django

Web framework for Python

Django Book: pdf version

compiled by Suvash Sedhain

bir2su.blogspot.com

Visit www.djangobook.com for online version of the book

The Django Book

Table of contents

Beta, English

Chapter 1: Introduction to Django

If you go to the Web site djangoproject.com using your Web browser — or, depending on the decade in which you're reading this destined-to-be-timeless literary work, using your cell phone, electronic notebook, shoe, or any Internet-superceding contraption — you'll find this explanation:

> "Django is a high-level Python Web framework that encourages rapid development and clean, pragmatic design."

That's a mouthful — or eyeful or pixelful, depending on whether this book is being recited, read on paper or projected to you on a Jumbotron, respectively.

Let's break it down.

Django is a high-level Python Web framework…

A high-level Web framework is software that eases the pain of building dynamic Web sites. It abstracts common problems of Web development and provides shortcuts for frequent programming tasks.

For clarity, a dynamic Web site is one in which pages aren't simply HTML documents sitting on a server's filesystem somewhere. In a dynamic Web site, rather, each page is generated by a computer program — a so-called "Web application" — that you, the Web developer, create. A Web application may, for instance, retrieve records from a database or take some action based on user input.

A good Web framework addresses these common concerns:

- **It provides a method of mapping requested URLs to code that handles requests.** In other words, it gives you a way of designating which code should execute for which URL. For instance, you could tell the framework, "For URLs that look like `/users/joe/`, execute code that displays the profile for the user with that username."
- **It makes it easy to display, validate and redisplay HTML forms.** HTML forms are the primary way of getting input data from Web users, so a Web framework had better make it easy to display them and handle the tedious code of form display and redisplay (with errors highlighted).
- **It converts user-submitted input into data structures that can be manipulated conveniently.** For example, the framework could convert HTML form submissions into native data types of the programming language you're using.
- **It helps separate content from presentation via a template system**, so you can change your site's look-and-feel without affecting your content, and vice-versa.
- **It conveniently integrates with storage layers** — such as databases — but doesn't strictly require the use of a database.
- **It lets you work more productively, at a higher level of abstraction**, than if you were coding against, say, HTTP. But it doesn't restrict you from going "down" one level of abstraction when needed.
- **It gets out of your way**, neglecting to leave dirty stains on your application such as URLs that contain ".aspx" or ".php".

Django does all of these things well — and introduces a number of features that raise the bar for what a Web framework should do.

The framework is written in Python, a beautiful, concise, powerful, high-level programming language. To develop a site using Django, you write Python code that uses the Django libraries. Although this book doesn't include a full Python tutorial, it highlights Python features and functionality where appropriate, particularly when code doesn't immediately make sense.

…that encourages rapid development…

Regardless of how many powerful features it has, a Web framework is worthless if it doesn't save you time. Django's philosophy is to do all it can to facilitate hyper-fast development. With Django, you build Web sites in a matter of hours, not days; weeks, not years.

This is possible largely thanks to Python itself. Oh, Python, how we love thee, let us count the bullet points:

- Python is an **interpreted language**, which means there's no need to compile code. Just write your program and execute it. In Web development, this means you can develop code and immediately see results by hitting "reload" in your Web browser.
- Python is **dynamically typed**, which means you don't have to worry about declaring data types for your variables.
- Python syntax is **concise yet expressive**, which means it takes less code to accomplish the same task than in other, more verbose, languages such as Java. One line of python usually equals 10 lines of Java. (This has a convenient side benefit: Fewer lines of code means fewer bugs.)
- Python offers **powerful introspection and meta-programming** features, which make it possible to inspect and add

behavior to objects at runtime.

Beyond the productivity advantages inherent in Python, Django itself makes every effort to encourage rapid development. Every part of the framework was designed with productivity in mind. We'll see examples throughout this book.

..and clean, pragmatic design

Finally, Django strictly maintains a clean design throughout its own code and makes it easy to follow best Web-development practices in the applications you create.

That means, if you think of Django as a car, it would be an elegant sports car, capable not only of high speeds and sharp turns, but delivering excellent mileage and clean emissions.

The philosophy here is: Django makes it easy to do things the "right" way.

Specifically, Django encourages loose coupling: the programming philosophy that different pieces of the application should be interchangeable and should communicate with each other via clear, concise APIs.

For example, the template system knows nothing about the database-access system, which knows nothing about the HTTP request/response layer, which knows nothing about caching. Each one of these layers is distinct and loosely coupled to the rest. In practice, this means you can mix and match the layers if need be.

Django follows the "model-view-controller" (MVC) architecture. Simply put, this is a way of developing software so that the code for defining and accessing data (the model) is separate from the business logic (the controller), which in turn is separate from the user interface (the view).

MVC is best explained by an example of what *not* to do. For instance, look at the following PHP code, which retrieves a list of people from a MySQL database and outputs the list in a simple HTML page. (Yes, we realize it's possible for disciplined programmers to write clean PHP code; we're simply using PHP to illustrate a point.):

```
<html>
<head><title>Friends of mine</title></head>
<body>

<h1>Friends of mine</h1>

<ul>

<?php
$connection = @mysql_connect("localhost", "my_username", "my_pass");
mysql_select_db("my_database");
$people = mysql_query("SELECT name, age FROM friends");
while ( $person = mysql_fetch_array($people, MYSQL_ASSOC) ) {
?>
<li>
<?php echo $person['name'] ?> is <?php echo $person['age'] ?> years old.
</li>
<?php } ?>

</ul>

</body>
</html>
```

While this code is conceptually simple for beginners — because everything is in a single file — it's bad practice for several reasons:

1. **The presentation is tied to the code.** If a designer wanted to edit the HTML of this page, he or she would have to edit this code, because the HTML and PHP core are intertwined.

   By contrast, the Django/MVC approach encourages separation of code and presentation, so that presentation is governed by templates and business logic lives in Python modules. Programmers deal with code, and designers deal with HTML.

2. **The database code is tied to the business logic.** This is a problem of redundancy: If you rename your database tables or columns, you'll have to rewrite your SQL.

   By contrast, the Django/MVC approach encourages a single, abstracted data-access layer that's responsible for all data access. In Django's case, the data-access layer knows your database table and column names and lets you execute SQL queries via Python instead of writing SQL manually. This means, if database table names change, you can change it in a single place — your data-model definition — instead of in each SQL statement littered throughout your code.

