set of `Product` models from our design in *Chapter 2*. However, this relationship from `Orders` to `Products` requires a lot of additional information that pertains to the actual relationship, not the order or product themselves.

For example, CranStore.com sells a lot of canned Cranberries around late November. When a customer places an order for canned Cranberries, we need to know not just what they ordered, but how many cans they wanted and how much we charged them for each can. A simple `ManyToManyField` is not sophisticated enough to capture this information. This is the purpose behind this field's `through` argument.

In Django the through argument on a `ManyToManyField` lets us store information about the relationship between two models in an intermediary model. This way we can save things like of a product in an order, without too much complexity in our design. The downside of using the through argument is that we can no longer perform operations such as add and create, directly on an object instance's `ManyToManyField`.

If our Order model used a standard `ManyToManyField`, we could add `Product` objects to an order simply by calling add like so:

```
order_object.products.add(cranberry_product)
```

This is no longer possible with the through field, we must instead create an instance of the intermediary model directly to add something to our Order object's products field, as in the following:

```
ProductInOrder.objects.create(order=order_object, product=cranberry_
product)
```

This creates a new `ProductInOrder` instance and updates the products field of `order_object` to include our `cranberry_product`. We can still use the usual `ManyToManyFieldQuerySet` operations, such as `filter` and `order_by`, on fields using the `through` argument.

In addition we can filter the `Order` model using fields on the `ProductInOrder` intermediary model. To find all open `Orders` that include canned Cranberries, we could perform the following:

```
Product.objects.filter(productinorder__product=cranberry_product,
status_code__short_name='OPEN')
```

The full `ProductInOrder` model is listed below:

```
classProductInOrder(models.Model):
    '''
    The ProductInOrder model represents information about a specific
    product ordered by a customer.
    '''
    order = models.ForeignKey(Order)
    product = models.ForeignKey(Product)
    unit_price = models.DecimalField(max_digits=7, decimal_places=2)
    total_price = models.DecimalField(max_digits=7, decimal_places=2)
    quantity = models.PositiveIntegerField()
    comments = models.TextField(blank=True)
```

These three models represent our basic order taking system. Now that we have an idea of the data we will be collecting for each order, we can begin building a set of views so that customers can place orders through the Web.

Taking orders: Views

We will build four views for our initial order taking system. These will perform basic e-commerce operations such as: review a shopping cart, add an item to the cart, remove an item from the cart, and checkout. We will discuss the mechanics of the shopping cart system in the next section. For now, we will just focus on these four operations, how they translate to Django views, and what their templates would look like.

The most important view is the `shopping_cart` view, which allows a customer to review the products they have selected for purchase. The code for this view appears below:

```
defshopping_cart(request, template_name='orders/shopping_cart.html'):
    '''
    This view allows a customer to see what products are currently in
    their shopping cart.
    '''
    cart = get_shopping_cart(request)
    ctx = {'cart': cart}
    return render_to_response(template_name, ctx,

context_instance=RequestContext(request))
```

The view retrieves the shopping cart object for this request, adds it to the template context, and returns a rendered template whose only variable is the shopping cart, called cart. We will discuss `get_shopping_cart` shortly, but for now let's just note that we are using this as a helper function in our views. The retrieval operation is very simple and would easily fit on one line in this view function. However, by using a helper function, if we ever need to make changes in the future to how our cart works, we have a single piece of code to change and can avoid having to touch every view.

The next two operations are related: `add_to_cart` and `remove_from_cart`. The code for these views is as follows:

```
def add_to_cart(request, queryset, object_id=None, slug=None,
                slug_field='slug', template_name='orders/add_to_cart.
                html'):
    '''
    This view allows a customer to add a product to their shopping
    cart. A single GET parameter can be included to specify the
    quantity of the product to add.
    '''
    obj = lookup_object(queryset, object_id, slug, slug_field)
    quantity = request.GET.get('quantity', 1)
    cart = get_shopping_cart(request)
    cart.add_item(obj, quantity)
    update_shopping_cart(request, cart)
    ctx = {'object': obj, 'cart': cart}
    return render_to_response(template_name, ctx,

context_instance=RequestContext(request))

def remove_from_cart(request, cart_item_id,
                     template_name='orders/remove_from_cart.html'):
    '''
    This view allows a customer to remove a product from their
    shopping cart. It simply removes the entire product from the cart,
    without regard to quantities.
    '''
    cart = get_shopping_cart(request)
    cart.remove_item(cart_item_id)
    update_shopping_cart(request, cart)
    ctx = {'cart': cart}
    return render_to_response(template_name, ctx,

context_instance=RequestContext(request))
```

A careful review of these two functions will reveal that they have two different interfaces for managing cart items. The `add_to_cart` view works with an object directly, an instance of our `Product` model from *Chapter 2*, usually. The `remove_from_cart` function needs a `cart_item_id` variable. This is simply an integer value that indexes into the shopping cart's item list.

We must use two different interfaces here, because once we've added a product to the cart, we need a unique way of talking about that particular product at that particular index in the cart. If our remove function tries to remove items based on the item's model or a primary key value, we would have difficulty when the same item has been added to our cart twice.

The solution is to give every item added to the cart a unique identifier. The `Cart` model in the next section handles this unique ID internally; all we need to do is render it in our templates when necessary. This way we can add arbitrary objects to our shopping cart as often as we want and retain the ability to remove a specific addition, whether it was performed in error or because the customer changed their mind.

When adding an item to the cart, we simply need to look up the item's `slug` or `object_id` value from the Django ORM and store the object directly into the cart. When added, the cart will assign it an identifier.

There is another helper function in these views called `lookup_object`, which performs the actual retrieval of our `Product` or other object that we're adding to the cart. This helper function looks like this:

```
def lookup_object(queryset, object_id=None, slug=None, slug_
field=None):
    if object_id is not None:
        obj = queryset.get(pk=object_id)
    elif slug and slug_field:
        kwargs = {slug_field: slug}
        obj = queryset.get(**kwargs)
     else:
         raise Http404
     return obj
```

This allows some flexibility and reuse for our `add_to_cart` view, because we can define which lookup fields (`object_id` or `slug` and `slug_field`) to use for retrieving the object. If we decided to use a different `Product` model than the one written in *Chapter 2* and this alternative model did not use slug fields as identifiers, then we would still be able to use our `add_to_cart` view without any modification. Separating the lookup with a helper function allows us to write clean, short, and easy to read views.

This view is not so flexible, however, that we can feel comfortable calling it a "generic view", without some additional work. Generic views, patterned after Django's built-in generic views discussed in *Chapter 2*, allow almost every aspect of their functionality to be configurable. For example, if these views were really generic, we could specify a custom class to use as our shopping cart, instead of what `get_shopping_cart` gives us. We could even add a keyword argument to `get_shopping_cart` that would allow different cart classes and pass through a value from our view's arguments.

The last view in our order taking system is the checkout view. This will act as a stepping stone into our payment processor system. In the previous chapter we saw a very simple payment processor; we will build a more advanced one here, and discuss the issue in detail in *Chapter 4*. The checkout view is listed below, note that it is specifically designed for the Google Checkout API we discussed previously:

```
def checkout(request, template_name='orders/checkout.html'):
    '''
    This view presents the user with an order confirmation page and
    the final order button to process their order through a checkout
    system.
    '''
    cart = get_shopping_cart(request)
    googleCart, googleSig = sign_google_cart(cart)
    ctx = {'cart': cart,
           'googleCart': googleCart,
           'googleSig': googleSig,
           'googleMerchantKey': settings.GOOGLE_MERCHANT_KEY,
           'googleMerchantID': settings.GOOGLE_MERCHANT_ID}
    return render_to_response(template_name, ctx,

context_instance=RequestContext(request))
```

The various Google context variables are needed to compose a Checkout API form in the rendered template. The additional merchant key and ID information is stored in our project's settings file. There is also a helper function, `sign_google_cart`, which generates a Google-compatible representation of our shopping cart as well as a cryptographic signature.

The Google-compatible representation of our shopping cart is simply an XML file that conforms to the specifications documented by the Checkout API. The signature is a base64 encoded HMAC SHA-1 signature generated from the shopping cart XML and our secret Google merchant key. This prevents tampering when our checkout form is submitted to the Google Checkout system. More information on this process is available at: `http://code.google.com/apis/checkout/developer/index.html`.

We will write these functions and examine the XML file in detail later in this chapter.

Shopping carts and Django sessions

Django makes it very easy to build a shopping cart for our web-based e-commerce stores and applications. In fact, Django does an excellent job of solving a more general problem: persisting temporary data between browser requests in a simple way. This is what Django's `session` framework gives us.

Any pickleable Python object can be stored using the session framework. The session object is attached to incoming requests and is accessible from any of our views. Requests from users that are not logged in, or haven't even created accounts, get session objects too. Django manages all of this by associating a cookie in the user's browser with a session ID. When the browser issues a request, the `SessionMiddleware` attaches the appropriate session object to the request so that it is available in our views.

Sessions, and the cookies that manage them, have an expiration date. This expiration is not necessarily tied to the expiration of browser cookies and can be controlled in our views using the session object's `set_expiry()` method. This method allows for four expiration possibilities:

- Never
- When the user's browser is closed
- After a specified amount of time has elapsed
- The default global Django setting

If the user happens to clear their browser's cookies, then any previous session information will be lost and they will receive a new session object the next time they visit the site. The information attached to these abandoned sessions is retained by Django in the database until it's manually cleared using the cleanup command to `django-admin.py`.

As we can use Django's session framework to store any pickleable Python object (see `http://docs.python.org/library/pickle.html` for more information on pickling), it makes storing shopping cart information very easy. We will build a simple Cart class that represents a list of items with methods that can add and remove from the list. Each item in the list is represented by an inner-class called `Item` that manages the item's ID in a particular shopping cart and a quantity value. This code appears as shown:

```
class Cart(object):
    class Item(object):
        def __init__(self, itemid, product, quantity=1):
            self.itemid = itemid
            self.product = product
            self.quantity = quantity
```

```
    def __init__(self):
         self.items = list()
         self.unique_item_id = 0

    def _get_next_item_id(self):
        self.unique_item_id += 1
        return self.unique_item_id
    next_item_id = property(_get_next_item_id)

    def add_item(self, product, quantity=1):
        item = Item(self.next_item_id, product, quantity)
        self.items.append(item)

    def remove_item(self, itemid):
        self.items = filter(lambda x: x.itemid != itemid, self.items)
```

`Cart` objects manage their own list of unique item IDs by incrementing a variable each time we add a new item. This is the purpose of the `next_item_id` property. At removal time, we can use the item's unique ID to easily find it in the items list and remove it.

As a further convenience, particularly when we need to access the contents of a shopping cart in our templates, we can make the `Cart` class iterable. This is a Python convention that allows container classes to function like sequences or lists. For example, using a non-iterable version of the `Cart` class, we could access the items list like so:

```
for item in cart_instance.items:
        ...
```

By transforming our cart into an iterable, however, we can simplify the above to just:

```
for item in cart_instance:
        ...
```

What's the big deal? We saved a few keystrokes and some syntax. But in the future, if we need to change the interface to our items list, the iterable approach prevents us from relying on semantic knowledge of the internal workings of our shopping cart. The underlying storage of items in the cart can change and we only need to update our iterable code. To make our `Cart` class iterable, we add the following two methods:

```
def __iter__(self):
     return self.forward()
def forward(self):
     current_index = 0
     while (current_index<len(self.items)):
         item = self.items[current_index]
         current_index += 1
         yield item
```

Later, if our shopping cart needs change, the `Cart` class's `forward()` method will act as a single point of control. We can implement any changes needed, but retain the same iterable interface in our views and templates.

Checking out: Take two

Now we can adapt our checkout process to use our new shopping cart. Previously we created a simple checkout process using Django templates and the Google Checkout buy-it-now button. The obvious limitation of this approach was that customers can only purchase one item at a time. Using our shopping cart, we allow customers to select as many items as they like, but this requires us to modify the checkout process.

Earlier we discussed the checkout view and a helper method, `doGoogleCheckout()`, which performed two actions. First, it converted our internal representation of the customer's shopping cart to a format that Google Checkout API can understand. Second, it created a cryptographic signature of this shopping cart using our secret Merchant Key identification. This signature is how Google Checkout's API verifies that the purchase request is coming from our web application.

Here is the `doGoogleCheckout` function in its entirety:

```
Import hmac, sha, base64
from django.template.loader import render_to_string
from django.conf import settings

def sign_google_cart(cart):
    cart_cleartext =render_to_string('orders/googlecheckout.xml',
                                     {'cart': cart})
    cart_sig = hmac.new(settings.GoogleMerchantKey,
    cart_cleartext, sha).digest()
    cart_base64 = base64.b64encode(cart_cleartext)
    sig_base64 = base64.b64encode(cart_sig)
    return cart_base64, sig_base64
```

As stated earlier, Google Checkout API-compatible shopping carts are represented as XML documents. Here we render this XML document using Django's standard template rendering functions. We provide a context that includes an instance of our internal Cart class.

After rendering the XML, we capture it as the string variable `cart_cleartext`. This is what we use to generate a cryptographic signature. The Checkout API is specific about how to sign our shopping cart: it must use the HMAC SHA-1 algorithm. Fortunately, Python makes this sort of signature generation very easy.

Finally, both the cleartext version of our Checkout API-compatible shopping cart and the digital signature are base-64 encoded so that they can be submitted for processing through an HTML form field in an HTTP POST request.

These two base-64 encoded values are returned to our checkout view and are included in the template context. They will be rendered in the template as fields on an HTML form that submits directly to the Google Checkout API.

The last key piece of this is the XML template to render the cleartext shopping cart. The specific format and XML schema are documented at the following URL: `http://code.google.com/apis/checkout/developer/Google_Checkout_XML_API.html`.

The actual design of this checkout XML file will depend on the specific application because it includes things such as shipping and tax tables, coupon codes, and other merchant-specific information. For our example application, here is our basic `googlecheckout.xml` template. Note the use of Django template tags and variables, just as if it were an HTML document.

```
<?xml version="1.0" encoding="UTF-8"?>
<checkout-shopping-cart xmlns="http://checkout.google.com/schema/2">
<shopping-cart>
<items>
      {% for item in cart %}
<item>
<item-name>{{ item.product.name }}</item-name>
<item-description>{{ item.product.description }}</item-description>
<unit-price currency="USD">{{ item.product.price_in_dollars }}
</unit-price>
<quantity>{{ item.quantity }}</quantity>
</item>
      {% endfor %}
</items>
</shopping-cart>
<checkout-flow-support>
<merchant-checkout-flow-support>
<shipping-methods>
<flat-rate-shipping name="SuperShip Ground">
<price currency="USD">2.50</price>
</flat-rate-shipping>
</shipping-methods>
</merchant-checkout-flow-support>
</checkout-flow-support>
</checkout-shopping-cart>
```

This is the simplest possible shopping cart XML. It works by iterating over the items in our cart; it generates an `<item>` element for each one that includes child elements that capture details about the item we're selling, including its price. It also includes a very simple flat-rate shipping fee. There are dozens of additional tags you can use in the checkout XML. It is a very flexible system that can handle a wide variety of scenarios. We will discuss payment processing and checkout APIs in even more depth in the next chapter.

Super-simple customer reviews

Users of our e-commerce application can now sign up for accounts, browse our product catalog, add items to a shopping cart, and submit payments through the checkout system. To complete the circle, we will build simple, quick customer reviews using Django's built-in comments framework.

The comments framework is designed as a general purpose tool to attach comments to any Django object. To use it for customer reviews, we can attach comments to our `Product` model. Doing this is very easy, in fact it is basically automatic—just another convenience of working in Django.

To implement comments you only need to do two things: add `django.contrib.comments` to your `INSTALLED_APPS` project setting and add `django.contrib.comments.urls` to your root URL patterns under the `comments/` URL. Once you've done this, you can begin using comments in your templates.

For our product detail page, we will add a section for user reviews and a form for customers to complete their own review. All of these will be driven by the comments framework. In order to access the comment template tags, we need to first load the comments module into our template using `{% load comments%}`.

Once we have the comments utility loaded, we can begin using it. The following Django template snippet will create the customer review section of our product detail page:

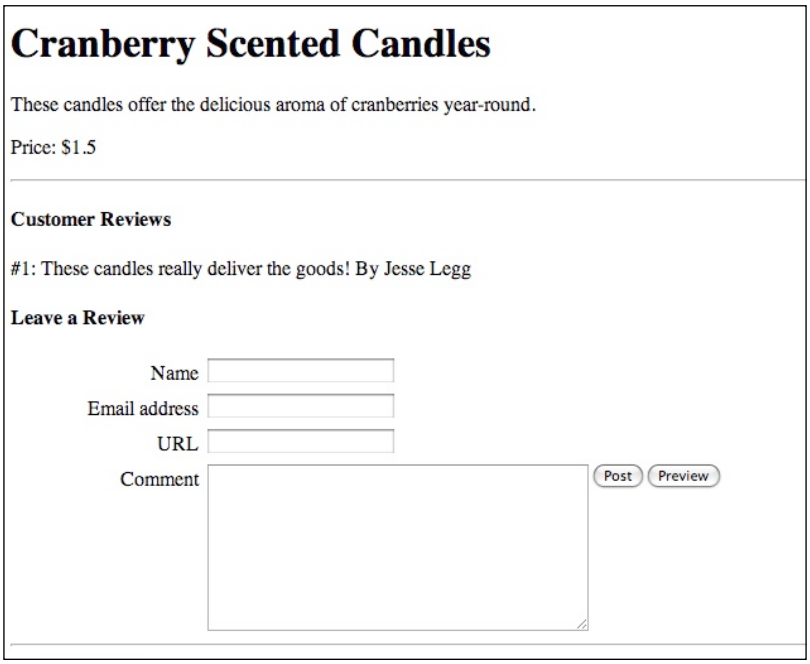
```
<h4>Customer Reviews</h4>
{% get_comment_list for object as comment_list %}
{% for comment in comment_list %}
<p>#{{ forloop.counter }}: {{ comment.comment }} By {{ comment.user_
name }}</p>
{% endfor %}
```

The `{% get_comment_list %}` tag is used to obtain the list of reviews for this particular Product object and store it in a template variable called `comment_list`. We can then iterate over this list and display the results using normal HTML and Django template markup. This is all that is required to add customer reviews or product comments to every item in your catalog.

Adding a form for customers to leave their own reviews is equally as simple as the following Django template snippet indicates:

```
<h4>Leave a Review</h4>
{% render_comment_form for object %}
```

Using the `{% render_comment_form %}` template tag creates the Django comments framework's default form. These simple defaults, with a little bit of CSS styling, will look like the following screenshot:



You will not be able to customize the fields or specify your own CSS classes in the HTML using this template tag. For that you will need to either create your own default comment form template, called `comments/form.html`, or use the `{% get_comment_form %}` template tag and place the Django form and field variables in your template manually. The full documentation for Django's comments framework is available here: `http://docs.djangoproject.com/en/dev/ref/contrib/comments/`.

Summary

This chapter aimed to cover the broad topic area surrounding customers and their orders. This is a very large concern and begins to tread into a much larger subject known as Customer Relationship Management (CRM). The e-commerce tools we built in this chapter included:

- Simple customer account creation and profile components
- A system for taking and tracking orders
- Implementation of a custom shopping cart
- Integration of our custom cart with the Google Checkout XML API
- Very basic customer product review and feedback functions

In the next chapter we will discuss the order payment processing mechanism in more depth, including building a generic framework to process payments from vendors other than Google Checkout. Later on in the book we will revisit the order and customer models we created here to add additional features, such as coupon codes and customer discounts.

4

Building Payment Processors

The goal of this chapter is to extend the basic system we've designed so far with a more robust and flexible payment system. Currently we have a shopping cart design that is independent of our payment process. We convert the cart to a format that is compatible with the Google Checkout API and then we can submit our customer orders for payment. We will now extend this system to achieve several things:

- Create a pluggable design so that payment systems can be reused, swapped out, or replaced
- Migrate our simple Google Checkout processor to our new system
- Discuss payment processors for Amazon FPS and PayPal services
- Explore advanced Django and Python design concepts such as class-based generic views and callable objects

After we have built our generic payment system, implementing additional payment services such as Amazon and PayPal will be relatively straightforward.

An additional goal in this chapter is to build a system that is not constrained to the payment services mentioned, but could be extended as a basis for any web-based payment system that can handle shopping cart orders and form-based payment initiation. Using the design laid out in this chapter, new implementations for other third-party services should be relatively straightforward.

Building a generic payment processor app

We start by creating an empty framework to act as the skeleton of any payment processors we will eventually build. This design will assume that there are two key components to a payment processor:

- Shopping cart translation
- Checkout submission

The first step is always to convert the contents of our customer's shopping cart into a format that can be understood by the payment service. We handled this in *Chapter 3, Handling Customers and Their Orders*, using the Django template system to render an XML file that represented a customer order in an XML document the Google Checkout API could understand. Many web services use an XML format for transmitting e-commerce data and fortunately Django makes it very easy to generate these files.

Step two is to take our newly translated shopping cart data and walk the customer through a checkout process. Initially we'll render a confirmation page listing the contents of the shopping cart and the quantity and price for each item. If the user confirms this information, their order will be submitted to the payment service where they will complete the payment.

The process described in this book will complete the payment process through the service provider interface. Often these pages can be customized with a business logo and other design elements, but the interaction will be entirely handled by the web service provider. For most organizations this is a highly recommended practice.

Medium to small companies generally don't have the resources to commit to the security practices necessary to secure a checkout service running on their own site. Despite the inability to brand and control this portion of your sales process, it is still preferable for its enhanced security. It also provides peace of mind if there is a major breach or other crisis; your business will not be required to deal with it directly. You can rely on the payment service to fix the problem quickly, because it will affect all of their clients and their own bottom line. For this reason, as well, we will focus on the biggest, most reliable payment services.

Class-based views

The approach we will take to implement our payments framework involves a focus on reuse and simplification of code. As this code will generally live in the view-layer of our application, one excellent pattern to use here is the class-based view. Unlike functional views, class-based views allow the usual gains in robustness inherent in object-oriented programming.

Using this method, we will create a standard Python class to act as a superclass for specialized versions of our views. This parent class will implement all of the basic functions for the view, but avoids specific details related to our application. Application specific logic will be implemented in child classes using polymorphism to modify our parent's basic set of methods.

These classes become suitable for use as views in URL patterns once we make them callable. In Python, any class can be made "callable" by implementing a `__call__` method. This allows object instances of the class to be called such as functions. The following code creates a callable object from a `MyCallable` class:

```
>>> obj = MyCallable(initial_data=[...])
>>> # call the object...
>>> result = obj()
```

Note that this is equivalent to the following:

```
>>> obj = MyCallable(initial_data=[...])
>>> result = obj.__call__()
```

The `__call__` method is a Python special method that allows any object to be executed using the function syntax shown in the first example. In Python any object that can be called this way is generally referred to as **callable** or as a **callable object**. You can use Python's introspection to check whether an object you are working with is callable or not:

```
>>> if callable(obj): print "True!"
```

Implementing a callable object has many advantages over simply writing a method and referring to that method anywhere you need to obtain certain functionality from the object. It is a form of abstraction; you can write your code to expect a callable object and just call it. The implementation details, including the names of the methods that run, are up to the class.

In our specific example, Django URL patterns were designed to use functions as views. But really it is more accurate to say that they are designed to use callables. Thus by writing callable classes, we can construct object instances directly in the URL patterns. The classes themselves don't matter as long as they are callable. This is one of several examples that make Python a unique and interesting language to work with.

Implementing a checkout view base class

The base class for our class-based view checkout system will be called `CheckoutView`. All class-based generic views are going to share a basic set of requirements: be callable, store a template name, create a response to an incoming request that renders the template, and provide whatever extra functionality, such as form creation, that the view needs.

Some of this information, such as the template name and any form classes, can be stored as data attributes on the view. Others, such as rendering a request, will be implemented as a method. Anything that is possible with standard Django views can be implemented using a class-based design. For complex designs, the class-based solution will often be easier to read, manage, and reuse later.

Using function-based generic views, the typical usage pattern in Django is to include arguments from the URL pattern to specify the view behavior. For example, Django's built-in `list_detail.object_list` generic view takes several arguments: an object ID, a slug value, the queryset to filter, and so on. These are passed either from the URL itself, via a regular expression, or a dictionary of parameters defined in the URLs file. Either way, we have to write extra code.

Using class-based views, it makes more sense to take advantage of polymorphism and inheritance rather than passing arguments to a function. This simplifies our URL patterns and cleanly separates view code into a typical class hierarchy. This approach means we can avoid passing a lot of data from our URLs but still achieve the goal of code reuse.

There still may be data attributes we can treat as constructor parameters, the template name for example, but most of the data should exist as class attributes. An excellent demonstration of this is the `extra_context` dictionary. This is a dictionary of additional key-value pairs to be added to the template context when it is rendered. It can get unwieldy to define this dictionary in the `urls.py` file, but using a class-based view it becomes a simple matter of adding a data attribute to the class. When the view is rendered we can update the context with the value of `self.extra_context`.

Let's demonstrate these concepts by revealing our base `CheckoutView` class:

```
class CheckoutView(object):
    template = 'payments/checkout.html'
    extra_context = {}
    cart = None
    cart_class = Cart
    request = None
    form = None
    form_class = None

    def __init__(self, template=None, form_class=CheckoutForm,
                       context_class=RequestContext):
        self.context_class = context_class
        self.form_class = form_class
        if template is not None:
            self.template = template
```

```
    def __unicode__(self):
        return self.__class__.__name__

    def get_shopping_cart(self, request):
        self.cart = request.session.get('cart', None) or self.cart_
class()

    def __call__(self, request):
        self.request = request
        return self.return_response()

    def return_response(self):
        self.form = self.form_class()
        context = {'cart': self.cart, 'form': self.form}
        context.update(self.get_extra_context())
        return render_to_response(self.template, context,
                                  context_instance=self.context_
                                  class(self.request))

    def get_extra_context(self):
         return self.extra_context
```

Here are the key features of this class: we made it callable by defining a `__call__` method, we store the view parameters as class attributes, and the generic base-class provides default values for some of these attributes.

Saving the order

As we will be processing payments using third-party payment processors, we need to save the order before sending it to the customer to make their payment. We also need to be able to create an order from our shopping cart object so that we can refer to it when we begin shipping after receiving payment.

Most payment processors allow us to add unique information specific to an order reference number for this purpose. When payment succeeds, we can update the status of the order in our system from `'NEW'` to something meaningful like `'PAID'`. Most payment processors will notify vendors via e-mail when an order has been paid and some even allow this to be automated. We will explore this topic more in *Chapter 5, From Payment to Porch: An Order Pipeline*.

Using the contents of the customer's shopping cart, we create an `Order` object from *Chapter 3*. This order will record all of the `Products` and quantities the customer is ordering. We save this prior to rendering the final checkout view from which the customer will navigate to the payment processor to complete payment.

We will implement this as a new method on our `CheckoutView` class called `save_order`. The `save_order` method is listed below:

```
def save_order(self):
    cart = self.get_shopping_cart()
    make_order_from_cart(order, customer=self.request.user)
```

`save_order` creates an order by passing the current shopping cart to the `make_order_from_cart` function. A new order object is created every time the customer submits their cart for checkout, which leads to a common problem with shopping cart services: abandonment. A shopping cart is abandoned when a customer adds items, but never completes the checkout process.

The design above will save these abandoned carts as Order objects with status set to `'NEW'`. We will address this abandonment issue in the next chapter, but saving these as Orders is not necessarily a bad thing. For example, it can be useful business knowledge to record what products are almost purchased, but given up at the last minute. There may be other analysis we can do as well. Maybe at peak traffic times, the server is responding too slowly and more shopping carts get abandoned out of user frustration. This would help make a business case for a faster server or other upgrades.

The `make_order_from_cart` function has been added to our shopping cart module from *Chapter 3*. Its code is listed below:

```
from coleman.orders.models import StatusCode

def get_status_code(name):
    code, created = StatusCode.objects.get_or_create(short_name=name)
    return code

def make_order_from_cart(cart, customer=None):
    '''
    Takes a shopping cart object and generates a new order object for
    the provided customer. If no customer is provided, an "anonymous"
    order is created.
    '''
    order = Order.objects.create(customer=customer,
    status_code=get_status_code('NEW'))
    for item in cart:
        total_price = item.product.get_price() * item.quantity
        ordered_item = ProductInOrder(
               order=order,
```

```
                product=item.product,
                unit_price=item.product.get_price(),
                total_price=total_price,
                quantity=item.quantity)
            ordered_item.save()
        return order
```

We start by creating the `Order` object, then looping over each item in the shopping cart. Remember that the shopping cart stores `Item` objects, not `Products` directly. This helps us manage quantities and other information. We can access the actual Product object via the Item's product attribute. We do this to create our `ProductInOrder` intermediary model, which creates the list of `Products` in our `Order`. Refer to the design of this model in the previous chapter.

The `save_order` method is called from our `return_response`, which renders the final checkout view to the user. With our `Order` saved, we can begin building specialized checkout procedures for specific payment processors.

A Google Checkout class

The Checkout API is capable of several checkout patterns. In *Chapter 2, Setting Up Shop in 30 Minutes*, we implemented the simplest pattern involving a single "Buy It Now" button. In *Chapter 3* we implemented a custom shopping-cart solution. In the previous section we built a generic approach to implement checkout functionality for any service. We will now implement our Google Checkout shopping cart solution using this generic framework. Later in this chapter we will also implement another checkout service, Amazon's Flexible Payment System, on top of our generic framework.

To build our Google Checkout class, we inherit all of the functionality from our generic checkout view and add some additional data and methods. We will store our Google API and merchant key as class attributes as well as the shopping cart XML template name. Our XML template will be treated like the checkout HTML template, which will give us the flexibility to use different XML shopping cart templates for the same website if needed.

We'll also implement a shopping cart conversion method that converts our site's `Cart` model into a Google Checkout cart using the XML template. This is the same conversion function we used in the previous chapter to render the XML template and calculate a base-64 signature. The conversion method will update our class's `extra_context` data member to automatically include the Google cart when the view is rendered.

This system will be represented by a subclass called `GoogleCheckoutView`. The code listing for this class is below:

```
class GoogleCheckoutView(CheckoutView):
    google_cart_template = 'payments/googlecheckout.xml'
    merchant_key =    getattr(settings, 'GOOGLE_MERCHANT_KEY', '')
    merchant_id = getattr(settings, 'GOOGLE_MERCHANT_ID', '')

    def convert_shopping_cart(self):
        cart = self.get_shopping_cart()
        cart_cleartext = render_to_string(self.google_cart_template,
                                          {'cart': cart})
        cart_sig = hmac.new(self.merchant_key, cart_cleartext, sha).
digest()
        cart_base64 = base64.b64encode(cart_cleartext)
        sig_base64 = base64.b64encode(cart_sig)
        self.extra_context.update({'googleCart': cart_base64,
                                   'googleSig': sig_base64})

    def return_response(self):
        self.convert_shopping_cart()
        return super(CheckoutView, self).return_response()
```

The only new code here is the `return_response` method, which uses polymorphism to call our shopping cart conversion method before returning the view response as normal by calling the parent class's implementation. This method has been modified slightly from our original version earlier in the book. Instead of returning the cart and signature data, it adds them to the `extra_context` attribute. This attribute is automatically added to the template context when the view is rendered from `return_response`.

Much more functionality is available in the XML version of the Checkout API, but if your implementation requires support for the HTML Checkout API version, we can implement another view to support this very easily. The HTML API uses a similar approach to the XML, but is not cryptographically signed and submits a full HTML form with hidden input fields for all products in the cart.

An HTML version of the `GoogleCheckoutView` class is as follows:

```
class GoogleCheckoutHTMLView(CheckoutView):
    google_cart_template = 'payments/googlecheckout.html'
    merchant_key = getattr(settings, 'GOOGLE_MERCHANT_KEY', None)
    merchant_id = getattr(settings, 'GOOGLE_MERCHANT_ID', None)
```

```
    def convert_shopping_cart(self):
        cart = self.get_shopping_cart()
        checkout_form = render_to_string(self.google_cart_template,
                                         {'cart': cart})
        self.extra_context.update({'checkout_form': checkout_form})

     def return_response(self):
          self.convert_shopping_cart()
          return super(CheckoutView, self).return_response()
```

In this case, we render a special HTML template called `google_cart_template`, just as we did in the XML version. We store the resulting HTML fragment string in the `extra_context`. The HTML form in this fragment will contain all hidden fields and the submit button that features the Google logo. An example of this HTML template looks like this:

```
<form method="POST"
action="https://sandbox.google.com/checkout/api/checkout/v2/
checkoutForm/Merchant/REPLACE_WITH_YOUR_SANDBOX_MERCHANT_ID"
accept-charset="utf-8">
    {% for item in cart %}
    {% with counter as forloop.counter %}
    <input type="hidden" name="item_name_{{ counter }}"
               value="{{ item.product.name }}"/>
    <input type="hidden" name="item_description_{{ counter }}"
               value="{{ item.product.description }}"/>
    <input type="hidden" name="item_price_{{ counter }}"
               value="{{ item.price_in_dollars }}"/>
    <input type="hidden" name="item_currency_{{ counter }}"
               value="USD"/>
    <input type="hidden" name="item_quantity_{{ counter }}"
               value="{{ item.quantity }}"/>
    <input type="hidden" name="item_merchant_id_{{ counter }}"
               value="PROD{{ item.product.id }}"/>
    {% endwith %}
    {% endfor %}
    <input type="hidden" name="_charset_" />

    <!-- Button code -->
    <input type="image"
      name="Google Checkout"
```

```
      alt="Fast checkout through Google"  src="http://sandbox.
google.com/checkout/buttons/checkout.gif?merchant_id=REPLACE_WITH_
YOUR_SANDBOX_MERCHANT_ID&w=180&h=46&style=white&variant=text&loc=e
n_US"
       height="46"
       width="180" />
</form>
```

We've just written two different implementations of the Google Checkout API in a very small amount of code by building on our generic checkout view framework. The Google Checkout API has numerous additional features that could be integrated to the above designs. Additional features are fully described in the Checkout API documentation.

An Amazon Checkout class

Amazon Web Services provides developers with another option for payment processing. Their Flexible Payments Service (FPS) is slightly more complex than the Google Checkout API, but also offers some additional robustness. It uses a significantly different technical approach, however, and will require some adjustments to our payment processor designs.

FPS allows developers to create more advanced payment mechanisms, as well. For example, you can use FPS to set up recurring payments, such as subscriptions, as well as single-click payments whose parameters will be remembered over time. Google Checkout API offers similar advanced functionality. These features of both services will not be implemented here, but are well documented on the Web. Later in this book we will discuss using Amazon's payment services for digital goods, such as music and software.

The FPS uses a similar interaction flow to Google Checkout. We present our customer with the full contents of their shopping cart, show them a **Pay with Amazon** checkout button, which when clicked directs them to a co-branded Amazon payments page. After their payment is complete, they are redirected to our e-commerce website and we are given a payment token. This payment token can be used to charge the customer right away or saved for later, after an order is fulfilled, for example.

The payment token portion is the main feature that differs between Checkout and FPS. The Google Checkout API does not charge the customer right away, either. But instead of passing back a token, it creates a record in the vendor's checkout order dashboard. It may also notify an e-mail address. The vendor is required to log in to Checkout and charge the order.

With Amazon, there is no dashboard. This means we have to provide a callback URL to the Amazon service, which handles the return of the customer and stores their token for later processing. A token just means the customer has been authorized for payment, but has not yet been charged. We submit the token separately, when we're ready to charge the order.

As a result of these API differences, processing FPS payments requires additional steps. We will build a `CheckoutView` subclass as before, but customize it for the Amazon service. We will also need a view to handle the callback that FPS makes after authorizing the customer's payment. This callback will extract the token, which the FPS service adds as part of URI parameters. We will need to save this token and associate it with the specific order we're handling in order to charge the customer.

Let's tackle the `CheckoutView` subclass first. The `AmazonCheckoutView` class will include a `convert_shopping_cart` method as we saw earlier, but instead of converting to an XML format, it constructs a standard Internet URI. We will use this URI as the action attribute on an HTML form that is rendered to the customer. This form will act as the final checkout button, just like the Google Checkout API created. Our Amazon checkout button will use the Amazon branded **Pay with Amazon** image as a submit button.

When the user submits this form, the URI we generate will be submitted as an HTTP POST. This, kicks off the Amazon authorization step; the user can sign in or create an Amazon account and pay with the usual payment methods that Amazon supports. When payment authorization completes, Amazon will redirect the user to the callback URL mentioned earlier. This page will do two things: first, it presents a thank you message or other confirmation and encourages the customer to continue browsing our e-commerce site.

Second, when our Django backend handles the request, it will extract the token information Amazon adds to our URL, and saves it to the database in a model that associates this customer's order with the token. Later on, when we fulfill the order, for example, this token will be submitted to Amazon to actually charge the customer's account. This view could also attempt to charge the customer right away. In the next chapter we will discuss how to optimize this process using an order pipeline and build some tools to simplify the procedure.