3. **The URL is coupled to the code.** If this PHP file lives at `/foo/index.php`, it'll be executed for all requests to that address. But what if you want this same code to execute for requests to `/bar/` and `/baz/`? You'd have to set up some sort of includes or rewrite rules, and those get unmanageable quickly.

By contrast, Django decouples URLs from callback code, so you can change the URLs for a given piece of code.

4. **The database connection parameters and backend are hard-coded.** It's messy to have to specify connection information — the server, username and password — within this code, because that's configuration, not programming logic. Also, this example hard-codes the fact that the database engine is MySQL.

   By contrast, Django has a single place for storing configuration, and the database-access layer is abstracted so that switching database servers (say, from MySQL to PostgreSQL) is easy.

What Django doesn't do

Of course, we want this book to be fair and balanced. With that in mind, we should be honest and outline what Django *doesn't* do:

- Feed your cat.
- Mind-read your project requirements and implement them on a carefully timed basis so as to fool your boss into thinking you're not really staying home to watch "The Price is Right."

On a more serious note, Django does not yet reverse the effects of global warming.

Why was Django developed?

Django is deeply rooted in the problems and solutions of the Real World. It wasn't created to be marketed and sold to developers, nor was it created as an academic exercise in somebody's spare time. It was built from Day One to solve daily problems for an industry-leading Web-development team.

It started in fall 2003, at — wait for it — a small-town newspaper in Lawrence, Kansas.

For one reason or another, The Lawrence Journal-World newspaper managed to attract a talented bunch of Web designers and developers in the early 2000s. The newspaper's Web operation, World Online, quickly turned into one of the most innovative newspaper Web operations in the world. Its three main sites, LJWorld.com (news), Lawrence.com (entertainment/music) and KUsports.com (college sports), began winning award after award in the online-journalism industry. Its innovations were many, including:

- The most in-depth local entertainment site in the world, Lawrence.com, which merges databases of local events, bands, restaurants, drink specials, downloadable songs and traditional-format news stories.
- A summer section of LJWorld.com that treated local Little League players like they were the New York Yankees — giving each team and league its own page, hooking into weather data to display forecasts for games, providing 360-degree panoramas of every playing field in the vicinity and alerting parents via cell-phone text messages when games were cancelled.
- Cell-phone game alerts for University of Kansas basketball and football games, which let fans get notified of scores and key stats during games, and a second system that used artificial-intelligence algorithms to let fans send plain-English text messages to the system to query the database ("how many points does giddens have" or "pts giddens").
- A deep database of all the college football and basketball stats you'd ever want, including a way to compare any two or more players or teams in the NCAA.
- Giving out blogs to community members and featuring community writing prominently — back before blogs were trendy.

Journalism pundits worldwide pointed to World Online as an example of the future of journalism. The New York Times did a front-page business-section story on the company; National Public Radio did a two-day series on it. World Online's head editor, Rob Curley, spoke nearly *weekly* at journalism conferences across the globe, showcasing World Online's innovative ideas and site features. In a bleak, old-fashioned industry resistant to change, World Online was a rare exception.

Much of World Online's success was due to the technology behind its sites, and the philosophy that computer programmers are just as important in creating quality 21st Century journalism as are journalists themselves.

This is why Django was developed: World Online's developers needed a framework for developing complex database-driven Web sites painlessly, easily and on journalism deadlines.

In fall 2003, World Online's two developers, Adrian Holovaty and Simon Willison, set about creating this framework. They decided to use Python, a language with which they'd recently fallen in love. After exploring (and being disappointed by) the available Python Web-programming libraries, they began creating Django.

Two years later, in summer 2005, after having developed Django to a point where it was efficiently powering most of World Online's sites, the World Online team, which now included Jacob Kaplan-Moss, decided it'd be a good idea to open-source the framework. That way, they could give back to the open-source community, get free improvements from outside developers, and generate some buzz for their commercial Django-powered content-management system, Ellington (http://www.ellingtoncms.com/). Django was open-sourced in July 2005 and quickly became popular.

Although Django is now an open-source project with contributors across the planet, the original World Online developers still provide central guidance for the framework's growth, and World Online contributes other important aspects such as employee time, marketing materials and hosting/bandwidth for the framework's Web site (http://www.djangoproject.com/).

Who uses Django?

Web developers around the world use Django. Some specific examples:

- World Online, of course, continues to use Django for all its Web sites, both internal and for commercial clients. Some of its Django-powered sites are:
  - http://www.ljworld.com/
  - http://www.lawrence.com/
  - http://www.6newslawrence.com/
  - http://www.visitlawrence.com/
  - http://www.lawrencechamber.com/
  - http://www2.kusports.com/stats/
- The Washington Post's Web site, washingtonpost.com, uses Django for database projects and various bits of functionality across the site. Some examples:
  - The U.S. Congress votes database, http://projects.washingtonpost.com/congress/
  - The staff directory and functionality that lets readers contact reporters, appearing as links on most article pages.
  - Faces of the Fallen, http://projects.washingtonpost.com/fallen/
- Chicagocrime.org, a freely browsable database of crime reported in Chicago and one of the original Google Maps mashups, was developed in Django.
- Tabblo.com, an innovative photo-sharing site, uses Django. The site lets you piece together your photos to create photo pages that tell stories.
- Texasgigs.com, a local music site in Dallas, Texas, was written with Django.
- Grono.net, a Polish social-networking site, started replacing its Java code with Django. It found that Django not only was faster (and more fun) to develop in — it performed better than Java and required less hardware.
- Traincheck.com was developed in Django. The site lets you send text-messages from your cell phone to get subway train schedules for your immediate location.

An up-to-date list of dozens of sites that use Django is located at http://code.djangoproject.com/wiki/DjangoPoweredSites

About this book

The goal of this book is to explain all the things Django does — and to make you an expert at using it.

By reading this book, you'll learn the skills needed to develop powerful Web sites quickly, with code that's clean and easy to maintain.

We're glad you're here!

table of contents ◊ next »

Chapter 2: Getting started

Let's get started, shall we?

Fortunately, installing Django is easy. Because Django runs anywhere Python does, Django can be configured in many ways. We've tried to cover the common scenarios for Django installations in this chapter.

Installing Python

Django is written in 100% pure Python code, so you'll need to install Python on your system. Django requires Python 2.3 or higher.

If you're on Linux or Mac OS X, you probably already have Python installed. Type `python` at a command prompt (or in Terminal, in OS X). If you see something like this, then Python is installed:

```
Python 2.4.1 (#2, Mar 31 2005, 00:05:10)
[GCC 3.3 20030304 (Apple Computer, Inc. build 1666)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Otherwise, if you see an error such as "`command not found`", you'll have to download and install Python. See http://www.python.org/download/ to get started. The installation is fast and easy.

Installing Django

Installing an official release

Most people will want to install the latest official release from http://www.djangoproject.com/download/. Django uses the standard Python `distutils` installation method, which in Linux land looks like:

1. Download the tarball, which will be named something like `Django-1.0.tar.gz`.
2. `tar xzvf Django-*.tar.gz`
3. `cd Django-*`
4. `sudo python setup.py install`

If everything worked, you should be able to import the module `django` from the Python interactive interpreter.

```
>>> import django
>>> django.VERSION
(1, 0, 'official')
```

The Python interactive interpreter:

The Python interactive interpreter is a command-line program that lets you write a Python program interactively. To start it, just run the command `python` at the command line. Throughout this book, we'll feature example Python code that's printed as if it's being entered in the interactive interpreter. The triple greater-than signs (">>>") signify a prompt.

Installing Django from Subversion

If you want to work on the bleeding edge, or if you want to contribute code to Django itself, you should install Django from its Subversion repository.

Subversion is a free, open-source revision-control system similar to CVS, and the Django team uses it to manage changes to the Django codebase. At any given time, you can use a Subversion client to grab the very latest Django source code, and, at any given time, you can update your local version of the Django code — known as your "local checkout" — to get the latest changes and improvements made by Django developers.

The latest-and-greatest Django development code is referred to as "the trunk."

To grab the latest Django trunk:

1. Make sure you have a Subversion client installed. You can get the software free from http://subversion.tigris.org/ and excellent documentation from http://svnbook.red-bean.com/
2. Check out the trunk using the command `svn co http://code.djangoproject.com/svn/django/trunk django_src`
3. Symlink `django_src/django` so that `django` is within your Python `site-packages` directory, or update your PYTHONPATH to point to it.

When installing from Subversion, you don't need to run `python setup.py install`.

Because the Django trunk changes often with bug fixes and feature additions, you'll probably want to update it every once in a while — or hourly, if you're really obsessed. To update the code, just run the command `svn update` from within the `django_src` directory. When you run that command, Subversion will contact our Web server, see if any code has changed and update your local version of the code with any changes that have been made since you last updated. It's quite slick.

Setting up a database

Django's only prerequisite is a working installation of Python. However, this book focuses on one of Django's sweet spots, which is developing *database-backed* Web sites — so you'll need to install a database server of some sort, for storing your data.

If you just want to get started playing with Django, skip ahead to Starting a project, but trust us — you'll want to install a database eventually. All of the examples in the book assume you've got a database set up.

As of version 1.0, Django supports five database engines:

- PostgreSQL (http://www.postgresql.org/)
- SQLite 3 (http://www.sqlite.org/)
- MySQL (http://www.mysql.com/)
- Microsoft SQL Server (http://www.microsoft.com/sql/)
- Oracle (http://www.oracle.com/database/)

We're quite fond of PostgreSQL ourselves, for reasons outside the scope of this book, so we mention it first. However, all those engines will work equally well with Django.

SQLite also deserves special notice: It's an extremely simple in-process database engine that doesn't require any sort of server set up or configuration. It's by far the easiest to set up if you just want to play around with Django.

Using Django with PostgreSQL

If you're using PostgreSQL, you'll need the `psycopg` package available from http://initd.org/projects/psycopg1. Make sure you use version 1, not version 2 (which is still in beta).

If you're using PostgreSQL on Windows, you can find precompiled binaries of `psycopg` at http://stickpeople.com/projects/python/win-psycopg/.

Using Django with SQLite 3

You'll need SQLite 3 — not version 2 — and the `pysqlite` package from http://initd.org/tracker/pysqlite. Make sure you have `pysqlite` version 2.0.3 or higher.

Using Django with MySQL

Django requires MySQL 4.0 or above; the 3.x versions don't support transactions, nested procedures, and some other fairly standard SQL statements. You'll also need the `MySQLdb` package from http://sourceforge.net/projects/mysql-python.

Using Django with MSSQL

Using Django with Oracle

Using Django without a database

As mentioned above, Django doesn't actually require a database. If you just want to use it to serve dynamic pages that don't hit a database, that's perfectly fine.

With that said, bear in mind that some of the extra tools bundled with Django *do* require a database, so if you choose not to use a database, you'll miss out on those features. (We'll highlight these features throughout this book.)

Starting a project

If this is your first time using Django, you'll have to take care of some initial setup.

Run the command `django-admin.py startproject mysite`. That'll create a `mysite` directory in your current directory.

Note

`django-admin.py` should be on your system path if you installed Django via its setup.py utility. If it's not on your path, you can find it in `site-packages/django/bin`; consider symlinking to it from some place on your path, such as /usr/local/bin.

A project is a collection of settings for an instance of Django — including database configuration, Django-specific options and application-specific settings. Let's look at what `startproject` created:

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

These files are:

`manage.py`
A command-line utility that lets you interact with this Django project in various ways.

`settings.py`
Settings/configuration for this Django project.

`urls.py`
The URL declarations for this Django project; a "table of contents" of your Django-powered site.

Where should this code live?

If your background is in PHP, you're probably used to putting code under the Web server's document root (in a place such as `/var/www`). With Django, you don't do that. It's not a good idea to put any of this Python code within your Web server's document root, because it risks the possibility that people may be able to view your code over the Web. That's not good for security.

Put your code in some directory **outside** of the document root, such as `/home/mycode`.

The development server

Change into the `mysite` directory, if you haven't already, and run the command `python manage.py runserver`. You'll see something like this:

```
Validating models...
0 errors found.

Django version 1.0, using settings 'mysite.settings'
Development server is running at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

You've started the Django development server, a lightweight Web server you can use while developing your site. We've included this with Django so you can develop things rapidly, without having to deal with configuring your production Web server (e.g., Apache) until you're ready for production. This development server watches your code for changes and automatically reloads, helping you make many rapid changes to your project without needing to restart anything.

Although the development server is extremely nice for, well, development, resist the temptation to use this server in anything resembling a production environment. The development server can only handle a single request at a time reliably, and it has not gone through a security audit of any sort. When the time comes to launch your site, see Chapter XXX for information on how to deploy Django.