Let's examine the `convert_shopping_cart` and related methods of our `AmazonCheckoutView` class:

```
Class AmazonCheckoutView:
    default_aws_params = {'currencyCode': 'USD',
                          'paymentMethod': 'ABT,ACH,CC',
                          'version': '2009-01-09',
                          'pipelineName': 'SingleUse'}

    def make_signature(self, params):
        path = u'%s?' % self.endpoint
        keys = params.keys()
        keys.sort()

        sign_string = path
        for key in keys:
            sign_string += '%s=%s&' % (urlquote(k),
                                       urlquote(params[k]))
        sign_string = sign_string[:-1]
        sig = hmac.new(self.aws_secret_key, sign_string, sha).digest()
        return base64.b64encode(sig)

    …

    def convert_shopping_cart(self):
        cart = self.get_shopping_cart()
        site = Site.objects.get_current()
        params = dict(self.default_aws_params)
        caller_reference = self.create_reference()
        callback_path = reverse('amazon_callback',
                                kwargs={'reference': callerReference})
        callback_url = 'http://%s%s' % (site.domain, callback_path)
        params.update({'callerKey': self.aws_access_key,
                       'callerReference': caller_reference,
                       'paymentReason': 'Purchase from %s' %
                        site.name,
                       'returnURL': callback_url,
                       'transactionAmount': cart.total()})
        signature = self.make_signature(params)
        params.update({'signature': signature,
                       'signatureMethod': 'HmacSHA256',
                       'signatureVersion': '2'})
        urlstring = urllib.urlencode(params)
        aws_request = '%s?%s' % (self.endpoint, urlstring)
        self.extra_context.update({'requestUrl': aws_request})
```

We begin by retrieving the shopping cart for this request. We also need information about our site, specifically about the domain we're running on. This information is retrieved via a call to Django's sites framework as `Site.objects.get_current()`. With this information, we are able to construct the callback URL where we will have Amazon return the customer after they've authorized payment. This is stored in the `callback_url` variable.

Next we build the list of parameters to pass to Amazon FPS. This includes other information such as our Amazon Web Services access key, a `caller_reference` variable, a description of the purchase, and a total cost of the payment.

The `caller_reference` variable is simply a reference number for the order associated with this purchase. We can use this reference number to look up the corresponding Order object from our database. Notice how this avoids passing any information about the products the customer has ordered. Instead we pass this reference to our Order object, which contains all of the products in the order. Our callback URL can use this reference number to obtain the correct order and associate it with the returned Amazon token.

Once we've constructed this basic set of parameters, we need to generate a base-64 encoded digital signature, very similar to the signature we generated for Google Checkout. After computing this signature, we add it to the FPS parameters and `urlencode` the whole set of data. We now have full FPS URI and we add it to the view context so that our template can render the appropriate HTML form (the **Pay with Amazon** button).

One final note regarding the `caller_reference` parameter. In our simple design, we will use the primary key of the `Order` object for this request. This gives us a simple way to look up the `Order` on the callback. Other applications may have different requirements, depending on how you store order data. As a result, we've broken the retrieval of this reference number into a `create_reference` method. In our case it's very simple:

```
def create_reference(self):
    if self.order:
        return self.order.id
    raise AttributeError("No order exists for this view.")
```

It just returns the ID of the Order object in the view. If our needs changed later, we would only need to update this method to return the appropriate value. Remember, though, that we must be able to use this reference number to find our customer's order in the callback routine. The simplest approach is to use the primary key.

The Amazon Callback view

Now that we can generate a checkout button that contains the necessary information and send the customer to Amazon to authorize payment, we next need to implement the callback view. When we defined the `callback_path` in our `convert_shopping_cart` method, we included the reference number directly on the URL. The URL pattern was designed this way for simplicity, but the reference number could also be retrieved from GET parameters. Our URL pattern looks like this:

```
url(r'^amazon/(?P<reference>\d+)/$', 'amazon_callback',
     name='amazon_callback'))
```

The view code is implemented as a regular Django functional view, not a class-based view. The reuse potential of this view is low, so for simplicity a standard Django view seems appropriate. The view expects one argument, the reference value, from the URL. Inside the view, we look up the information Amazon has added to the request as HTTP GET parameters. These parameters include copies of the parameters we constructed in our `convert_shopping_cart` method. This is useful for debugging.

Two new values are also present in these parameters, both of which Amazon has generated. These are `tokenId` and `status`. The `tokenId` is the key piece, which we will save with our Order object so that we can charge the customer. Without a `tokenId`, we cannot get Amazon to send us payment, so it's a very important piece of data. The status simply reflects the status of the authorization request, whether it was denied or not. We will save this value as well. The full view code for this function is as shown:

```
def amazon_callback(request, reference,
                    template_name='payments/amazon_complete.html'):
    order = get_order_by_reference(reference)
    token = request.GET.get('tokenId')
    status = request.GET.get('status')

    # save the Amazon FPS token for this order to the database
    save_amazon_transaction(order, token, status)
    return render_to_response(template_name, {},
                              context_instance=RequestContext(request))
```

There are two helper methods, `get_order_by_reference` and `save_amazon_transaction`. The code for these functions is as follows:

```
def get_order_by_reference(reference):
     return Order.objects.get(id=reference)

def save_amazon_transaction(order, token, status):
    try:
        transaction = AmazonTransacion.objects.get(order=order)
    except AmazonTransaction.DoesNotExist:
        transaction = AmazonTransaction.objects.create(order=order)
    transaction.token = token
    transaction.status = status
    transaction.save()
```

These are simply using Django's ORM to retrieve and store the information we need. Breaking the view up with these helper functions simplifies the view code and improves readability.

Finally, the view renders a template whose default name is `payments/amazon_complete.html`. This is where we thank the customer for their order and encourage them to continue browsing. At this point we've successfully taken their order, recorded it in the database, and authorized the payment. The next step is to fulfill the order and actually charge the customer. We will discuss this "order pipeline" in the next chapter.

PayPal and other payment processors

The Web has a full ecosystem of payment processing services. We have detailed the use of two of the major players: Google Checkout and Amazon Flexible Payment Services. These two companies provide excellent solutions for payment processing. The primary benefit of building your application for these services is that they are extremely developer friendly. Libraries are available in many languages and their documentation is complete and well tested.

Most payment processors follow a similar design as the two we've discussed. One of the oldest services on the web, PayPal, offers a variety of payment services-everything from "Buy Now" style buttons to shopping cart integration.

We will not implement a PayPal payment processor here, but will discuss the general approach to building one. Unfortunately, the PayPal payment APIs are not particularly friendly for Python developers. They have not yet published official Python tools, though they do offer implementations for PHP, Java, Ruby, ASP, and ColdFusion. Some attempts to implement PayPal APIs as community Python projects do exist, but their success has been limited.

One of the difficulties with PayPal is that they offer so many different services and APIs that it's difficult to evaluate the right one for your e-commerce application. Further complicating matters, some of PayPal's more advanced tools require merchants to pay monthly service fees in addition to standard per transaction costs.

The PayPal service that is most similar to the payment processors we've built in this chapter is called **Express Checkout**. It works by the same pattern we've seen before: show the user their shopping cart with a checkout button, authorize payment at PayPal's site using a PayPal account, and return to our application for a confirmation page.

Instead of submitting a form directly to the payment processor, PayPal's Express Checkout service requires you to process a form in a back-end view. This view makes a call to PayPal's API and sends an HTTP Redirect to the customer's browser. The PayPal API call obtains a token, which you add to the redirection URL when you return the redirect response. This token is used to identify the order throughout the payment process, similar to Amazon's FPS, and it will be included when PayPal redirects to your site taking the user's payment.

When the customer returns to our application, we can make another call to the PayPal API with the returned token to obtain details about how the checkout went. We can then create or update our Order object or other Django models as needed.

Other PayPal services follow totally different patterns. PayPal offers advanced functionality, such as processing the entire payment through your web application. This method means the customer never leaves your site and makes you responsible for managing their data. Some organizations may prefer this approach, but it does present unique complications and potential for security disasters.

If we were to implement the entire payment processing directly on our site, we need to ensure that we handle all requests via HTTPS with an SSL certificate and that our web servers are tightly secured. An unsecure web server could open the potential for security risks because all of the processing is handled by our server. Even though we do not need to store credit card information or other payment details, there is always potential for our processing to be snooped or otherwise intercepted.

For small to medium sized organizations, the traditional objective is to look professional in the way they handle payments. Some may find that an offsite payment service such as Google Checkout, FPS, or PayPal Express Checkout appears amateurish. This is an unfortunate viewpoint, however, because using such services mitigates a substantial amount of risk from online payments. It also simplifies the design, development, and maintenance of payment solutions, which saves both time and money.

Summary

This chapter focused on extending our payment system from a simple set of views to a rudimentary payment framework that can be reused and extended as needed. Topics discussed include:

- Support for a full shopping cart experience on both Google Checkout and Amazon's Flexible Payment Services
- Extended our Google Checkout processor to use both XML and HTML APIs
- Use of class-based generic views in Django apps
- Discussed the implementation of PayPal
- Reviewed security issues related to on-site payment processing

In the next chapter we will begin to tie the various components we've built thus far into a user-friendly backend. We call this an "order pipeline" and the goal of it is to simplify the handling of orders, interacting with the customer, and arranging shipments.

5
From Payment to Porch: An Order Pipeline

So far we've built a variety of useful e-commerce tools. These include a product catalog, a customer information model, an order and shopping cart interface, and payment processors. These apps cover the customer interaction portion of the e-commerce selling process. In this chapter we will build a simple set of tools for handling the steps that come after we've received a customer's payment. This includes:

- Updating/assigning status information to our orders
- Using Google Checkout API's automatic payments system
- Calculating shipping and handling charges
- A simple CRM tool to allow customer feedback on orders

The process that happens after an order is completely submitted and paid for by the customer tends to be very specific to a company and the products they're selling. However, the outline above reflects typical areas of need in almost all e-commerce operations. As in the rest of this book, we will build a very simple set of tools with emphasis on highlighting particular Django techniques that can simplify or enhance the development process.

Adding status information to orders

When we built the `Order` model in *Chapter 3, Handling Customers and Their Orders*, we included a field and related model to manage the status of particular orders. This status was designed to be simple and lightweight, but capable of describing a variety of circumstances. Some simple example status messages could be:

- Awaiting Payment
- Payment Received
- Shipped
- Closed

When we initially constructed the simple checkout view in *Chapter 3*, we did not integrate our `Order` model. This was added in the last chapter as part of our more general purpose checkout and payment processors. We wrote the following method into our payment processor base class:

```
def save_order(self):
    cart = self.get_shopping_cart()
    self.order = make_order_from_cart(order, customer=self.request.
                                      user)
```

Here we convert the customer's shopping cart into an `Order` object. This happens prior to completing the checkout process with the payments provider (Google Checkout or Amazon). As a result, we will inevitably accumulate `Order` objects for shopping carts that have been abandoned. This happens when the customer adds items to their carts, proceeds to our checkout page, but does not take the final step to checkout with the payment processor.

The `make_order_from_cart` function was a module-level function added to our cart module. It is listed below along with the helper function `get_status_code`:

```
def make_order_from_cart(cart, customer=None):
    '''
    Takes a shopping cart object and generates a new order object
    for the provided customer. If no customer is provided, an
    "anonymous" order is created.
    '''
    order = Order.objects.create(customer=customer,
    status_code=get_status_code('NEW')) for item in cart:
    total_price = item.product.get_price() * item.quantity
    ordered_item = ProductInOrder(order=order,
                                  product=item.product,
                                  unit_price=item.product.get_price(),
                                  total_price=total_price,
                                  quantity=item.quantity)
    ordered_item.save()
    return order

def get_status_code(name):
    code, created = StatusCode.objects.get_or_create(short_name=name)
    return code
```

These two functions create a new `Order` object from our shopping cart class and, if necessary, constructs a new `StatusCode` object to represent the status of the order. In the case of `make_order_from_cart` we created a `StatusCode` with the short description of `'NEW'`. The `NEW` status code is designed to mark orders that have just come in and have not yet completed the payment portion of the checkout process.

Some system of clean-up would be necessary, either a simple cron job that runs periodically or a clean-up/delete script that could be run manually, to delete `Order`s whose status is `'NEW'` but that have been abandoned. Abandoned orders could be defined as those with a `'NEW'` status created more than two weeks ago. You would have a query like the following:

```
>>> twoweeks = datetime.today() - datetime.timedelta(days=14)
>>> old_orders = Order.objects.filter(status_code__short_name='NEW',
                                      date_placed__lte=twoweeks).
                                      delete()
```

Using the system we've built in this book, it is very safe to delete `NEWOrder`s after even a short period of time because every time a customer proceeded to checkout we would create a unique, new `Order`. This was a design decision to simplify the order taking process and prevent the need for cookies or sessions to manage a customer's order and shopping cart combination. Instead of attempting to update the same `Order` object when a customer makes changes, we simply create a new one.

This is the entry point for all orders in our system. When a customer does not abandon their order and proceeds to the payment service to enter their payment information, two things should happen. First, the payment service will notify us, the e-commerce vendor, which payment succeeded. This notification must include our systems unique `Orderid`, which we can provide as extra information that will be returned to us when we render the checkout request for the customer. Second, upon receipt of notice from the payment service, we should locate the `Order` that was paid for in our system and update its status. This will be a new status code of our choosing to reflect the fact that payment has been received and the order should enter our fulfillment process.

Depending on the payment service we're using, the notification of payment could arrive in several ways. For example, Google Checkout by default sends an e-mail notification to the administrative address we entered when we created our API account. This shows the order details including the extra information we include, like our own `Orderid`.

With this information, a manual solution for low-volume sites could be as simple as logging into the Django admin, locating the `Order`, and updating its status. We can automate this process, however; using Google Checkout's order processing interface we can build a sophisticated, automatic integration with our Django system.

The order processing aspect of the Checkout API uses a secure SSL callback, which is hosted on our e-commerce site. Google Checkout will notify this callback when an order payment is completed. This requires an SSL certificate and secure Apache installation; see the Apache documentation for more information on configuring this for your site.

This SSL channel delivers order notifications to our system in a secure, timely manner. We can respond, after charging an order, for example, through another set of SSL-secured endpoints on the Checkout API side. This in effect performs all of the operations that are available through the Checkout dashboard, but in a custom way that is integrated with our Django project. The Amazon Flexible Payment Service offers a similar callback approach, as do many other payment services.

Even relatively simple systems with low order volume could benefit from integrating order processing. Despite the extra work involved in developing a Django interface and configuring a secure web server, having all the order management happen in a single tool may be a productivity boost to staff managing orders. The extra work involved in implementation, however, could be significant and requires a skilled developer.

SSL and security considerations

We've already briefly mentioned SSL, but we should stop and emphasize its importance in securing e-commerce applications. **Secure Sockets Layer** (**SSL**) is the standard approach to encrypting web communications. Google Checkout requires the secure endpoints we discussed in the previous section because they will be transmitting potentially sensitive order information. Without using SSL, this information would be sent in clear text and could be intercepted or read by anyone positioned between Google and our servers.

With the growing use of public wireless access points, mobile devices, and other "un-trusted" connections to the Internet, securing communications between browser and server has become essential. But similar risks are involved with server-to-server communications and these connections also require protection. Fortunately, SSL is capable of handling both.

A certificate is required to implement SSL on your server. This certificate must be purchased and issued from a trusted authority. There are numerous companies selling these authenticated certificates on the Web; prices and packages vary widely amongst vendors. A purchased certificate will typically require renewal on an annual basis.

Once in possession of a certificate, implementation of a secure server in Apache requires enabling `mod_ssl` and creating additional configuration sections that use port 443 (instead of 80) and include the `SSLEngine on` directive. The certificate file should be installed in a secure location (usually readable only by the root account) and specified in the `SSLCertificateFile` and `SSLCertificateKeyFile` directives.

After these changes, our Django application can be configured as normal within the SSL configuration section. More sophisticated approaches exist to limit the secured views to a subset of all our site's views. For example, we could write a WSGI script that loads a special, secure settings file and points to a different `ROOT_URLCONF` that contains only our secure URL patterns and views.

Order processing overview

The order processing system begins immediately after a customer has placed their order. As mentioned, this is dependent on the payment processing service, but in the Google Checkout world the next step is authorization. During this phase, Google performs some anti-fraud checks on the customer's payment and then verifies that the payment method is good for the complete amount of the order.

If the customer's payment passes these safety checks, Google changes the payment status to **Ready to Charge**. This means the merchant can now log in to the Checkout service and charge the order. By default, this is a manual process for the merchant that requires logging into Google Checkout.

Once the merchant has charged the order, they must begin the fulfillment process, which includes picking the product from their shelf or warehouse, preparing it for shipping, packing, and sending. Once these steps are completed, the merchant can log in to Google Checkout and change the payment status to **Shipped**. These manual steps, charging and shipping, are what the order processing system seeks to automate, or at the very least, integrate into the tools that run our e-commerce websites.

There are two Google Checkout APIs to help with this task: the Notification API and the Order Processing API. The Notification service simply provides a means for Google Checkout to notify your application when new orders arrive, pass risk checks, or when the order is charged. This is a one-way API, your application cannot communicate with Checkout, it only receives updates.

The Order Processing API is used to initiate actions with the Checkout service. These actions include not only charging and shipping, but also refunds and cancellations. You can also use this API to manage the status of individual items in an order. This is useful for situations where items will be fulfilled or shipped separately. You can also use this API to add shipping tracking numbers to orders or partial orders, and allow your application to send messages to your buyers automatically.

These two APIs are very powerful and have myriad uses depending on the needs of your business. Not all features of the API must be implemented, so solutions can be tailored directly to the application needs.

Other payments services, such as Amazon FPS and PayPal, support a similar set of tools. For simplicity, our discussion here will cover Google Checkout, but the basic flow is similar to other vendors. The support for programmable APIs and callbacks varies from vendor to vendor, so some features of the Checkout API may not be available in FPS or vice-versa.

Notification API

There are two possible implementations of Google Checkout notifications. The first, called simply the Notification API, is a push-based solution. A data pull-based solution, called Notification History API, is also available. Developers can implement either or both, depending on their needs, but for our examples we will stick with the push-based Notification API.

To implement support for Notification API, we must only use Django views that are secured using SSL. In addition, each request that comes into our application via a secure view must be verified as an authenticated Google Checkout notification. This is accomplished using HTTP Basic Authentication, wherein the Authorization header includes a base64 encoded pairing of our Checkout Merchant ID and Merchant Key.

These ID and key values should remain secret, which is part of the necessity of using SSL for Notification API communications. All notification requests should be verified by our view. This is done by inspecting the authorization header, which we can do using a decorator function. The decorator appears below:

```
def verify_googlecheckout(view_func):
    '''
    Decode the Authorization header for Google Checkout Notification
    API requests and verify that it matches MERCHANT_ID:MERCHANT_KEY
    when base64 decoded.
    '''
    def _verified_view_func(request, *args, **kwargs):
        merchant_key = getattr(settings, 'GOOGLE_MERCHANT_KEY', None)
        merchant_id = getattr(settings, 'GOOGLE_MERCHANT_ID', None)
        b64string = request.META.get('Authorization', None)
        If b64string:
            cleartext = base64.b64decode(b64string)
            auth_id, auth_key = cleartext.split(":")
            if auth_id != merchant_id or auth_key != merchant_key:
                raise Http404
            return view_func(request, *args, **kwargs)
        else:
            raise Http404
    return _verified_view_func(request, *args, **kwargs)
```

The Notification API is configured in the Google Checkout Merchant Center by editing the Integration settings and entering an API callback URL. This is the primary URL where Notification information will be sent to us from Google Checkout.

When our view receives a notification it is sent using HTTP POST and the message is stored as XML in the request body. The contents of the POST are not the usual key-value combinations we're used to working with in Django views. As a result, the `request.POST QueryDict` in our view will not be usable for our purposes. Instead we will need to access the raw POST data. This is done through the `raw_post_data` attribute on the request object. This should contain a valid XML document that represents the Notification API message.

There are four kinds of notifications: new orders, risk information, order state change, and amount notifications. The usefulness of each of these is dependent on the application. For example, an e-commerce site that needs to handle lots of refunds and cancellations will be certain to implement support for amount notifications because they cover that type of information.

Our example in this chapter will use the new order notification data to update the order status in our Django model. We will ignore the other notification types, but the implementation for each of them will follow the same pattern. The specific actions required, however, are determined by the application or business need.

When a new order notification arrives, it includes several key pieces of information. This includes the shopping cart element, as it appeared in our Checkout XML submissions generated by the payment processors in *Chapters 3* and *Chapter 4, Building Payment Processors*. It also includes: the buyer's billing and shipping address, a buyer ID that is unique to Google Checkout, the order total, and order adjustment information that corresponds to the shipping method and whether any coupons or gift certificates were used.

For our purposes, the `<shopping-cart>` element is important here because we can use it to locate the Order object in our Django application's database. When we built the cart XML template in earlier chapters, we did not include the Order ID in our local database. We should modify our shopping cart XML to include this information in a special tag called `<merchant-private-data>`.

The `<merchant-private-data>` tag can contain any information we choose. We should structure this information as valid XML and choose a tag name that makes sense for parsing later on. Here is an example tag to add to the shopping cart XML for our payment processor:

```
<shopping-cart>
   ...
<merchant-private-data>
<django-order-id>{{ order.id }}</django-order-id>
</merchant-private-data>
</shopping-cart>
```

The `<merchant-private-data>` tag must be added as a subtag to the shopping cart in our XML file. It should contain private data for the order as a whole.

We could add merchant data to the shopping cart XML for specific items. This could tie the items a customer is ordering to Product model IDs in our Django database. The tag to use for this is `<merchant-private-item-data>` and it must be added as a subtag to a specific `<item>` tag in the shopping cart XML. This is not necessary for our example here, but it would look something like this:

```
<item>
    ...
<merchant-private-item-data>
<django-product-id>{{ product.id }}</django-product-id>
</merchant-private-item-data>
</item>
```

With this modification to our payment processor, we gain the ability to tie new order notifications to specific Order objects in our system. Let's examine the code that makes this possible. We'll start with the notification callback view:

```
@verify_googlecheckout
def notification_callback(request):
    '''
    A simple endpoint for Google Checkout's Notification API. This
    handles New Order notifications and ignores all other information.
    '''
    if request.method is not 'POST':
        raise Http404
    dom = parseString(request.raw_post_data)
    notify_type = get_notification_type(dom)
    if notify_type is 'new-order-notification':
        order_id = get_order_id(dom)
        order = Order.objects.get(id=order_id)
        set_status_code(order, 'NEW_CHKOUT')
        return HttpResponse('OK')
    raise Http404
```

Despite the short length of this snippet, there is a lot going on. First, we've wrapped the view in our verification decorator to make sure we have authentic Notification API requests. Next, we use the `raw_post_data` attribute on the `HttpRequest` object as input to an XML parser. This is using the `parseString` function from Python's `xml.dom.minidom` package. Finally, if this is a new order notification, we process it using a series of additional helper functions.

When we use `parseString`, we get a Python DOM object. This has a variety of methods for traversing the XML DOM tree. In our case, we'll stick to some very simple techniques. First, to determine the notification type, we simply look at the tag name for the whole XML document we receive. A New Order notification XML will look something like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<new-order-notification xmlns="http://checkout.google.com/schema/2"
serial-number="85f54628-538a-44fc-8605-ae62364f6c71">
<shopping-cart>
      ...
</shopping-cart>
</new-order-notification>
```

Whereas a risk information notification XML would look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<risk-information-notification xmlns="http://checkout.google.com/
schema/2"
serial-number="0b95f758-0332-45d5-aced-5da64c8fc5b9">
<risk-information>
      ...
</risk-information>
</risk-information-notification>
```

So by examining the root XML tag, we can determine the type of request. This is handled by a helper function that simply extracts the `tagName` of the root document element:

```
def get_notification_type(xmldoc):
    return xmldoc.documentElement.tagName
```

We traverse the DOM in a similar fashion to obtain the order ID information from our merchant-private-data tag.

```
def get_private_data(xmldoc, private_data_tag='merchant-private-
                     data'):
    docEl = xmldoc.documentElement
    return docEl.getElementsByTagName(private_data_tag)[0]

def get_order_id(xmldoc, order_tag='django-order-id'):
    private_data = get_private_data(xmldoc)
    order_id = private_data.getElementsByTagName(order_tag)[0]
    return order_id.nodeValue
```

The full set of DOM traversal methods are described in the Python `xml.dom` module documentation.

Once we obtain the order ID, we simply look it up in our Order model and change its status. You can use whatever status you like here, but the goal is to indicate that the order has been submitted and payment information was entered in the Checkout API system. Checkout is still working on verification for the payment at this point, so we do not want to go any further than this.

At some point the Notification API will let us know that the order payment has passed fraud and verification checks. This means the order is chargeable. We can handle these types of notifications as well by adding conditional clauses to our notification view. The full set of Notification API messages is documented at the following URL:

`http://code.google.com/apis/checkout/developer/Google_Checkout_XML_API_Notification_API.html`.

Order Processing API

Unlike the Notification API, Checkout's Order Processing API is not initiated by requests from the Checkout system. Instead it is available as a method of updating Checkout order information by making secure HTTP calls to the Checkout servers. It allows your application to issue three types of commands: financial, fulfillment, and archiving.

Financial commands are those that modify an order's financial status. This includes things such as charging an order, refunding all or part of an order, cancelling an order, and reauthorizing a customer's credit card (in cases where it failed initially).

Fulfillment commands include information about an order's fulfillment state, particularly with regard to shipping status. It can also include problems such as backorders and returned items.

Archiving commands are simply housekeeping tools to manage the list of orders displayed in the Google Checkout Merchant Center. This is useful for removing old orders that no longer need your attention. It can also unarchive orders that were previously complete, but now need to be reopened to make changes (for example when an item is returned).

All Order Processing API requests must be sent over SSL connections and use HTTP Basic authentication. The endpoints for Order Processing requests are documented in the Checkout API, but as of this writing they exist at the following URLs (for sandbox and live data, respectively):

`https://sandbox.google.com/checkout/api/checkout/v2/request/Merchant/MERCH_ID`.

`https://checkout.google.com/api/checkout/v2/request/Merchant/MERCH_ID`.

The messages you send to these endpoints are represented as XML documents in the format appropriate to the specific command you want to issue. The schemas for these XML documents are well documented and include many customizable tags. It is convenient to use Django's template system to define XML templates for the commands you wish to implement. A full implementation of all the possible Order Processing commands is a very significant undertaking. Here we will demonstrate just the basics of how to get started.

First, it is necessary to discuss how we will make these API calls using SSL and Basic authentication. Python's `urllib2` module provides convenient tools to accomplish this task. However, the Order Processing API requires the use of a client-side SSL certificate to secure the API transaction. For this purpose we'll need to build a special `urllib2` handler. There are various ways to achieve this, but the easiest is to subclass the default `urllib2HTTPSHandler`. An example of this approach is demonstrated on the Three Pillars Software blog at `http://www.threepillarsoftware.com/soap_client_auth`.

You will use the same SSL certificate for these requests that your secure web server uses to secure incoming connections. Securing your e-commerce application is a subject worthy of a book in itself, and it is highly recommended that you research this subject if you're planning to build a Checkout integration to this depth.

Using a custom handler we can create a URL opener like so:

```
>>> mykey = '/path/to/ssl_key_file'
>>> mycert = '/path/to/ssl_cert_file'
>>> opener = urllib2.build_opener(HTTPSClientAuthHandler(mykey,
                                  mycert))
>>> opener.add_handler(urllib2.HTTPBasicAuthHandler())
# add our HTTP Basic Authentication information...
>>> opener.add_password(user=settings.GOOGLE_MERCHANT_ID,
                        passwd=settings.GOOGLE_MERCHANT_KEY)
```

We can use this opener to send our requests to the Order Processing API. This would generally work by rendering an XML template specific to the command we want to issue, including context data pertaining to the order we want to modify, and then opening a request using our opener object.

```
# T and C are Django Template and Context objects
>>> xml_cmd_document = T.render(C)
>>> api_endpoint_url = \ 'https://checkout.google.com/api/checkout/v2/
request/Merchant/ID'
>>> response = opener.open(api_endpoint_url, xml_cmd_document)
```

As you can see, integrating your e-commerce application to this level with Google Checkout (or any payment system) is a very complex endeavor. It requires knowledge of secure HTTP transactions as well as the Order Processing API and XML. Google provides ample documentation for the entire process in their online documentation. If interested, we recommend starting with the XML API Developer's Guide for checkout processing:

`http://code.google.com/apis/checkout/developer/Google_Checkout_XML_API_Processing.html`.

One potential implementation strategy for these APIs is to use the web-based Merchant Center for charging and shipping orders, but implement a notification callback so that your application becomes aware of the order status as it moves through the fulfillment process. This way your web application can update the customer as to the status of their order, without any additional work on your part. Your e-commerce tools will know the status of any order because of the Notification API. This avoids the complexity involved in integrating charging and shipping features directly into your application, but still allows you to empower your customers.

Calculating shipping charges

Handling shipping and its related charges is one of the most difficult aspects of an e-commerce platform. In many countries there are a wide variety of shipping companies, each with their own list of services and fees. The issue is further complicated by the fact that shipping costs must be computed before accepting payment from a payment service.

If you ship only a few products with very similar characteristics, it can be relatively easy to manage. You'll often know exactly how much to charge for shipping and can hardcode those values into your payment processor (in the Checkout shopping cart XML, for example). Often this is unfeasible, though, because you sell so many products in such different sizes and packaging.

There are no one-size-fits-all solutions to these problems, but tools such as Django and Python can simplify your life if you struggle with generating accurate shipping charges. The large shipping companies such as FedEx, DHL, UPS, and even the United States Postal Service, now offer web services APIs to perform shipping calculations. We'll discuss each of these services briefly, and then demonstrate another option for shipping calculations, which is builtin to Google Checkout.

Notably, the United States Postal Service offers some of the most useful and sophisticated web-based APIs. Not only can you calculate shipping charges, but you can standardize and verify addresses, look up ZIP codes, and even schedule carrier pickups. All of these services are free and use a very simple XML API, similar to the Checkout APIs we discussed earlier. The services are well documented and easily accessed by developers. It's clear that the USPS has tried to build a very web-friendly toolset. The only downside is that it isn't useful for international developers.

FedEx has implemented a set of web services using the **Simple Object Access Protocol** (**SOAP**) and **Web Service Definition Language** (**WSDL**) standards. Their documentation is good and they provide code samples in several languages. However, you will be required to sign-up for a developer account before accessing many of the resources. There is also an excellent third-party Python library called **python-fedex** that is a light wrapper around the FedEx tools. It supports shipment creation, cancellations, and tracking by tracking number. It is available on Google Code: `http://code.google.com/p/python-fedex/`.

UPS offers a similar set of services as the previous two, but there is a significant amount of work to get a developer key and browse the documentation. Their API can do typical activities such as package tracking, rate and service calculations, address validation, and so on. In my experience, however, the UPS tools are not the easiest to use. This includes using their website, which can be a daunting experience for new, non-technical users attempting their first shipment.

Integrating any of these API services into our e-commerce Django application could follow several patterns. One method is to attempt to retrieve shipping calculations from our checkout view. This is an acceptable solution for many applications, though you are relying on the shipper's web service to return relatively quickly, otherwise the rendering time on your checkout view will be extended (this can be mitigated somewhat with a short timeout and default values). Another alternative is to use AJAX to retrieve shipping values from the appropriate web service, and dynamically update the HTML checkout form.

Tracking information is decidedly easier. In the simplest case, we could attach a tracking field to our `Order` model. This would have to be populated manually, unless we went the full step of integrating our label making/shipment printing tool with our web service. This would be possible using FedEx or UPS, but is a particularly advanced feature that may require significant development time.

The main alternative to a custom implementation of shipping services is to let your payment processor handle it. Google Checkout does this very effectively. When we created our shopping cart XML in earlier chapters, we created as simple a document as possible. There are many additional features, however, one of which is carrier-calculated shipping.

The carrier-calculated shipping service in the Checkout API allows you to specify one or more shipping carriers that the customer can choose from. The checkout process queries the carrier's appropriate API automatically and retrieves shipping information. For this to work, however, it is required that you provide a weight for each item. This could be done using the `ProductDetail` and `ProductAttribute` models we built in *Chapter 2*.

This feature currently supports UPS, FedEx, and USPS only. There are several optional features, including adding fixed handling charges or automatically increasing/decreasing shipping calculations by a certain percentage.

To implement carrier-calculated shipping, we must add several tags to our shopping cart XML file from previous chapters. The most important thing is that each `<item>` block must now contain an `<item-weight>` tag whose attributes specify the item's actual weight. The following XML fragment adds a weight of 5.5 pounds to the listed item:

```
<item>
<item-name>Fresh Cranberries</item-name>
  ...
<item-weight unit="LB" value="5.5"/>
</item>
```

After adding weights to all our ordered items, we can create the carrier-calculated shipping rules. These are added to the `<checkout-flow-support>` tag of our XML document and look like the following:

```
<checkout-flow-support>
<merchant-checkout-flow-support>
<shipping-methods>
<carrier-calculated-shipping>
<carrier-calculated-shipping-options>
<carrier-calculated-shipping-option>
<price currency="USD">10.00</price>
<shipping-company>FedEx</shipping-company>
<carrier-pickup>REGULAR_PICKUP</carrier-pickup>
<shipping-type>Priority Overnight</shipping-type>
</carrier-calculated-shipping-option>
</carrier-calculated-shipping-options>

<shipping-packages>
<shipping-package>
<height unit="IN" value="6"/>
<length unit="IN" value="24"/>
<width unit="IN" value="15"/>
<ship-from id="BOS">
<city>Boston</city>
<region>MA</region>
<country-code>US</country-code>
<postal-code>02110</postal-code>
</ship-from>
</shipping-package>
</shipping-packages>
</carrier-calculated-shipping>
</shipping-methods>
      ...
</checkout-flow-support>
```

Note that the `<price>` tag in the `<carrier-calculated-shipping-options>` section is the default price for shipping using this method. This is required for cases where Google Checkout is unable to reach the shipper's web service or when another technical error prevents automatic calculation.

The `<carrier-calculated-shipping>` is composed of two parts: the shipping options and the shipping packages. The former is where you can define each vendor (FedEx, UPS, or USPS), their service and their pickup option. The shipping packages section is where you define the package size (optional) and the location you will be shipping from. These details are used by Google Checkout's system to present the customer with a total for shipping charges when they enter payment.

Checkout API also supports flat-rate shipping, which you can combine with the carrier-calculated results to provide customers with even more choices. This is done using a `<flat-rate-shipping>` tag added to the `<shipping-methods>` section of the checkout flow support. An example follows:

```
<flat-rate-shipping name="USPS Priority Mail>
<price currency="USD">4.00</price>
</flat-rate-shipping>
<flat-rate-shipping name="Free Shipping>
<price currency="USD">0.00</price>
</flat-rate-shipping>
```

You can learn more about carrier-calculated shipping at the following documentation URL:

`http://code.google.com/apis/checkout/developer/Google_Checkout_XML_API_Carrier_Calculated_Shipping.html`.

A simple CRM tool

Django makes it relatively easy to combine the information gathered from the order and shipment process into a simple Customer Relationship Management (CRM) tool. We can simply wrap a generic view to display a list of the logged-in user's orders.

```
@login_required
def order_list(request, *args, **kwargs):
    queryset = Order.objects.filter(customer=request.user)
    return list_detail.object_list(request, queryset, *args, **kwargs)
```

This uses the standard Django `object_list` generic view we've seen from earlier chapters. A detail view on a specific Order object is equally as simple. We will wrap the `list_detail.object_detail` generic view to ensure that only the current user's `Order`s can be inspected:

```
@login_required
def order_detail(request, *args, **kwargs):
    queryset = Order.objects.filter(customer=request.user)
    return list_detail.object_detail(request, queryset, *args,
                                     **kwargs)
```

At first glance these wrapper views seem superfluous, but they are necessary to ensure that the user who is logged-in can see only their own orders and no others. The Django generic view framework makes this easy, of course, by allowing us to pass the same filtered queryset in both instances. This requires wrapping in our own view code and cannot be defined in the URL pattern, because the request object is unavailable. We must have access to request to obtain the logged-in user.

With the list and detail views in place, we can write simple templates, as we did for our `Product` catalog. Because only site administrators and the customer can access the details for their order, we could use Django's comments framework, again, to implement a very simple customer feedback feature. Customer feedback is a primary concern of CRM tools.

The comments framework may lack a complete set of functionality for serious CRM applications, but it provides a simple, reasonably secure channel of feedback on a per-order basis. It could also be easily extended or a substitute could be put in place using the same "pluggable app" design that Django's comments exemplifies.

Other payment services

This chapter has focused on Google Checkout as a payment service provider, but numerous other providers exist. Amazon offers a Checkout service as well as a Flexible Payments Service. We will detail the FPS technology later, in the chapter on selling digital goods.

The Amazon Checkout service functions very similarly to Google Checkout. For example, constructing a shopping cart and submitting it to the Amazon Checkout service involves creating an XML document, rendering it with our shopping cart content information, signing, and submitting it to the appropriate `HTTP POST` URL. The views and utility functions written in this chapter could easily be adapted for Amazon's service. In some cases the only difference will be in the format and tags used for the XML file.

PayPal is another popular payment vendor. Their payment workflow tends to be different than Google or Amazon, but this depends entirely on the API you are writing against. As of this writing, PayPal offers a dozen or so payment APIs, each with different objectives or target applications.

The most similar PayPal API to what we've discussed in this chapter is their Express Checkout. The simplest integration with this API does not involve transmitting the shopping cart contents. An order details feature is available for use with Express Checkout that provides a similar user experience to the checkout tools we've already discussed. However, PayPal APIs do not rely on an XML document for transmission of this information; they instead use a Name-Value Pair (NVP) interface. These are just encoded `HTTP GET` parameters added to special endpoint URLs. Alternatively, PayPal also offers a SOAP interface.

Summary

This chapter has explored some advanced e-commerce features related to pipe-lining the order, payment, and shipping process. This sort of integration is traditionally very complex and expensive. Prior to web frameworks like Django it would often involve a team of developers and numerous home-grown modules and tools. Using a modern web framework combined with innovative third-party APIs provided by Google Checkout and shipping companies, it now becomes feasible for a small development team to implement on a reasonable schedule. These features include:

- Tracking an order's status as it moves through our system
- Retrieving information from the payment processor to automatically update order information
- Shipping services APIs for calculating delivery costs
- Integrating carrier-calculated shipping costs with our Google Checkout shopping cart
- A simple pair of views for receiving customer feedback

These are relatively advanced features that may not be necessary for all e-commerce sites. However, for medium-to-large sized organizations that struggle to find an off-the-shelf e-commerce solution that fits their needs, this sort of integration is easily within reach and could be customized for any industry or product. This is part of what makes Django and other frameworks exciting, if unexplored, territory for e-commerce businesses.