Changing the host or the port

By default, the `runserver` command starts the development server on port 8000, listening only for local connections. If you want to change the server's port, pass it as a command-line argument:

```
python manage.py runserver 8080
```

You can also change the IP address that the server listens on. This is especially helpful if you'd like to share a development site with other developers:

```
python manage.py runserver 0.0.0.0:8080
```

will make Django listen on any network interface, thus allowing other computers to connect to the development server.

Now that the server's running, visit http://127.0.0.1:8000/ with your Web browser. You'll see a “Welcome to Django” page, in pleasant, light-blue pastel. It worked!

What's next?

Now that we've got everything installed and the development server running, let's write some basic code that demonstrates how to serve Web pages using Django.

« previous ◊ table of contents ◊ next »

Chapter 3: The basics of dynamic Web pages

In the previous chapter, we explained how to set up a Django project and run the Django development server. Of course, that site doesn't actually do anything useful yet — all it does is display the "It worked!" message. Let's change that.

This chapter introduces how to create dynamic Web pages with Django.

Your first view: Dynamic content

As our first goal, let's create a Web page that displays the current date and time. This is a good example of a *dynamic* Web page, because the contents of the page are not static — rather, the contents change according to the result of a computation (in this case, a calculation of the current time).

This simple example doesn't involve a database or any sort of user input — just the output of your server's internal clock.

To create this page, we'll write a **view function**. A view function, or **view** for short, is simply a Python function that takes a Web request and returns a Web response. This response can be the HTML contents of a Web page, or a redirect, or a 404 error, or an XML document, or an image...or anything, really. The view itself contains whatever arbitrary logic is necessary to return that response.

Here's a view that returns the current date and time, as an HTML document:

```
from django.http import HttpResponse
import datetime

def current_datetime(request):
    now = datetime.datetime.now()
    html = "<html><body>It is now %s.</body></html>" % now
    return HttpResponse(html)
```

Let's step through this code one line at a time:

- First, we import the class `HttpResponse`, which lives in the `django.http` module.
- Next, we import the `datetime` module from Python's standard library — the set of useful modules that comes with Python. The `datetime` module contains several functions and classes for dealing with dates and times, including a function that returns the current time.
- Next, we define a function called `current_datetime`. This is the **view function**, and, as such, it takes an `HttpRequest` object as its first parameter. Each view function takes an `HttpRequest` object as its first parameter. In this case, we call that parameter `request`.

  Note that the name of the view function doesn't matter; Django doesn't care what it's called, and it doesn't have to be named in a certain way in order for Django to recognize it. We're calling it `current_datetime` here, because that name clearly indicates what it does, but it could just as well be named `super_duper_awesome_current_time`, or something equally revolting. Django doesn't care. (How does Django find this function, then? We'll get to that in a moment.)
- The first line of code within the function calculates the current date/time, as a `datetime.datetime` object, and stores that as the local variable `now`.
- The second line of code within the function constructs an HTML response using Python's format-string capability. The `%s` within the string is a placeholder, and the percent sign after the string means "replace the `%s` with the value of the variable `now`."

  (A note to the HTML purists: Yes, we know we're missing a DOCTYPE, and a `<head>`, and all that stuff. We're trying to keep it simple.)
- Finally, the view returns an `HttpResponse` object that contains the generated HTML. Each view function is responsible for returning an `HttpResponse` object. (There are exceptions, but we'll get to those later.)

Your first URLconf

So, to recap, this view function returns an HTML page that includes the current date and time. But where should this code live, how do we tell Django to use this code?

The answer to the first question is: This code can live anywhere you want, as long as it's on your Python path. There's no other requirement — no "magic," so to speak. For the sake of putting it *somewhere*, let's create a file called `views.py`, copy this view code into that file and save it into the `mysite` directory you created in the previous chapter.

Your Python path

The Python path is the list of directories on your system where Python looks when you use the Python `import` statement.

For example, let's say your Python path is set to `['', '/usr/lib/python2.4/site-packages', '/home/mycode']`. If you execute the Python code `from foo import bar`, Python will first check for a module called `foo.py` in the current directory. (The first entry in the Python path, an empty string, means "the current directory.") If that file doesn't exist, Python will look for the file `/usr/lib/python2.4/site-packages/foo.py`. If that file doesn't exist, it will try `/home/mycode/foo.py`. Finally, if *that* file doesn't exist, it will raise `ImportError`.

If you're interested in seeing the value of your Python path, start the Python interactive interpreter and type `import sys`, followed by `print sys.path`.

Generally you don't have to worry about setting your Python path — Python and Django will take care of things for you automatically behind the scenes. (If you're curious, setting the Python path is one of the things that the `manage.py` file does.)

How do we tell Django to use this view code? That's where URLconfs come in.

A **URLconf** is like a table of contents for your Django-powered Web site. Basically, it's a mapping between URL patterns and the view functions that should be called for those URL patterns. It's how you tell Django "For this URL, call this code, and for that URL, call that code."

When you executed `django-admin.py startproject` in the previous chapter, the script created a URLconf for you automatically: the file `urls.py`. Let's edit that file. By default, it looks something like this:

```
from django.conf.urls.defaults import *

urlpatterns = patterns('',
    # Example:
    # (r'^mysite/', include('mysite.apps.foo.urls.foo')),

    # Uncomment this for admin:
#     (r'^admin/', include('django.contrib.admin.urls')),
)
```

Let's step through this code one line at a time:

- The first line imports all objects from the `django.conf.urls.defaults` module, including a function called `patterns`.
- The second line calls the function `patterns()` and saves the result into a variable called `urlpatterns`. The `patterns()` function gets passed only a single argument — the empty string. The rest of the lines are commented out.

The main thing to see here is the variable `urlpatterns`. This defines the mapping between URLs and the code that handles those URLs.

By default, everything in the URLconf is commented out — your Django application is a blank slate. (As a side note, that's how Django knew to show you the "It worked!" page in the last chapter: If your URLconf is empty, Django assumes you just started a new project and, hence, displays that message.)

Let's edit this file to expose our `current_datetime` view:

```
from django.conf.urls.defaults import *
from mysite.views import current_datetime

urlpatterns = patterns('',
    (r'^now/$', current_datetime),
)
```

We made two changes here. First, we imported the `current_datetime` view from its module (`mysite/views.py`, which translates into `mysite.views` in Python import syntax). Next, we added the line `(r'^now/$', current_datetime),`. This line is referred to as a **URLpattern** — it's a Python tuple in which the first element is a simple regular expression and the second element is the view function to use for that pattern.