6
Searching the Product Catalog

Search is an extremely powerful and important addition to any e-commerce website. A lousy search engine means the customer will not find what they want, will not make a purchase, and will probably go someplace else. Far too many websites fail to provide their users with an adequate search engine. It's not clear, exactly, why this has been so, but it could be partly due to the difficulty of implementing search technology.

Not only is Django easy to integrate with open source search engine software, but there are dozens of community projects that add sophisticated search functionality to any Django app with a minimum of effort. This chapter will explore Django and search, including:

- The naïve, but simple, search strategy
- Simple MySQL-based index searches
- An overview of open source search engine tools
- Configuring the Sphinx, Whoosh, and Xapian engines
- Using the Haystack Django search module

Stupidly simple search

By far the easiest search functionality in Django is to simply use the ORM filter method and a `contains` clause. We'll call this the naïve approach to search, but it can be useful for quick, relatively simple search needs or when we want to specifically search an individual field for an exactly matching query term.

The `contains` clause makes use of the SQL `LIKE` statement, which performs a case-sensitive match against the table column. Django also supports the `icontains` clause, which performs the same function but uses SQL's `ILIKE` statement to perform case-insensitive matches.

A naive filter-based search can be performed like so:

```
>>> results = Product.objects.filter(name__icontains='cranberry
                                     sauce')
```

Note that the `icontains` lookup requires the search term to exactly match the field contents. Thus in the previous example any `Product` object with a name field containing just `'cranberry'` or just `'sauce'` will not be included in the results. Only exact matches for `'cranberry sauce'` are matched.

We can affect a more keyword-like approach by splitting our search term on whitespace and passing it to a Django in clause:

```
>>> terms = 'cranberry sauce'.split()
>>> results = Product.objects.filter(name__in=terms)
```

Here the results `QuerySet` will include any `Product` object whose name exactly matches on of the terms `'cranberry'` or `'sauce'`. Now we've created the opposite problem, namely that a `Product` whose name field is `'cranberry sauce'` (the exact term used for searching) will not be found in the results `QuerySet`. There are many similar hacks one could attempt, but they would only marginally increase our results.

Even if these methods returned better results, they still lack any kind of ordering. The `order_by` ORM method can be used to order on fields, but this is completely disconnected from our search. There is no concept of relevance when using the ORM filter method. This is as it should be because most of the time this functionality is not desired. The ORM methods are designed to be quick methods of retrieving database objects.

As you can see, Django's built-in ORM is not designed for the search problem. They are of very limited use for the kinds of search functionality to which users have grown accustomed in web applications, especially in the e-commerce space. Amazon.com's extremely sophisticated search functionality is the standard many users now expect.

To achieve better search results, we must move on to more sophisticated tools. Django has very limited support for searching builtin to the framework, but in the next section we will examine the primary method that is included. It's not fully automated, however, and will require us to directly manipulate our database tables to get up and running.

MySQL simple index searches

If you are using the MySQL database system and MyISAM tables, a very simple Django search interface exists automatically. This is a feature builtin to the Django ORM layer that allows you to perform boolean full-text searches directly from a `filter()` method call on any `QuerySet`.

The only caveat to this is that you must build a full text index on the columns you want to search. This is a one-time step, but it's by far the easiest way to get search up and running on your Django application.

The ORM search syntax looks like this:

```
>>> results = Product.objects.filter(name__search='+cranberry -sauce')
```

Note the use of `+` and `-` characters, which act as operators to explicitly define the search criteria. This is a boolean search, which is very different from what you might expect based on the use of major search engines such as Google or Bing. Boolean search looks for the presence or absence of the terms provided in the simplest way possible. If the term is prefixed with a `+` symbol, it is explicitly included, while a `-` symbol excludes the term outright.

The order of the results of this sort of search is not sorted by relevance as you might expect. In fact, when using this built-in ORM search method, the results do not include any relevance score. This means it's not possible to use `order_by()` on the results `QuerySet` to sort by relevance.

The above search query `'+cranberry -sauce'` will return a `Product QuerySet` for objects whose name field contains the word cranberry, but does not contain the word sauce. If we were to modify this query to be `'cranberry -sauce'` the results would be slightly different. This query means `'cranberry'` is optional, so Product objects with this term in their name will also be included in the results. Anything containing the term 'sauce' will be excluded.

Under the hood, this search method is performing a `MATCH() AGAINST()` SQL function. The above query search will translate to SQL that looks something like this:

```
SELECT ... WHERE MATCH(name) AGAINST (+cranberry -sauce IN BOOLEAN
MODE);
```

This SQL is currently hard-coded into Django and cannot be changed. However, you are free to write your own `MATCH() AGAINST()` routines and add them to your Django managers. For example, the following Django ORM method call should produce equivalent SQL as the above:

```
Product.objects.extra(where=Product.objects.extra(where=
    ['MATCH(name) AGAINST(+cranberry -sauce)IN BOOLEAN MODE'])
```

This could be extended to a manager method that adds the relevance score, like so:

```
def search(self, query):
     exp = 'MATCH(name) AGAINST(%%s IN BOOLEAN MODE)'
     return self.extra(where=[exp], params=[query],
                               select={'relevance': exp})
```

In order to support any of these search methods, we first must create a full-text index on our table's columns. In this case, the name column is the only one we're searching. The index is created by issuing the following SQL statement directly to MySQL:

```
CREATE FULLTEXT INDEX name_idx ON products_product (name);
```

We only need to create the index once and MySQL will subsequently index new data as rows are added to the products table.

Even with relevance scores added, boolean searches are very limited in their usefulness. They often give good results when searching a single, short field. But imagine the results of the boolean operation for fields with very large amounts of text, such as a product description or blog post body. Boolean search tends to return many false positives for text-rich data.

MySQL also supports natural language searching on columns with full-text indexes. The syntax is very similar to boolean searching, but is not builtin to Django's ORM syntax. But we can perform a natural language search by slightly modifying our manage method from above:

```
def search(self, query):
    exp = 'MATCH(name) AGAINST(%%s IN NATURAL LANGUAGE MODE)'
    return self.extra(where=[exp], params=[query],
                              select={'relevance': exp})
```

We can translate this to a more complete search manager class by adding a constructor method that stores a list of fields to include in our full-text search:

```
class SearchManager(models.Manager):
    def __init__(self, fields):
        models.Manager.__init__(self)
        self.fields = fields
```

```
    def search(self, query):
        meta = self.model._meta
        db_columns = ['%s.%s' % (meta.db_table,
                                 meta.get_field(name).column
                                 for name in self.fields]
        columns = ','.join(db_columns)

        exp = 'MATCH(%s) AGAINST (%%s IN NATURAL LANGUAGE MODE)' % \
                                  columns
        return self.extra(where=[exp], params=[query],
                          select={'relevance': exp})
```

We can add this search manager to any of our models and specify what model attribute fields we would like to perform the full-text search over. These fields all need to have MySQL `FULLTEXT` indexes created and should generally store text data.

```
class Product(models.Model):
    ...

    search = SearchManager(['name', 'description'])
```

MySQL's built-in natural language search is an improvement over our boolean mode search, but as far as search engine techniques goes, it's still relatively simple. It doesn't support features such as word stemming or phrase searching. And as mentioned earlier, full-text indexes in MySQL are only supported for MyISAM database tables and only work for `CHAR`, `VARCHAR`, and `TEXT` column types.

Another limitation of both the MySQL natural language search and boolean mode search is that it can only search on one model per search. More sophisticated search solutions would allow us to search across many different model classes in one query.

In the next sections we will explore some open source search engine solutions and how to integrate them with Django for more advanced functionality.

Search engines overview

There has been an explosion of search engine technology available in the open source community in recent years. Many of these have grown out of various academic and commercial projects and most are extremely high quality. We'll discuss a handful of these tools in this section and go on to integrate two of them with Django later in this chapter.

Sphinx

The **Sphinx** full-text search engine is a free and open source search engine product that has the added benefit of official, paid-support packages. It is available at `http://www.sphinxsearch.com/`. Sphinx is written in C++ and is available for most UNIX platforms, including Mac OS X. A Windows version is also available, but is officially not recommended for production use. It is fast and has good relevance.

Sphinx includes a search daemon that can be run as a background process on any system. Sphinx also includes API libraries for several popular programming languages, including Python. Sphinx requires you to define your search indexes using a configuration file. This file is read by the included indexer utility, which produces the full-text indexes. This indexer must be run on a timely basis to update the index for your database tables.

For use with Django, Sphinx is a good choice due to David Cramer's excellent django-sphinx utility. This is a wrapper around the Sphinx API that allows Sphinx-specified functionality to be attached to Django models. The django-sphinx utility is available from: `http://github.com/dcramer/django-sphinx`.

Unlike some of the other Django community projects for search, the django-sphinx application doesn't do any index definition or other search engine tweaking from Django itself. The search engine and indexes are configured entirely via the Sphinx configuration file. This is a matter of preference, as some of the other Django tools allow you to define indexes directly in Python.

Solr

Solr is a search engine built on top of the Apache Lucene search library and it is available at `http://lucene.apache.org/solr/`. Solr is written in Java and includes many advanced features. It requires a Java Servlet Container within which to run. This makes it a popular choice for enterprise applications where Java and Servlets are very common. To run a standalone Solr server, you'll need a container like Apache's Tomcat or the Jetty application server.

Solr support can be achieved in Django using the Haystack application, which we will discuss later in this chapter. Additional Solr support is also available from the **django-solr-search** project (also called **solango**). If you are already using Solr as a search platform internally, these solutions make it easy to attach support to any Django applications you may build. Django-solr is at: `http://code.google.com/p/django-solr-search/`.

One nice feature of the Django integration is that you can define the search parameters for your Django models and have solango generate the necessary Solr XML schemas automatically. Solango also supports more advanced Solr features like faceting and highlighting.

Whoosh

Whoosh is a search engine written entirely in Python. As a result it requires no compilation, making it relatively simple to install and maintain. It's an excellent solution for Python projects that need to add search functionality and don't already have a search engine in place. It features "fast indexing and retrieval" but the performance is likely less than you would see on some of the other search engines we've discussed, which are written in C++ or Java. Whoosh is available at: `http://whoosh.ca/`.

Whoosh is a very compelling tool if you're interested in a pure-Python solution or don't want to bother with the more complicated setups required with other tools. It is relatively full-featured, and includes much of the advanced functionality you'll find in Sphinx, Solr, or Xapian. We will work with Whoosh and Haystack in the coming pages.

It should be noted that at the time of this writing, Whoosh is probably not a practical choice for anything but very small-scale or testing situations. Besides potential performance problems, Whoosh has several technical issues that have rendered it unusable on many Django projects.

Xapian

The **Xapian** project is another open source, C++ search engine library. It is available at `http://xapian.org/`. It has bindings for many languages, including Python. It is a very powerful search engine and has an excellent community project for Django called **djapian**, available at `http://code.google.com/p/djapian/`.

The Xapian search engine library is a little different than the other tools we've covered. Unlike Solr or Sphinx, for example, there is no application to run under Xapian, it is simply a library. Djapian does a nice job of wrapping this functionality and storing the generated indexes in our Django database. It even includes Django management commands to perform index builds and other operations from the command line.

Haystack

Haystack is an ambitious Django project created by Daniel Lindsley that exposes a common interface to Django for a variety of pluggable search engine backends. These include Whoosh, Solr, and Xapian. This is an excellent tool for getting up and running quickly on any of these search platforms. The advantage of Haystack is that the interface for index definition and search queries remains the same, regardless of the engine being used on the backend. It is available at `http://haystacksearch.org/`.

Haystack also includes a lot of extra functionality, such as a set of predefined URL patterns, views, and forms. Not to mention niceties like Django management commands and template tags for "More like this" and term highlighting. Haystack is very Django-specific, however, and it is not recommended as a general-purpose search tool or in cases where heavy usage is expected.

Configuring the Sphinx search engine

The Sphinx full-text search engine was created by Andrew Aksyonoff and is available from `http://www.sphinxsearch.com`. It is a standalone search engine, meaning it is built and run as a background application and communication occurs directly from client applications, like our Django site. Installing on Windows is simply a matter of downloading and unzipping the pre-compiled binaries from the **Downloads** section of the homepage and configuring it as a Windows service. Linux and Mac OS X users will need to compile from source (see the Installation section of the Documentation for details).

Sphinx can index data from a variety of sources. In our examples, we will connect it to our Django database running on MySQL. It also supports the PostgreSQL database and can even index text files, including HTML and XML documents. For our purposes we will examine a simple MySQL configuration using our Django database.

Sphinx includes two important tools: the search daemon (`searchd`) and an indexing application (`indexer`). As Sphinx is a standalone application, the full-text indexes used in searching must be generated by the `indexer` tool. The indexes are not stored as part of your database, but as Sphinx-specific files. The location of these files, as well as all other indexer parameters, is specified in a configuration file, usually called `sphinx.conf`.

The `sphinx.conf` file is the heart of the search configuration. It generally specifies two things: sources and indexes. Sources correspond to the database queries we will build indexes against. Indexes include all the different parameters for indexing our data, including morphology, stop words, minimum word lengths, and so on. We will detail these settings shortly.

Defining the data source

We begin by defining our data source. Using our `Products` model from earlier in the book, we can define a source that includes the name and description fields for our products. Django automatically generated tables for our models, but we need to know which tables we want to tell Sphinx to index. The Django convention is to use application and model names for our tables, but this can be overridden. If you aren't sure what tables are used in your project you can always verify by running `django-admin.py sql all`.

In our case, the `Product` table should be `products_product`. The Sphinx configuration file has a very simple syntax. We start by defining a source and giving it a name. This name will be used for reference when we define indexes later. Here is a complete sources section for `products_product`:

```
source products_product
{
   type = mysql
   sql_host = localhost
   sql_user = root
   sql_pass = xxxxxx
   sql_db   = coleman_book

   sql_query = SELECT id, name, slug, description FROM products_
   product
}
```

The usual database parameters are included at the top. The important part is the `sql_query` statement. This is the query used to retrieve the data we want to index. It should include all of the fields we plan to search as well as the primary key value. Sphinx only supports primary keys that are positive integers, which is the default for Django models.

This source defines the ID, name, slug, and description columns in the `products_product` table as the source's data. You can write any SQL here that the database backend will support. This means we could include a `WHERE` clause to filter the source data arbitrarily. The results can also be filtered by application logic, however, so most of the time it's better to index larger data sets and let our Django application filter what it needs. For very large sets of data, however, it may be useful to segment sources this way.

There can be as many source definitions in the `sphinx.conf` as are needed for a project. These will generally differ based on the tables and fields that need indexing. For example, we could define a simpler source that just included the name field for an index that would exclusively search product names.

When we define indexes, we can index as many different sources as needed and all of the results will be combined into one large set. This is convenient, but complicated by the fact that all primary key values must be unique across all sources. If this is not true, searches will return unexpected results. This makes it difficult to combine Django models, for instance, because they all typically define a similar set of primary key values. To keep things simple, in our example usage, we will only index the single source defined above.

Defining the indexes

Index definitions are very similar to source definitions. We start by naming the index and then define its source and other properties, as below:

```
index products_product
{
    source  = products_product
    path    = /usr/local/sphinx/var/data/products_product
    morphology = stem_en
    stopwords = stopwords-en.txt
    min_word_len = 2
}
```

The index definition includes the source, defined earlier in the file, as well as a path to the filesystem location where Sphinx will store this index. Sphinx uses special flat index files to persist indexes. These are loaded from disk when the search daemon begins and must be reloaded as new documents appear in the sources and the index is updated. We will discuss more on updating indexes later in this section.

In addition to these mandatory statements, we have morphology, stop words, and `min_word_len` parameters. These affect the construction of the index in complicated but very useful ways. Unlike a simple MySQL `FULLTEXT` index, Sphinx supports these advanced pre-processing functions.

The `morphology` parameter allows Sphinx to apply "morphology preprocessors". This is a list of filters that uses natural language knowledge in attempt to generate more accurate search results. The `stem_en` preprocessor is a Sphinx built-in that applies an English language "stemmer." A stemmer normalizes words by indexing their root, instead of longer variations. For example, the word 'cranberries' would be indexed as 'cranberry' in an effort to provide accurate search results for either term. The stemming function is applied to indexed data and search terms.

The `stopwords` parameter specifies a text file that includes a list of "stop words." These are words that will be ignored by the indexer and usually include very common words in the indexed language. In our example, we're providing a list of words in English, such as 'the' or 'and'. The `stopwords` parameter allows you to specify multiple files separated by commas.

Finally, the `min_word_len` parameter instructs Sphinx not to index words shorter than two characters. This is an optional parameter (as are `morphology` and `stopwords`), but it is useful in tweaking the search results for some data sets.

There are numerous other indexer options, but these are some of the more powerful ones. The Sphinx documentation provides a complete list with thorough descriptions of their usages.

Building and testing our index

Now that we've defined our sources and indexes, we can build the first index of our data. This is accomplished by running the indexer tool included in the Sphinx package. This will generate the index files and store them in the location specified in our index section's path statement.

Once indexes have been created, we can test our search results in the command line using Sphinx's search tool:

```
$ search cranberry juice
Sphinx 0.9.8.1-release (r1533)
Copyright © 2001-2008, Andrew Aksyonoff

displaying matches:
1.     document=9628, weight=2
id=9628
name=CranStore Cranberry Juice
slug=cranstore-cranberry-juice
description=A refreshing Cranberry cocktail
...

words:
1.     cranberry: 3 documents, 11 hits
2.     'juice: 43 documents, 114 hits
```

This is a simple way to test your Sphinx indexes and to debug search results. It can be especially useful when trying to understand how Sphinx is using preprocessing filters, such as word stemming, to alter your search results. This command line utility is flexible and includes several flags for added power. For example, if you have multiple indexes, you can specify which index to run a test search against with the `-i` or `--index` parameter:

```
$ search—index products_product cranberry juice
```

When the configuration is to your liking, you can run the Sphinx search engine background process by issuing the `searchd` command. Under a Windows environment, this will usually happen automatically when you install Sphinx as a service. `Searchd` can be added to your system's initialize scripts to automatically execute when the system starts.

Searching Sphinx from Python

The Sphinx application includes a Python API interface that allows you to perform searches and return results in Python programs. All of the API commands are defined in the Sphinx documentation, but their specific usage within Python can vary.

We will take a quick tour of the Sphinx Python API and then discuss how it can be integrated into our Django models. First, a simple Python snippet that generates the interface to a Sphinx server running on our local machine:

```
SERVER = 'localhost'
PORT = 5555
client = sphinxapi.SphinxClient()
client.setServer(SERVER, PORT)
```

Now we have a client object that exposes the API listed in the Sphinx documentation. We can perform simple operations, like queries, using this client. To replicate the command line we used in the previous section, we could issue the following Python command:

```
results = client.Query('cranberry juice')
```

If we need to specify the index or indexes to use, we can pass them as a string containing the index names separated by commas as shown:

```
results = client.Query('cranberry juice', 'products_product, products_
                        other')
```

The results variable can be accessed like a dictionary to retrieve information about the search results:

```
total_hits = results['total']
matches = results['matches']
```

We can now access the columns for each of our results in the `matches` variable to load the primary key values:

```
keys = [r['attrs']['id'] for r in matches]
```

We could go on from here by using these primary key values to find a `QuerySet` in our Django model for this table. But this is quickly getting complicated and the performance of doing it this way is not good. Fortunately, the Django community includes an application for Sphinx that wraps all of this functionality into a simple tool. It is called **django-sphinx** and was written by David Cramer.

Simplifying searching with django-sphinx

You can download the **django-sphinx** application from its repository on GitHub by visiting the following URL and clicking on the **Download** button: `http://github.com/dcramer/django-sphinx/`.

It is also available through the `easy_install` tool by issuing the following command:

```
$ sudo easy_install django-sphinx
```

Once installed, you can add the django-sphinx layer to any model and take advantage of a very simple interface to search and retrieve Django model objects from your database tables. The only requirement is to create the source and indexes exactly as we did earlier and specify which index `django-sphinx` should search when you attach it to your models.

For example, to use django-sphinx on our `Product` model from *chapter 2*, we would change the model definition to include the `SphinxSearch` manager.

```
from djangosphinx.models import SphinxSearch

class Product(models.Model):
    category = models.ForeignKey('CatalogCategory',
                                 related_name='products')
    name = models.CharField(max_length=300)
    slug = models.SlugField(max_length=150)
    description = models.TextField()
```

```
    photo = models.ImageField(upload_to='product_photo', blank=True)
    manufacturer = models.CharField(max_length=300, blank=True)
    price_in_dollars = models.DecimalField(max_digits=6, decimal_
                                           places=2)
    price_in_euros = models.DecimalField(max_digits=6, decimal_
                                         places=2)
    price_in_pounds = models.DecimalField(max_digits=6, decimal_
                                          places=2)
    search = SphinxSearch('products_product')
```

The index we want to use is specified as the first parameter when we attach the manager object to our model. Multiple indexes are supported with the index keyword argument:

```
SphinxSearch(index='products_product products_other')
```

Our `Product` model now has a custom manager that we can use to query the Sphinx search engine and get back Django model objects for the corresponding results. This is done using the query method on the `SphinxSearch` manager:

```
products = Product.search.query("cranberry juice")
```

We can filter the products `QuerySet` just like we would anywhere else in Django:

```
expensive_matches = products.filter(price_in_dollars__gte=100)
```

The django-sphinx layer also injects the Sphinx result weights, allowing you to order the search results by their relevance weight:

```
products = Product.search.query('cranberry juice').order_by('@weight')
```

In the `SphinxSearch` manager definition on our `Product` model, django-sphinx will allow you to further define weights on your indexed fields. This way we can weigh the name field, for example, heavier than the slug or description fields.

```
search = SphinxSearch(index='products_product',
                      weights={
                      'name': 100,
                      'slug': 50,
                      'description: 80})
```

One last excellent feature of django-sphinx is the ability to generate the Sphinx configuration files automatically. The resulting `sphinx.conf` will likely require hand tuning, but it can help get you up and running quickly. To use this feature, add the `SphinxManager` to your models, as described above, and then use django-sphinx's `generate_config_for_model` helper method:

```
generate_config_for_model(Product)
```

The Whoosh search engine

Whoosh is a search engine written entirely in Python. It's slightly easier to install and run than the Sphinx search engine and some would argue it generally feels "more pythonic". To install Whoosh you can simply `easy_install Whoosh` or visit `http://whoosh.ca` to get the latest development version.

The fact that Whoosh is pure Python is very convenient for developers who are not interested in or lack knowledge of Java or compiling UNIX software. It can get you up and running quickly and it supports integration with Django, as we'll see shortly.

Just like in Sphinx, Whoosh needs to define and build a set of indexes on our data. The Whoosh documentation is very extensive (another advantage of a pure Python tool) and explains all the indexing options in great detail. We will present a quick tutorial here, before continuing on to using Whoosh with Django.

In Sphinx we defined our indexes using the index section of our `sphinx.conf`. In Whoosh, they use a `Schema` object, which is defined purely in Python (of course). The schema performs the same role as the index section: it defines the fields and additional options for the index we will build around our data. Unlike Sphinx, Whoosh requires the schema to include "type" information for each field we want to index. This type information affects how Whoosh indexes our data.

The Sphinx approach to indexing was to treat each field in our source definition as text whose contents would be chunked and preprocessed. Whoosh supports this as well, but it also provides alternative methods of handling fields. This is the idea behind field types. The `TEXT` type is used for fields of text, like Sphinx, but the `KEYWORD` and `ID` field types are treated differently.

Fields with the `KEYWORD` type are treated as lists of keywords or terms, separated by commas or spaces. This would be useful for indexing lists of tags. The `ID` type is used on fields that store single term values. The documentation suggests using this type for things like URLs and e-mail addresses.

For our `Product` model, the field types for the name and description would be `TEXT`. The slug field is probably best treated as `TEXT`, but could be considered as `ID` field (if you wanted to search based on slug). The corresponding Whoosh schema definition would like this:

```
from whoosh.fields import Schema, TEXT
from whoosh.analysis import StemmingAnalyzer

schema = Schema(name=TEXT, slug=TEXT, description=TEXT)
```

The Whoosh search engine stores indexes on the disk in files specified when you create an index. This is similar to the path statement in the index section of our Sphinx configuration. Remember, this is a pure Python implementation, so instead of using an indexer tool or other utility to generate our indexes, we have to write a Python script.

To do this we need a Whoosh index object, which defines the location of the index files as well as the schema to be used for indexing. Whoosh includes a convenient function we can use to create this object, called `create_index_in`.

```
from whoosh.index import create_index_in

index = create_index_in('indexes', schema)
```

This will create our index files in a directory called `indexes` and use the schema object we defined earlier. Likewise, when we're ready to load indexes we've previously created, Whoosh includes a convenient function for that too:

```
index = open_dir('indexes')
```

To actually index data, we have to obtain a Whoosh `IndexWriter` object and pass it our field data. The following code snippet will index all of our `Product` objects:

```
writer = index.writer()

for product in Product.objects.all():
    writer.add_document(name=product.name,
                        description=product.description)
writer.commit()
```

This indexes all of our `Product` objects as documents in the index defined earlier. The last line calls the `commit()` method of the `IndexWriter` object, which writes the full index to the disk in the location we specified with `create_index_in`.

If this feels like a very manual process, similar to the exploration of the Sphinx Python API from earlier, it's because it is. We're almost finished, however. Now that we've created an index for all of our `Product` objects, we can perform a search query by creating a Whoosh `Searcher` object from our index:

```
searcher = index.searcher()
```

And then, we search our index by calling the find method:

```
results = searcher.find("name", u"cranberry sauce")
```

Notice how this method requires us to specify the search field in addition to the search term. The results are returned as a dictionary object whose keys are the field names and corresponding values. The Whoosh API includes additional objects and methods for searches and queries, including a special query language. But the Whoosh engine alone doesn't help us add search to our Django models directly. For that we can use Haystack.

Haystack search for Django

Haystack is a general purpose search application for Django. It supports multiple search engine backends with a standardized integration. To install Haystack, use the download link from `http://github.com/toastdriven/django-haystack` and run `setup.py` to install it.

Haystack currently supports three search engine backends: Solr, Whoosh, and Xapian. These backends are specified to Haystack with the Django setting `HAYSTACK_SEARCH_ENGINE`. We will be using Haystack with the Whoosh search engine, so our settings file will need to include the following:

```
HAYSTACK_SEARCH_ENGINE='whoosh'
HAYSTACK_WHOOSH_PATH='/path/to/indexes'
HAYSTACK_SITECONF='project.search_sites'
```

As we discussed in the previous section, Whoosh stores indexes in files on the filesystem. When used with Haystack the location of these indexes is defined by the `HAYSTACK_WHOOSH_PATH` setting. Now that we've configured our search engine backend, we can move on to configuring the rest of Haystack.

Haystack uses a common Django design known as the registration pattern. This is the same approach as with Django's built-in admin application. It involves using a register method to attach functionality to your models. This allows code to be reused through subclassing and simplifies configuration. For more information, examine the `ModelAdmin` class in the `django.contrib.admin` module and the admin site documentation.

When you add haystack to your project setting's `INSTALLED_APPS`, it will trigger a search for a `search_indexes.py` file in each of your app modules. This file defines the indexes on your models.

In addition, Haystack needs a global configuration file, specified by the Django setting `HAYSTACK_SITECONF`. Typically this site conf file will be very simple:

```
import haystack

haystack.autodiscover()
```

Haystack requires us to specify our indexes, just as in Sphinx and Whoosh. Haystack uses a more Django-like syntax, however, and it will automatically translate and create our index definitions to the format appropriate for our chosen search engine backend. Just as our Django models are defined by subclassing `django.db.models.Model`, our indexes are defined by subclassing `haystack.indexes.SearchIndex`. These subclasses then specify the fields we want to index along with their type.

To index our `Product` model using Haystack, we would write the following `SearchIndex` subclass in our app's `search_indexes.py`:

```
from haystack import indexes
from haystack import site
from coleman.products.models import Product

class ProductIndex(index.SearchIndex):
    text = indexes.CharField(document=True, use_template=True)
    name = indexes.CharField(model_attr='name')
    slug = indexes.CharField(model_attr='slug')
    description = indexes.CharField(model_attr='description')

site.register(Product, ProductIndex)
```

Haystack's approach to search differs significantly from what we've seen before. The indexes we define will be fed into our search engine, but queries and results are all handled by the Haystack application logic, which translates everything automatically. Every Haystack `SearchIndex` requires one field to be marked `document=True`. This field should be the same name across all your indexes (for different models). It will be treated as the primary field for searching.

In our product index, this field is called text. It has an additional special option: `use_template=True`. This is a way of indexing the output of a simple template as a model's data, rather than just using a model attribute. The remaining fields will have the attribute specified by the `model_attr` keyword argument looked up and used as index data. This can be a normal object attribute, a callable, and can even go through relationships using the double-under syntax (`category__name__id`).

This index definition may seem awkward at first due to the extra field and `document=True`, but it is required in part to standardize an interface with multiple backends. Because the text field will be present in all indexes, our search engine knows for certain it can index this field for all our data. It's a little complex, but if you're interested in learning more the Haystack documentation is very complete.

One last thing about our index definition: the `use_template=True` argument. This argument lets us define a template for the data to be indexed in the text field. The template must be named according to our model and the indexed field, so in the `ProductIndex` example we would create a `product_text.txt` template that looks like this:

```
{{ object.name }}
{{ object.manufacturer }}
{{ object.description }}
```

This gives us additional flexibility in defining the data we want indexed. Think of this template as allowing you to construct a more formal "document" out of your Django models. Haystack will render this template for all `Product` objects it processes and pass the resulting string to the search engine to use as the indexed data. Standard template filters are available here and we can also call methods on our object to include their results in our index.

Haystack searches

Haystack includes a default URLs file that defines a stock set of views. You can get search up and running on your Django site simply by adding the Haystack URLs to your root URLConf:

```
(r'^search/', include('haystack.urls')),
```

Haystack also needs us to define a search template, called `search/search.html` by default. This template will receive a paginator object that contains the list of search results. Unlike django-sphinx, which provides a `QuerySet` of model objects, our Haystack search results will be instances of the Haystack `SearchResult` class. This class includes access to the model through an object attribute, but it should be noted that this will cause a database connection and affect performance.

In addition, the `SearchResult` object has all of the fields included in the `SearchIndex` we previously defined. In order to save on database hits, we can also store data in the index that is available as part of our `SearchResult` objects. This means we can store a model attribute, for example, directly in our `SearchIndex` and not need to access the object attribute in `SearchResults` (causing a database hit).

To do this we can specify the `stored=True` argument, which is the default for all `SearchIndex` field attributes. In our `ProductIndex` example, when we process the `SearchResult` list we can access the slug data via `result.slug` or `result.object.slug`, but the latter will incur a performance penalty.

Now that our indexes have been defined and we've added a search view, we need to actually generate the index files. To simplify this task, Haystack provides a Django management command, which we can pass to the `django-admin.py` or `manage.py` utilities. This command is called `rebuild_index` and it will cause Haystack to process the models in our database and create indexes for those that define one.

The rebuild command will completely recreate our indexes (by clearing them and updating against all model data). An alternative is the `update_index` management command, which will only update, but not clear, our indexes. The update command can take an age parameter, which will only update model objects from a certain time period. For example, running `./manage.py update_index --age=24` will refresh the index for model objects update in the last 24 hours. To use this, we need to add a `get_updated_field` method to our `SearchIndex` definition that returns a model field to use for checking age. Often this can be a `pub_date DateTime` field on our model, as in the following:

```
class ProductIndex(indexes.SearchIndex):
    ...

def get_update_field(self):
return 'pub_date'
```

You will need to schedule a periodic call to the `update_index` command to ensure that new data is indexed and will appear in search results. This is necessary for any search engine indexer. In the case of Haystack, we can use our server's cron daemon to schedule a nightly run of `update_index` with this example crontab line:

```
0 2 * * * django-admin.py update_index–age=24
```

It is not necessary to schedule the index update this way, especially if your data does not change frequently (for example, your e-commerce site doesn't add new products on a daily basis). In these cases you can do periodic, manual index updates.

However, there are other reasons you may wish to update an index besides capturing new data. If you store field information for your models in the `SearchField` objects using `stored=True`, you will need to reindex to refresh this stored data. Search results may use this stored data to avoid a database lookup on actual models, so reindexing ensures any changes (to price, for example) are reflected in search results on a regular basis.

Haystack for real-time search

One of the unique features of Haystack is that it can be used to build a real-time search engine for your data. In the typical approach to search, including the Haystack usage described above, indexes are built at some regular interval. Any new data added to the site will not be indexed until the next run of the `update_index` command, either manually or via a scheduled cron job. However, Haystack offers an alternative to this approach, which is the `RealTimeSearchIndex`.

Using the `RealTimeSearchIndex` is simple: instead of writing your indexes as subclasses of `SearchIndex`, you simply subclass `index.RealTimeSearchIndex` instead. Using this real-time class attaches signal handlers to your model's `post_save` and `post_delete` signals.

These signal handlers cause Haystack to immediately update your indexes whenever a model object is saved or deleted. This means your indexes are updated in real time and search results will reflect any changes instantly. This functionality could be useful in e-commerce applications by allowing staff to search over product sales data in real time or for cases where the price of a good or service changes many times throughout the day.

Real-time search of this nature causes significant load on your server resources if you have many object updates and deletions. Before deploying this solution, you should make sure your search engine and database server can handle the increased load that frequent indexing will create.

Haystack is extremely powerful considering how unassuming it appears at first glance. It can be used to produce very sophisticated search strategies across even the largest of Django sites. It also follows many of the Django conventions that have evolved over the past couple of releases and is a great project to reference as an example reusable application. Because it supports multiple search engine backends it has the performance and flexibility to fit many development needs. For these reasons and more, it is a highly recommended tool.

Xapian/Djapian

The Djapian project is a search layer for Django that specifically supports the Xapian open source search engine. It was created in Russia by Alex Koshelev and is currently under continued development. Xapian is a search engine library, not specifically a search engine application. Djapian wraps this library and applies this library and attaches the full-text search functionality to a Django model. This follows the pattern we've seen in the previous examples of a Django layer on top of a search engine application or library.

You can get the Djapian library from Google Code at the following URL: `http://code.google.com/p/djapian/`.

Once installed, we can begin using Djapian in any Django project by adding it to our `settings.py` file in the `INSTALLED_APPS` setting and creating a `DJAPIAN_DATABASE_PATH` setting to a directory for our search indexes.

As before, indexes must be created in Djapian. As it is wrapping the Xapian library, our indexes are defined in Python, unlike Sphinx where indexes and sources are defined in external configuration files. This index definition is very Django-like, similar to the indexes we built for Haystack. They start by subclassing a base indexer class and then define specific fields for the object we're indexing.

The new index is associated with a model class by adding it to the index space. Djapian uses an index space object, which is just a wrapper around the file system storage routines needed to save index information to disk. The location on the file system where the index space lives is defined in the `DJAPIAN_DATABASE_PATH` setting.

An indexer for our `Product` model in Djapian would look like this:

```
from djapian import space, Indexer
from coleman.products.models import Product

class ProductIndexer(Indexer):
fields = ['name', 'description'], tags = [('price_in_dollars', 'price_
in_dollars')]
space.add_index(Product, ProductIndexer, attach_as='indexer')
```

Once we've defined an index, we can use Djapian's management command to process our `Product` models and build the indexes. We do this by issuing the index command to `django-admin.py` or `manage.py`. To build the initial index, we must pass the `--refresh` argument:

```
$ django-admin.py index–rebuild
```

Later calls to index will update the indexes and don't require the `rebuild` flag, unless we want to delete our indexes and build from scratch.

A really convenient feature of Djapian is the ability to test search results in a special index shell. You can access this feature by passing the `indexshell` management command:

```
$ django-admin.py indexshell
>>> use 0.1.0
>>> query "cranberry sauce"
[<Hit: model=products.Product pk=200, percent=100, rank=0,
weight=0.4444>]
```

This allows us to test the quality of the search results from our indexes in a quick and effective way, without writing any views or other Django code. This can be very useful for testing purposes and to evaluate the indexes you've created.

Searching indexes

When we write `Indexer` objects and add them to our space object, they are also attached as an attribute to our model class. This happened when we included the `attach_as='indexer'` keyword argument to the `add_index` method above. This attribute is what we will use to work with the search engine from Django code.

To perform a search query over our `Product` models, for example, we obtain the indexer for the model and call the search method:

```
product_indexer = Product.indexer results = product_indexer.
search('cranberry sauce').prefetch()
```

This searches our product indexes for `'cranberry sauce'` and stores the results in `results`. The results variable will be a Djapian `ResultSet` object, which is similar to Django's built-in `QuerySet` objects. It doesn't support the full set of methods that `QuerySet` does, but you can loop over the items returned, `count()` its length, slice it, and order the results with the `order_by()` method, among other operations. It's also compatible with Django's built-in `Paginator` class.

Advanced Xapian features

Djapian takes advantage of some of the advanced Xapian features in easy and convenient ways. For example, we can use the Xapian spell checker simply by chaining a method call to the end of our search operation:

```
results = product_indexer.search("cranberri sauce").spell_correction()
```

This will instruct Xapian to try and correct out spelling errors in the query string. To find out what Xapian decided to use, call `get_corrected_query_string` on the results object:

```
>>> results.get_corrected_query_string()
cranberry sauce
```

Xapian also supports word stemming, as we discussed in the section on Sphinx. This normalizes word variations to a single form in attempt to improve search results. Activating the Xapian stemming function using Djapian is extremely easy, just add a `DJAPIAN_STEMMING_LANG` setting to your project settings file and set it to the language of your choice. For English, use:

```
DJAPIAN_STEMMING_LANG = "en"
```

As you can see, Djapian does an excellent job of integrating with the Xapian open source search engine library. It offers yet another powerful tool for adding search to any Django project.

We should also note that the Xapian search engine can also be used with Haystack and all of the discussion of Haystack in previous sections applies, whether the backend uses Whoosh or Xapian. Xapian's advantage over Whoosh is primarily that it's written in C++ instead of Python. This means, generally, that it should be some degree faster. This sort of metric, however, is dependent on a lot of factors and should not discourage you from working with Whoosh or other Python search engines.

Summary

Throughout this chapter we have explored the landscape of search engine options for our Django projects. On some level, the diversity is really astounding considering Django's relatively short existence. This is an area where the Django and Python community really shines and is a testament to both the popularity of the framework and the quality of its design. To review, this chapter covered:

- Creating a simple, homegrown search engine using MySQL indexes
- The Sphinx search engine and its use with django-sphinx
- A tour of the pure-Python search engine, Whoosh
- Demonstration of the powerful Haystack Django application
- Quick examples of using Xapian and Djapian

We also covered some continuing Django development themes such as pluggable backends and the registration pattern. The community projects presented in this chapter are significant for their designs in addition to their usefulness as tools. Anyone interested in improving their Python and Django development skills could benefit greatly by studying the code presented in these projects.

7
Data and Report Generation

The needs of every e-commerce application can vary widely when it comes to reports, metrics, and data exports. Some businesses will want to capture detailed profiles of their customers and what they are purchasing in order to optimize promotions and marketing activities for their particular needs. Others will be interested in making data available internally, to provide the boss updates on how many jars of cranberry preserves sold in December last year versus this year.

In this chapter, we will discuss a toolbox of Python libraries and Django applications to assist with whatever reporting needs that may arise. These topics include:

- Serializing and exposing data
- Tracking and improving search engine rank using sitemaps
- Generating charts and graph-based reports
- Exporting information via RSS and Atom feeds
- Salesforce integration

We will be using a variety of tools, many builtin to Django. As in other chapters, however, we will discuss some third-party libraries. These are all relatively stable and mature, but as with all open source technology, new versions could change their usage at any time.

Exposing data and APIs

One of the biggest elements of the web applications developed in the last decade has been the adoption of so-called **Web 2.0** features. These come in a variety of flavors, but one thing that has been persistent amongst them all is a data-centric view of the world. Modern web applications work with data, usually stored in a database, in ways that are more modular and flexible than ever before. As a result, many web-based companies are choosing to share parts of their data with the world in hopes of generating "buzz", or so that interested developers might create a clever "mash-up" (a combination of third-party application software with data exposed via an API or other source).

These mash-ups take a variety of forms. Some simply allow external data to be integrated or imported into a desktop or web-based application. For example, loading Amazon's vast product catalog into a niche website on movie reviews. Others actually deploy software written in web-based languages into their own application. This software is usually provided by the service that is exposing their data in the form of a code library or web-accessible API.

Larger web services that want to provide users with programmatic access to their data will produce code libraries written in one or more of the popular web-development languages. Increasingly, this includes Python, though not always, and typically also includes PHP, Java, or Perl. Often when an official data library exists in another language, an enterprising developer has ported the code to Python.

Increasingly, however, full-on code libraries are eschewed in favor of open, standards-based, web-accessible APIs. These came into existence on the Web in the form of remote procedure call tools. These mapped functions in a local application written in a programming language that supports **XML-RPC** to functions on a server that exposed a specific, well-documented interface. XML and network transport protocols were used "under the hood" to make the connection and "call" the function.

Other similar technologies also achieved a lot of use. For example, many web-services provide **Simple Object Access Protocol** (**SOAP**) interface, which is the successor to XML-RPC and built on a very similar foundation. Other standards, sometimes with proprietary implementations, also exist, but many new web-services are now building APIs using REST-style architecture.

REST stands for **Representational State Transfer** and is a lightweight and open technique for transmitting data across the Web in both server-to-server and client-to-server situations. It has become extremely popular in the Web 2.0 and open source world due to its ease of use and its reliance on standard web protocols such as HTTP, though it is not limited to any one particular protocol.

A full discussion of REST web services is beyond the scope of this book. Despite their simplicity, there can arise many complicated technical details. Our implementation in this chapter will focus on a very straightforward, yet powerful design.

REST focuses on defining our data as a resource that when used with HTTP can map to a URL. Access to data in this scheme is simply a matter of specifying a URL and, if supported, any look-up, filter, or other operational parameters. A fully featured REST web service that uses the HTTP protocol will attempt to define as many operations as possible using the basic HTTP access methods. These include the usual `GET` and `POST` methods, but also `PUT` and `DELETE`, which can be used for replacement, updating, or deletion of resources.

There is no standard implementation of a REST-based web service and as such the design and use can vary widely from application to application. Still, REST is lightweight enough and relies on a well known set of basic architectures that a developer can learn a new REST-based web service in a very short period of time. This gives it a degree of advantage over competing SOAP or XML-RPC web services. Of course, there are many people who would dispute this claim. For our purposes, however, REST will work very well and we will begin by implementing a REST-based view of our data using Django.

Writing our own REST service in Django would be very straightforward, partly because URL mapping schemes are very easy to design in the `urls.py` file. A very quick and dirty data API could be created using the following super-simple URL patterns:

```
(r'^api/(?P<obj_model>\w*)/$', 'project.views.api')
(r'^api/(?P<obj_model>\w*)/(?P<id>\d*)/$', 'project.views.api')
```

And this view:

```
from django.core import serializers

def api(request, obj_model, obj_id=None):
    model = get_model(obj_model.split("."))
    if model is None:
        raise Http404
    if obj_id is not None:
        results = model.objects.get(id=obj_id)
    else:
        results = model.objects.all()
    json_data = serializers.serialize('json', results)
    return HttpResponse(json_data, mimetype='application/json'))
```

This approach as it is written above is not recommended, but it shows an example of one of the simplest possible data APIs. The API view returns the full set of model objects requested in JSON form. JSON is a simple, lightweight data format that resembles JavaScript syntax. It is quickly becoming the preferred method of data transfer for web applications.

To request a list of all products, for example, we only need to access the following URL path on our site: `/api/products.Product/`. This uses Django's `app.model` syntax to refer to the model we want to retrieve. The view uses `get_model` to obtain a reference to the Product model and then we can work with it as needed. A specific model can be retrieved by including an object ID in the URL path: `/api/products.Product/123/` would retrieve the Product whose `ID` is `123`.

After obtaining the results data, it must be encoded to JSON format. Django provides serializers for several data formats, including JSON. These are all located in the `django.code.serializers` module. In our case, we simply pass the results `QuerySet` to the serialize function, which returns our JSON data. We can limit the fields to be serialized by including a field's keyword argument in the call to serialize:

```
json_data = serializers.serialize('json', results,
fields=('name','price'))
```

We can also use the built-in serializers to generate XML. We could modify the above view to include a format flag to allow the generation of JSON or XML:

```
def api(request, obj_model, obj_id=None, format='json'):
    model = get_model(*obj_model.split())
    If model is None:
        raise Http404
    if obj_id is not None:
        results = model.objects.get(id=obj_id)
    else:
        results = model.objects.all()
    serialized_data = serializers.serialize(format, results)
    return HttpResponse(serialized_data,
                                    mimetype='application/' + format)
```

Format could be passed directly on the URL or better yet, we could define two distinct URL patterns and use Django's keyword dictionary:

```
(r'^api/(?P<obj_model>\w*)/$', 'project.views.api'),
(r'^api/(?P<obj_model>\w*)/xml/$', 'project.views.api',
{'format': 'xml'}),
(r'^api/(?P<obj_model>\w*)/yaml/$', 'project.views.api',
{'format': 'yaml'}),
(r'^api/(?P<obj_model>\w*)/python/$', 'project.views.api',
{'format': 'python'}),
```

By default our serializer will generate JSON data, but we've got to provide alternative API URLs that support XML, YAML, and Python formats. These are the four built-in formats supported by Django's serializers module. Note that Django's support for YAML as a serialization format requires installation of the third-party PyYAML module.

Building our own API is in some ways both easy and difficult. Clearly we have a good start with the above code, but there are many problems. For example, this is exposing all of our Django model information to the world, including our `User` objects. This is why we do not recommend this approach. The views could be password protected or require a login (which would make programmatic access from code more difficult) or we could look for another solution.

Django-piston: A mini-framework for data APIs

One excellent Django community project that has emerged recently is called **django-piston**. Piston allows Django developers to quickly and easily build data APIs for their web applications using a REST-style interface. It supports all the serialization formats mentioned above and includes sophisticated authentication tools such as OAuth as well as HTTP Basic.

The official repository for django-piston is hosted on bitbucket at the following URL: `http://bitbucket.org/jespern/django-piston/wiki/Home`.

Complete documentation on the installation and usage of Piston are available on the bitbucket site and in the readme file.

Piston supports the full set of HTTP methods: `GET`, `POST`, `PUT`, and `DELETE`. `GET` is used for the retrieval of objects, `POST` is used for creation, `PUT` is used for updating, and `DELETE` is used for deletion. Any subset of these operations can be defined on a model-by-model basis. Piston does this by using class-based "handlers" that behave somewhat like class-based generic views.

To define a handler on our `Product` model, we would write something like this:

```
from piston.handler import BaseHandler
from coleman.products import Product


class ProductHandler(BaseHandler):
    allowed_methods = ('GET',)
    model = Product

    def read(self, request, post_slug):
          ...
```

The `ProductHandler` defines one operation, the `GET`, on our `Product` model. To define the behavior when a `GET` request is made to a `Product` object, we write a read method. Method names for other HTTP operations include: create for `POST`, update for `PUT`, and delete for `DELETE`. Each of these methods can be defined on our `ProductHandler` and added to the `allowed_methods` class variable and Piston will instantly enable them in our web-based API.

To utilize our `ProductHandler`, we must create the appropriate URL scheme in our `urls.py` file:

```
from piston.resource import Resource
from coleman.api.handlers import ProductHandler


product_resource = Resource(ProductHandler)
(r'^product/(?P<slug>[^/]+)/', product_resource)
```

Our Product objects and their data are now accessible using the URL above and the Product slug field, as in: `/api/product/cranberry-sauce/`.

Piston allows us to restrict the returned data by including fields and exclude attributes on our handler class:

```
class ProductHandler(BaseHandler):
    fields = ('name', 'slug', 'description')
    exclude = ('id', 'photo')
        ...
```

Piston also makes it very easy to request our data in a different format. Simply pass the format as a `GET` parameter to any Piston-enabled URL and set the value to any of the formats Piston supports. For example, to get our Cranberry Sauce product information in YAML format use: `/api/product/cranberry-sauce/?format=yaml`.

Adding authentication to our handlers is also very simple. Django-piston includes three kinds of authentication handlers in the current release: HTTP BASIC, OAuth, and Django. The Django authentication handler is a simple wrapper around the usual Django auth module. This means users will need cookies enabled and will be required to log in to the site using their Django account before this auth handler will grant API access.

The other two handlers are more suitable for programmatic access from a script or off-site. HTTP BASIC uses the standard, web-server based authentication. In a typical Apache configuration, this involves defining user and password combinations in an `htpasswd` file using the `htpasswd` command line utility. See the web server's documentation for more details. It's also possible to configure Apache authentication against Django's auth module to support HTTP BASIC auth against the Django database. This involves adding the `django.contrib.auth.handlers.modpython` handler to the Apache configuration. See the Django manual for additional details.

To attach BASIC authentication to the handler for our `Product` model, we will include it in our `urls.py` file as part of the `Resource` object definition:

```
from piston.authentication import HttpBasicAuthentication

basic_auth = HttpBasicAuthentication(realm='Products API') product_
resource = Resource(handler=ProductHandler, auth=basic_auth)
```

Our Product URLs will now be available only to clients who have passed HTTP BASIC authentication with a user name and password.

As we've seen, Piston makes building a REST-based API for our Django projects extremely easy. It also uses some Django design principles we've seen earlier. For example, the authentication tools are designed to be pluggable. We can examine the `HttpBasicAuthentication` class in `piston.authentication` as a template to write our own. A custom authentication class can be plugged in to the `Resource` definition with just a small change to the code. Despite being easily customizable, Piston's default setup includes enough built-in functionality for the majority of data API needs.

Django's syndication framework

In cases where a full-fledged data API would be overkill, but exporting some data in a standard format is required, Django's syndication tools may be the perfect fit. Having been originally designed for newspaper websites, Django includes robust support for exporting information in syndication formats. This is usually done using the popular RSS or Atom feed formats.

Syndication feeds allow us to render our data in a standard format that can be consumed by human-controlled reader software, such as Google Reader or NetNewsWire, and also by machines running software tools.

Feeds began as a way of consuming content from multiple sources in a single location using reader software. Today, however, lots of variations on this theme exist. For example, Twitter is itself one big feed generating application (one can even consume Twitter content in RSS/Atom format).

It's often not necessary to have a reason to syndicate our data, as it is increasingly considered a courtesy that enables others to consume our information in their own way. In an e-commerce situation, we could export a collection of weekly or daily promotional sales as RSS or Atom feeds, to which our customers could subscribe and then read at their convenience in their preferred feed reader.

Feeds could also be parsed by "affiliate" sites or even by physical devices. Imagine a retail store that posted their weekly specials in RSS format then connected this feed to an LED sign or LCD TV that can consume RSS information. The sign could be posted in the store window and would be constantly refreshing that store's advertised sales.

Django's syndication framework includes lots of flexibility in the way it generates feeds. In the simplest form, we just need to write a class specific to the model whose information we want to export. This class defines an items method, which is used to retrieve the contents of the feed in an appropriate order. A simple Django template is used to define the content of the feed.

Let's start by building a feed class for our `Product` model:

```
from django.contrib.syndication.feeds import Feed
from coleman.products.models import Product

class AllProducts(Feed):
    title = 'CranStore.com's Product Catalog'
    link = '/feed/'
    description = 'An updating feed of Products available on our site'

    def items(self):
        return Product.objects.all()
```

This is the simplest example of generating a feed in Django. We can enhance this in many ways, all of which are described in the Django documentation. The syndication framework is very similar to django-piston in that after defining a feed, we must update our `urls.py` file to include it on our site:

```
from coleman.feeds import AllProducts

feeds = {'products': AllProducts}

urlpatterns = patterns(
(r'^feeds/(?P<url>.*)/$', 'django.contrib.syndication.views.feed',
     {'feed_dict': feeds}))
```

Our Product models are now exporting to an RSS feed at the URL `/feeds/products/`. Notice how the framework uses the feed dictionary's keys as the final portion of the URL. We can create as many feeds as we wish and only need to update the feed dictionary. The URL pattern will be reused by the syndication framework.

In Django, the template system can be used in various ways. When dealing with syndicated feeds, templates are used to control the output to the feed. Regardless of format, the template is translated appropriately and the same templates can be used for any feed the framework will produce (versions of RSS and Atom).

Django does not require us to create a feed template. In cases where no template exists, it will default to the string representation of the model (the `__unicode__` or `__str__` methods). In most cases, though, we will want to create appropriate templates.

The feed templates live in a feeds directory beneath our site's templates location. Recall the feed dictionary we used earlier; the keys to this dictionary not only affect the URL where the feed lives, but also what templates are rendered. In our previous example, the feed dictionary contained a key called products for our `AllProducts` feed.

Django will attempt to load two templates for this feed, both based off the dictionary key: `feeds/products_title.html` and `feeds/products_description.html`. Note that despite ending in `.html` extensions, these templates are not required to contain HTML and are never complete HTML documents with head and body tags, only fragments. The title template will be rendered for the feed's title element and the description will be rendered in the body.

These templates will have access to two variables, `obj` and `site`, which correspond to the object we're rendering the feed template against and the site where it lives. We access these values using normal Django template double-brace syntax.

Our feed templates can contain HTML, but the results may vary depending on what application is used to consume our feeds. Some have a limited amount of HTML knowledge and most will ignore any attempt to format our output using CSS or other design tools. Using basic HTML tags is recommended for best results.

An example of a feed title template for our `AllProducts` feed would be:

```
{{ obj.name }}
```

And the corresponding description template could look like this:

```
<h1>{{ obj.name }} - {{ obj.get_price }}</h1>
<p>{{ obj.description }}</p>
```

In our case `obj` will be an instance of our `Product` model, so we can use it just as we would in any other Django template.

Django sitemaps

Traditionally, very large sites have provided a directory of the information they've made available using a navigational device known as a sitemap. The intent was originally to help users find information they were looking for with the least amount of effort. Turns out, however, there were better solutions to this problem, namely search engines.

Why dig through a long list of irrelevant information when you can simply type in some keywords and get a much more accurate list of potential matches? Sitemaps made a lot of sense before search engine technology was available on a wide scale, but as we discussed in *Chapter 6, Searching the Product Catalog*, adding search to our applications is now super easy. Even sites that don't offer a search engine can be searched using Google.

So why does Django include a module for automatic sitemap generation? These sitemaps are a little different from the traditional sitemaps, which were usually designed in HTML for human consumption. The sitemaps produced by Django are XML files that are designed to inform machines about the layout and content of your site.

When we say machines, we're really talking about search engines and, more precisely, Google. Google uses sitemaps to build its index of your site. This is very important for achieving a high position in Google's search results.

Django's sitemap module is in `django.constrib.sitemaps`. To get started, we need to add this to our `INSTALLED_APPS` setting and create a URL in our root URL configuration. The URL requires us to pass in a dictionary that resembles the feed dictionary we discussed in the previous section. It takes a string and maps it to a `Sitemap` class for a specific section of our site. For example, we may have a `Products` sitemap that lists all of our product pages as well as other sitemaps that list manufacturers, special deals, blog posts from our corporate blog, or any other piece of content we've put on the Web.

A sitemap class for our `Products` would look something like this:

```
from django.contrib.sitemaps import Sitemap
from coleman.products.models import Product

class ProductSitemap(Sitemap):
    def items(self):
        return Product.objects.all()
```

We could filter our `Product` module instead of calling all and limit it by some useful metric. Say, for example, we had a discontinued field but the `Product` object remained in our database. We may not want Google to index discontinued products so we could return `Product.objects.filter(discontinued=False)` in the sitemap items method.

The items returned for the sitemap will be assumed to have a `get_absolute_url` method. This method will be used by the Django sitemap framework to construct the URLs for all of our objects in the sitemap. It is important to make sure your `get_absolute_url` methods are correct and functional for any object you want indexed in a sitemap.

Once we've generated a sitemap for our site, we want to make sure to tell Google about it. The way to do this is register for a Google Webmaster account at `http://google.com/webmaster`. In addition to submitting our sitemap, Google's Webmaster Tools let us see all kinds of interesting metrics about our site and how Google evaluates it. This includes what keywords we score highly on and where any incoming links originate. It also lets us track how changes to our site affect our position in the Google index.

If you're building an e-commerce site with lots of content, a very large product catalog, for example, it is highly recommended that you generate a sitemap and submit it to Google. Several books have been written about "Search Engine Optimization" that include lots of search engine magic. Building a sitemap is among the best and most realistic tactics for improving your search result position.

ReportLab: Generating PDF reports from Python

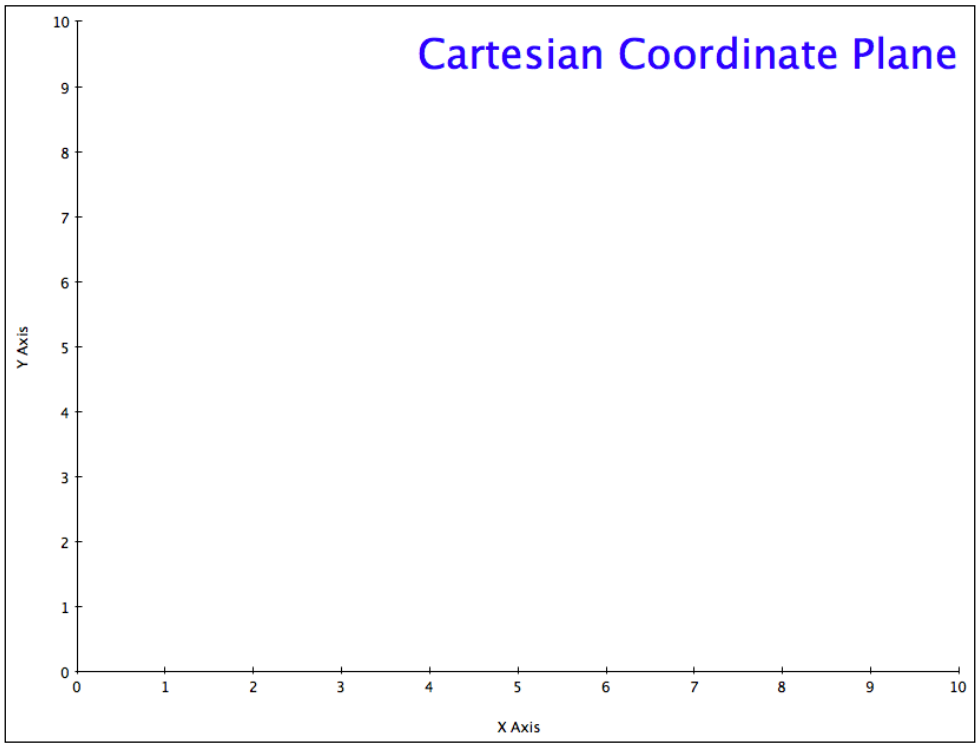
The Python community offers dozens of libraries designed to generate graphics, reports, PDF files, images, and charts. It can be somewhat overwhelming choosing which tool is appropriate for the job. In this section, we will experiment with the ReportLab toolkit, which is a Python module that allows us to create PDF files. ReportLab can be integrated with Django to generate dynamic PDFs on-the-fly for the data stored in our Django models.

ReportLab is an open source project available at `http://www.reportlab.org`. It is a very mature tool and includes binaries for several platforms as well as source code. It also contains extension code written in C, so it's relatively fast. It is possible for ReportLab to insert PNG and GIF image files into PDF output, but in order to do so we must have the **Python Imaging Library** (**PIL**) installed. We will not require this functionality in this book, but if you need it for a future project, see the PIL documentation for setup instructions.

The starting point in the ReportLab API is drawing to canvas objects. Canvas objects are exactly what they sound like: blank slates that are accessible using various drawing commands that paint graphics, images, and words. This is sometimes referred to as a low-level drawing interface because generating output is often tedious. If we were creating anything beyond basic reporting output, we would likely want to build our own framework or set of routines on top of these low-level drawing functions.

Drawing to a canvas object is a lot like working with the old LOGO programming language. It has a cursor that we can move around and draw points from one position to another. Mostly, these drawing functions work with two-dimensional (x, y) coordinates to specify starting and ending positions.

This two-dimensional coordinate system in ReportLab is different from the typical system used in many graphics applications. It is the standard Cartesian plane, whose origin (x and y coordinates both equal to 0) begins in the lower-left hand corner instead of the typical upper-right hand corner. This coordinate system is used in most mathematics courses, but computer graphics tools, including HTML and CSS layouts, typically use a different coordinate system, where the origin is in the upper-left.



ReportLab's low-level interface also includes functions to render text to a canvas. This includes support for different fonts and colors. The text routines we will see, however, may surprise you with their relative crudeness. For example, word-wrapping and other typesetting operations are not automatically implemented. ReportLab includes a more advanced set of routines called PLATYPUS, which can handle page layout and typography. Most low-level drawing tools do not include this functionality by default (hence the name "low-level").

This low-level drawing interface is called `pdfgen` and is located in the `reportlab.pdfgen` module. The ReportLab User's Guide includes extensive information about its use and a separate API reference is also available.

The ReportLab canvas object is designed to work directly on files. We can create a new canvas from an existing open file object or by simply passing in a file name. The canvas constructor takes as its first argument the filename or an open file object. For example:

```
from reportlab.pdfgen import canvas

c = canvas.Canvas("myreport.pdf")
```

Once we obtained a canvas object, we can access the drawing routines as methods on the instance. To draw some text, we can call the `drawString` method:

```
c.drawString(250, 250, "Ecommerce in Django")
```

This command moves the cursor to coordinates `(250, 250)` and draws the string `"Ecommerce in Django"`. In addition to drawing strings, the canvas object includes methods to create rectangles, lines, circles, and other shapes.

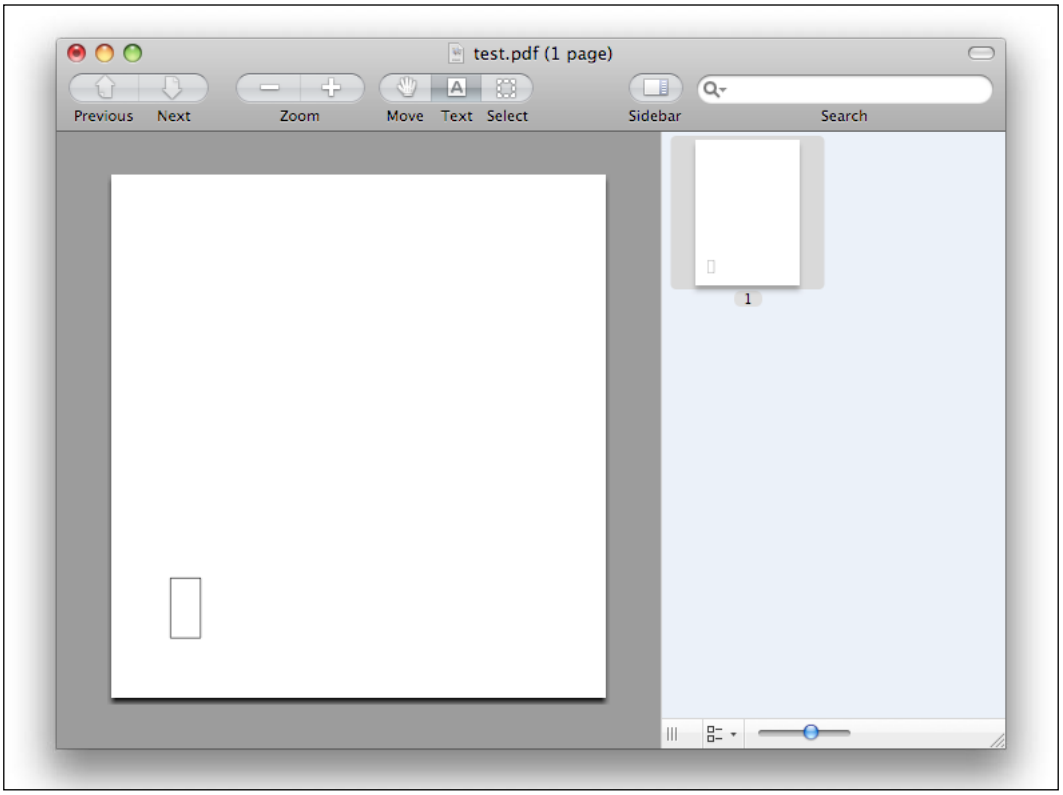
Because PDF was originally designed for printed output, consideration needs to be made for page size. Page size refers to the size of the PDF document if it were to be output to paper. By default, ReportLab uses the A4 standard, but it supports most popular page sizes, including letter, the typical size used in the US. Various page sizes are defined in `reportlab.lib.pagesizes`. To change this setting for our canvas object, we pass in the `pagesize` keyword argument to the canvas constructor.

```
from reportlab.lib.pagesizes import letter

c = canvas.Canvas('myreport.pdf', pagesize=letter)
```

Because the units passed to our drawing functions, like `rect`, will vary according to what page size we're using, we can use ReportLab's units module to precisely control the output of our drawing methods. Units are stored in `reportlab.lib.units`. We can use the inch unit to draw shapes of a specific size:

```
from reportlab.lib.units import inch

c.rect(1*inch, 1*inch, 0.5*inch, 1*inch)
```

The above code fragment draws a rectangle, starting one inch from the bottom and one inch from the left of the page, with sides that are length 0.5 inches and one inch, as shown in the following screenshot:



Not particularly impressive is it? As you can see, using the low-level library routines require a lot of work to generate very little results. Using these routines directly is tedious. They are certainly useful and required for some tasks. They can also act as building blocks for your own, more sophisticated routines.

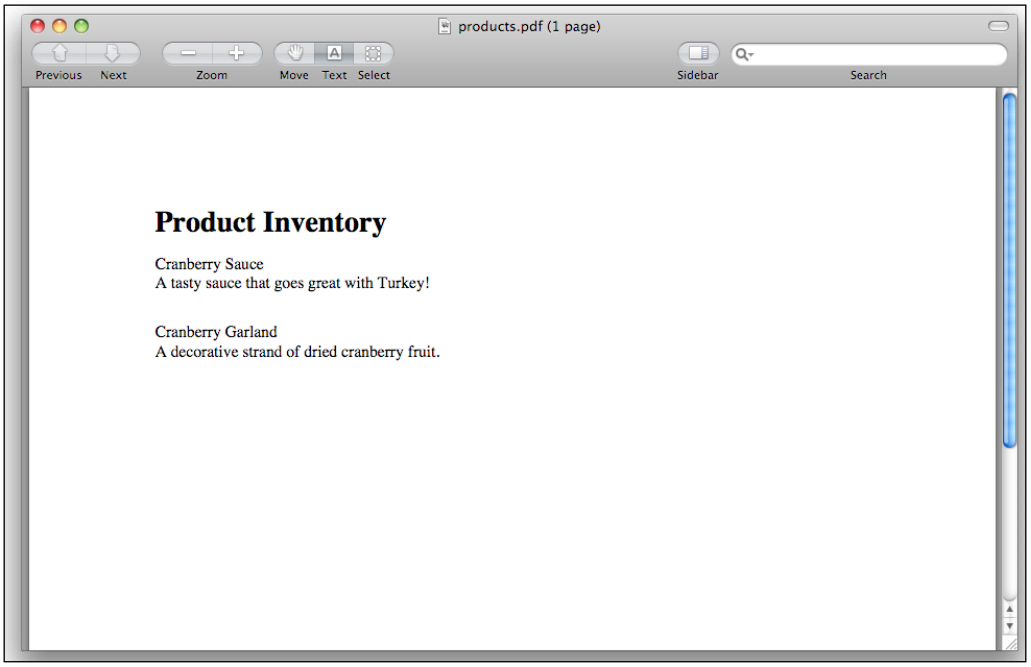
Building our own library of routines would still be a lot of work. Fortunately ReportLab includes a built-in high-level interface for creating sophisticated report documents quickly. These routines are called PLATYPUS; we mentioned them earlier when talking about typesetting text, but they can do much more.

PLATYPUS is an acronym for "Page Layout and Typography Using Scripts". The PLATYPUS code is located in the `reportlab.platypus` module. It allows us to create some very sophisticated documents suitable for reporting systems in an e-commerce application.

Using PLATYPUS means we don't have to worry about things such as page margins, font sizes, and word-wrapping. The bulk of this heavy lifting is taken care of by the high-level routines. We can, if we wish, access the low-level canvas routines. Instead we can build a document from a template object, defining and adding the elements (such as paragraphs, tables, spacers, and images) to the document container.

The following example generates a PDF report listing all the products in our `Product` inventory:

```
from reportlab.platypus.doctemplate import SimpleDocTemplate
from reportlab.platypus import Paragraph, Spacer
from reportlab.lib import sytles

doc = SimpleDocTemplate("products.pdf")
Catalog = []
header = Paragraph("Product Inventory", styles['Heading1'])
Catalog.append(header)
style = styles['Normal']
for product in Product.objects.all():
    for product in Product.objects.all():
        p = Paragraph("%s" % product.name, style)
        Catalog.append(p)
        s = Spacer(1, 0.25*inch)
        Catalog.append(s)
doc.build(Catalog)
```



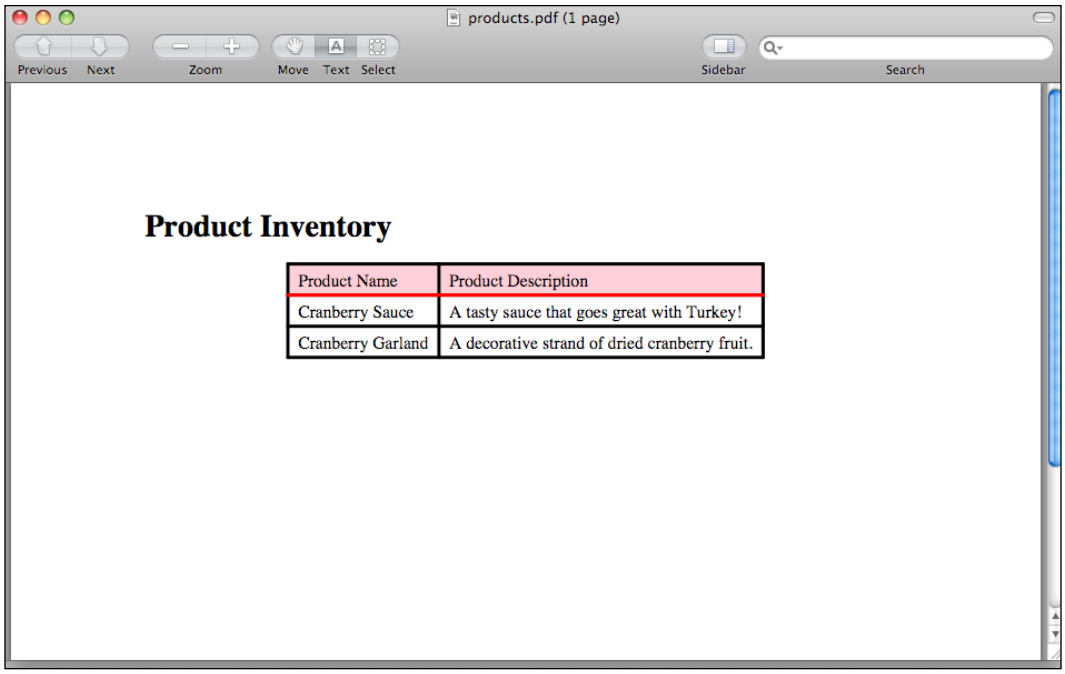
The previous code generates a PDF file called `products.pdf` that contains a header and a list of our product inventory. The output is displayed in the accompanying screenshot.

The document template used above, `SimpleDocTemplate`, is a class derived from ReportLab's `BaseDocTemplate` class. It provides for management of the page, the PDF metadata such as document title and author, and can even be used for PDF encryption. By inheriting from `BaseDocTemplate`, we could create our own document templates that correspond to specific reports or types of report we might generate on a regular basis (a product deals flyer, for example).

In addition to the paragraph and spacer elements we saw in the previous example, documents also support very sophisticated table definitions. We can render the same product inventory in a table format using ReportLab's `Table` class. Tables can be styled in myriad ways, including setting background and line colors around specific cells, rows or columns, drawing boxes around subsets of cells, and adding images to specific cells. A very simple demonstration is as follows:

```
doc = SimpleDocTemplate("products.pdf")
Catalog = []
header = Paragraph("Product Inventory", styles['Heading1'])
Catalog.append(header)
style = styles['Normal']
headings = ('Product Name', 'Product Description')
allproducts = [(p.name, p.description) for p in Product.objects.all()]
t = Table([headings] + allproducts)
t.setStyle(TableStyle(
                   [('GRID', (0,0), (1,-1), 2, colors.black),
                    ('LINEBELOW', (0,0), (-1,0), 2, colors.red),
                    ('BACKGROUND', (0, 0), (-1, 0), colors.pink)]))
Catalog.append(t) doc.build(Catalog)
```

This code defines a two-column table with headings **Product Name** and **Product Description**, and then renders the product inventory into each row in the table. We set some style rules for the `Table` element, including a black grid throughout the table, a red line beneath the headers, and a light pink background on the header row. The PDF output looks like this:



Notice how the padding and centering of the table is handled automatically by the ReportLab package. Now we're getting more sophisticated, but there is a lot more to the ReportLab library. We will end our tour by looking at ReportLab's chart functions.

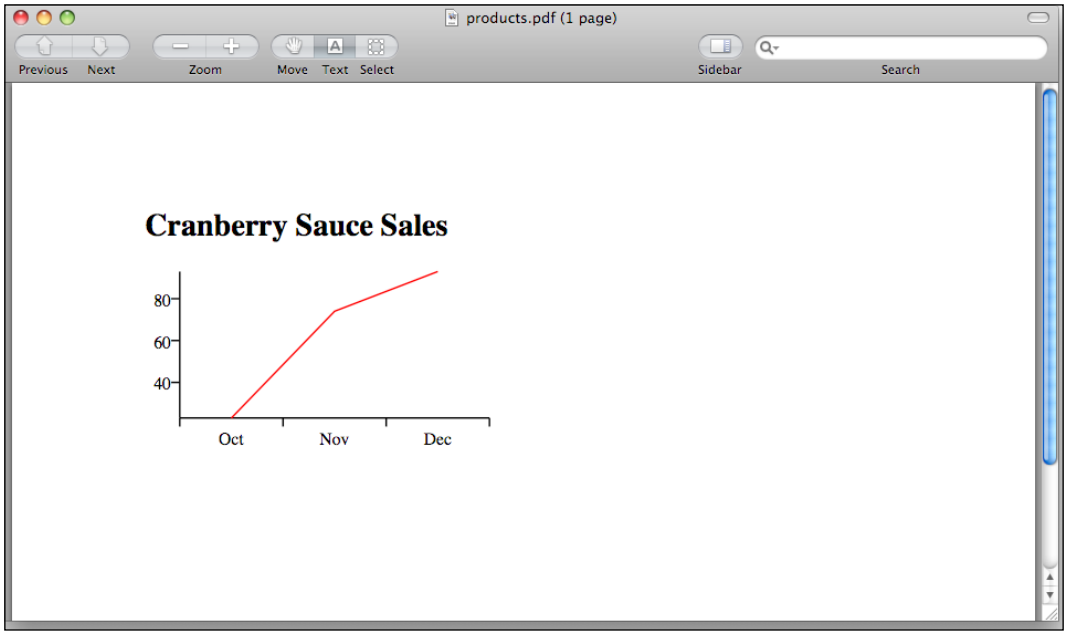
Creating charts in ReportLab is straightforward, but very powerful. One aspect of the charting module worth mentioning is that it does not have to be output to PDF files, but is general purpose enough to render to other formats. PostScript and bitmap output is available, as well as support for the SVG XML format is now supported by most browsers.