In a nutshell, we just told Django that any request to the URL `/now/` should be handled by the `current_datetime` view function.

A few things are worth pointing out:

- Note that, in this example, we passed the `current_datetime` view function as an object without calling the function. This is a key feature of Python (and other dynamic languages): Functions are first-class objects, which means you can pass them around just like any other variables. Cool stuff, eh?

- There's no need to add a slash at the beginning of the `'^now/$'` expression in order to match `/now/`. Django automatically puts a slash before every expression.
- The caret character (`'^'`) and dollar sign character (`'$'`) are important. The caret means "require that the pattern matches the start of the string," and the dollar sign means "require that the pattern matches the end of the string."

  This concept is best explained by example. If we had instead used the pattern `'^now/'` (without a dollar sign at the end), then *any* URL that starts with `now/` would match — such as `/now/foo` and `/now/bar`, not just `/now/`. Similarly, if we had left off the initial caret character (`'now/$'`), Django would match *any* URL that ends with `now/` — e.g., `/foo/bar/now/`. Thus, we use both the caret and dollar sign to ensure that only the URL `/now/` matches. Nothing more, nothing less.

To test our changes to the URLconf, start the Django development server, as you did in Chapter 1, by running the command `python manage.py runserver`. (If you left it running, that's fine, too. The development server automatically detects changes to your Python code and reloads as necessary, so you don't have to restart the server between changes.) The server is running at the address `http://127.0.0.1:8000/`, so open up a Web browser and go to `http://127.0.0.1:8000/now/` — and you should see the output of your Django view.

Hooray! You've made your first Django-powered Web page.

How Django processes a request

We should point out several things about what just happened. Here's the nitty-gritty of what goes on when you run the Django development server and make requests to Web pages:

- The command `python manage.py runserver` looks for a file called `settings.py`. This file contains all sorts of optional configuration for this particular Django instance, but one of the most important settings is one called `ROOT_URLCONF`. The `ROOT_URLCONF` setting tells Django which Python module should be used as the URLconf for this Web site.

  Remember when `django-admin.py startproject` created the files `settings.py` and `urls.py`? Well, the auto-generated `settings.py` has a `ROOT_URLCONF` that points to the auto-generated `urls.py`. Convenient.
- When a request comes in — say, a request to the URL `/now/` — Django loads the URLconf pointed-to by the `ROOT_URLCONF` setting. Then it checks each of the URLpatterns in that URLconf in order, comparing the requested URL with the patterns one at a time, until it finds one that matches. When it finds one that matches, it calls the view function associated with that pattern, passing a `HttpRequest` object as the first parameter to the function. (More on `HttpRequest` later.)
- The view function is responsible for returning an `HttpResponse` object.

With this knowledge, you know the basics of how to make Django-powered pages. It's quite simple, really — just write view functions and map them to URLs via URLconfs.

URLconfs and loose coupling

Now's a good time to point out a key philosophy behind URLconfs, and behind Django in general: the principle of **loose coupling**. Simply put, loose coupling is a software-development approach that values the importance of making pieces interchangeable. If two pieces of code are "loosely coupled," then making changes to one of the pieces will have little-to-no effect on the other.

Django's URLconfs are a good example of this principle in practice. In a Django Web application, the URL definitions and the view functions they call are loosely coupled; that is, the decision of what the URL should be for a given function, and the implementation of the function itself, reside in two separate places. This lets a developer switch out one piece without affecting the other.

In contrast, other Web development platforms couple the URL to the program. In basic PHP (http://www.php.net/), for example, the URL of your application is designated by where you place the code on your filesystem. In the CherryPy Python Web framework (http://www.cherrypy.org/), the URL of your application corresponds to the name of the method in which your code lives. This may seem like a convenient shortcut in the short term, but it can get unmanageable in the long run.

For example, consider the view function we wrote above, which displays the current date and time. If we wanted to change the URL for the application — say, move it from `/now/` to `/currenttime/` — we could make a quick change to the URLconf, without having to worry about the underlying implementation of the function. Similarly, if we wanted to change the view function — altering its logic somehow — we could do that without affecting the URL to which the function is bound. Furthermore, if we wanted to expose the current-date functionality at *several* URLs, we could easily take care of that by editing the URLconf, without having to touch the view code.

That's loose coupling in action. And we'll continue to point out examples of this important philosophy throughout this book.

404 errors

In our URLconf thus far, we've only defined a single URLpattern — the one that handles requests to the URL `/now/`. What happens when a different URL is requested?

To find out, try running the Django development server and hitting a page such as `http://127.0.0.1:8000/hello/`

or `http://127.0.0.1:8000/does-not-exist/`, or even `http://127.0.0.1:8000/` (the site “root”).

You should see a “Page not found” message. (Pretty, isn’t it? We Django people sure do like our pastel colors.) Django displays this message because you requested a URL that’s not defined in your URLconf.

The utility of this page goes beyond the basic 404 error message: It also tells you precisely which URLconf Django used and every pattern in that URLconf. From that information, you should be able to tell why the requested URL threw a 404.

Naturally, this is sensitive information intended only for you, the Web developer. If this were a production site deployed live on the Internet, we wouldn’t want to expose that information to the public. For that reason, this “Page not found” page is only displayed if your Django project is in **debug mode**. We’ll explain how to deactivate debug mode later. For now, just know that every Django project is in debug mode automatically when you start it.

Your second view: Dynamic URLs

In our first view example, the contents of the page — the current date/time — were dynamic, but the URL (“/now/”) was static. In most dynamic Web applications, though, a URL contains parameters that influence the output of the page.

As another (slightly contrived) example, let’s create a second view, which displays the current date and time offset by a certain number of hours. The goal is to craft a site in such a way that the page `/now/plus1hour/` displays the date/time one hour into the future, the page `/now/plus2hours/` displays the date/time two hours into the future, the page `/now/plus3hours/` displays the date/time three hours into the future, and so on.

A novice might think to code a separate view function for each hour offset, which might result in a URLconf that looked like this:

```
urlpatterns = patterns('',
    (r'^now/$', current_datetime),
    (r'^now/plus1hour/$', one_hour_ahead),
    (r'^now/plus2hours/$', two_hours_ahead),
    (r'^now/plus3hours/$', three_hours_ahead),
    (r'^now/plus4hours/$', four_hours_ahead),
)
```

Clearly, this line of thought is flawed. Not only would this result in redundant view functions, but the application is fundamentally limited to supporting only the predefined hour ranges — one, two, three or four hours. If, all of a sudden, we wanted to create a page that displayed the time *five* hours into the future, we’d have to create a separate view and URLconf line for that, furthering the duplication and insanity. We need to do some abstraction here.