To create charts we need to use ReportLab's `Drawing` object. This is what ReportLab calls a "Flowable". We've already seen several examples of flowable objects: `Paragraph`, `Spacer`, and `Table`. Flowables can be added to a document template as we've already seen. In addition to the `Drawing` object, ReportLab provides several classes that represent different kinds of charts: line charts, bar charts, pie charts, and so on. We instantiate an object from one of these classes and manipulate it by adding our data, our axis labels, and so on. Once our chart object is set up, we can add it to our `Drawing` flowable and then add the `Drawing` to our document template.

The following example renders a simple line chart with fictitious sales data for our CranStore.com website. It shows sales of cranberry sauce through the months of October, November, and December. The code is as follows:

```
from reportlab.graphics.charts.linecharts import HorizontalLineChart
from reportlab.graphics.shapes import Drawing
sales = [ (23, 74, 93) ]
months = ['Oct', 'Nov', 'Dec']

doc = SimpleDocTemplate("products.pdf")
Catalog = []
style = styles['Normal']
p = Paragraph("Cranberry Sauce Sales", styles['Heading1']) Catalog.
append(p)
d = Drawing(100, 100)
cht = HorizontalLineChart()
cht.data = sales
cht.categoryAxis.categoryNames = months
d.add(cht)
Catalog.append(d)
doc.build(Catalog)
```

The output in PDF format appears as follows:

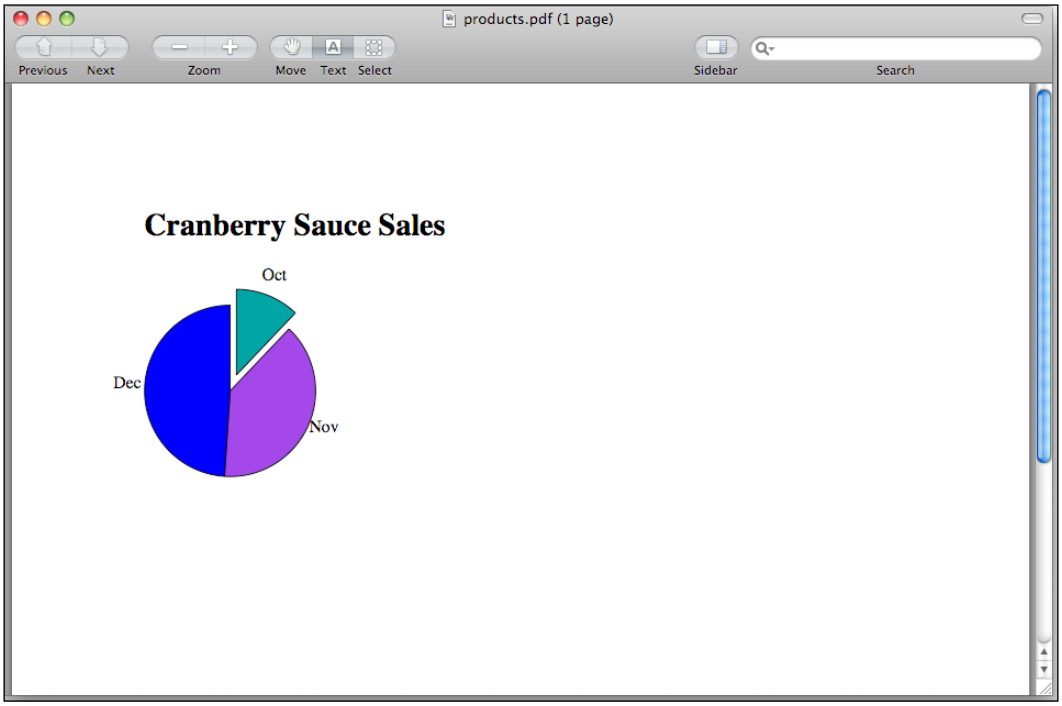


Notice that our sales data is a list of tuples. This list could contain more than one element. If it did, then our horizontal line chart would contain two lines: one for each data series defined in the sales list. Most of ReportLab's line chart classes allow for this type of data definition.

Generating a pie chart is equally as simple. We can even add a fancy pop-out on a specific pie slice:

```
doc = SimpleDocTemplate("products.pdf")
Catalog = []
style = styles['Normal']
p = Paragraph("Cranberry Sauce Sales", styles['Heading1']) Catalog.
append(p)
d = Drawing(100, 125)
cht = Pie()
cht.data = sales[0]
cht.labels = months
cht.slices[0].popout = 10
d.add(cht)
Catalog.append(d)
doc.build(Catalog)
```

When rendered the pie chart appears as follows:



As you can see, ReportLab allows us to quickly generate interesting report output in PDF and other formats. As it is a Python library, we can easily integrate it with our existing Django code. But what if we want to publish our report information to the Web? The next section will explain how to write Django views that generate custom, dynamic PDF output.

Creating PDF views

When working with ReportLab, we briefly mentioned that we can create canvas or document templates using either a file name or an open Python file, or file-like, object. In our previous examples, we created a new file using a file name passed as a string to our ReportLab constructors. If we wanted, we could have opened a file manually and used it as a constructor argument:

```
f = open('products.pdf')
doc = SimpleDocTemplate(f)
```

This is important because when working with Django views, we want to be able to return this PDF file after generating it from our database information. One idea is to generate the file on disk and redirect the user's browser to the newly create PDF. But this is a lot of extra effort and will require maintenance of the disk to prevent old report requests from using up our server's storage space.

A much better alternative is to use ReportLab's support for file-like objects with our Django response object. Django views always return an instance of the `HttpResponse` class. It turns out that this response object is itself a file-like object; it supports a write method and can therefore be used with our ReportLab routines. This works like so: our view code constructs a response object and passes it to our report generation code. When our report is created, control returns to our view, which returns our response object that now contains the report.

Here is an example Django view that generates a pie chart using the routine listed in the previous section. The pie chart code has been moved to a `make_pie_chart` function, which takes and returns a file object:

```
def piechart_view(request):
          response = HttpResponse(mimetype='application/pdf')
doc = SimpleDocTemplate(response)
Catalog = []
style = styles['Normal']
p = Paragraph("Cranberry Sauce Sales", styles['Heading1'])
Catalog.append(p)
d = Drawing(100, 125)
cht = Pie()
```

```
cht.data = sales[0]
cht.labels = months
cht.slices[0].popout = 10
d.add(cht)
Catalog.append(d)
doc.build(Catalog)
return response
```

The key to these specialized views is setting the appropriate `mimetype` on the `HttpResponse` object. Browsers will interpret the mimetype and launch the appropriate in-browser handler for the file returned by the view. You can specify other mimetypes in this way, as well. If we were using ReportLab to generate a PNG file instead of PDF, we would use this:

```
response = HttpResponse(mimetype='image/png')
```

This is an extremely powerful feature of Django and allows us to write views that generate reports based on parameters defined in URL patterns or using `GET` and `POST` values. You could even manipulate the report's display by processing `GET` parameters and adjusting the resulting report (for example, breaking out the smallest element of the pie chart if a certain `GET` flag is included).

If we wanted to use a toolkit besides ReportLab, the implementation pattern would be exactly the same. Anything that can work with Python file-like objects can be used in a view, as we've done previously. We can generate any type of file, even spreadsheet data in a comma-separated value format, as long as we provide the appropriate `mimetype`.

Salesforce.com integration

We will end this chapter with a quick overview of Salesforce integration. Salesforce is a cloud-based data management tool that allows organizations to store and manage their data in a collaborative way. Information about important contacts, related organizations, and any other custom piece of data that a team may need to share can be stored in Salesforce.

One advantage of using Salesforce to manage your data is that it includes many built-in reporting mechanisms. These are especially useful in medium-sized organizations that have enough people to make sharing data difficult, but do not have the time or resources to build a custom internal solution, like those we've seen earlier.

The difficulty with Salesforce, however, is that data must be entered. If you're collecting information via a Django application, it is stored in a local database. You could expose this data as we did in the beginning of this chapter using an API, but once exposed you still need tools to manipulate it.

Taking the Salesforce approach, we can push our Django database information into a Salesforce account, which can include custom objects that represent a subset of data we've collected in Django. These objects are defined via the Salesforce website, but typically resemble a table definition you would make in any SQL database.

Salesforce is accessible via a SOAP API and several Python wrapper libraries exist. One such library is `salesforce-beatbox`, which is available at: `http://code.google.com/p/salesforce-beatbox/`.

The beatbox application lets you receive and send information to your Salesforce instance. It requires very little in the way of Python code to get up and running. You will need to generate a security token from your Salesforce preferences page, but after you obtain this you can connect to the Salesforce API like so:

```
import beatbox

USER = 'sf_user@email.com'
PASSWD = 'xxxxx'
SECURITY_TOKEN = 'gds#mklz!'

svc = beatbox.PythonClient() tokenpass = '%s%s' % (PASSWD, SECURITY_
                                        TOKEN) svc.login(USER, tokenpass)
```

The `svc` object should now successfully be connected to Salesforce and we can begin querying our objects using the Salesforce query language, which they call SOQL:

```
result = svc.query("SELECT id,name FROM Merchandise__c")
```

The `result` variable contains a list of dictionary objects. Each dictionary has the keys specified in the `SELECT` statement and the corresponding values in the Salesforce table. Any custom object created in Salesforce will include a `__c` suffix when referenced in a query. We've created a `Merchandise` object in Salesforce, which we query above using the `merchandise__c` table.

Creating objects in Salesforce is equally as simple. To do so we need to construct a dictionary that includes the values for the object we want to create. A particularly important key that is required in all cases is `type`. The type key specifies what kind of Salesforce object we are creating. For example, the following dictionary sets up the new `Merchandise` object that we will create in Salesforce:

```
merch_data = {'name': 'Cranberry Sauce',
                      'price_in_dollars__c': '1.50',
                      'type': 'Merchandise__c'}
```

Once we've created our data, we can send it to Salesforce using our `svc` object:

```
svc.create(data)
```

Updating Salesforce information is equally as straightforward. The only difference is that our data dictionary must include an `id` key whose value is the Salesforce unique ID number. This is in addition to a type key, which is also required for updates, as well as the fields which we want to update and their new values. It is not required that every field has a key in the update dictionary, only those we want to change. For example:

```
updated_data = {'id': '02111AC2DB987', 'type': 'Merchandise__c',
                'price_in_dollars__c': '1.75'}
svc.update(updated_data)
```

Create and update calls to the Salesforce API can also perform bulk updates using their bulk API. This happens automatically when using beatbox when you construct your new or updated data as a list, instead of a single dictionary. Bulk updates are limited to 200 items per call, so some logic is required to slice your data lists to conform to this limit. It is also a requirement that all of the data you are sending to the Salesforce bulk API is of the same type.

Salesforce Object Query Language

When querying our Salesforce instance using the query API function above, we used a subset of SQL that Salesforce calls SOQL. SOQL is much simpler than most SQL implementations, but allows us to filter and slice our Salesforce data. Once we've issued a query to the Salesforce API, we can use Python to manipulate the returned data if we need further filtering.

SOQL exists exclusively for querying Salesforce. As a result, almost all SOQL commands resemble the `SELECT` statement in standard SQL implementations. But unlike most SQL systems, SOQL has limited the features of the `SELECT` statement. You cannot perform arbitrary joins, for example. Here is an example SOQL query:

```
SELECT Name, Description
FROM Merchandise__c
WHERE Name LIKE 'Cranberry'
```

In cases where we have relationships defined on our Salesforce objects, SOQL allows us to perform queries on these relationships using a special dot-syntax. Imagine the `Merchandise` object queried in the previous SOQL statement included a relationship to a `Manufacturer` object. To `SELECT` based on the related Manufacturer's field information would look like this:

```
SELECT Name, Description, Manufacturer.Address
FROM Merchandise__c
WHERE Manufacturer.Name == 'CranCo'
```

This SOQL query would return the name and description of all `Merchandise` in Salesforce that was manufactured by "CranCo". It would also include the manufacturer's address field in the resulting data. When using the Python beatbox module, these results are again returned as a list of dictionaries.

The Salesforce API is relatively small. We've seen two of the most important functions in these examples: query and create. There are additional functions for deleting, updating, and even defining new object types. The beatbox module supports almost all of the functions in the API, but not every one. For example, it does not currently support the undelete function. This compatibility information is documented in the beatbox `README.txt`.

With data in Salesforce, it suddenly becomes accessible to entire teams without any additional development work. This can be very useful in small organizations whose access to developers is limited. Non-technical staff can use Salesforce to run reports and pull information in an intuitive, web-based application. Often this data would be locked-up inside our web application, accessible just to those who can perform a Django ORM call or write a SQL query.

Practical use-cases

The techniques and examples in this chapter have been varied, but all center on the primary theme of data integration. As e-commerce applications become more sophisticated and web-based businesses grow more competitive, the advanced functionality we've covered here becomes increasingly important. Django's rapid development nature and excellent community makes implementation of these tools faster and easier than ever before.

Building a RESTful data API is important for transmitting data between different systems and machines. This data could be shared across the Web or across an internal corporate network. It could support applications built in another department or in an affiliate-marketing style. A data API unlocks our information for whomever we wish to share it with.

Feeds allow a similar kind of data transmission, though on a somewhat higher level than a data API. It can let human users, in addition to machines, parse our data and is an increasingly popular delivery mechanism for consuming content.

Generating visual reports is a still higher-level form of communicating our data, specifically geared for human use. The tools we've discussed could allow an e-commerce platform to automatically send sales reports to interested members of an organization or to produce other metrics suitable for printing and hard-copy distribution.

Finally, integration with a web-based service like Salesforce.com demonstrates how our e-commerce applications can transmit, receive, and interact with other related applications. Increasingly, web-based third-party services are a cost-effective method of analyzing, managing, and working with large volumes of information. Integrating our application demonstrates Django's flexibility in this area.

Summary

In this chapter we've covered several different facets of manipulating data from our Django application. This included:

- Machine-accessible API functions
- Human and machine-readable feed exports
- Visual display output to PDF
- Transmission to and from a web-based service

We've seen that it is very easy to work with data from our Django application, convert it to new formats, transmit it across the wire, and display it in powerful ways. As Django is written in Python the ecosystem of tools becomes very liberating here. In addition to ReportLab, there are a dozen other graphical report and charting tools available in Python. We could have written an entire chapter on any one of them. This continues to demonstrate the advantages of Django as a web-framework.

8
Creating Rich, Interactive UIs

JavaScript is a browser-based programming language that allows front-end developers to build everything from simple enhancements to full-blown applications. In this chapter, we will explore the use of JavaScript to enhance web applications and integrate them with Django. These enhancements include:

- Writing effective, clean JavaScript
- Serializing Django models into JSON
- Utilizing JavaScript framework utilities
- Progressive enhancement
- Building an AJAX rating tool

The history of JavaScript implementations has been bumpy and inconsistent. We will spend the first part of this chapter reviewing some JavaScript basics and highlighting some of the language's peculiarities. Despite these problems, JavaScript is an exciting, powerful programming language that has grown from providing simple HTML enhancements to powering large, browser-based UIs.

JavaScript: A quick overview

The foundation of JavaScript includes features and philosophies from a variety of programming paradigms. It has a syntax that resembles C, advanced features derived from functional programming, and it's own, extremely powerful event-driven programming interface.

There are many web developers programming in JavaScript, but unfortunately there are a lot of bad habits around and language ignorance exists. Developers are often required to write JavaScript without adequate training or resources. This is further exacerbated by the practice of *snippet* programming, which is very prevalent amongst some web developers.

Understanding event-driven programming and the close relationship between JavaScript events and the browser are key to developing in the language. An event-driven language focuses on handler functions that execute only when the environment enters a certain state (called an event). Browsers fire events for almost everything that happens: when the mouse moves, when a link is clicked, or when a form is submitted.

We program in JavaScript by writing handler functions and attaching them to events. These are called listeners and there can be many listeners operating on the same event types. An important source of frustration for most developers is the inconsistency of event handling between browsers. This is the source of JavaScript's somewhat unfair reputation as a bug-prone language. Because the specific event API was not standardized early in web browser development, some browsers exhibit vastly different behaviors (or no behavior at all) for the same state or event.

As in many other web technologies, early Internet Explorer versions implemented drastically different versions than their competitors. These problems, which originated almost a decade ago, persist to this day because of the number of users who are still using old versions of this browser.

Fortunately there are numerous solutions that now exist to handle these cross-browser problems. The best is to employ a JavaScript framework. These frameworks have been designed to handle all of the special-case situations that are browser dependent, without the developer needing to think, or even know, about them. At the time of this writing, the most popular frameworks include jQuery YUI, Dojo, and dozens of others.

In Django 1.2, jQuery is included with the framework as part of the media files in the automatic admin module, `django.contrib.admin`. This gives jQuery a somewhat special status as far as JavaScript frameworks and Django are concerned. In the future it will likely be used to provide enhanced UI functionality in the built-in admin interface. Despite this blessing by the Django team, developers remain free to use whatever JavaScript framework they want in their own applications.

By using a framework, developers can write better, faster, more compatible JavaScript, and focus on the specific tasks at hand, not squashing browser-based bugs.

In addition to event-driven programming, JavaScript has a few other odd features that cause problems for many developers. One is the lack of modules, namespaces, and other methods of organizing code. All functions and objects defined in a JavaScript file enter the browser's global namespace. This can cause major headaches and extremely difficult debugging scenarios.

This lack of modularity has many workarounds. Some frameworks, like YUI, provide a single global object to which developers can attach their own objects and functions. Other times, developers will wrap their JavaScript files in an anonymous function that will prevent exposing local variables to the global namespace. This looks like the following snippet:

```
(function() {
// Javascript code

}());
```

The other major practical consideration most developers encounter is the division of code amongst files. Using a typical development approach, it would be easy to segment your JavaScript application into a half-dozen or more `.js` files. This is good practice in any language, including JavaScript, but when it comes time for deployment a major problem looms.

JavaScript files must be embedded in the web pages they will be used. The addition of six `.js` files to a web page will incur a significant performance hit in the form of six additional client-server HTTP requests, download times for each file, and the separate parsing of each by the browser.

Arguably, the biggest issue here is actually the multiple server requests. Many web applications will not have a need to optimize to this level, but if you do, a six-fold increase in the number of requests needed for your JavaScript files is probably unacceptable.

What is needed is a JavaScript build tool. Often these are custom, hand-written scripts specific to the project at hand, but they almost always seek to do two things: concatenate and minify.

Concatenating JavaScript files is exactly what it sounds like: taking several files and stringing them together to form a single, larger file. This single file is then served to each web page, instead of each individual file. This has the advantage of allowing developers to segment their code in a manageable way, but avoids the performance penalties when deploying to the server.

Minification is a process by which the JavaScript is shrunk. Similar to with compression, when a `.js` file is minified a variety of tactics are used. Whitespace is usually removed; long variable and function names are replaced by shortened, single character names, and so on. The original minification process was developed by Douglas Crockford and is called `jsmin.c`. It has since been ported to a variety of languages and used as the basis for other development tools, like Packer and YUI Compress.

Minification occurs in addition to any client-server compression in use with our web servers. It also has one major disadvantage: minified code is almost unreadable and effectively impossible to debug. Still, the performance and bandwidth benefits are worth it and most serious web applications should take advantage of both concatenation and minification in their build process.

JavaScript Object Notation

Another key component to modern JavaScript development is **JSON**, which stands for **JavaScript Object Notation**. JSON is a data format typically used for transmitting data over the wire from a web application's back-end to client JavaScript running in the browser. It has other uses too: it can be used as a general purpose data format, a configuration file format, and more. Usually, though, it is used in dynamic web applications.

JSON's use has grown rapidly in some web development communities. It is now preferred by many developers for data exchange and largely takes the role formerly occupied by XML in many development shops. XML was a key component in AJAX programming, which is still the commonly used acronym for these types of dynamic JavaScript applications. However, many developers have recently begun abandoning the XML portion of AJAX for JSON due to its lighter weight, direct translation path into JavaScript objects, and generally simpler implementation requirements.

In fact, JSON is essentially JavaScript's built-in object notation with some additional rules to ensure it can be properly transmitted between machines. JSON also bears a lot of resemblance to object syntax in other languages. Specifically, it looks very similar to a Python dictionary. By sticking to a simple, common syntax, JSON avoids XML's *angled bracket tax* that sometimes makes reading and writing XML data more tedious.

Python and Django include special modules for converting Python objects into JSON strings. We saw how this could be done in our discussion of `django-piston` and the API design in *Chapter 7, Data and Report Generation*.

Event-driven programming

Event-driven programming is a programming technique geared around the idea of events and listeners. An applications behavior is controlled in a non-linear fashion: when the user takes action, it results in an event. That event can have one or more functions listening to it and when it fires, those listeners go in to action.

For example, all `<a>` tags in modern browsers support a click event. This is fired when the user clicks a link embedded in the web page. Using JavaScript, developers can write a listener function that responds whenever a link in an `<a>` tag is clicked.

Usually a browser will follow a link by loading the referenced URL as a fresh page in the browser. By listening for click events, however, we can prevent this default behavior or enhance it in some way. This is useful when developing interactive tools because we can design a user interface using `<a>` tags that have no `href` attribute, listen for clicks on them, and perform updates to the UI as necessary.

Web browsers provide event hooks for almost every-minute detail of user interaction. We can detect when the user pushes down a keyboard key and when they release. We can listen for mouse movements as well as mouse button clicks. Events are fired when the browser window is changed in some way, like being resized or when the back button is pressed.

By hooking into these events we can build very powerful applications that do almost anything inside the browser that we could imagine. Event-driven programming is at the heart of JavaScript. Not all browsers support all events the same way, so it's recommended that developers working in JavaScript test their code across several browsers. There are also many references online and in book form. A particularly useful website is `QuirksMode.org`.

JavaScript frameworks: YUI

We will briefly introduce two JavaScript frameworks in this chapter. The first is **YUI**, which was developed by Yahoo! and is used in web applications throughout the Yahoo! website. It is a very extensive, very centralized, and professional framework. Its official homepage is: `http://developer.yahoo.com/yui/`.

YUI includes more than just JavaScript tools; it also provides a CSS framework that standardizes font sizes and other settings across all browsers, including a `reset.css` that sets the same set of defaults across all browsers. In addition, you currently have a choice between YUI versions 2 and 3. The two are similar, but version 3 differs enough from its predecessor that tutorials and other web-based resources are not completely caught up.

YUI takes a more *kitchen sink* approach than some libraries. It includes a lot of very powerful built-in utilities, as opposed to using extension plugins. These utilities include everything from standardized event listener routines to full UI widgets, such as auto-complete fields, dialog panels, and data tables. It also includes very high-quality, extensive documentation.

As mentioned earlier, concatenating and minifying JavaScript is important to improve script performance and simplify development. YUI includes a configurator tool that automatically constructs the appropriate `.js` file for use in our web application's `<link>` tag. This tool lets us choose individual components and rolls them up into a single file, which can be served directly from the YUI site.

JavaScript frameworks: jQuery

JQuery is another excellent JavaScript framework. It takes a slightly different approach from YUI in that many of the utilities are provided through plugins, often developed by community developers. The community around jQuery is very large, the code is of an extremely high quality, and the documentation is generally very good. There is also a great deal of print and web-based resources dedicated to this framework.

The jQuery philosophy is to be efficient and fast, allowing developers to write the least amount of code necessary. The hallmark of any jQuery script is the $ object. This is the main interface to jQuery and is used all over the place. Its primary function is to act as a DOM selector. This means it selects and returns elements from the HTML document within which our jQuery script executes. To select all anchor tags, for example, we can use the following syntax:

```
$("a")
```

Developers new to jQuery's way of doing things may be surprised at first by how easy it can be. The best place to begin learning the framework is to work through the tutorials at `http://docs.jquery.com/Tutorials`.

Graceful degradation and progressive enhancement

Graceful degradation and progressive enhancement are two terms used to describe an effective approach to developing web-based applications using JavaScript. The goal is to obtain compatibility with the most browser configurations as possible while providing as much enhanced functionality as we can.

There are a lot of potential risks when developing in JavaScript and the biggest is that different browsers implement their own version of the JavaScript and DOM APIs. Recent browsers do a great job of standardizing their implementation to be as compatible with other browsers as possible. But older browsers differ wildly.

By employing graceful degradation and progressive enhancement techniques, you can ensure that even when your JavaScript code breaks, the user is still able to accomplish their task.

The other goal is to work with browsers that do not support JavaScript at all. This may seem odd at first; after all don't all modern browsers support JavaScript? Unless the user has disabled support on purpose, they should be able to parse our scripts.

While true, this point of view is short-sighted because it excludes the increasing population of users who are interacting with web applications using devices other than a computer. This includes mobile devices, everything from iPhones to basic cell phones, which have very different implementations of JavaScript.

Many other new devices are web enabled and more become so every day. Video game consoles have web browsers built-in, as do some television sets. With new kinds of internet appliances, refrigerators with LCD screens, clock-radios, and mysterious Apple tablets, the list continues to increase.

Designing for such myriad products is only going to get more difficult and JavaScript really complicates the matter. The best bet to support the widest range of devices is to employ progressive enhancement. Remember that the applications we build today could exist for years beyond what we expect; we want to ensure they'll be compatible as much as possible with whatever comes next.

To build a progressively enhanced version of your application requires a fair amount of design and planning. But the benefits, especially over the long term, can be decreased maintenance, better user experience, and higher-quality code.

Creating product ratings

One of the innovations in e-commerce web applications has been the use of user-generated content in the form of product reviews and ratings. We've seen some of this in our implementation of customer reviews earlier in the book: users could write short comments about any product in the product catalog. This is too much unstructured information, however. Numeric or *star* ratings are a more structured alternative that can be provided by users more simply. We can use this as a form of feedback, too, and generate an average rating for all of our products.

Ratings are intended to be quick, easy feedback for customers who wish to contribute, but are not interested in writing a full comment. Many ratings examples exist and it is now a well recognized and even expected idiom for web-based product catalogs.

Using traditional HTTP `POST` forms, though, does not provide the user experience we would like. It is almost a requirement that we use JavaScript for this type of interaction because users have grown so accustomed to the instant, uninterrupted experience.

In addition to the usability needs of our users, we also need to make sure we capture the rating information in our Django backend. JavaScript and AJAX techniques are the only universal mechanisms of providing this functionality across all browser technologies.

The ratings tool will divide cleanly into two separate aspects: the Django module that lives on the backend and the JavaScript that lives in the browser. We will create our Django module first as a new application in our `coleman` project, which we will call ratings.

The ratings app will include an extremely simple model that allows us to associate a numeric rating submitted by a user with one of any of our products. In the interest of reuse, we will not construct a `ForeignKey` directly to our product model, but instead use Django's `contenttypes` framework to create a `GenericForeignKey`. This allows us to rate objects other than just products. Perhaps later we'd like to allow users to rate manufacturers or other entities as well.

Our rating model looks like this:

```
class Rating(models.Model):
    rating = models.IntegerField()
    content_type = models.ForeignKey(ContentType)
    object_id = models.PositiveIntegerField()
    content_object = generic.GenericForeignKey('content_type',
                                               'object_id')
```

Next we need to build a view through which our JavaScript code can interact with Django, submitting a rating from a user that Django will record as a new `Rating` object. Django's `HttpRequest` objects allow us to test whether our view is accessed via AJAX (that is using `XMLHttpRequest`) or using a standard HTTP request. This method is called `is_ajax()` and is available on any `HttpRequest` object.

There are some cases where a view would be designed to handle either an AJAX-style request or an HTTP request. By checking `is_ajax`, we could branch and return JSON when AJAX is used or render a standard template in the non-AJAX case. For our simple rating view, however, we will only handle AJAX calls and raise a not found message otherwise. The following simple example illustrates this technique:

```
def myview(request):
    if request.is_ajax():
        # do something ajax-y
        return HttpResponse(...)
    else:
        raise Http404
```

Design aside: User experience and AJAX

The previous code segment is very simple, but it has an important implication: we are only handling the AJAX case. That means the interface for ratings in our application will only work for users who have JavaScript enabled. This could violate graceful degradation so we must be careful how we implement it in the browser.

This example is intended to illustrate a point about the Web and user experience: it's often difficult to implement advanced functionality in browsers without JavaScript. But it is important to provide these users with an excellent browser experience. Without AJAX, our star-based rating system would require several page loads and an awkward form. In the early days of the Web, this would be commonplace, but on modern sites they're rare and confusing.

Distracting the user with two click-throughs and a form to fill out distances us from the rating tool's original goal: to allow for quick and simple customer feedback. In e-commerce applications, such distractions could be costly. It's a philosophical decision, but in many cases it would seem better to drop the functionality altogether than to wreck the user experience.

An excellent way to design for these user-experience issues is to examine what others do. Netflix and Amazon produce some of the best AJAX-powered interfaces on the Web. Simply disabling JavaScript in your browser and using sites such as these can reveal a whole other world. You'll notice functionality missing that you may be used and sometimes functionality is replaced with a gracefully degraded interface.

Another factor to consider in all of this is development time. If your non-AJAX interface adds twice the development cost and actually hurts the user experience, it is probably a bad business decision. On the other hand, if your functionality is critical and must support all browsers, but you'd like to offer an enhanced version to JavaScript-enabled browsers, then the effort may well pay off.

In fact, this is really what graceful degradation in web interface design should be about and is often called *progressive enhancement*. First build it without JavaScript or any expectation that it will ever use AJAX. When it's built and working, layer AJAX on top, perhaps by adding an autocomplete field instead of a regular text field or make the form submission asynchronous so that the user's browser does not advance to a new page. Working backwards like this is a great way to ensure your web interfaces work well for all users.

Often you don't care about this or you're building something so complicated that it cannot be done through any other means. As the population of users without JavaScript is relatively small, it is becoming increasingly common to ignore them for complex tools. Be warned, though, that many mobile-enabled browsers have no JavaScript support and if your application is at all targeted toward mobile users, you should ensure to accommodate their entire browser.

This is an increasingly difficult issue, because supporting the whole range of browsers is difficult, costly, and time consuming. When planning a web-based application, this should be a major consideration. It can be difficult to restrict development from the cutting edge just to ensure the small percentage of users without JavaScript can use it. The ramifications of failing to do so will differ across businesses, though, so the decision should be made specifically for one's own application in their own industry.

Product rating view

With these design considerations in mind, we can begin to construct our rating view. Because our aim is to keep the ratings app reusable, our rating view should be written to support ratings for any Django model type. Our rating model does this by using `django.contrib.contenttypes` and `GenericForeignKey`.

There are a variety of ways to handle this situation, but almost any solution for the view will also involve the `contenttypes` framework. We can construct the view in such a way as to include an `object_id` and `content_type` parameter. With these two parameters we can obtain the `ContentType` object and do with it whatever we need. We've seen some of this before and it is a common need in Django applications.

Our ratings app `views.py` file will appear as follows:

```
from django.contrib.contenttypes.models import ContentType
from django.core.serializers.json import DjangoJSONEncoder
from django.http import Http404
from django.db.models import get_model
from coleman.ratings.models import Rating


def lookup_object(queryset, object_id=None, slug=None, slug_
field=None):
    if object_id is not None:
        obj = queryset.get(pk=object_id) elif slug and slug_field:
        kwargs = {slug_field: slug}
        obj = queryset.get(**kwargs)
    else:
```

```
            raise Http404
        return obj

    def json_response(response_obj):
        Encoder = DjangoJSONEncoder()
        return HttpResponse(Encoder.encode(response_obj))

    def rate_object(request, rating, content_type, object_id):
        if request.is_ajax():
            app_label, model_name = content_type.split('.')
            rating_type = ContentType.objects.get(app_label=app_label,
                                                  model=model_name)
            Model = rating_type.model_class()
            obj = lookup_object(Model.objects.all(), object_id=object_id)
            rating = Rating(content_object=obj, rating=rating)
            rating.save()
            response_dict = {'rating': rating, 'success': True}
            return json_response(response_dict)
        else:
            raise Http404
```

The important part is the view itself: `rate_object`. This view implements the `ContentType` retrieval we've been talking about and uses it to create a new rating object. The rating view parameter should be a positive or negative integer value, though our application logic could use text values that are converted to appropriate integers if we were interested in writing cleaner URLs.

The `json_response` helper function takes an object and serializes it to JSON using the built-in `DjangoJSONEncoder`. This encoder has support for encoding a few additional data types such as dates and decimal values. There is room for improving this function later, by implementing our own custom JSON encoder class, for example, so it's helpful to breakout this code into a small function.

Finally, the `lookup_object` helper function is something we've seen in earlier chapters. Its job is to retrieve a specific object from a `QuerySet` given an `id` or `slug` value. There are many variations on this function; we could have passed a `Model` class instead of `QuerySet`, for example. But the primary goal of all of them is to perform generic lookups.

With our view in place, we can now create rating URLs for all of our Django objects. We can integrate these URLs into a star-rating tool in our templates and ultimately manage their use using JavaScript.

Constructing the template

Creating a rating tool in our template has three components: the tool's image content (stars or dots, and so on), CSS to activate the appropriate rating on hover, and the JavaScript that handles clicks and submits the rating to our Django view. This section will focus on the first two tool elements: the image and CSS.

There are a variety of star rating CSS and image examples on the Web. Some are licensed for reuse in applications, but not all. The technique detailed in this section is a simplified version of the one demonstrated by Rogie at Komodo Media at the following URL: `http://www.komodomedia.com/blog/2005/08/creating-a-star-rater-using-css/`. Rogie has written about other approaches to this problem, including some with better cross-browser support. See the tutorials at `komodomedia.com` for additional information.

We have created the stars images for this book using a simple graphics editor and the star character from the Mac OS X webdings font. The star has two states: inactive and active. Thus we need two images, one for each state. The star images are shown below:

With our stars created, we can begin developing a set of stylesheet rules that reveals our creation. The CSS rules need to decorate `<a>` tags so that we have a hook for our JavaScript code later. Let's start with the following basic HTML, which we will enhance as we go:

```
<a href="">Rate 1 Star</a>
<a href="">Rate 2 Stars</a>
<a href="">Rate 3 Stars</a>
<a href="">Rate 4 Stars</a>
```

Notice our `href` attribute is not yet filled in. We will be using Django's `{% url %}` template tag to handle this URL, but for clarity it will be omitted until later. The next step is to add class declarations for each of the four types of rating actions:

```
<a class="one" href="">Rate 1 Star</a>
<a class="two" href="">Rate 2 Stars</a>
<a class="three" href="">Rate 3 Stars</a>
<a class="four" href="">Rate 4 Stars</a>
```

Adding CSS classes to our `<a>` tags gives us a place to begin when styling our star rating tool. We will also want to wrap the whole set of rating links in a `<ul>` tag and each individual link in `<li>`:

```
<ul class="rating-tool">
   <li><a class="one" href="">Rate 1 Star</a></li>
   <li><a class="two" href="">Rate 2 Stars</a></li>
   <li><a class="three" href="">Rate 3 Stars</a></li>
   <li><a class="four" href="">Rate 4 Stars</a></li>
</ul>
```

Wrapping the links as an unordered list lets us make sure the links are displayed horizontally, as a set of side-by-side stars.

Now that we have our HTML ready, we can begin writing CSS rules. The first rule applies to our `<ul>` class. It will let us define the size of our rating tool and sets the background to our inactive star image.

```
.rating-tool {
    list-style-type: none;
    width: 165px;
    height:50px;
    position: relative;
    background: url('/site_media/img/stars.png') top left repeat-x;
}
```

Because our star image is 40 pixels wide and 50 pixels tall, the size of our rating tool, which will support four stars, is set to 165 pixels wide.