A word about pretty URLs

If you’re experienced in another Web development platform, such as PHP or Java, you may be thinking: “Hey, let’s use a query-string parameter!” That’d be something like `/now/plus?hours=3`, in which the hours would be designated by the `hours` parameter in the URL’s query string (the part after the `?`).

You *can* do that with Django — and we’ll tell you how later, if you really must know — but one of Django’s core philosophies is that URLs should be beautiful. The URL `/now/plus3hours/` is far cleaner, simpler, more readable, easier to recite to somebody aloud and …just plain prettier than its query-string counterpart. Pretty URLs are a sign of a quality Web application.

Django’s URLconf system encourages pretty URLs by making it easier to use pretty URLs than *not* to.

Wildcard URLpatterns

Continuing with our `hours_ahead` example, let’s put a wildcard in the URLpattern. As we mentioned above, a URLpattern is a regular expression, and, hence, we can use the regular expression pattern `\d+` to match one or more digits:

```
from django.conf.urls.defaults import *
from mysite.views import current_datetime, hours_ahead

urlpatterns = patterns('',
    (r'^now/$', current_datetime),
    (r'^now/plus\d+hours/$', hours_ahead),
)
```

This URLpattern will match any URL such as `/now/plus2hours/`, `/now/plus25hours/` or even `/now/plus100000000000hours/`. Come to think of it, let’s limit it so that the maximum allowed offset is 99 hours. That means we want to allow either one- or two-digit numbers; in regular expression syntax, that translates into `\d{1,2}`:

```
(r'^now/plus\d{1,2}hours/$', hours_ahead),
```

(When building Web applications, it’s always important to consider the most outlandish data input possible, and decide whether the application should support that input or not. We’ve curtailed the outlandishness here by limiting the offset to 99 hours. And, by the way, The Outlandishness Curtailers would be a fantastic, if verbose, band name.)

Regular expressions

Regular expressions (or "regexes") are a compact way of specifying patterns in text. While Django URLconfs allow arbitrary regexes for powerful URL-matching capability, you'll probably only use a few regex patterns in practice. Here's a small selection of common patterns:

Symbol	Matches
`.` (dot)	Any character
`\d`	Any digit
`[A-Z]`	Any character from A-Z (uppercase)
`[a-z]`	Any character from a-z (lowercase)
`[A-Za-z]`	Any character from a-z (case-insensitive)
`[^/]+`	All characters until a forward slash (excluding the slash itself)
`+`	One or more of the previous character (e.g., `\d+` matches one or more digit)
`?`	Zero or more of the previous character (e.g., `\d*` matches zero or more digits)
`{1,3}`	Between one and three (inclusive) of the previous character

For more on regular expressions, see Appendix XXX, Regular Expressions.

Now that we've designated a wildcard for the URL, we need a way of passing that data to the view function, so that we can use a single view function for any arbitrary hour offset. We do this by placing parentheses around the data in the URLpattern that we want to save. In the case of our example, we want to save whatever number was entered in the URL — so let's put parentheses around the `\d{1,2}`:

```
(r'^now/plus(\d{1,2})hours/$', hours_ahead),
```

If you're familiar with regular expressions, you'll be right at home here; we're using parentheses to *capture* data from the matched text.

The final URLconf, including our previous `current_datetime` view, looks like this:

```
from django.conf.urls.defaults import *
from mysite.views import current_datetime, hours_ahead

urlpatterns = patterns('',
    (r'^now/$', current_datetime),
    (r'^now/plus(\d{1,2})hours/$', hours_ahead),
)
```

With that taken care of, let's write the `hours_ahead` view.

..admonition:: Coding order

> In this case, we wrote the URLpattern first and the view second, but in the previous example, we wrote the view first, then the URLpattern. Which technique is better?
>
> Well, every developer is different.
>
> If you're a big-picture type of person, it may make most sense to you to write all of the URLpatterns for your application at the same time, at the start of your project, then coding up the views. This has the advantage of giving you a clear to-do list, and it essentially defines the parameter requirements for the view functions you'll need to write.
>
> If you're more of a bottom-up developer, you might prefer to write the views first, then anchor them to URLs afterward. That's OK, too.
>
> In the end, it comes down to what fits your brain the best. Either approach is valid.

`hours_ahead` is very similar to the `current_datetime` view we wrote earlier, with a key difference: it takes an extra argument, the number of hours of offset. Here it is:

```
from django.http import HttpResponse
import datetime

def hours_ahead(request, offset):
    offset = int(offset)
    dt = datetime.datetime.now() + datetime.timedelta(hours=offset)
    html = "<html><body>In %s hour(s), it will be %s.</body></html>" % (offset, dt)
    return HttpResponse(html)
```

Let's step through this code one line at a time:

- Just as we did for our `current_datetime` view, we import the class `django.http.HttpResponse` and the `datetime` module.
- The view function, `hours_ahead`, takes *two* parameters: `request` and `offset`.
  - `request` is an `HttpRequest` object, just as in `current_datetime`. We'll say it again: Each view *always* takes an `HttpRequest` object as its first parameter.
  - `offset` is the string captured by the parentheses in the URLpattern. For example, if the requested URL were `/now/plus3hours/`, then `offset` would be the string `'3'`. If the requested URL were `/now/plus21hours/`, then `offset` would be the string `'21'`. Note that captured strings will always be *strings*, not integers, even if the string is composed of only digits, such as `'21'`.

    We decided to call the variable `offset`, but you can call it whatever you'd like, as long as it's a valid Python identifier. The variable name doesn't matter; all that matters is that it's the second argument to the function (after `request`).
- The first thing we do within the function is call `int()` on `offset`. This converts the string value to an integer.

  Note that Python will raise a `ValueError` exception if you call `int()` on a value that cannot be converted to an integer, such as the string `'foo'`. However, we don't have to worry about catching that exception, because we can be certain `offset` will be a string containing only digits. We know that because the regular-expression pattern in our URLconf — `\d{1,2}` — captures only digits. This illustrates another nicety of URLconfs: They provide a fair level of input validation.
- The next line of the function shows why we called `int()` on `offset`. On this line, we calculate the current time plus a time offset of `offset` hours, storing the result in `dt`. The `datetime.timedelta` function requires the `hours` parameter to be an integer.
- Next, we construct the HTML output of this view function, just as we did in `current_datetime`. A small difference in this line from the previous line is that it uses Python's format-string capability with *two* values, not just one. Hence, there are two `%s` symbols in the string and a tuple of values to insert — `(offset, dt)`.
- Finally, we return an `HttpResponse` of the HTML — again, just as we did in `current_datetime`.