Next we style the `<li>` elements so that they align horizontally instead of vertically. There are several ways of achieving this, but here we will use the `float: left` rule:

```
.rating-tool li {
margin:0; padding: 0;
float: left;
}
```

With our `<li>` tags style appropriately, we can start styling the `<a>` tags themselves. These tags are the crux of our rating tool. We must ensure several things to make this work: that the link text is hidden, that each `<a>` width is set to match the width of an individual star, and that we position the `<a>` completely within our rating-tool `<ul>`. We will also set the `z-index` rule, which will function as an image mask:

```
.rating-tool li a {
    display: block;
    width: 50px;
    height: 50px;
    text-indent: -10000px;
    text-decoration: none;
    position: absolute;
    padding:0;
    z-index:5;
}
```

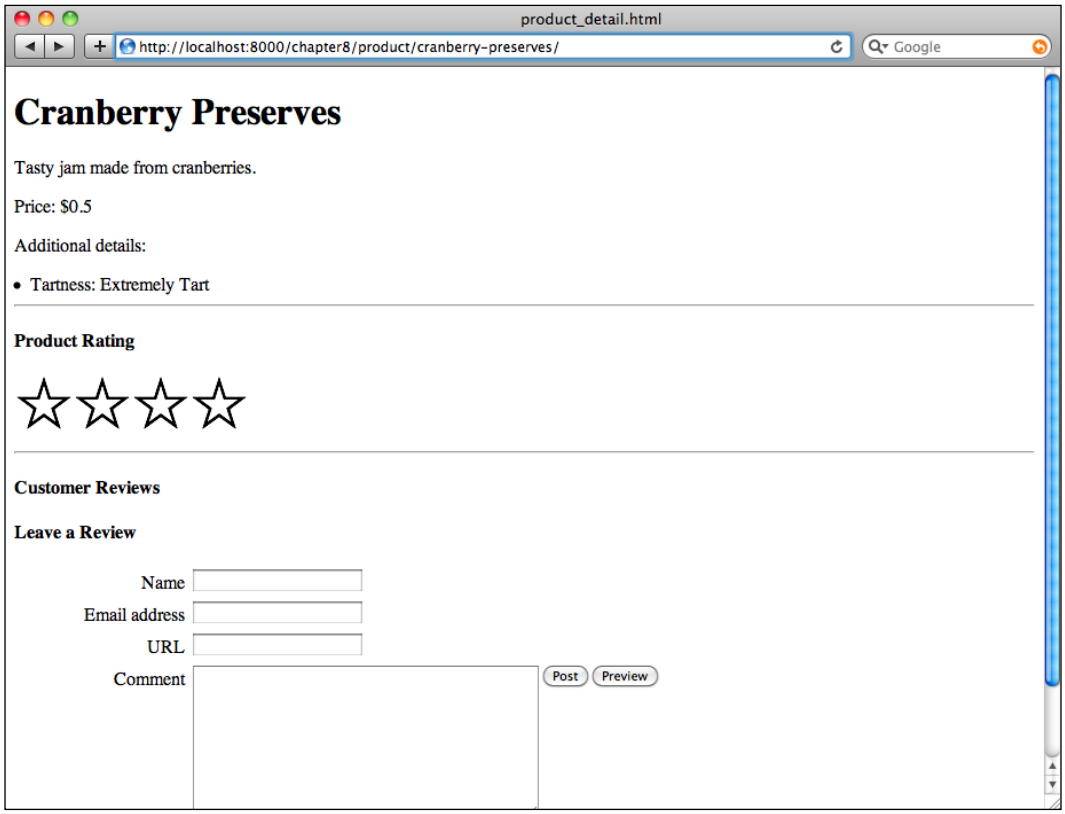
Each `<a>` also defines a `:hover` pseudoclass. When the link is hovered, the background is changed using a CSS image replacement technique and the `z-index` mask is adjusted to show only the portion of stars currently hovering. In other words, when the user hovers over the second star, they should see both the first and second stars activate. This is completed with the help of the next set of rules:

```
.rating-tool li a:hover {
    background: url('/site_media/img/stars.png') left bottom;
    z-index: 1;
    left:0px;
}
```

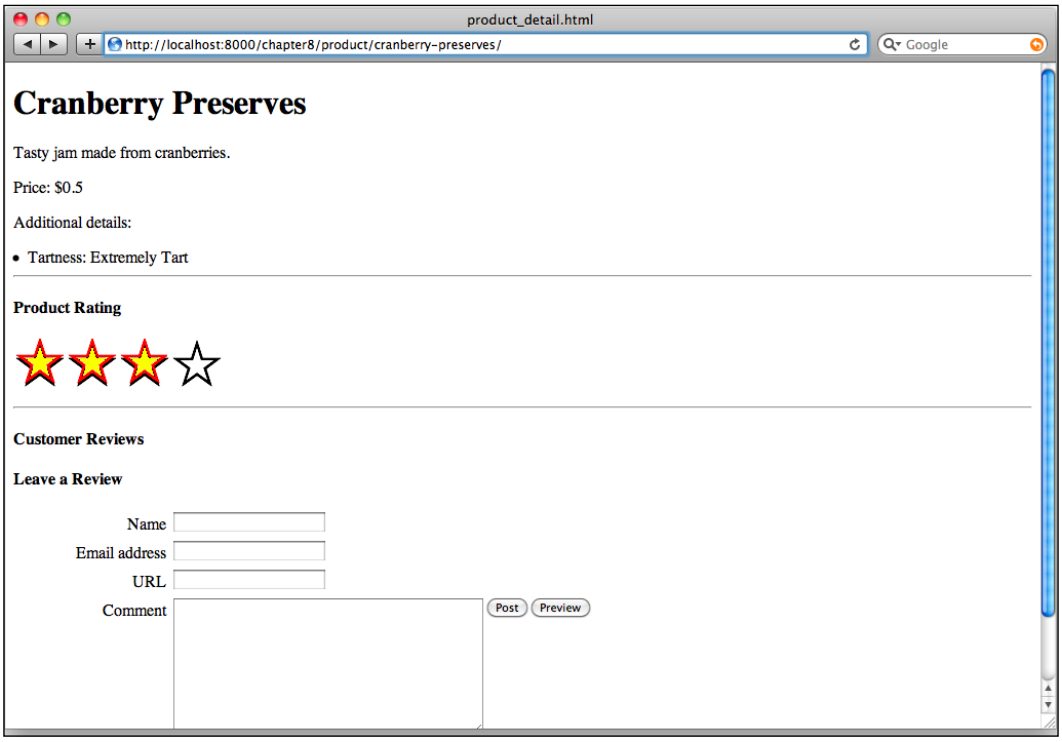
Each anchor tag should define a new width for itself specific to which star we're revealing. The fourth star is going to reveal the entire rating tool by setting a width of 200 pixels when hovered. These rules work in conjunction with the previous, more general set of rules.

```
.rating-tool a.one { left: 0px; }
.rating-tool a.one:hover { width: 50px; }
.rating-tool a.two { left: 50px; }
.rating-tool a.two:hover { width: 100px; }
.rating-tool a.three { left: 100px; }
.rating-tool a.three:hover { width: 150px; }
.rating-tool a.four { left: 150px; }
.rating-tool a.four:hover { width: 200px; }
```

This finishes the CSS rules for our rating tool. You can see the results of this design in the following screenshots:



The activation state of the rating tool reveals yellow stars when the mouse hovers over the rating control, as in the following screenshot:



This is just one of many possible techniques for implementing a rating tool. It works well because it uses a pure combination of HTML and CSS for the graphics effect.

One last step is required before we can begin work on the JavaScript code that will power our rating tool. Earlier we mentioned that we purposefully excluded the `href` attribute for our `<a>` tags. We now need to wire these up to our Django view.

To do this we will use Django's built-in `{% url %}` tag and a named URL pattern. Our rating view from the previous section lives in `coleman.ratings.views`. Our `urlpatterns` will then look like this:

```
urlpatterns = patterns(
    '',
    url(
      r'^(?P<content_type>[^/]+)/(?P<object_id>\d+)/(?P<rating>[1-
5])/$',
      'coleman.ratings.views.rate_object',
      name='rate_object'),
 )
```

Using the `url()` function, we can create a named URL pattern, which we've called `rate_object`. Using a named URL pattern means we can reference it easily from templates using `{% url %}`.

The `{% url %}` template tag takes a path to a view or the name of a URL pattern. For example, we could have written:

```
{% url coleman.ratings.views.rate_object ... %}
```

But when a view could potentially be used in multiple URL patterns, it's easier to use a named URL:

```
{% url rate_object ... %}
```

Using the `{% url %}` tag is an important habit because it prevents the hard coding of absolute URLs into HTML templates. This will save tremendous amounts of time if we ever decide to change our view or change the layout of URLs on our site. Instead of manually having to rewrite each `href` attribute that linked to the changed URL, Django will do it for us automatically.

The `{% url %}` tag also takes arguments to be passed on the URL line and will construct the appropriate URL using these arguments. For example, our `rate_object` view takes three arguments: `content_type`, `object_id`, and `rating`. These correspond directly to our view function arguments. A rating URL may look like this:

```
/rating/products.Product/125/4/
```

This translates to giving a four rating to the `Product` with the primary key `125`. We could hard-code this in using template variables, but instead we'll use the `{% url %}` tag. Assume that the template receives a template variable named object that contains the `Product` we will be rating. An equivalent `{% url %}` tag will look like this:

```
{% url rate_object content_type='products.Product' object_id=object.
id rating=4 %}
```

Notice how we are still able to reference template variables, like object, in the tag.

Now that we have a working `{% url %}` tag, we can integrate into the HTML for our rating tool. Each star will correspond to a rating with a matching value from one to four. The Django template code will look like this:

```
<ul class="rating-tool">
  <li>
   <a class="one"
     href="{% url rate_object content_type='products.Product'
```

```
                    object_id=object.id rating=1
                %}">Rate 1 Star</a>
    </li>
    <li>
     <a class="two"
       href="{% url rate_object content_type='products.Product'
                   object_id=object.id rating=2
                %}">Rate 2 Stars</a>
     </li>
     <li>
       <a class="three"
         href="{% url rate_object content_type='products.Product'
                     object_id=object.id rating=3
                  %}">Rate 3 Stars</a>
     </li>
     <li>
         <a class="four"
           href="{% url rate_object content_type='products.Product'
                       object_id=object.id rating=4
                   %}">Rate 4 Stars</a>
    </li>
</ul>
```

Writing the JavaScript

We will build our JavaScript rating tool using YAHOO's YUI JavaScript utilities discussed earlier in the chapter. These utilities simplify the code considerably and insure against cross-browser compatibility problems.

In order to take advantage of the YUI library, we have to embed it in our templates using a `<script>` like normal JavaScript code. The following `<script>` will bundle all of the YUI utilities we'll be working with:

```
<script type="text/javascript" src="http://yui.yahooapis.com/
combo?2.8.0r4/build/yahoo-dom-event/yahoo-dom-event.js&2.8.0r4/
build/connection/connection_core-min.js&2.8.0r4/build/json/json-
min.js"></script>
```

That long string of a URL in the `src` attribute is generated using YUI's excellent dependency configurator. This will automatically choose the most efficient script tag to embed in your application based on the YUI modules you are using. You can access the configurator via: http://developer.yahoo.com/yui/articles/hosting/.

Our JavaScript code is divisible into three parts: the main routine, which listens for the browser's ready event and kicks off our other code, a function to set up our click listeners, and a handler routine that takes action when a click is detected.

The main routine is very simple and is run as soon as the browser loads our JavaScript file from the server. This should happen late in the page load process and we can take an extra step to encourage this by embedding our `<script>` tag at the end of our template's `<body>`.

When the script loads, we will instruct YUI to set up another listener. This time we'll be listening for a `DOMReady` event. This is one of the ways YUI helps us: the `DOMReady` event is a YUI construct that will fire when our page's DOM has stabilized. This means the elements have rendered and the HTML has been fully parsed. This is part of YUI's Event utility package and we use it like this:

```
YAHOO.util.Event.onDOMReady(init);
```

The `init` part is a function we will write called init. When the YUI tool has detected the DOM is stable, this code will call init and execute its function body. Let's write the `init` routine now:

```
function init(e) {
    var rating_tool = document.getElementById('rating'),
    rating_links = rating_tool.getElementsByTagName('a');

    for (var i=0; i < rating_links.length; i++) {
         YAHOO.util.Event.addListener(rating_links[i], 'click',
handleClick, i+1);
    }
 }
```

The `init` function gathers up the `<a>` elements on our pages that are used for our rating tool. It does so by finding the `<ul>` tag, to which we've added the ID attribute rating. It then loads all the `<a>` elements within the rating tool's `<ul>` tag and begins to process them.

As the code loops through all of our `<a>` ratings, it attaches to each one an event listener. These listeners all employ the same function: `handleClick`. It also includes the counter variable `i`, which we use to keep track of which stars were clicked.

At this point code execution stops. Nothing else happens during the page load process and our user can go about their business. Eventually, they will find a product they like and attempt to give it a rating. When they click on the `<a>` tag corresponding to their star rating, it will activate the handler we set up in the `init` function.

These click handlers are where the magic happens. They need to do several things. First, they prevent the browser from taking the usual action when you click on an `<a>`. This is typically to load the next page, linked to via the `href` attribute. If we do not prevent the browser from doing this, we'll transport the user to a possibly broken URL or otherwise break our JavaScript.

YUI will automatically provide the `handleClick` function with its first argument. This is an event object that contains information about what event has fired and what object it has fired on. It is a very useful object and we will need it shortly.

It is possible to prevent default behaviors using pure JavaScript, but the YUI routines perform better across all browsers. Events and event handling is one area where browsers can differ greatly. The YUI Event utility provides a `preventDefault` function that we can use to stop the browser from performing its usual action:

```
YAHOO.util.Event.preventDefault(e);
```

In the above code, `e` is the event object YUI has passed to our handler function. This call to `preventDefault` on the event object will cease the browser's default behavior as soon as our handler completes execution.

The next task our handler needs to perform is to submit the rating to our Django backend. This will finally link the frontend to the backend code we wrote earlier. We want to submit this information asynchronously. This means the browser will send off the request to our Django `rate_object` view and then go about its business, allowing the user to continue to interact with our application.

When Django has finished processing the request, the browser will be notified and an event is fired. This event is handled by code we set up at the time of the asynchronous call. We will use YUI's Connection Manager to perform this operation, which greatly simplifies this process.

First, we need to define our callback handler. YUI's Connection tool uses a JavaScript object with success and failure properties. These properties are functions that will run in the case of a successful response from the server or an unsuccessful one, respectively. Unsuccessful asynchronous calls are the result of the standard kinds of HTTP failure conditions: `404` and `500` errors, especially.

The `callback` handler object will look like this:

```
callback = {
    success: function(o){
        var result = YAHOO.lang.JSON.parse(o.responseText);
        if (result.success == true) {
             alert("You've rated this object " + result.rating);
        }
```

```
        },
        failure: function(o){
             alert("Failed to rate object!");
        }
    };
```

Note that we've taken a very simplified approach here, notifying the user of their rating via `alert()`. A production setup would likely want to take additional steps to update the UI to indicate a rating had been made, possibly by modifying the CSS rules we detailed earlier.

There are a couple of other things going on as well. First, both the success and failure functions receive an object `o`. This object contains information about the asynchronous call. To access the raw server response in either case, for example, you can use the `o.responseText` property.

Our success function employs another YUI tool, the JSON utility. This includes a safe JSON parser. Our `o.responseText` will contain JSON data but will contain it as a raw JavaScript string. One way to convert this JSON to actual JavaScript data objects is with `eval`. But `eval` can be dangerous and YUI's JSON utility provides us with a safe parse function to evaluate the server's response.

Once evaluated, the result variable contains a standard JavaScript object with properties translated from the JSON sent back from the server. We can use this as we would any JavaScript data.

With our `callback` handler in place, we can return to the Connection Manager call we began to make a few paragraphs ago. This call needs two other pieces of information: the HTTP method to use and the URL to send our asynchronous call. The method is simply the usual HTTP `GET` or `POST`. Our Django view has no particular method requirement so we will use POST.

The URL part is more difficult. Remember when we developed our Django template we included the rating URL using the `{% url %}` template tag. We can now extract that URL and use it with our connection manager function.

To access the appropriate URL, we again use our event object provided by the event utility. Using the YUI event module's `getTarget` function, we can obtain the `<a>` tag that was clicked. This `<a>` tag will contain the URL we need in its `href` attribute:

```
var target = YAHOO.util.Event.getTarget(e)
```

We can access the `href` via the `getAttribute` DOM method:

```
var url = target.getAttribute('href')
```

We're now ready to send off our asynchronous request, by calling the YUI Connection Manager's `asyncRequest` function:

```
YAHOO.util.Connect.asyncRequest('POST', url, callback)
```

The complete JavaScript routine for our rating tool appears as follows:

```
(function(){
    function handleClick(e, count) {
        var target = YAHOO.util.Event.getTarget(e),
        url = target.getAttribute('href'),
        callback = {
           success: function(o){
              var result = YAHOO.lang.JSON.parse(o.responseText);
              if (result.success == true) {
                alert("You've rated this object " + result.rating);
             }
           },
           failure: function(o){
                alert("Failed to rate object!");
       }
    };

   YAHOO.util.Event.preventDefault(e);
   YAHOO.util.Connect.asyncRequest('POST', url, callback);
   };

  function init(e) {
       var rating_tool = document.getElementById('rating'),
       rating_links = rating_tool.getElementsByTagName('a');
       for (var i=0; i < rating_links.length; i++) {
            YAHOO.util.Event.addListener(rating_links[i], 'click',
                                         handleClick, i+1);
     }
   }
YAHOO.util.Event.onDOMReady(init);
}());
```

Debugging JavaScript

JavaScript is notoriously difficult to debug. In our rating tool example there are many potential failure points. Primary among them is what happens when Django fails to return a correct response. We will be notified of this case because of the failure function in our `callback` object.

But how do we debug the problem? There is an increasing amount of tools available to perform debugging of this sort. Two of the simplest and easiest to get installed are FireBug, for FireFox, and Safari's Web Developer functions, which now come with Safari 4.4 and Google Chrome.

These tools let you inspect the HTML DOM, see the JavaScript error console, and see the results of XHR calls, such as those made by `asyncRequest`. Any error page that Django generates as a result of an AJAX call will be returned to these web development debuggers in the `resources` section and can be inspected to determine what exception Django has thrown and where it occurred in our view code.

Another common practice is to insert debugging `alert()` statements at critical points in the JavaScript code. This is not an elegant method, and one has to be sure to remove any debugging statements when pushing code to production use, but it is quick and effective.

If using a JavaScript debugging tool such as Firebug (see above), another popular debugging technique involves the `console` global object. This is a global variable inserted by debugging tools to allow advanced debugging functionality. It can be used similarly to alert statements by calling `console.log()` and passing a string to log as debugging output. There are many additional functions in the console API, though specific features may or may not be available depending on the browser or debugging tool in use.

In addition, most JavaScript frameworks include their own set of tools for debugging applications. See the documentation for your framework of choice to get more information.

Summary

This chapter has presented a simple example of using Django with enhanced, AJAX-based user interfaces in the browser. There is much, much more to this topic. In fact, entire books exist on this material alone.

We have given quick coverage to several important topics:

- Writing Django views for AJAX applications
- Building a functional JavaScript interface enhancement
- Using the YUI framework for a real-world AJAX tool
- Exploring the basic issues involved with JavaScript development

Django makes it very easy to build AJAX functionality into your project. Writing successful JavaScript is harder, however, and it is highly recommended that those who are interested in doing more pick up one of many excellent books available on the language.

9
Selling Digital Goods

Selling digital goods and content presents unique challenges for e-commerce developers. Though we don't need to worry about calculating shipping and coordinating physical delivery, we do need to consider new factors, such as securing download links and quickly verifying payments. This chapter will examine the following advanced topics:

- Using Amazon S3 for secure content-storage and delivery
- Adding S3 storage to Django's file and image fields
- Creating functions for working with Amazon's Aggregated Payments Service
- Examining the use of Digital Delivery with Google Checkout

Different types of digital sales have different requirements. Selling a subscription service is simpler than selling digital music or video files, for example. We will begin with a survey of these two kinds of sales and present the technology considerations for each. We'll then examine the features of Amazon and Google's digital payment services and develop a rudimentary infrastructure for handling them in Django.

Subscription sales

A subscription-based sale describes a situation where our web application provides access to a service for a fee on a recurring basis. This could be monthly, but it isn't always. In the digital world, weekly or even daily subscriptions might make sense.

The issue of payment is complicated in subscription sales by automatic renewals. Until recently, recurring subscription payments were not well supported in popular payment processors. This is changing quickly and both Amazon FPS and Google Checkout now offer built-in support for this type of payment.

A big advantage of subscription support in a payment service provider is that the renewal process is handled automatically. The customer is billed and given a receipt, while our applications are notified of the new order. Using some of the payment processing techniques discussed earlier in this book, very little human intervention is needed. The payment processor can also provide reminders, history, and other information to subscribers about their payments.

Subscription services are useful in a wide variety of applications. The example that immediately springs to mind is a content site that charges for access to premium content. But this doesn't necessarily need to be text-based content. For example, this type of payment is often used for access to web or desktop-based software. It could also provide access to video or audio content, through a web browser or desktop application.

Our applications for this type of service would need to keep track of user access level and subscription status. When their subscription expires, the user's account should be updated to prevent access. In Django, the built-in permissions system is rather simple, but could be used to satisfy this need.

More complicated subscription tracking could be handled using a user profile model or specialized Django application that tracks and manages subscription access.

Digital goods sales

Selling digital goods is trickier than subscription sales. For one, the user normally downloads a digital asset after making their purchase. This may be a music or video file, but it could also be a software download, a serial number and/or a registration key.

Providing downloads complicates the matter in part because of potential for delays in payment authorization. When selling a physical product, there is usually time between the purchase request and packaging and shipping. This interval allows the payment processor to authorize and charge payment.

Most users expect a digital download almost immediately, leaving us little time to verify whether the user's payment is legitimate. This opens the possibility for fraud or other failure states. Fortunately, the big payment processing services have taken many steps to solve these problems with little inconvenience to the customer or our application's usability.

After purchase, customers are typically given a special URL download link, either in a response web page or through an e-mail. They can then download their purchase from this link. This necessitates a storage scheme where only authorized, paying customers are able to download content.

It's also helpful to provide customers with the fastest possible download, not just for their immediate convenience, but also to provide confidence in the service, encourage repeat sales, and to cut down on support costs due to failed downloads. A **Content Delivery Network** (**CDN**) is one possible solution to this problem, which we will discuss later in this chapter.

Finally, there is the issue of securing the download. Often this involves providing a special, one-time download link to the customer. This link ensures that our digital good is only available to paying users. We will explore one method of doing this using Amazon's S3 file storage service and their CloudFront content delivery service.

Content storage and bandwidth

Storage and bandwidth are important concerns when selling digital goods. Digital content sales tend to involve the transfer of a significant amount of data. Video is especially data intensive, as even with modern compression codecs, file sizes still amount to hundreds of megabytes.

Even a moderately popular video download can quickly consume many gigabytes of bandwidth. Imagine our fictional Cranberry Merchant website decides to sell a short, 20-minute instructional video on Cranberry farming. The video is compressed using a high-performance H.264 codec and is sold as a direct download (as opposed to streaming).

The video quickly becomes a small hit and soon hundreds of amateur Cranberry farmers are clamoring to learn from it. Over 200 downloads were sold in the first day it launched. Each 30-minute, high-definition video file weighs in at 800MB. Video sales consumed 160GB of bandwidth in a single day.

Every web hosting provider offers different bandwidth pricing. Some claim to offer unlimited or several terabytes of bandwidth on their inexpensive shared hosting plan. Buyers beware: it's unlikely these services would allow the relatively simple scenario outlined in the last paragraph without seriously throttling connections or shutting off the hosting account altogether.

More serious hosting providers typically offer a soft capped bandwidth service included in the monthly hosting fees. Beyond this cap, the hosting provider usually charges an overage fee per gigabyte. As of this writing, $0.30 cents per GB is a common charge in these situations. In other cases no bandwidth is included in the hosting plan at all and all bandwidth is charged by the gigabyte. These plans typically hover around the $0.20 cents per GB range.

Let's assume our hosting provider offers 50GB per month in bandwidth, then charges $0.30 cents overage. In a single day our overage usage hit $33.00 and for the rest of the month we'll be paying for all the bandwidth we use. It should be noted, too, that service providers offering plans like this may also struggle to keep up and could end up throttling our customer's download connections if they occur all at once, or take other steps to reduce the load on their networks.

In the alternative example, where we pay for all the bandwidth, our first day of sales would rack up $32.00 in bandwidth fees at $0.20 per GB.

Bandwidth costs have implications for how digital goods are priced. Though most of the time the sale price should easily cover bandwidth charges, it is still very important to understand the pricing structure for our hosting provider. What if in addition to for-sale video products, we also offered several free instructional videos available for download or streaming? The consumption pattern of these videos will likely differ greatly and must be accounted in our costs. If using the same hosting system for website materials, including HTML, image files, and the like, this bandwidth must also be included. Ideally the sale price of our video would be able to subsidize the bandwidth required for the rest of the site.

As we can see, different situations can result in very complex cost structures. Some applications choose to offload heavy content files, like video, to a service like Amazon S3. At the time of writing, S3 charges begin at $0.17 per GB and can go as low as $0.10 for extremely high volume users. This is among the best available. Whether this is the most economical approach depends on a lot of factors, but the difference between $0.10/GB and $0.30/GB overage represents a potential two-thirds cost savings.

We will explore how to offload digital content files to Amazon S3 using Django, including securing them as private download links in the next section.

Django and Amazon S3

By default, Django models with `FileField` use local filesystem storage for uploaded files. Recent versions, however, have implemented an excellent pluggable storage system. This means we can substitute our own file storage mechanism for the default filesystem storage.

Custom storage can be implemented for almost any backend storage system. Essentially anything Python can connect with and store data on can be used as a storage backend. For many popular storage services, community plugins already exist. One outstanding project is David Larlet's **django-storages**, available at:

`http://code.welldev.org/django-storages/`.

The django-storages application supports lots of popular storage services and even allows us to store our files in a database. For the purpose of this section, however, we are interested in its support of Amazon's S3 storage service.

S3 stands for **Simple Storage Service**. It is a product from Amazon's Web Services group that includes extremely competitive pricing for storage and bandwidth, a high availability around the world, and special integration with the Amazon CloudFront content delivery network. It also supports permissions-based authentication mechanisms and a service-level agreement that covers very rare downtime.

Amazon S3 can be accessed using either a REST or SOAP API. Wrapper libraries are available in almost all web-development languages, including Python. The django-storages app mentioned bundles everything needed to use S3 as a storage module. For developers interested in more direct access to S3 functionality, there is a community project called **boto**, available on Google Code at: `http://code.google.com/p/boto/`. In addition to S3, boto supports operations for the other Amazon Web Services tools.

In part due to the excellent design of the S3 storage backend included with django-storages, integrating S3 support into our Django models is relatively easy. In most cases all that are required are a few additions to our Django settings file. These settings specify the access keys for our Amazon S3 account, provided upon sign up with the service.

An example settings configuration looks like this:

```
DEFAULT_FILE_STORAGE = 'backends.s3.S3Storage'
AWS_ACCESS_KEY = 'xxxxxxxxxxxxx'
AWS_SECRET_KEY = 'xxxxxxxxxxxxx'
AWS_STORAGE_BUCKET_NAME = 'content4sale'
```

These settings and the django-storages app are all that we need to convert our `FileField` and `ImageField`s to S3. The `AWS_STORAGE_BUCKET_NAME` setting specifies the S3 *bucket* where our files will live. S3 uses buckets somewhat like file directories on the hard drive. Buckets only live at one level, though, meaning there are no subdirectories or subfolders.

Files in a bucket are accessed using a key. This key effectively becomes a file name and can simulate subfolder hierarchies by including the usual / character as part of its key. For example, if we wanted to store videos in our bucket under a video subdirectory, we could set the file key to something like this:

```
videos/cranberry_farming_intro.mp4
```

Technically this is not a file path, just a key that looks like one. However, web browsers, humans, and other tools will generally not notice the difference.

File-level authorization and security takes place at the S3 service level and can be managed with a variety of desktop and web-based tools.

We should also note that if we need to use a combination of S3 and filesystem storages, we can do so by not setting the `DEFAULT_FILE_STORAGE` setting and instead using the `storage=` keyword argument on the `FileField` or `ImageField` where we'd like to store the results in S3.

Query string request authentication

Amazon's S3 service allows developers to provide access to private files stored in an S3 bucket using **Query String Request Authentication**. This results in a URL that can be accessed by third parties without any passwords or other complications.

These special URLs can include an expiration time expressed in UNIX epoch format. Any request made after this time will be denied. This limits the potential for multiple downloads and other abuses. It's not foolproof, but is a quick and easy solution to the private downloads problem and it could be sophisticated enough for many applications.

Implementing Query String Request Authentication is a matter of constructing a standard S3 URL with a set of query string parameters. These parameters include our AWS access key, the expiration time, and a signature. The signature is the key component as it allows S3 to verify the validity of the requests made to the URL.

To create the signature parameter of the authenticated URL, we need to first create a string that describes the request we'll be allowing access and specifies the expiration time. We then apply the HMAC-SHA1 hashing function to the string, then base64 and URL encode the result.

AWS documentation includes more details on the process, including the specific format of the string we need to generate. In short, this string will resemble the following:

```
GET\n
\n
\n
1264516089\n

/files/movies/cranberry_instructional.mp4
```

Note that the expiration time is required both in our request string and in the final, authenticated URL.

An example method of generating the URL-encoded signature in Python is as follows:

```
import hmac, hashlib
import base64
import urllib
s = """GET\n
\n
\n
1264516089\n
/files/movies/cranberry_instructional.mp4"""
digest = hmac.new(s, digestmod=hashlib.sha1).hexdigest()
digest64 = base64.b64encode(digest)
signature = urllib.quote(digest64)
```

Now that we have the signature, constructing the authenticated URL is simple. For this example, our authenticated request URL will look like this:

```
/files/movies/cranberry_instruction.mp4?AWSAccessKey=0PN32DSASDX33&E
xpires=1264516089&Signature=M2IwNWU4MDYzOGVmOTIzZWNhMTNjZDA5OGJmYmU4
YWQ2N2Q3OTU1Yg%3D%3D
```

We can provide this URL to the customer after they purchase our instructional video in an HTML template or e-mail, and they will have temporary access to download the content in their browser. Currently there is no easy way to determine when a third party has successfully completed an authenticated download. This could be managed at the application level, however, by providing the user with a sufficient amount of time before their URL expires and optionally allowing them to regenerate an expired request for content they've purchased.

About Amazon AWS services requests

Communication with most Amazon web services can be handled using two different methods: REST and Query or SOAP. The REST and Query approach has been used throughout this book. It relies exclusively on standard HTTP functionality to pass data parameters to AWS functions. This involves constructing and signing a query string like that used in the previous section.

Amazon's AWS documentation guides use the following pseudo grammar to explain how to construct this string:

```
StringToSign = HTTPVerb + "\n" +
ValueOfHostHeaderInLowercase + "\n" +
HTTPRequestURI + "\n" +
CanonicalizedQueryString
```

`HTTPVerb` is one of the usual HTTP methods like `GET` and `POST`. The host header value is simply the web services hostname we will be submitting requests to (that is. `fps.sandbox.amazonaws.com` or `fps.amazonaws.com`), and the URI is the path portion of the HTTP request. Often this is just `/`.

The `CanonicalizedQueryString` phrase is the set of HTTP parameters, usually a `GET` string, sorted, and excluding the signature parameter (which we will add before making the request). These parameters are passed as normal to the web services URL when we make the request. A partial query string looks like this:

```
version=2009-01-09&SignatureMethod=HmacSHA256&callerReferenceSender=jd
l_123123&FundingAmount=25.0&pipelineName=SetupPrepaid
```

SOAP interfaces use an entirely different approach and have not been implemented for this book.

Amazon FPS for digital goods

One of the advantages of implementing Amazon Flexible Payments for digital goods transactions is that the workflow is basically the same for physical or digital goods. Using the FPS Co-Branded UI, your customers submit their payment to the Amazon processor and are returned to a URL that was specified when the transaction started. In the case of physical goods, this is usually a receipt or thank you page. For digital goods, we would serve up the media download right away. Flexible Payment Services was discussed in *Chapter 4, From Payment to Porch: An Order Pipeline*.

Amazon offers another API, however, that may be of interest to digital content merchants. The FPS Aggregated Payments API is designed to aggregate many small transactions into a single, combined transaction. This has many advantages, but the most important is to reduce transaction charges.

An aggregated payment system is designed for applications that sell many different products to the same customer within a short period of time. FPS Aggregated Payments allows the developer to control many of these parameters. It also supports prepaid instruments that work like gift cards or prepaid debit cards and allow the customer to purchase from our application against an amount they pay up-front.

Here are some simple examples of services where this sort of payment processing is useful: single-song music sales, games with point or item purchases, and Facebook or other social media applications.

The Aggregated Payments API has many of the features of the standard FPS service, like refunds and instant payment notification. It also supports using either the prepaid or postpaid aggregation.

Prepaid payments

The prepaid version of the API follows the same pattern as Amazon FPS. Customers are initially driven to a Co-branded UI page via a **Pay with Amazon** link. Here they submit their payment information, select a prepayment amount, and so on. Upon completion, the user is returned to our application along with some information from Amazon. We capture this information and allow the user to browse and purchase.

When we make a sale, our application must communicate that fact with Amazon FPS, including the information we captured after the customer's prepayment. With future sales, we continue the process, updating FPS with the new purchase information. Customers can check their balance from the Amazon Payments website and FPS takes care of tracking transactions and balances.

Obtaining a prepaid token

Let's examine this process from the development side. The first step is to generate a special URL where we will send the customer to authorize payment of the prepaid amount. This requires the usual FPS steps of generating a query string and signing it using the HMAC-SHA256 function. The following code constructs this URL:

```
CBUI_ENDPOINT='https://authorize.payments-sandbox.amazon.com/
cobranded-ui/actions/start'

def sign_aws_request(params, hostheader, url_path='/',
                                    http_method="GET"):
    import hmac, hashlib
    import base64
    import urllib
    keys = params.keys()
    keys.sort()
    sign_string = '%s\n%s\n%s\n' % (http_method, hostheader, url_path)
    for key in keys:
        sign_string += '%s=%s&' % (urllib.quote(key),
```

```
            urllib.quote(params[key]).replace('/', '%2F'))
            sign_string = sign_string[:-1]
            digest = hmac.new(settings.AWS_SECRET_KEY,
            sign_string, digestmod=hashlib.sha256).digest().strip()
        return base64.b64encode(digest)

    def get_prepaid_token(ref_sender, ref_funding, amount, return_url):
         query_dict = {'callerKey': settings.AWS_ACCESS_KEY,
                               'callerReferenceFunding': ref_funding,
                               'callerReferenceSender': ref_sender,
                               'FundingAmount': str(amount),
                               'pipelineName': 'SetupPrepaid',
                               'returnURL': return_url,
                               'version': '2009-01-09',
                               'SignatureVersion': '2',
                               'SignatureMethod': 'HmacSHA256'}
        signature = sign_aws_request(query_dict,
                     'authorize.payments-sandbox.amazon.com',
                     url_path='/cobranded-ui/actions/start')
        query_dict.update({'signature': signature})
        cbui_url = CBUI_ENDPOINT + '?' + urllib.urlencode(query_dict)
        return cbui_url
```

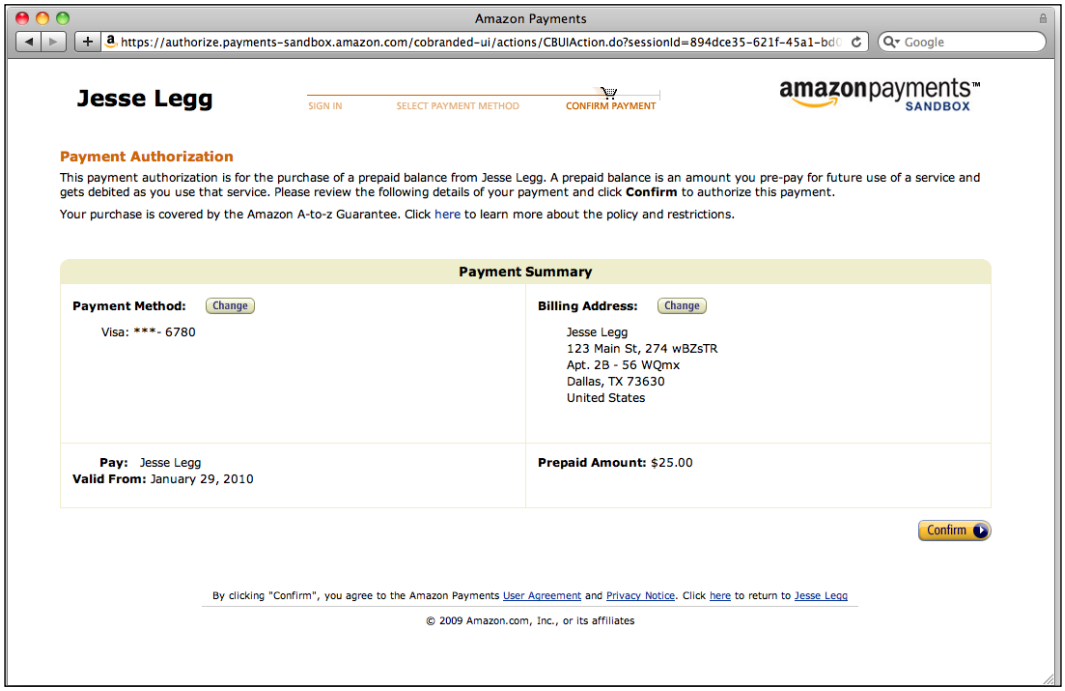
The URL returned from the `get_prepaid_token` function will be used as the `href` attribute in our **Pay Now** link or button. This function needs four arguments. The amount and `return_url` arguments are almost self explanatory. Amount is the dollar amount we will allow the customer to prepay. The `return_url` is a string representation of a URL in our application to return after the customer has authorized payment. This URL is important because it will receive the response data from the API request, which our application must process.

The remaining arguments, `ref_sender` and `ref_funding`, are somewhat mysterious. The FPS documentation explains them like this:

- `callerReferenceSender`: This is a unique identifier that you supply to specify this payment token and the sender in your records
- `callerReferenceFunding`: This is a unique identifier that you supply to specify this payment token and its funding in your records

These are required arguments and are used to relate the user funding activities in our application with the transaction in Amazon's FPS. Both should be unique to all of our prepaid requests. We could achieve this by appending an order `ID` number to the Django username, using a hash of order and amount information, or through many other methods.

Upon successfully generating an authorization URL, we can embed it as a link in our application's templates. Customers clicking the link to initiate a prepaid transaction will be directed to Amazon's **Co-Branded User Interface** (**CBUI**). This is an Amazon-hosted page that first prompts the user to log in to their Amazon Payments account and then presents them with an authorization screen like the one that follows:



This page simply verifies the customer's payment method, how much they're prepaying and some other billing details. By clicking the **Confirm** button, the user is returned to our application via the `return_url` parameter discussed previously. No money has changed hands at this point, to do that our application must fund the prepaid request.

To better understand what happens next, we'll outline the entire aggregated payments process for prepaid instruments. We've just seen what obtaining authorization in step 1 looks like. The remaining steps are as follows:

1. Obtain authorization from customer using CBUI.
2. Our application receives prepaid token at `return_url`.
3. Prior to use, our application funds the prepaid instrument using the token from step 2.

4. The customer makes a purchase.
5. Our application makes a `Pay` request to the FPS API.
6. The customer's prepaid balance is reduced by the amount of purchase.

Steps 4 through 6 will repeat until the customer's prepaid balance reaches zero. At that time, if the customer plans to make more purchases, we will need to authorize a new prepaid instrument. The full amount the customer has prepaid is deposited to our application's Amazon Payments account upon funding in step 3. No money actually exchanges hands at this point; the `Pay` requests are simply bookkeeping to manage the status of the customer's prepayment.

We should note, too, that the customer cannot see their prepaid balance from the Amazon Payments site itself. Only our application can retrieve their current balance, so this becomes an important piece of functionality. Fortunately, it is relatively easy to obtain balances for display in our application, as we'll see shortly.

If this sounds complicated, that's because it is. These series of steps are necessary, however, to secure the transaction to Amazon's standards. Aggregated payments are very recent innovations and diligence in the security of financial information is extremely important. Implementing an equivalent system from scratch would be quite
a task, which is why the Aggregate Payments API is so powerful.

Upon redirection to our return URL, Amazon FPS will include several pieces of information about the authorization. The most important among these is the funding token, sent in the HTTP `GET` parameter called `fundingTokenID`. This ID will be used in funding the prepaid instrument.

The response data also includes another important element, the `prepaidInstrumentId`. This represents the customer's prepaid instrument. It may help to think of this as the customer's "gift card" that we will be loading with funds. This instrument ID is used during funding and later on during `Pay` requests.

The final important response value is `prepaidSenderTokenID`. We will need this to actually use the prepaid balance when the customer makes a purchase in our application.

Funding the prepaid token

Funding the prepaid request requires us to make an API call back to the FPS service. We pass in the `fundingTokenId` sent back from the authorization request, as well as the `prepaidInstrumentId`, the amount we will be funding, and a `CallerReference` value that uniquely identifies this funding request.

To construct the REST request URL for the funding operation, we can use the following Python code:

```
def aws_timestamp():
    return strftime("%Y-%m-%dT%H:%M:%S.000Z", gmtime())

def fund_prepaid_token(funding_token_id, prepaid_id, caller_ref,
amount):
    timestamp = aws_timestamp()
    params={'Action': 'FundPrepaid',
                  'AWSAccessKeyId': settings.AWS_ACCESS_KEY,
                  'SignatureVersion': '2',
                  'SignatureMethod': 'HmacSHA256',
                  'Version': '2009-01-09',
                  'Timestamp': timestamp,
                  'FundingAmount': str(amount),
                  'PrepaidInstrumentId': prepaid_id,
                  'SenderTokenId': funding_token_id,
                  'CallerReference': caller_ref}
    signature = sign_aws_request(params, 'fps.sandbox.amazon.com')
    params.update({'signature': signature})
    fps_url = FPS_ENDPOINT + '?' + urllib.urlencode(params)
    return fps_url
```

We will sign the query parameters just as we did for the Co-branded UI request, except we will be using a different endpoint and thus different host header. The `sign_aws_request` Python function can be used to sign all of our FPS requests:

```
def sign_aws_request(params, hostheader, url_path='/',
                                    http_method="GET"):
    import hmac, hashlib
    import base64
    import urllib

    keys = params.keys()
    keys.sort()
    sign_string = '%s\n%s\n%s\n' % (http_method, hostheader, url_path)
    for key in keys:
        sign_string += '%s=%s&' % (urllib.quote(key),
                                                urllib.
quote(params[key]).replace('/', '%2F'))
        sign_string = sign_string[:-1]

         digest = hmac.new(settings.AWS_SECRET_KEY,
                     sign_string, digestmod=hashlib.sha256).digest().
strip()
    return base64.b64encode(digest)
```

Let's examine the `fund_prepaid_token` function in more depth. The important request parameters are `Action`, `PrepaidInstrumentId`, `SenderTokenId`, and `CallerReference`. The `Action` parameter is used in all FPS requests to specify what our request is attempting to do. We can think of it as the function we want to run against the services API. In this case we're calling `FundPrepaid`.

`PrepaidInstrumentId` is a unique identifier for the payment instrument (the customer's "gift card") that we will be funding. The value for this parameter was provided in the response from the prepaid token authorization request we made in the previous section. The response value was sent as the `prepaidInstrumentID` parameter (note the lower case). FPS generates this value internally during the initial authorization request.

`SenderTokenId` was also sent back to us in the response step of the previous section as the `fundingTokenID` parameter. This value is a key to funding the prepaid instrument and is required in all `FundPrepaid` requests. This token represents the balance amount between when the customer authorizes funds and before their prepaid instrument is funded. Without this value we could not fund the "gift card".

Finally, the last parameter is `CallerReference`. This is an important value that we will need to store for future reference. It could be analogous to an order ID or other unique reference number in our application. When we receive a response to our funding request, it will include a transaction ID, which we will want to associate with the corresponding `CallerReference` value. We will build Django models to store all of this information later in the chapter.

If for some reason our funding request fails, due to a network outage or other anomalies, and we do not receive a transaction ID, we can use the `CallerReference` value to submit the request again within seven days. This works by submitting the exact same request, including the other parameters, as we did initially. If the other parameters differ, FPS will return a duplication error.

Unlike the Co-branded UI request for authorization we issued earlier, no `return_url` parameter is needed. This is because our application will be processing the response without any user involvement and can read the response values directly.

`FundPrepaid` requests only return two values, the transaction ID and a transaction status. Status messages are one of five values: `Cancelled`, `Failure`, `Pending`, `Reserved` or `Success`. A `Success` value means we can allow the customer to proceed and begin purchasing against their prepaid balance, while we make `Pay` requests for each purchase.

Prepaid pay requests

Now that we've obtained the customer's permission and their payment method using the Co-branded UI authorization request, and funded their prepaid instrument, the next step is to make payments against the prepaid balance at the time of purchase.

When our customer finds a digital good to buy, we want to deduct the purchase amount from their prepaid instrument. To do this we perform another FPS API call, this time to the `Pay` action. We need to include three important parameters in this call: `SenderTokenID`, `TransactionAmount`, and `CallerReference`.

- `SenderTokenID` is the token we received during the authorization process where it was returned as the `prepaidSenderTokenID` parameter in the API response. This represents the customer's prepaid token.
- `TransactionAmount` is the purchase amount for the good we'll be selling. If the amount charged exceeds the balance available in the prepaid instrument, the API request will fail and will include errors data that describes the failure condition.
- `CallerReference` is a new unique identifier that we will need to store to track this `Pay` request. As with other FPS requests, if there is a network failure we can resend the `Pay` request using the exact parameters as the initial attempt for up to seven days.

For successful `Pay` requests, the response information will include the usual data, namely `TransactionId` and `TransactionStatus`. We can inspect these values to determine the success or failure of our payment attempt. If successful we can direct the customer to their purchased content, and their prepaid balance will be reduced by the amount of the purchase.

We can generate `Pay` requests for the FPS API using the following Python function:

```
def make_payment(sender_token_id, transaction_amount, caller_
reference):
    timestamp = aws_timestamp()
    params={'Action': 'Pay',
                  'AWSAccessKeyId': settings.AWS_ACCESS_KEY,
                  'SignatureVersion': '2',
                  'SignatureMethod': 'HmacSHA256',
                  'Version': '2009-01-09',
                  'Timestamp': timestamp,
                  'TransactionAmount': str(transaction_amount),
                  'CallerReference': caller_reference,
                  'SenderTokenId': sender_token_id}
```

```
    signature = sign_aws_request(params, 'fps.sandbox.amazon.com')
    params.update({'signature': signature})
    fps_url = FPS_ENDPOINT + '?' + urllib.urlencode(params)
    return fps_url
```

We will use the URL returned from `make_payment` to submit our request and await the API's response.

Checking prepaid balances

The FPS API action to check prepaid balances is called `GetPrepaidBalance`. It requires a signal parameter, `PrepaidInstrumentId`. This value is the unique identifier for the prepaid instrument and originated as the `prepaidInstrumentID` response value during the Co-branded UI authorization step.

This is not the same value used for `Pay` requests, `SenderTokenId`, though the two IDs play similar roles. This one cannot be used for the prepaid balance.

The following Python function will ready a `GetPrepaidBalance` FPS API call and return it as a URL where we can submit our request:

```
def get_prepaid_balance(prepaid_instrument_id):
    timestamp = aws_timestamp()
    params={'Action': 'PrepaidBalance',
                  'AWSAccessKeyId': settings.AWS_ACCESS_KEY,
                  'SignatureVersion': '2',
                  'SignatureMethod': 'HmacSHA256',
                  'Version': '2009-01-09',
                  'Timestamp': timestamp,
                  'PrepaidInstrumentId': prepaid_instrument_id}
    signature = sign_aws_request(params, 'fps.sandbox.amazon.com')
    params.update({'signature': signature})
    fps_url = FPS_ENDPOINT + '?' + urllib.urlencode(params)
    return fps_url
```

Postpaid payments

The postpaid API works slightly differently than the prepaid equivalent. The overall process is the same but the API calls and their necessary parameters differ. The postpaid approach also has different financial implications.

As we mentioned in our discussion of the Aggregated Payments API, the goal is to combine many smaller purchases into one large purchase to reduce transaction fees. The prepaid approach does this by charging the customer up front. Postpaid works like a type of credit system, where we initially grant the customer a limited purchase amount. As they buy from our application, we issue `Pay` requests that accrue against the customer's limit.

The customer can purchase as much as they like up to the limit we initially grant. When they reach the limit they are unable to make additional purchases until their debt balance is settled. We can also settle debts at an agreed to interval, every day or week, for example.

When the debt is settled, the balance is transferred from the customer's payment method, selected during the authorization step, into our Amazon Payments account. Our application must settle the debt by calling the `SettleDebt` FPS API function. We do this whenever we detect that the balance has reached its maximum or is at the agreed upon interval.

Obtaining a postpaid token

The overall process of using the Postpaid API mirrors the steps we took for the Prepaid API. To obtain the Postpaid Token we begin by constructing a URL to use in our **Pay Now** buttons. This is the Co-branded UI where the customer reviews the terms of the postpaid transaction, configures, and then authorizes payment.

There are a variety of customizations in this step, including setting usage limits and expiration dates for the credit we'll be extending to the customer. For our purposes, we will keep it simple and implement a basic postpaid credit account that never expires.

We need to supply four parameters to the CBUI authorization system to begin a postpaid transaction: `callerReferenceSender`, `callerReferenceSettlement`, `creditLimit`, and `globalAmountLimit`.

The `callerReferenceSender` and `callerReferenceSettlement` parameters are unique identifiers for this postpaid transaction. The first is intended to associate the customer and the token, while the second associates the token and its settlement. Our application should generate unique values for these parameters according to some consistent method, as discussed in the prepaid section.

The next two parameters control the terms of the credit. We set a maximum credit amount using the `creditLimit` parameter. This is equivalent to a credit limit on a credit card: the customer cannot spend beyond this amount without settling the transaction. The second value, `globalAmountLimit`, is the maximum charge the customer can incur against their credit limit in our purchase.

These two parameters give our application flexibility in how we handle credit. For a service that is processing many small transactions, we may wish to set a relatively high `creditLimit`, but restrict the amount of any one-time purchase, so that if the customer fails to pay, we limit our losses. Other times it may make sense to set these two values equivalent to each other.

The following Python function generates a Postpaid CBUI URL:

```
def get_postpaid_token(ref_sender, ref_settlement, credit_limit,
                       global_limit, return_url):
    params = {'callerKey': settings.AWS_ACCESS_KEY,
                    'callerReferenceSettlement': ref_settlement,
                    'callerReferenceSender': ref_sender,
                    'creditLimit': str(credit_limit),
                    'globalAmountLimit': str(global_limit),
                    'returnURL': return_url,
                    'pipelineName': 'SetupPostpaid',
                    'SignatureVersion': '2',
                    'SignatureMethod': 'HmacSHA256',
                    'Version': '2009-01-09'}
    signature = sign_aws_request(params,
                                 'authorize.payments-sandbox.amazon.com',
                                  url_path='/cobranded-ui/actions/start')
    params.update({'signature': signature})
    cbui_url = CBUI_ENDPOINT + '?' + urllib.urlencode(params)
    return cbui_url
```

This function uses the same AWS signing code we used for Prepaid API calls.

Our `return_url` will again capture the response data the FPS API sends when redirecting the user back to our site. The response values include: `creditInstrumentID`, `creditSenderTokenID`, and `settlementTokenID`.

The `creditInstrumentID` acts as a unique identifier for the postpaid instrument. As with the Prepaid API, this value is not used for making payments against the instrument, but can be used to obtain information and present it to the customer, such as debt balances. It can also be used when writing off debt from a customer's credit balance.

The `creditSenderTokenID` value is used to make `Pay` requests against the credit instrument.

The `settlementTokenID` value is used during the settlement process as a parameter to the `SettleDebt` FPS function.

Again, we will need these values for later operations.

Postpaid pay requets

Unlike in the Prepaid API, postpaid instruments do not need to be funded before they can be used. Instead we can begin issuing `Pay` requests immediately after obtaining the Postpaid Token. When the user makes a purchase, our application can record it and issue the `Pay` API call. If the `Pay` fails, the customer may have reached their credit limit or attempted to purchase something more than their global limit allows. Our application could check for these conditions and if necessary begin settlement.

The `Pay` request is identical to that used in the Prepaid API. We will use the `creditSenderTokenID` value returned during the authorization step as the value for `SenderTokenID`. This way our `make_payment` function will also allow us to make postpaid credit `Pay` requests:

```
def make_payment(sender_token_id, transaction_amount, caller_
                 reference):
    timestamp = aws_timestamp()
    params={'Action': 'Pay',
                 'AWSAccessKeyId': settings.AWS_ACCESS_KEY,
                 'SignatureVersion': '2',
                 'SignatureMethod': 'HmacSHA256',
                 'Version': '2009-01-09',
                 'Timestamp': timestamp,
                 'TransactionAmount': str(transaction_amount),
                 'CallerReference': caller_reference,
                 'SenderTokenId': sender_token_id}
    signature = sign_aws_request(params, 'fps.sandbox.amazon.com')
    params.update({'signature': signature})
    fps_url = FPS_ENDPOINT + '?' + urllib.urlencode(params)
    return fps_url
```

Our application can now connect to the URL returned from this function and in so doing, submit the `Pay` request.

Settling debts

As discussed, when the customer reaches their credit limit or at some agreed upon interval (over night, for example), we must settle the debt. We do this by issuing a call to the FPS `SettleDebt` function.

The `SettleDebt` action takes three parameters: `CallerReference`, `CreditInstrumentId`, and `SenderTokenId`. These are similar to the parameters we've already seen.

As before, `CallerReference` is a unique identifier for this settlement transaction that we will want to store in the database for potential future reference. It can be any value as long as it is unique.

The `CreditInstrumentId` was sent back after the Co-branded UI authorization step. It originated as the `creditInstrumentID` in that response data. It represents a unique identifier for our customer's credit instrument.

Lastly is the `SenderTokenId`. This was also sent back to the authorization step. It originated in the response data as the `settlementTokenID` value.

We can generate the `SettleDebt` API request call using the following Python function:

```
def settle_debt(credit_instrument_id, sender_token_id, caller_
reference):
    timestamp = aws_timestamp()
    params={'Action': 'SettleDebt',
                  'AWSAccessKeyId': settings.AWS_ACCESS_KEY,
                  'SignatureVersion': '2',
                  'SignatureMethod': 'HmacSHA256',
                  'Version': '2009-01-09',
                  'Timestamp': timestamp,
                  'CallerReference': caller_reference,
                  'CreditInstrumentId': credit_instrument_id,
                  'SenderTokenId': sender_token_id}
    signature = sign_aws_request(params, 'fps.sandbox.amazon.com')
    params.update({'signature': signature})
    fps_url = FPS_ENDPOINT + '?' + urllib.urlencode(params)
    return fps_url
```

Our application can use the URL value returned by this function to submit the `SettleDebt` request.

Writing off debt

Sometimes, perhaps due to a download problem or our own sense of generosity, we may want to relieve a customer of some amount of debt. This is different from a refund, though it may appear like a refund to the user.

Writing off debt simply reduces the debt balance for a credit instrument. To do this we use the `WriteOffDebt` FPS API action. This call requires three parameters: `AdjustmentAmount`, `CallerReference`, and `CreditInstrumentId`.

- `AdjustmentAmount` represents the amount of debt we intend to write off. This is a unique parameter in that it cannot simply be expressed as a dollar amount, but has a special format that includes currency.

- `CallerReference` is a unique identifier used as a reference in our application to this `WriteOffDebt` call.
- The `CreditInstrumentId`, as we have seen, is a unique identifier for the credit instrument from which we will be writing off debt. This was provided as a response value during the authorization step where we obtained the postpaid token.

We can initiate a `WriteOffDebt` call using the following Python function:

```
def write_off_debt(adjustment_amount, credit_instrument_id,
                           caller_reference,
                           currency_code='USD'):
    timestamp = aws_timestamp()
    params={'Action': 'SettleDebt',
                  'AWSAccessKeyId': settings.AWS_ACCESS_KEY,
                  'SignatureVersion': '2',
                  'SignatureMethod': 'HmacSHA256',
                  'Version': '2009-01-09',
                  'Timestamp': timestamp,
                  'CallerReference': caller_reference,
                  'CreditInstrumentId': credit_instrument_id,
                  'AdjustmentAmount.CurrencyCode': currency_code,
                  'AdjustmentAmount.Value': adjustment_amount}
    signature = sign_aws_request(params, 'fps.sandbox.amazon.com')
    params.update({'signature': signature})
    fps_url = FPS_ENDPOINT + '?' + urllib.urlencode(params)
    return fps_url
```

Notice how we specified both a value and currency code for the `AdjustmentAmount` parameter. This is a requirement of the FPS API for the `WriteOffDebt` call.

As usual the response data includes a `TransactionId` and status code, similar to the other FPS API calls that we have seen.

Getting debt balances

Just like balances for prepaid instruments, customers are unable to see their current debt balance from the Amazon Payments service. We provide this functionality for them by making a call to the `GetDebtBalance` API function.

The `GetDebtBalance` call can also be useful to determine an amount to write off for an entire debt, if necessary.

`GetDebtBalance` takes a single parameter, `CreditInstrumentId`, a unique identifier for the credit instrument we're inspecting. This value was returned in the response data after the initial authorization step. We can initiate the `GetDebtBalance` call with the following Python function:

```
def get_debt_balance(credit_instrument_id):
    timestamp = aws_timestamp()
    params={'Action': 'PrepaidBalance',
                  'AWSAccessKeyId': settings.AWS_ACCESS_KEY,
                  'SignatureVersion': '2',
                  'SignatureMethod': 'HmacSHA256',
                  'Version': '2009-01-09',
                  'Timestamp': timestamp,
                  'CreditInstrumentId': credit_instrument_id}
    signature = sign_aws_request(params, 'fps.sandbox.amazon.com')
    params.update({'signature': signature})
    fps_url = FPS_ENDPOINT + '?' + urllib.urlencode(params)
    return fps_url
```

Django integration

Now we can begin integrating aggregated payments into our Django application. Because of the numerous variables to keep track of, we will need to retain our reference values. Since we're building in Django, a `Model` subclass seems appropriate.

But what values need to be stored? There are a variety of IDs, tokens and the like, do we need all of them? It turns out that every important value we'll need can be retrieved, if necessary, using the `CallerReference` value we include in each of our requests. In addition, Amazon recommends storing all `TransactionID` values returned from FPS service calls and associating them with the correct `CallerReference`.

For the most part, the FPS response values we need will be used right away. Our storing these reference values is mostly a back-up plan in case we run into problems. It is also a good idea to store the request ID value, which is returned with every request sent to Amazon web services. This is for troubleshooting purposes, in case there is a problem and then AWS requires the request ID.

A Django model to persist this information may seem quite simple. But with only these two reference values, we have no way to relate the FPS transactions back to our application's order tracking. For this we need a relationship with our `Order` model.

```
class TokenReference(models.Model):
    order = models.ForeignKey(Order)
    funding_reference = models.CharField(max_length=128)
    sender_reference = models.CharField(max_length=128)
    transaction_id = models.CharField(max_length=35)
```

This makes sense because we will need to track orders for digital goods just like traditional sales.

View implementation

Our view code will be responsible for tracking the reference values stored in the `TokenReference` model. It is likely that we will want to redirect the user to the CBUI site ourselves, instead of providing a direct link, so that we can capture the reference information in our Django model.

To do this, we'll create a `cbui_redirect` view and wire our **Pay Now** button links to use it. It will take a dollar amount, which is the amount the customer will authorize for prepaid or postpaid credit.

```
def sender_reference(user):
    return '%s_%s' % (user.username, datetime.now())

def funding_reference(order):
    return '%s' % order.pk

def make_order(user, amount):
    # Create an Order object for this user's aggregated payments
      purchase
    return Order(customer=user, total_price=amount,
                      status_code=get_status_code("NEW"))

def cbui_redirect(request, amount=25.00, pay_type='prepaid'):
    order = make_order(request.user, amount)
    caller_sender_reference = sender_reference(request.user)
    caller_funding_reference = funding_reference(order)
    TokenReference.objects.create(order=order,
                         sender_reference=caller_sender_reference,
                         funding_reference=caller_funding_reference)

    if pay_type is 'prepaid':
        redirect_url = get_prepaid_token(
                                caller_sender_reference,
                                caller_funding_reference,
                                amount,
                                reverse('cbui_return'))
    elif pay_type is 'postpaid':
        redirect_url = get_postpaid_token(
                                caller_sender_reference,
                                caller_funding_reference,
```

```
                                    amount,
                                    reverse('cbui_return'))
        else:
            raise ValueError
    return HttpResponseRedirect(redirect_url)
```

The purpose of this view is to:

1. Create a new `Order` object for this aggregated payment request.
2. Generate unique reference numbers to identify the request to Amazon.
3. Redirect the user's browser to the Amazon Co-Branded UI page to finalize authorization.

This view also references a `cbui_return` view, which would be the view to handle the user's return from FPS's CBUI. It is likely that some additional order management is required in both these views. We could possibly stick a copy of the customer's order in the session object so that subsequent steps can look up the appropriate `TokenReference` object by the `Order` ID.

View implementations for FPS Aggregated Payments are complicated by the amount of information we need to pass around between our own view code and FPS API calls. Particularly problematic is the response data we receive in support of our `returnURL` value, when we sent the user to the CBUI. This includes the `fundingTokenID` and `prepaidInstrumentID` values.

There are a variety of possibilities: we could create another Django model to store the token information and relate it back to the user and/or order. This approach would require the security of our database to be strong, because anyone with access to this token information and our `AWS_SECRET_KEY` could potentially cause harm.

Another potential optimization would be to create an aggregated payments version of our shopping cart. This could include token information assigned to the shopping cart itself, which would make it easy to manage the customer's balance as they make purchases.

Amazon's Web Services and Flexible Payment Services in particular are generally very large, sometimes unwieldy APIs. They also suffer documentation problems, probably due to their complexity. But the examples we've begun in this chapter should be enough to get started implementing aggregated payments in your own applications. For more information, the full FPS Aggregated Payments documentation is available from: `http://aws.amazon.com/documentation/fps/`.

Google Checkout Digital Delivery

Google Checkout provides another option for digital goods sales using their Digital Delivery API. It does not include support for aggregated payments, which are often useful when selling digital goods due to the fee savings from their typical low sales price. But if you're already working with Google Checkout, as we did earlier in this book, integration with digital delivery is relatively simple.

Google Checkout's Digital Delivery arguably gives us less control over the delivery process, but does so at the benefit of simplifying development. Checkout takes care of just about all aspects of the purchase, from initiating payment to notifying the buyer with download instructions.

Digital Delivery for Checkout includes three delivery methods:

- E-mail
- Key/URL
- Description based

E-mail is the least recommended and most manual method. The customer will receive a confirmation page after submitting their payment. This page will let them know they should receive instructions regarding access to their purchase. We are responsible for sending these instructions when Checkout notifies us that a purchase is complete.

The **key/URL** method is more automatic. When the purchase is completed, the customer receives a confirmation page that includes the URL or key to access their content. This could work well for video or music downloads, in which case we could link the buyer directly to the download. We could even provide the link to our Amazon S3 storage bucket with Query String Authorization.

Another case where this method might work well is for software purchases. We could generate a software key for the buyer and this would be presented to them after they complete payment.

The final method is **Description based**, which is similar to e-mail, but the instruction text will be displayed at the end of their purchase, instead of a notice that instructions will be e-mailed to them. For example, perhaps the buyer is purchasing an upgrade for a video game character. The descriptive delivery method could be used to inform them that their character will be enhanced the next time they log in to the game world.

As with the rest of Google Checkout's API, all information is provided in XML format. When we generate the Checkout XML, we will include special digital delivery tags in the `<item>` element that corresponds to the digital purchase. This is where we include the key or URL information for that delivery method, or the instruction text for the description method.

An example `<item>` element for key/URL digital delivery looks like this:

```
<item>
   ...
   <digital-content>
     <display-disposition>OPTIMISTIC</display-disposition>
     <description> Please go to the Help->About menu in your trial
      version of our software to enter the registration key.
     </description>
     <key>1456-1514-3657-2198</key>
   </digital-content>
   ...
</item>
```

With the other Checkout services, it allows for easy integration with Django because we can render this information dynamically using Django's template language. This also means we could track purchases and keys in a Django model with similar ease. We discussed the Google Checkout XML format at length in the first three chapters of this book. For more information on digital delivery, refer to the latest Checkout XML API documentation.

Summary

This chapter has provided an overview of several advanced use cases for selling on the Web with Django. These included:

- Utilizing Django's storage back end to support content storage on S3
- Aggregated payments using Amazon's FPS service
- Simple sets of Django and Python functions for working in FPS
- Google Checkout's Digital Delivery XML

These services are almost literally expanding and changing on a daily basis. This reflects the rapid evolution of the Web as an e-commerce platform and budding attempts to sell digital content and goods. If some of these services feel rough around the edges, it is likely a reflection of this early phase.

10
Deployment and Maintenance Strategies

One of the major challenges in modern web development is managing deployment and maintenance tasks. Python and Django, as well as other frameworks, can simplify application development, but leave us somewhat on our own when it comes to deployment. Fortunately an array of tools that are improving all the time can assist us in this task. In this chapter we will examine some of these tools, including:

- The `mod_wsgi` extension for Apache to load our Python applications
- Automated remote deployment using **Fabric**
- Python environment isolation and project building with `zc.buildout`
- Virtual environments to easily develop and deploy multiple projects
- Python module distribution using `distutils`

Knowledge of these tools, at the very least, is important for any Django developer. If you're interested in boosting your productivity right away, as well as avoiding potential future headaches, you can do so easily by switching to virtual environments. In the author's opinion, learning virtualenv is one of the best investments in process you can make and is one of the most productive development tools in the Django and Python community.

Note also that many of these tools are changing fast. Fabric, for example, is expected to release a new 1.0 version sometime soon. We have done our best to cover the very latest versions of the tools in this chapter and throughout the book, but minor variations to syntax in new versions are likely unavoidable. In addition, these are the top tools available today, but developers are constantly creating, improving, and releasing new tools.

Apache and mod_wsgi

There are numerous methods of integrating Django applications with web servers. Each has specific advantages and disadvantages, and your options are sometimes limited by your server, hosting provider, and operating system. The new, official recommendation, at least for Apache servers, is to use `mod_wsgi`.

WSGI (pronounced *wizgy*) stands for **Web Server Gateway Interface**, and it is designed to be a standard means of connecting Python web applications with the servers that run them. This means that any WSGI-compliant Python web framework or application can run on any web server that implements the WSGI standard. Apache is a very popular open source web server that supports the WSGI standard through its `mod_wsgi` extension.

Web server configurations, especially those involving Apache, are a complex business. Web servers are like snowflakes and no two are ever the same. It's easy to argue about performance of module X over module Y using framework Z, but in reality there is a trade-off happening in every configuration. Sometimes you give up a little memory to gain some speed, and sometimes vice versa.

If you have a choice in the matter, `mod_wsgi` is generally an all-around solid and highly recommended deployment option. It may be possible to get improved performance out of another option, but if your situation warrants something more sophisticated, you likely don't need us to tell you.

How does WSGI compare to other Django server options, such as `mod_python`? In general, `mod_wsgi` is faster and uses less memory. In the case of it being *faster* we simply mean that `mod_wsgi`'s measured request throughput is often greater than `mod_python` under comparable load in similar servers. Memory usage is more obvious, in part because `mod_wsgi` is written strictly for support of WSGI applications. The `mod_python` option is designed not just for running Python-based web applications, but also to provide Python modules for interacting with the Apache server. As a result, additional memory is required to accommodate the additional modules. This memory usage claim holds up even under the simplest load analysis (try setting up two identical servers, automating a load test and running top).

There are two modes of configuration for `mod_wsgi`, embedded and daemon. Daemon mode is recommended in almost all cases. The difference is that embedded mode, like `mod_python`, runs the Python interpreter and WSGI application within an Apache process. Daemon mode allows us to run our WSGI applications in a separate process from Apache requests—a special WSGI-only process. This process can be threaded and has other special properties, such as running under an arbitrary user account. Though performance in both configurations can be made comparable, Apache's default setup makes embedded mode difficult to optimize without a deep understanding of Apache configuration. As a result, daemon mode is recommended just because it usually works.

These few paragraphs about performance are at best crude sketches of a topic that is very complex. For the vast majority of applications, server-level configuration, beyond the basics we've outlined, is not going to be the source of problems. Usually, bottlenecks and other performance issues will result from the Django application itself.

A great deal of additional information on `mod_wsgi` is available at the project's homepage on Google Code: `http://code.google.com/p/modwsgi/`.

A Django WSGI script

Django includes built-in support for WSGI, but in order to deploy our Django projects under `mod_wsgi`, we need to write a simple WSGI script. This is just some Python code that sets up our environment and kicks off Django's WSGI handler. Here is an example script file, called `django.wsgi`:

```
import os
import sys

os.environ['DJANGO_SETTINGS_MODULE'] = 'coleman.settings' sys.path.
append('/home/jesse/src/my-site-packages')
import django.core.handlers.wsgi
application = django.core.handlers.wsgi.WSGIHandler()
```

Specifying our settings module manually is required in all cases (it's a Django requirement). We can also modify the value of `sys.path`, if needed, to include our Django or other Python packages. The WSGI script will run under `mod_wsgi` and will therefore not receive any additions to the `$PYTHONPATH` environment variable we might use in our local, command-line development environment.

An example httpd.conf

Once we've put together our WSGI script, we need to write an Apache configuration file. The primary Apache directive to know is `WSGIScriptAlias`. This is similar to the more common `ScriptAlias` directive. An example `VirtualHost` configuration appears below:

```
<VirtualHost *:80>
ServerName coleman.localhost
WSGIScriptAlias / /Users/jesse/src/coleman/apache/django.wsgi

<Directory /Users/jesse/src/coleman/apache>
Order deny,allow
Allow from all
</Directory>
Alias /media/ /Users/jesse/src/coleman/media/
<Directory /Users/jesse/src/coleman/media>
Order deny,allow
Allow from all
</Directory>
</VirtualHost>
```

Configuring daemon mode

By default, this configuration uses `mod_wsgi` in embedded mode. For development or testing purposes this may not make much of a difference, but as we mentioned earlier, it is generally recommended to use daemon mode for any WSGI application, unless you're willing and able to tune Apache yourself. To enable daemon mode, we use the `WSGIDaemonProcess` and `WSGIProcessGroup` directives:

```
<VirtualHost *:80>
WSGIDaemonProcess book-site user=jesse threads=25
WSGIProcessGroup book-site
ServerName coleman.localhost
WSGIScriptAlias / /Users/jesse/src/coleman/apache/django.wsgi

<Directory /Users/jesse/src/coleman/apache>
  Order deny,allow
  Allow from all
</Directory>

Alias /media/ /Users/jesse/src/coleman/media/
```

```
<Directory /Users/jesse/src/coleman/media>
  Order deny,allow
  Allow from all
</Directory>
</VirtualHost>
```

The addition of these two directives changes things quite significantly. Let's examine `WSGIDaemonProcess` first.

The `WSGIDaemonProcess` directive activates `mod_wsgi`'s daemon mode. The first parameter, which is required, is a unique name that will be assigned to the WSGI daemon process. The remaining options are not required, but fine tune `mod_wsgi`'s mechanisms. The above example uses two options. The first, user, specifies the user account that our WSGI process will run under. The second, threads, specifies how many threads to initiate in this process.

`WSGIDaemonProcess` supports establishing multiple processes as well, using the processes option. Use of this option in combination with threads will control how many total request handlers will be available for your WSGI application. The default number of threads is 15, so using `processes=2` would spawn 30 total request handlers. These options are tunable and depend a lot on your specific application, server, hardware, and so on.

The `WSGIDaemonProcess` directive simply creates a WSGI process according to our desired options. To assign this process to a specific WSGI application, we need to use `WSGIProcessGroup`. This directive can occur in our top-level Apache configuration or within a `VirtualHost` context. It our example case, the WSGI application defined for the virtual host will be assigned to the WSGI process named book-site. A WSGI application can be bound to any WSGI process defined in the server or virtual host. In addition, multiple applications can be assigned to the same process in this same manner.

Whole books are dedicated to Apache configuration and the `mod_wsgi` site alone includes dozens of wiki pages with details and discussion about the module's use. In general, most applications will not need much more than a simple, straightforward configuration. It is important to understand the basics, however, because as your application grows, it could become important.

Thread-safety

An issue that arises in our configuration above is thread-safety. Considerable debate has followed the development of Django in regards to this topic, but as of most recent releases, the framework is generally considered to be thread-safe. This means it is reasonable to run Django applications using multi-threaded WSGI processes or under a multi-threaded web server (for example, Apache's 'worker' MPM).

Undiscovered threading bugs are always possible, albeit unlikely. More important, however, is to consider the implications of your own application code and how it works in a multi-threaded environment. Typically this sort of problem can be introduced when trying to use shared, global variables between threads in your application code. Particularly important is the fact that Django `QuerySet` objects are not thread-safe. They attempt to share a global `QuerySet` instance in application code between threads, which will have unpredictable results.

We should also mention that versions of Django prior to 1.0 had known thread-safety problems. If you're running an earlier Django version, for code compatibility reasons perhaps, you could potentially encounter threading bugs when running under Apache's worker MPM, multi-threaded WSGI processes, any Apache server running on the Windows operating system, or any other multi-threaded environment.

Automating deployment with Fabric

A big win for development team productivity can be automating as much of the routine tasks as possible. The scope of tasks that could be automated is practically unlimited; if it happens more than once, you might want to think about it. The reason for this is simple: deployment tasks are often repetitive and prone to error. Doing it correctly once, capturing the process and automating it prevents errors, reduces efforts, and minimizes time spent away from focusing on your application.

Fabric is a very recent tool written in Python that attempts to provide a simple framework for automating deployment activities. It allows us to automate common elements of a deployment process, including connecting to and issuing commands on servers via SSH, uploading and downloading files, and gathering input from the console when needed. Documentation and more information are available at `http://docs.fabfile.org/`.

Writing a Fabfile

Creating a set of deployment functions for our application is just a matter of writing some Python. Often this code will live in a file called `fabfile.py` somewhere in our project tree. The contents of this file can be whatever we want; it's just normal Python, but it will typically import the fabric module to allow us to perform our deployment work.

What's great about Fabric is that it lets us define all of our operations relative to an environment. The environment is a global dictionary containing the import information regarding our deployment situation. This includes things such as host names, user names, passwords, and so on. Thus we can write a set of deployment functions for our application and reuse them in a simple way, just by changing our environment values.

For example, to execute our Fabric script across multiple servers, we can define the environment's `hosts` list to include the server hostnames or IP address at the top of our `fabfile.py`:

```
env.hosts = ['ebook.server.com', '172.16.1.10']
```

With our environment defined, Fabric will automatically run our deployment functions across both of these servers, including logging in to SSH. In real-world scenarios, additional information, such as user names, can be included in the environment. By default, the current user is used as the value for username and passwords are obtained interactively.

The fabric module is loaded with useful functions. For example, the `fabric.operations` module includes:

- `sudo`: Used to run a remote command as the super user
- `put`: Used to upload a file to the remote server
- `run`: Used to run a remove command as the logged-in user
- `get`: Used to download a file from the remote server
- `local`: Used to run a command locally

These are all just Python functions that behave as you would probably expect. Some examples include:

```
put('media/js/site_wide.js', '/var/www/media/js/site_wide.js')
put('media/img/*.jpg', '/var/www/media/img/')
sudo('svn update /opt/svn/repos/myrepos')
run('touch django.wsgi')
local('django-admin.py test all')
```

With these basic set of operations, you can begin to compile a set of functions before the deployment tasks. This might include a test function to run Django unit tests, a minify function if you're minifying JavaScript, a function to compress and upload test data or import SQL commands to your database, and almost anything else.

Using the fab tool

Fabric provides a command line tool called `fab` that allows us to quickly run our Fabric operations from a shell prompt. The `fab` tool takes any number of arguments, each of which corresponds to a function we've written in our fabfile.

The fabfile library is auto discovered when using the `fab` tool. It will search the current directory for a `fabfile.py` or, in newer versions of Fabric, search for a `fabfile/` module. The `fabfile/` subdirectory in our project must contain `__init__.py` for this auto-discovery to succeed.

The `fab` command will execute our fabfile operations in the order we provide on the command line. An example usage follows:

```
$ fab run_tests prepare_deployment deploy
```

This will first execute our fabfile's `run_tests` function, then `prepare_deployment`, then `deploy`. For simplicity we could have wrapped all three functions inside the deploy operation and the result would be equivalent. But what happens when say, `run_tests` fails? Fabric will automatically detect Python exceptions and cease execution of the current and future operations.