With that view function and URLconf written, start the Django development server (if it's not already running), and visit `http://127.0.0.1:8000/now/plus3hours/` to verify it works. Then try `http://127.0.0.1:8000/now/plus5hours/`. Then `http://127.0.0.1:8000/now/plus24hours/`. Finally, visit `http://127.0.0.1:8000/now/plus100hours/` to verify that the pattern in your URLconf only accepts one- or two-digit numbers; Django should display a "Page not found" error in this case, just as we saw in the "404 errors" section above. The URL `http://127.0.0.1:8000/now/plushours/` (with *no* hour designation) should also throw a 404.

If you're following along while coding at the same time, you'll notice that the `views.py` file now contains two views. (We omitted the `current_datetime` view from the last set of examples for clarity.) Put together, `views.py` should look like this:

```
from django.http import HttpResponse
import datetime

def current_datetime(request):
    now = datetime.datetime.now()
    html = "<html><body>It is now %s.</body></html>" % now
    return HttpResponse(html)

def hours_ahead(request, offset):
    offset = int(offset)
    dt = datetime.datetime.now() + datetime.timedelta(hours=offset)
    html = "<html><body>In %s hour(s), it will be %s.</body></html>" % (offset, dt)
    return HttpResponse(html)
```

Django's pretty error pages

Take a moment to admire the fine Web application we've made so far…and break it!

Let's deliberately introduce a Python error into our `views.py` file, by commenting-out the `offset = int(offset)` line in the `hours_ahead` view:

```
def hours_ahead(request, offset):
    # offset = int(offset)
    dt = datetime.datetime.now() + datetime.timedelta(hours=offset)
    html = "<html><body>In %s hour(s), it will be %s.</body></html>" % (offset, dt)
    return HttpResponse(html)
```

Now load up the development server and navigate to `/now/plus3hours/`. You'll see an error page with a significant amount of information, including a `TypeError` message displayed at the very top: "unsupported type for timedelta hours component: str".

What happened?

Well, the `datetime.timedelta` function expects the `hours` parameter to be an integer, and we commented-out the bit of code that converted `offset` to an integer. That caused `datetime.timedelta` to raise the `TypeError`. It's the typical kind of small bug that every programmer runs into at some point.

The point of this example was to demonstrate Django's error pages. Take some time to explore the error page and get to know the various bits of information it gives you.

Some highlights:

- At the top of the page, you get the key information about the exception: the type of exception, any parameters to the exception (e.g., the `"unsupported type"` message in this case), the file in which the exception was raised and the offending line number.
- Under that, the page displays the full Python traceback for this exception. This is similar to the standard traceback you get in Python's command-line interpreter, except it's more interactive. For each frame in the stack, Django displays the name of the file, the function/method name, the line number and the source code of that line.

  Click the line of source code (in dark gray), and you'll see several lines from before and after the erroneous line, to give you context.

  Click "Local vars" under any frame in the stack to view a table of all local variables, and their values, in that frame, at the exact point in the code at which the exception was raised. This debugging information is invaluable.
- Note the "Switch to copy-and-paste view" text just under the "Traceback" header. Click those words, and the traceback will switch to a alternate version that can be easily copied and pasted. Use this when you want to share your exception traceback with others to get technical support — such as the kind folks in the Django IRC chat room or on the Django users mailing list.
- Next, the "Request information" section includes a wealth of information about the incoming Web request that spawned the error: GET and POST information, cookie values and meta information, such as CGI headers. If this information seems like gibberish to you at the moment, don't fret — we'll explain it later in this book.

  Below, the "Settings" section lists all of the settings for this particular Django installation. Again, we'll explain settings later in this book. For now, take a look at the settings to get an idea of the information available.

The Django error page is capable of displaying more information in certain special cases, such as the case of template syntax errors. We'll get to those later, when we discuss the Django template system. For now, uncomment the `offset = int(offset)` line to get the view function working properly again.

Are you the type of programmer who likes to debug with the help of carefully placed `print` statements? You can use the Django error page to do just that — just without the `print` statements. At any point in your view, temporarily insert an `assert False` to trigger the error page. Then, you can view the local variables and state of the program. (There's a more advanced way to debug Django views, which we'll explain later, but this is the quickest and easiest.)

Finally, it's obvious that much of this information is sensitive — it exposes the innards of your Python code and Django configuration — and it would be foolish to show this information on the public Internet. A malicious person could use it to attempt to reverse-engineer your Web application and do nasty things. For that reason, the Django error page is only displayed when your Django project is in debug mode. We'll explain how to deactivate debug mode later. For now, just know that every Django project is in debug mode automatically when you start it. (Sound familiar? The "Page not found" errors, described in the "404 errors" section above, work the same way.)

Exercises

Here are a few exercises that will solidify some of the things you learned in this chapter. (Hint: Even if you think you understood everything, at least give these exercises, and their respective answers, a read. We introduce a couple of new tricks here.)

1. Create another view, `hours_behind`, that works like `hours_ahead` but instead displays the date/time with an offset into the *past*, not the future. This view should bind to URLs in the style `/now/minusXhours/`, where `X` is the offset, in hours.
2. Once you've done that, be a good programmer and notice how similar the `hours_ahead` and `hours_behind` views are. How redundant! Eliminate the redundancy and combine them into a single view, `hour_offset`. The URLs should stay the same as before: e.g., `/now/minusXhours/` and `/now/plusXhours/`. Don't forget to change the HTML to say either "In X hour(s)" or "X hour(s) ago", depending on whether the offset is positive or negative.
3. We were lazy and hard-coded the plural form of "hour" in the URL, resulting in the grammatic atrocity `/now/plus1hours/`. Do your part to uphold proper English grammar, and improve the application so that it accepts the URL `/now/plus1hour/`.

For bonus points, be a perfectionist: allow `/now/plus1hour/` and `/now/plus2hours/` but disallow `/now/plus1hours/` and `/now/plus2hour/`.