In addition to Python exceptions originating in our functions, Fabric will also halt when a remote or local shell command does not return cleanly, after a failed file transfer or broken `sudo` command, for example. This behavior is configurable, of course, by changing the environment setting `warn_only` to `True`.

Fabric for production deployments

Fabric is a very Pythonic deployment tool. Other excellent tools exist that play a similar role, such as Ruby's Capistrano (which can be used for deploying anything, not just Ruby). But many developers still use the manual, SSH-and-type-some-commands method. This works, but automating these repetitive tasks can turn a half-dozen shell commands and a lot of thought into a single command that we can run while grabbing coffee (or automate for a continuous integration strategy).

Consider the typical production deployment. For Django applications there can be several complicating factors. One of the biggest is media. Media is typically served in a different fashion than the application itself. This might mean media files are served via a content-delivery network or a separate media server. Either way, changes to Django media files sometimes necessitate special steps in the deployment process.

Cascading Style Sheets (**CSS**) are good examples. CSS files frequently contain file paths to background images and other content, but do not have access to Django's `MEDIA_URL` template variable, because they are not rendered by Django. Relative file paths can solve this problem much of the time, as long as media is served up identically on production, staging, and development servers.

But when media can't be served identically, perhaps because production servers use a content-delivery network that we have a limited ability to customize, we face a deployment headache. A tool like Fabric can solve this problem in concert with an appropriate search-and-replace type script. It is as simple as writing custom deploy functions that extract our application code, deploy to the corresponding site, and run a site-specific media processing script to make any necessary adjustments.

Another example is using Fabric to manage deployments from source code repositories. The server-centric functions we've written to include special handlers for media files could also intelligently manage the deployment interaction with a source code repository. This means we could write Fabric functions to roll out new code releases and roll back to previous releases automatically. A simplification like this is ideal because it provides a simple way for anyone (even non-technical staff) to manage production servers.

One-off and complicated situations like this are often unavoidable. If developers are burdened by a complicated deployment process, they are almost certainly going to perform less testing and encounter more problems during roll outs of new code. This is where a tool like Fabric can become especially powerful.

zc.buildout

In addition to the problem of managing and configuring our applications on remote servers, which Fabric handles very well, there is another problem inherent in Python and Django-based development. That is the issue of packages and dependencies.

Consider the scenario where we're running a production version of our e-commerce store on the same server as our development version. The development site is where we do testing as we implement new features and fix bugs. Now suppose our production site is running on Django version 1.0, but we've decided for the next version we need to upgrade to Django 1.2 because we need some of the new framework features.

During the development of the new version of our site, the production instance must continue running with absolutely no problems. If we've been using the naive approach of installing Django into our server's system `site-packages`, we face a problem. We cannot upgrade to 1.2 at the system level because that will break the production site. But we need to upgrade in order to test and develop our new version.

Our first goal is to avoid global, system-level installation of packages as much as possible. This is relatively easy to do: don't install Python packages globally using `setup.py` or `easy_install` and avoid installations using an operating system's package manager, such as Debian's `apt-get`.

The second goal is to find a tool that will simplify the complexity of managing and running different versions of our packages (in this case Django itself) amongst the different instances (production and development) of our site. The `zc.buildout` tool helps us meet this second goal.

Buildout is just one of several tools for tackling this sort of deployment problem in Python. It originates as part of the Zope framework, but is flexible enough for general use on any Python-based project. It has a moderate learning curve, but the time invested in this deployment process will be made up many times over with that efficiency it will bring. More information and full documentation is available at `http://buildout.org`.

Buildout bootstraps

Though buildout can be installed in the system-level `site-packages`, it also provides a bootstrapping script that can be used to set up and configure a project without the developer needing to install anything beyond a Python interpreter. The bootstrapping process installs the buildout package itself into the project, which can then be used to actually create the project build, installing all the relevant Python packages and providing a special project-specific Python interpreter.

This two-step process is an excellent way to get other developers up and running on a project. The bootstrapping tool and buildout configuration can be placed in a version control repository such that all a new developer needs to do is check out the code, run `bootstrap.py` to prepare for a buildout, and then run the `buildout` tool itself. This also holds true for server deployments, where the buildout can happen on a fresh server with just these simple commands.

There are other ways to use buildout without bootstrapping. If it is installed in the system site packages, then `bootstrap.py` is unnecessary and new developers or deployment scripts only need to run the `buildout` tool directly.

After running the `buildout` tool, our project directory will contain a half dozen or so additional files and directories. This is where buildout does its work. These subdirectories include: `bin`, `developer-eggs`, `eggs`, and `parts`. The `bin` is for executable files, like the buildout tool itself and the special Python interpreters. The `eggs` location is where buildout stores third-party eggs, while `developer-eggs` stores the eggs that we're currently developing (our application modules). `Parts` functions as a workspace that buildout uses while building things.

Let's take a moment to talk about eggs. Eggs are created by the `setuptools` module and are a quasi-standard way of packaging modules into a tidy unit (the standard way is built-in to Python and called `distutils`, of which `setuptools` is an enhanced version). They are typically used to package up a specific version of a Python module such that it can be easily shared with other developers.

The `.egg` format differs from a standard Python module (a simple directory with `__init__.py`) in that it includes metadata in addition to code. This is accomplished through the use of either an `EGG-INFO` subdirectory at the module level or an `.egg-info` directory at the module's parent level. An `.egg`'s metadata typically describes the package, including the author information, license, a homepage URL, and, importantly, any dependencies.

Typically when you `easy_install` something into your system-level Python, you are downloading the module from the **Python Package Index** (**PyPI**, formerly known as The Cheese Shop). This comes in `.egg` format and gets placed into your system's site-packages directory. Since the `.egg` includes additional information, like dependencies, `easy_install` will attempt to install these as well.

In our case, we want to avoid installing things at the system level. So buildout creates and uses the `eggs` and `developer-eggs` directories in our project hierarchy as a place to `easy_install` modules. It will also create a Python interpreter that is aware of these special locations for our project. After the buildout finishes, we will use this interpreter, not the system Python, to run our application.

buildout.cfg: The buildout section

The buildout tool uses a configuration file to figure out what it needs to do to install our project and its dependencies. This is a text file called `buildout.cfg` that uses a very simple format.

The first thing we define is a buildout section, which is the only required portion of `buildout.cfg`. This section has two goals. First, it defines the parts our project needs to build in order to function. These are almost equivalent to dependencies our project needs in order to run. In our case this would be, at the very least, Django. The second goal is to define the modules we're developing (our web application, for example, the Coleman project we've built in this book). These will be translated into `developer-eggs`.

Before going any farther, we should look at an example `[buildout]` section from `buildout.cfg`:

```
[buildout]
develop = .
parts = django
eggs = django-coleman
```

Here you can see the `develop` and `parts` definitions. In the example, we only supply a single value to each configuration option, but in real-world projects these options can have as many values as needed, each separated by a space character.

The `develop` option points to `.`—the current directory. This tells buildout that the modules we're developing live in the project's root directory and that it will find a `setup.py` file there, which it can use to turn our modules into an `.egg`. In order for scripts in our buildout installation to access this egg, though, it needs to be added to the `eggs/` directory. We tell buildout to install our egg there with the `eggs = django-coleman` option. The name of our egg must match the name we specify in our setup script. Clearly we have additional work to do for this to work. We must now write this setup script, and then we will revisit the parts option.

Writing the setup script

The `setup.py` file is a setup script for our project. It is used to package up the modules we develop. When using `setuptools`, this package will be an egg. There are other cases, such as using pure `distutils`, where the setup script doesn't generate an egg. We will discuss the differences between `setuptools` and `distutils` setup scripts later in the chapter, but since `zc.buildout` uses `setuptools`, we will write our `setup.py` accordingly.

Our setup script can include all kinds of metadata, which `setuptools` will automatically roll up into the generated `.egg` file. For this example, we will keep it extremely simple, however, and specify only what is needed to get our module built:

```
from setuptools import setup
setup(
name='django-coleman',
```

```
    version='0.0.1',
    package_dir={'': 'coleman'},
          )
```

The name and version arguments are self explanatory, though important because they affect the ultimate name of the generated `.egg` file (`django-coleman-0.0.1.egg`). The `package_dir` argument points `setuptools` to a file-system location relative to the `setup.py` file where it will find our Python module. This setup script implies the following project layout (minus buildout's configuration and working directories):

```
setup.py
coleman/
coleman/__init__.py
 ...
```

The setup script will package the contents of the `coleman/` subdirectory into our egg.

This more than satisfies the basic requirements for our setup script. When we run our `buildout` command, it will be able to use `setup.py` to turn our project into an egg, and then store it in `eggs/` where our buildout scripts or interpreters can find it. We have effectively built our project inside a fully isolated Python environment, shut off from the rest of our system. The final step is to finish up our `buildout.cfg` by discussing our project's parts.

buildout.cfg: The parts sections

Parts are buildout's way of defining additional modules or components that our project needs in order to function. A part can be almost anything, but are often third-party packages from Python Package Index. We specify our project's parts in the buildout section of our `buildout.cfg` file using the `parts = option`. This is a list of part names separated by whitespace. Each part will have its own section later in our `buildout.cfg` file.

For each part we define, buildout needs to know how to install and/or configure it. It does this using a mechanism called a recipe. Several recipes come included with buildout, but we can also write custom ones. Recipes are just normal Python classes so they can do anything a Python script can do. This includes interacting with the operating system, local storage, or even compiling C extension modules.

A recipe is a Python class with three required methods: `__init__`, `install`, and `update`. These methods must perform a certain set of functions for buildout to successfully build the part for our project. We will not write a custom recipe here because most of the time the built-in recipes will work for us. In addition, hundreds of additional recipes have been contributed by the Python community and are available for download from PyPI. If none of these fit your own needs, it is easy to create a custom one and the process is well documented in the buildout documentation.

Among the recipes that come included with buildout or are available in PyPI is `djangorecipe`. This is a recipe designed specifically for building out Django projects. It has a lot of features, including the ability to set up a Django project and point to a settings file. The end result is a script in our buildout `bin/` called Django that wraps up all of these things and wraps Django's standard `manage.py` utility.

To help explain this, let's look at an example. Earlier we wrote the buildout section of our `buildout.cfg` file, here we will add another section for our Django part:

```
[buildout]
develop = .
parts = django
eggs = django-coleman

[django]
recipe = djangorecipe
version = 1.0.2
eggs = ${buildout:eggs}
```

The `[django]` section corresponds to our django part. We tell buildout to use the `djangorecipe`, which it will obtain automatically from PyPI, to build this part. We also tell the recipe we want Django version 1.0.2, using the version option. Finally, the eggs option tells buildout we want the same eggs to be available to Django as those defined in our buildout section.

When we run buildout in the directory with `buildout.cfg` the output should resemble the following:

```
Getting distribution for 'zc.recipe.egg'.
Got zc.recipe.egg 1.2.2.
Getting distribution for 'djangorecipe'.
Got djangorecipe 0.20.
Uninstalling django.
Installing django.
django: Downloading Django from: http://www.djangoproject.com/
download/1.0.2/tarball/
Generated script '/Users/jesse/src/django-coleman/bin/django'.
```

The `bin/` directory now contains a django script. This is the wrapper around the standard `manage.py` and supports all the commands you might expect: `runserver`, `syncdb`, `shell`, and so on. Only this script now comes prepared with our modules and any other eggs we specify available to Django. It also creates a `project/` directory in our buildout location, which contains a stock Django project configuration, including `settings.py` file and root `urls.py`. If we already have these things created in a Django project, we can include them in the options for our django part:

```
[django]
recipe = djangorecipe
version = 1.0.2
eggs = ${buildout:eggs}
project = myproject
```

If a `myproject` directory exists and contains our own settings and URLs files, the `djangorecipe` will not create new ones.

Now we have an isolated Python environment with four things:

- Our `django-coleman` modules built into an egg and installed in the `eggs/` location
- A Django project module with settings and root `urls.py`
- An installation of Django v1.0.2
- A django wrapper script in `bin/` preconfigured to use our project settings and the installed version of Django

We're all set to work with this fully configured instance of our Django project. We can add additional parts to our buildout file as needed and rerun buildout to download and install any additions. The `buildout.cfg` file can now be put into version control and, if we really want to get fancy, we can set up our development server to automatically run buildout whenever our configuration changes.

One thing is missing, however, and that is a straightforward way to deploy this isolated version of our project to an actual web server. The `djangorecipe` can help here too. We just need to add the following option to our django part in `buildout.cfg`:

```
wsgi = True
```

With this addition, buildout will create a `django.wsgi` file in the `bin/` directory. This WSGI script is configured to load our isolated project as a WSGI application and can be dropped directly into our Apache configuration file if we're using `mod_wsgi`. For the curious, this WSGI script looks like this:

```
#!/usr/bin/python
import sys
sys.path[0:0] = [
'/Users/jesse/src/django-coleman/eggs/django-coleman-0.0.1.egg',
'/Users/jesse/src/django-coleman/eggs/djangorecipe-0.20-py2.6.egg',
'/Users/jesse/src/django-coleman/eggs/zc.recipe.egg-1.2.2-py2.6.egg',
'/Users/jesse/src/django-coleman/parts/django',
'/Users/jesse/src/django-coleman,
  ]

import djangorecipe.wsgi
application = djangorecipe.wsgi.main('myproject.development',
logfile='')
```

As an even further level of automation, if we run `mod_wsgi` in daemon mode, whenever our buildout script runs and makes changes to our isolated environment, it will automatically update the WSGI script. This update will update the file's time stamp, which will act as a notice to Apache's `mod_wsgi` that the script has changed and needs to be reloaded. The `mod_wsgi` module will reload our entire project without restarting Apache and without requiring us to type any additional commands. Now that's automation!

Sometimes we need to include additional third-party modules, but don't want them installed into our project's isolated environment. Maybe we want to try out some functions in a package we're considering for use. We can do this with buildout as well by defining a new part, installing an egg to it and having buildout generate a custom interpreter. Such a parts section would look like this:

```
[testout]
recipe = zc.recipe.egg
interpreter = testout
eggs = elementtree
```

If we were to add a parts section like this to our `buildout.cfg` file, a couple of things will happen. First, an `elementtree` egg will be downloaded from PyPI and setup in our eggs location. Second, a custom Python interpreter will be created in the `bin/` location called `bin/testout`. When we run this interpreter, we will see a standard Python shell with only the `elementtree` egg available from our isolated environment's eggs. Our Django instance will not include this egg, keeping them nicely separated. This is especially useful when trying out different versions of a PyPI package.

Adding packages individually in this way is sometimes useful, but often too complicated. Buildout seems to work best as a tool for deployment onto servers or for building project packages. It can be used in many ways, but for local development tasks, it can feel like overkill to make changes to a configuration just to test out a package. There are better solutions for this use case, one of which we will discuss in the next section on virtual environments.

Finally, buildout can be used for more than just deployments. If you're aim is to write reusable modules that you want to contribute to the community, you can use buildout to automatically run unit tests and then package everything up into an egg. It can even upload and register your eggs with PyPI. There is much, much more than buildout can do and an excellent starting point for additional information is the buildout tutorial at: `http://www.buildout.org/docs/tutorial.html`.

Virtualenv

In the previous section we saw how `zc.buildout` could be used to generate an isolated project environment, including project files and dependencies. Buildout automated the process using a configuration file and a special filesystem hierarchy. It is a powerful tool, but may not suit every developer's style. Getting an environment up and running is a fair amount of work and making changes to try out new things requires editing the configuration file and rebuilding everything.

Environment isolation is a definite need for any serious Python or Django developer, though. It becomes very difficult to work in a situation where multiple projects are overlapping and there are conflicting package needs. Developing enhancements to our applications while maintaining a working production copy, like upgrading to a newer version of Django, for example, is impossible without some tool to manage our packages.

A lot of developers will take the obvious approach of creating special local package directories in their own workspace and attaching this location to the `$PYTHONPATH` environment variable or `sys.path`, swapping around a `.pth` file in the system `site-packages` location, or managing a collection of symbolic links.

If you've ever tried these approaches you'll probably recognize the difficulties this creates. First among them is that distributed Python applications that use `setuptools` or `distutils` and a `setup.py` script require a lot of special attention to install. Second, tools like `easy_install` install to the system `site-packages` by default. Both of these kill productivity and defeat the purpose of having a package index.

Buildout solves this problem, but it's not the only solution. Virtual environments take a different approach. Instead of writing configuration files and running build scripts, you can configure a virtual environment, and then use all the standard Python practices for installing packages and configuring tools. No changes to `$PYTHONPATH` or `sys.path`, no symbolic links, no funny business.

A virtual environment in some ways resembles the isolated environments we saw with buildout. A virtual environment is isolated from all other virtual environments, but not from the system environment. This means if we have installed packages to our system `site-packages` location, by default they will be available in our virtual environments as well. Otherwise packages are not available until we install them to the virtual environment.

There are currently several different virtual environment tools. Among these, virtualenv has gained a large following and provides an excellent all-around solution to the virtual environment problem. Virtualenv was written by top Python developer Ian Bicking and is available from PyPI at `http://pypi.python.org/pypi/virtualenv`.

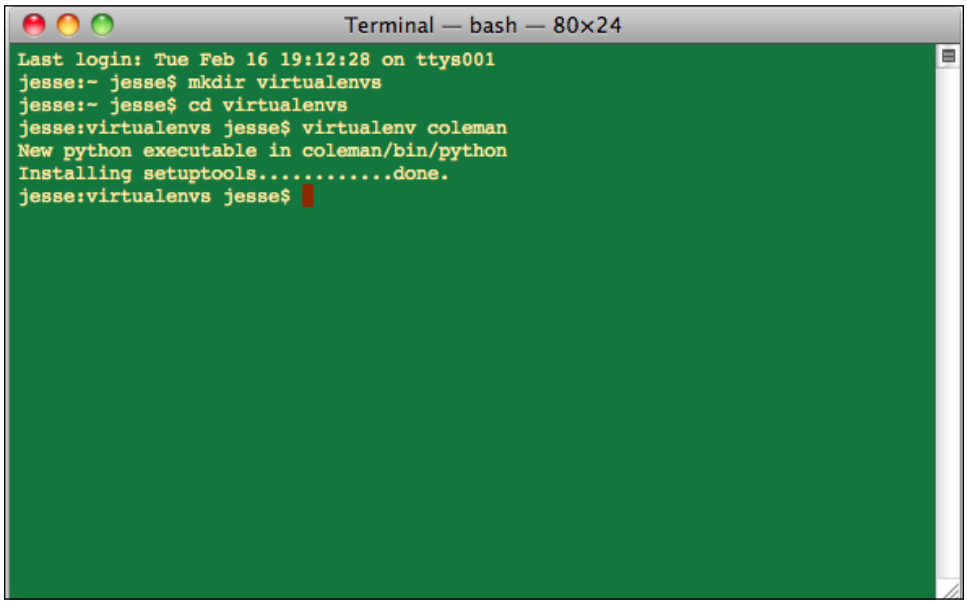
Creating an environment

Once installed, we can use the shell command virtualenv to manipulate environments. Every developer will have different preferences, but it's likely that you'll want to store all of your virtual environments in a single location, perhaps `/home/username/virtualenvs`.

When we change to our `virtualenvs` directory, we can issue the `virtualenv` command, following by a destination directory. This could be the name of our project or some other identifier to distinguish the environment we're creating. Let's create an environment for this book's `coleman` project:

```
$ virtualenv coleman
```

The results of this command are shown in the following screenshot:

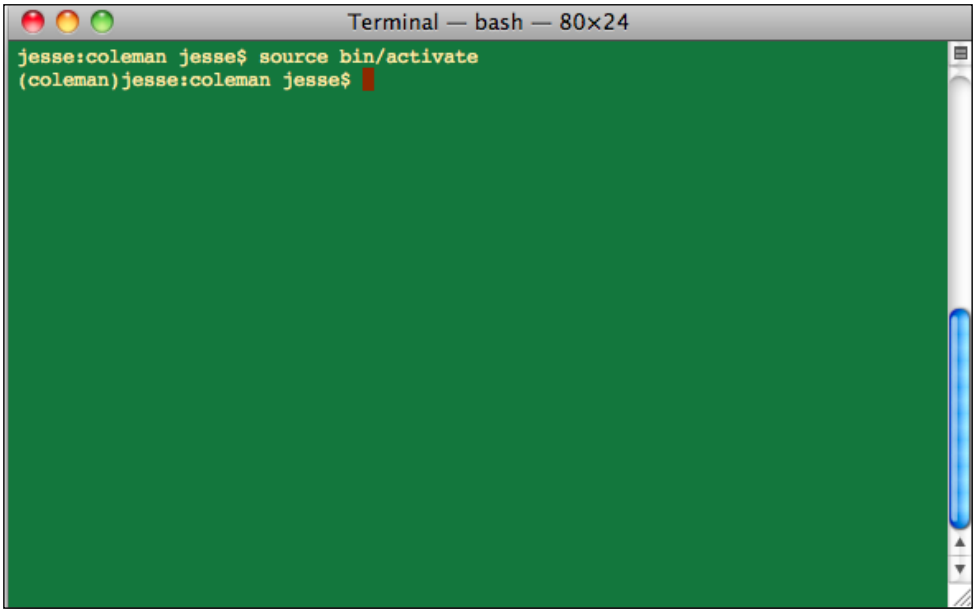


We've just created a complete virtual environment. If we navigate to our environment directory, we'll notice three subdirectories: `bin`, `include`, and `lib`. So far this is similar to what happened when we used buildout to create an environment.

Inside our virtual environment's `bin`, we see a copy of our Python interpreter. This interpreter is an actual interpreter binary, copied from our system's Python installation. This is unlike buildout, which creates wrappers around our system Python and provides it with an updated set of paths.

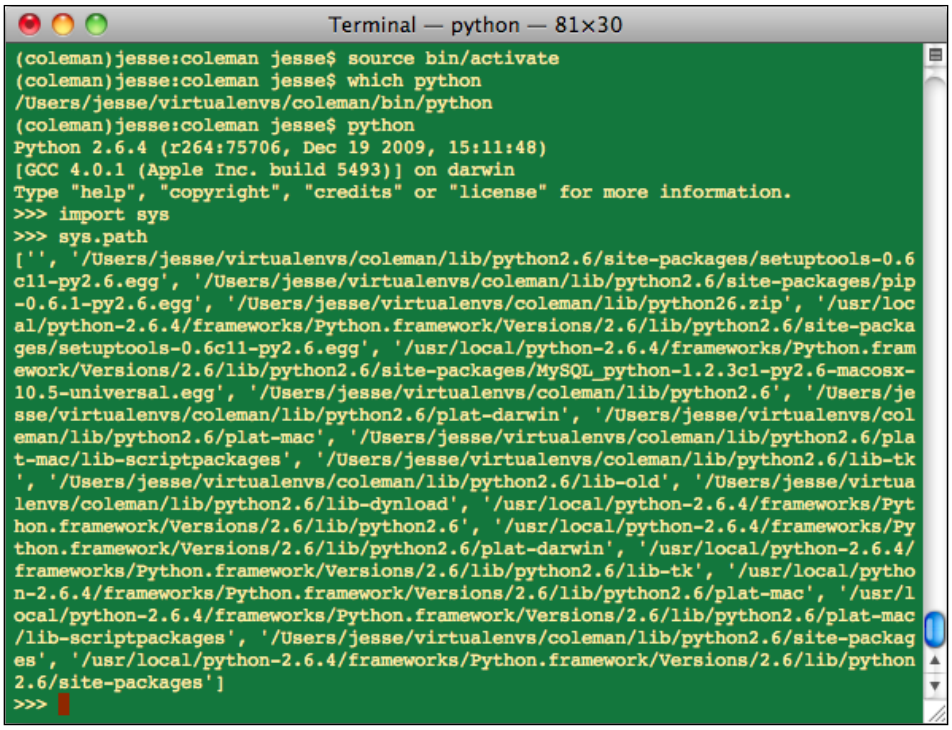
Even though the virtualenv interpreter is not editable, it does have knowledge of the virtual environment. If we were to run it as a Python shell, the path variable will reflect our virtual environment installation path. In order for this interpreter to be recognized as the default when we type `python` into a shell, however, we must first activate the environment.

Inside the `bin` location there is a shell script called `activate`. In order to activate the environment we have to run this shell script using the source command. Source is a built-in bash command that runs a script in our current shell (as opposed to starting a sub-shell). Running `activate` with source updates our shell environment and also adds a useful indicator to our shell prompt to inform us of what virtual environment we're currently using, as shown below:



The `activate` command has done two things: updated our shell's `$PATH` environment variable to include our virtual environment's bin location and updated a `$VIRTUAL_ENV` shell environment variable that points to our virtual environment directory.

When we ran virtualenv earlier, it actually performed some magic whereby our virtual environment's interpreter can detect our activated environment and adjust its path accordingly. The results of these operations are shown in the following screenshot:



Once we have activated an environment, we can deactivate it in a similar way by just typing `deactivate` at the shell prompt. Our virtual environment indicator should disappear from our prompt and we'll be back to using the system Python interpreter. We can activate other virtual environments by navigating to their location in the shell and running the new environment's `bin/activate`.

Working in the environment

Now that we've created a virtual environment for our project and activated it using the `bin/activate` script, we can go about our work. Because virtualenv has updated our shell path, every time we run python, our virtual environment's `python` will run.

Since we can now use the python command at will, something cool happens: we can begin installing packages using `setup.py` or `easy_install`. As long as our virtual environment is activated, we can `easy_install` to our heart's content and it all goes into our virtual environment.

In general, it is good practice when using virtual environments to keep the system `site-packages` location as light as possible. This prevents confusion and other potential mishaps. Our `easy_installs` from an activated virtual environment will include the system `site-packages` when looking for dependencies and such, but it is still recommended that most packages be installed virtually.

This will lead to some duplication if you are working on multiple projects that use the same set of packages. For these cases you may consider installing the package system-wide, especially if it's particularly important or difficult to install. The `MySQLdb` package and other database interfaces are good examples of candidates for system-wide installation.

Django would be an example of something not to install system wide, unless you are absolutely certain you won't be developing two separate projects that are running against different versions of the framework. Even if you are, still consider keeping all packages installed virtually. The disk space is there, why not use it?

Another important point to note is that the package manager for your operating system may, either by default or through a dependency negotiation, have installed packages to your Python's system `site-packages` without you necessarily being aware. It's a good idea to check out what packages and modules are installed in your system before switching to virtualenv. In fact, you may wish to empty your system `site-packages` altogether. You can uninstall any module by deleting its `.egg`, directory, and other metadata files.

Virtualenvwrapper

Doug Hellman has released a set of extension scripts for virtualenv called `virtualenvwrapper`. These are a set of shell functions that enhance the virtual environment configuration, primarily by organizing and structuring our environment locations. It includes pre and post hooks for several operations, including creating an environment, activating a new environment, and deactivating an environment.

The activation hooks allow for lots of interesting possibilities. For example, you could attach a post-activate hook that automatically runs a shell script or AppleScript to launch and configure your text editor whenever you change virtual environments.

The virtualenvwrapper scripts are a highly recommended enhancement to virtualenv. They are available from: `http://www.doughellmann.com/projects/virtualenvwrapper/`.

Distutils and module distributions

We've already seen some of what it takes to package an application. Packaging means combining the source code and other files for one or more Python modules into a single file that can be distributed to other Python developers. This could be a ZIP file or a Python egg.

We've already seen some of what `setuptools` can do. The `distutils` module is included with Python and is the *official* packaging tool that `setuptools` is built on. `Setuptools` is an extension to `distutils`, not an official part of Python.

One of the biggest differences between `distutils` and `setuptools` is that `setuptools` generates eggs, while `distutils` does not. Using `distutils`, developers create distributions as opposed to eggs. These distributions can take a couple of forms, namely source and built. Source distributions are generally just archived and compressed copies of a module's directories and source code. This is useful for distributing your modules to other developers.

Built distributions (or *binary* distributions) compile your modules into an executable format specific to a destination-build platform. This could be Window (`.exe`), Red Hat Linux (`.rpm`), or a variety of other formats.

Another noteworthy difference between `distutils` distributions and those created by `setuptools` is that `distutils` is unaware of dependencies and is unable to download them anyway, even if it knew about them. This feature is likely responsible for `setuptools` and `easy_install`'s popularity, despite not being an official part of Python.

To make use of `distutils` to distribute our modules, we need to write a setup script called `setup.py`, just as we did using `setuptools`. A lot the `setup.py` code is compatible with `distutils` and `setuptools`. Here is the simple example setup script from the `distutils` documentation:

```
from distutils.core import setup

setup(name='foo',
    version='1.0',
    py_modules=['foo'],)
```

This simple setup script includes two keyword arguments of metadata about our distribution, name and version, and a simple list of individual modules we want to include in our distribution. In this case, the `foo` module will be included and will correspond to a file called `foo.py` at the same level in the filesystem as our setup script.

This works for a single file or a short list of Python modules. But what about the case of packaging several packages, each with many modules? `Distutils` handles this as well:

```
setup(name='foo',
    version='1.0',
    packages=['foo', 'bar'],
    package_dir={'': 'src},)
```

Will look for modules in `src/foo/` and `src/bar/` relative to the setup script location and include them in the distribution output.

With this setup script in place and pointing to the appropriate modules or packages, we can run:

```
$ python setup.py sdist
```

The `sdist` command will generate a source distribution for our modules in a `.tar.gz` format. To create a built distribution we would use the following syntax for a Red Hat Linux .RPM file:

```
$ python setup.py bdist_rpm
```

Or, to create a Windows-compatible `.EXE` installation file we would use the following:

```
$ python setup.py bdist_windows
```

Installing distributions

In addition to providing distributions for developers to easily exchange modules between themselves, the `distutils` setup script allows users to install modules as well. This is done by passing the `install` command to the setup script:

```
$ python setup.py install
```

Usually, the process of installing a package like this involves downloading the `.tar.gz` source distribution, un-archiving the contents to the filesystem, and running the `setup.py` script as above.

Installing a package in this way places the Python modules included in our distribution into the user's `site-packages`. If they are running their default system Python, the package will be installed system-wide. If they are using virtualenv, for example, and have an active environment, the distribution will be installed to that environment.

Distutils metadata and PyPI

Even though `distutils` does not provide a mechanism for installing packages by directly downloading from PyPI, you can still define meta-data inside your `setup.py` script and use it to register and upload your distribution with the Python Package Index.

Uploading the distribution to PyPI also acts as a form of publishing your project. By providing version, author, and dependency metadata in `setup.py`, these requirements will be included on your distributions web page at the package index.

Registering and uploading your distribution with PyPI is done using the register `setup.py` command:

```
$ python setup.py register
```

Registering your distribution will transmit any metadata defined in `setup.py`, such as version and author's e-mail address, and create a project page for it in the index. This page will list all metadata you supplied in your `setup.py`. You can update a distribution's metadata by registering it again, with corrected metadata keyword arguments in your setup script.

You will be required to have a PyPI account before you can register your distributions. You can do this directly in the `setup.py` register command by choosing the second menu option when prompted.

Once your package is registered, you can upload it using the `upload` command:

```
$ python setup.py sdist bdist_wininst upload
```

This upload command will package and upload a source distribution as well as a built Windows install distribution. Both will be listed under the appropriate metadata in the PyPI.

Easy_install

Python packages deployed to PyPI using `setuptools` have an additional advantage: super-simple installation using the `easy_install` tool. This is a utility that with a single command, will find, download, and install a Python package from the package index. To install Django, for example, one can attempt it from the command line like so:

```
$ easy_install Django
```

This will download the latest released version of Django from PyPI and install it into the current Python environment, whether it is the system environment or a virtual environment. In this way it works very much like `setup.py`, with the enhanced function of automatically locating and downloading the appropriate package.

Pip

`Pip` is a powerful replacement for the `easy_install` tool. It is a widely recommended utility, especially for use in virtual environments, because it offers many enhancements to the original `easy_install` tool and some differences in philosophy too.

First, `pip` offers the ability to uninstall packages. It also keeps better track of why a package needs to be installed, what dependency it satisfies, and provides logs of these details. Likewise, no packages are installed until all of the required packages have completed downloading. These features allow us to keep cleaner environments that are easier to manage and maintain.

Second, and most important for use with virtualenv, is that `pip` will respect virtual environments. This means packages installed with `pip` while running virtualenv will install to the appropriate active environment location.

`Pip` also allows us to write requirements files. These are files that specify version requirements for a set of Python packages. This way we can simplify the installation of complicated sets of version-dependent packages.

`Pip` is quickly becoming the package installation system of choice in the Python community. Its use has many advantages over the traditional `easy_install` approach and is highly recommended.

Summary

In this chapter we have provided a quick overview of various deployment and maintenance tools. There are literally dozens of additional tools available in the Django and web development community. Many of these have reached an exceedingly high level of quality and are used in many top Django and Python development shops. This includes:

- `Mod_wsgi` to simplify Django deployments and reduce memory overhead
- Fabric to automatically perform remote deployment activities over Secure Shell (SSH)
- Buildout and Virtualenv to create isolated project environments
- Creating reusable, distributable packages using `distutils`

Investing in a deployment process is essential for projects of a certain size. Not only will automation help make the tasks easier, but it makes it repeatable by people potentially unfamiliar with a deployment situation. Having good tools and good documentation are the best way to help someone who may be trying to solve a deployment issue in the middle of the night.

Also remember that there is often no right approach to the problems any of these tools are trying to solve. In some cases there may be objectively better approaches—they may require fewer steps or fewer configurations to achieve the same thing. But even a slightly flawed process is going to be a marked improvement over nothing at all.

Index

Symbols

A

B

C

D

E

F

G

H

Q

R

S

T

U

V

W

X

Y

Z

Thank you for buying

Django 1.2 e-commerce

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: `www.packtpub.com`.

About Packt Open Source

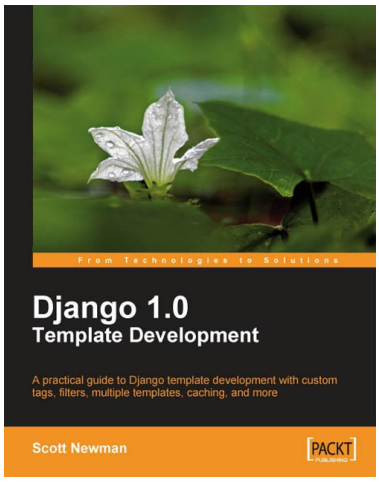
In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around Open Source licences, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each Open Source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

[PACKT] open source
community experience distilled
PUBLISHING

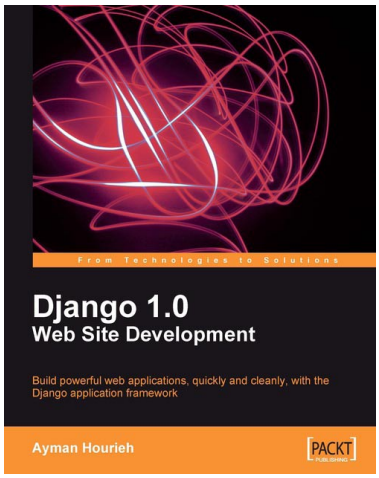


Django 1.0 Template Development

ISBN: 978-1-847195-70-8 Paperback: 272 pages

A practical guide to Django template development with custom tags, filters, multiple templates, caching, and more

1. Dive into Django's template system and build your own template
2. Learn to use built-in tags and filters in Django 1.0
3. Practical tips for project setup and template structure



Django 1.0 Website Development

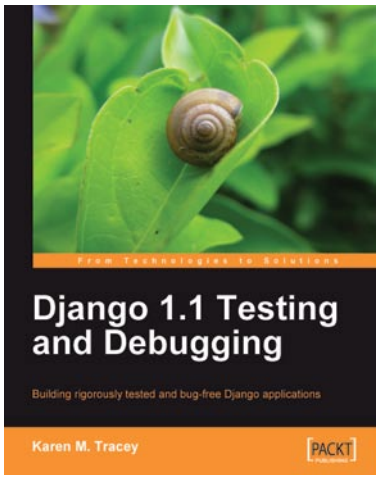
ISBN: 978-1-847196-78-1 Paperback: 272 pages

Build powerful web applications, quickly and cleanly, with the Django application framework

1. Teaches everything you need to create a complete Web 2.0-style web application with Django 1.0
2. Learn rapid development and clean, pragmatic design
3. No knowledge of Django required
4. Packed with examples and screenshots for better understanding

Please check **www.PacktPub.com** for information on our titles

[PACKT] open source
community experience distilled
PUBLISHING

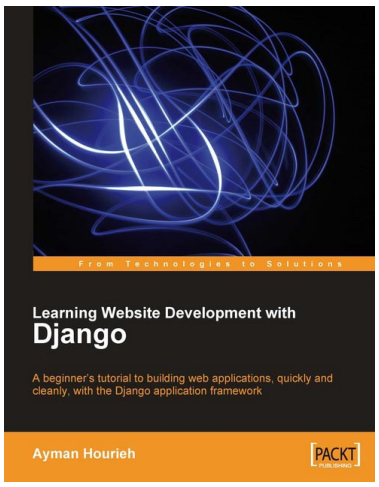


Django 1.1 Testing and Debugging

ISBN: 978-1-847197-56-6 Paperback: 436 pages

Building rigorously tested and bug-free Django applications

1. Develop Django applications quickly with fewer bugs through effective use of automated testing and debugging tools.
2. Ensure your code is accurate and stable throughout development and production by using Django's test framework.
3. Understand the working of code and its generated output with the help of debugging tools.



Learning Website Development with Django

ISBN: 978-1-847193-35-3 Paperback: 264 pages

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

1. Create a complete Web 2.0-style web application with Django
2. Learn rapid development and clean, pragmatic design
3. Build a social bookmarking application
4. No knowledge of Django required

Please check **www.PacktPub.com** for information on our titles