4. Similarly, we were lazy in the HTML display, saying `"In %s hour(s), it will be %s."` Fix this to remove the `hour(s)`. The `(s)` was such a cop-out! If the offset is singular, use `'hour'`; otherwise, use `'hours'`.

Answers to exercises

1. Here's one implementation of the `hours_behind` view:

```
def hours_behind(request, offset):
    offset = int(offset)
    dt = datetime.datetime.now() - datetime.timedelta(hours=offset)
    html = "<html><body>%s hour(s) ago, it was %s.</body></html>" % (offset, dt)
    return HttpResponse(html)
```

Not much is different between this view and `hours_ahead` — only the calculation of `dt` and the text within the HTML.

The URLpattern would look like this:

```
(r'^now/minus(\d{1,2})hours/$', hours_behind),
```

2. Here's one implementation of the `hour_offset` view:

```
def hour_offset(request, plus_or_minus, offset):
    offset = int(offset)
    if plus_or_minus == 'plus':
        dt = datetime.datetime.now() + datetime.timedelta(hours=offset)
        html = 'In %s hour(s), it will be %s.' % (offset, dt)
    else:
        dt = datetime.datetime.now() - datetime.timedelta(hours=offset)
        html = '%s hour(s) ago, it was %s.' % (offset, dt)
    html = '<html><body>%s</body></html>' % html
    return HttpResponse(html)
```

The URLpattern would look like this:

```
(r'^now/(plus|minus)(\d{1,2})hours/$', hour_offset),
```

In this implementation, we capture *two* values from the URL — the offset, as we did before, but also the string that designates whether the offset should be positive or negative. They're passed to the view function in the order in which they're captured.

Inside the view code, the variable `plus_or_minus` will be either the string `'plus'` or the string `'minus'`. We test that to determine how to calculate the offset — either by adding or subtracting a `datetime.timedelta`.

If you're particularly anal, you may find it inelegant that the view code is "aware" of the URL, having to test for the string `'plus'` or `'minus'` rather than some other variable that has been abstracted from the URL. There's no way around that; Django does not include any sort of "middleman" layer that converts captured URL parameters to abstracted data structures, for simplicity's sake.

3. To accomplish this, we wouldn't have to change the `hour_offset` view at all. We'd just need to edit the URLconf slightly. Here's one way to do it, by using *two* URLpatterns:

```
(r'^now/(plus|minus)(1)hour/$', hour_offset),
(r'^now/(plus|minus)([2-9]|\d\d)hours/$', hour_offset),
```

More than one URLpattern can point to the same view; Django processes the patterns in order and doesn't care how many times a certain view is referenced. In this case, the first pattern matches the URLs `/now/plus1hour/` and `/now/minus1hour/`. The `(1)` is a neat little trick — it passes the value `'1'` as the captured value, without allowing any sort of wildcard.

The second pattern is more complex, as it uses a slightly tricky regular expression. The key part is `([2-9]|\d\d)`. The pipe character (`'|'`) means "or," so the pattern in full means "match either the pattern `[2-9]` or `\d\d`." In other words, that matches any one-digit number from 2 through 9, *or* any two-digit number.

4. Here's a basic way of accomplishing this. Alter the `hour_offset` function like so:

```
def hour_offset(request, plus_or_minus, offset):
    offset = int(offset)
    if offset == 1:
        hours = 'hour'
    else:
        hours = 'hours'
    if plus_or_minus == 'plus':
        dt = datetime.datetime.now() + datetime.timedelta(hours=offset)
        output = 'In %s %s, it will be %s.' % (offset, hours, dt)
    else:
        dt = datetime.datetime.now() - datetime.timedelta(hours=offset)
        output = '%s %s ago, it was %s.' % (offset, hours, dt)
    output = '<html><body>%s</body></html>' % output
    return HttpResponse(output)
```

Ideally, though, we wouldn’t have to edit Python code to make small presentation-related changes like this. Wouldn’t it be nice if we could separate presentation from Python logic? Ah, foreshadowing...

« previous ◊ table of contents ◊ next »

Chapter 4: The Django template system

In the previous chapter, you may have noticed something peculiar in how we returned the HTML in our example views. Namely, the HTML was hard-coded directly in our Python code!

This arrangement leads to several problems:

- Obviously, any change to the design of the page would require a change to the Python code. The design of a site tends to change far more frequently than the underlying Python code, so it would be convenient if the frequency of HTML changes were separated from changes to Python code.
- Second, writing backend Python code and designing/coding HTML are two different disciplines, and most professional Web development environments split these responsibilities across separate people (or even separate departments). Designers and HTML/CSS coders shouldn't have to edit Python code to get their job done; they should deal with HTML.
- Similarly, it's most efficient if programmers can work on Python code and designers can work on templates at the same time, rather than one person waiting for the other to finish editing a single file that contains both Python and HTML.

For these reasons, it's much cleaner and more maintainable to separate the design of the page from the Python code itself. We can do this with Django's **template system**.

Template system basics

A Django template is a string of text that is intended to separate the presentation of a document from its data. A template defines placeholders and various bits of basic logic — **tags** — that regulate how the document should be displayed. Usually, templates are used for outputting HTML, but Django templates are equally capable of generating any text-based format.

Let's dive in with a simple example template. This template describes an HTML page that thanks a person for making an order from a company. Think of it as a form letter:

```
<html>
<head><title>Ordering notice</title></head>

<body>

<p>Dear {{ person_name }},</p>

<p>Thanks for placing an order from {{ company }}. It's scheduled to
ship on {{ ship_date|date:"F j, Y" }}.</p>

<p>Here are the items you've ordered:</p>
```

```
<ul>
{% for item in item_list %}
<li>{{ item }}</li>
{% endfor %}
</ul>

{% if ordered_warranty %}
<p>Your warranty information will be included in the packaging.</p>
{% endif %}

<p>Sincerely,<br />{{ company }}</p>

</body>
</html>
```

This template is basic HTML with some **variables** and **template tags** thrown in. Let's step through it:

- Any text surrounded by a pair of braces — e.g., `{{ person_name }}` — is a **variable**. This means "insert the value of the variable with the given name." (How do we specify the values of the variables? We'll get to that in a moment.)

- Any text that's surrounded by curly braces and percent signs — e.g., `{% if ordered_warranty %}` — is a **block tag**. The definition of a block tag is quite broad: A block tag just tells the template system to *do something*.

  This example template contains two block tags — the `{% for item in item_list %}` tag (a "for" tag) and the `{% if ordered_warranty %}` tag (an "if" tag). A "for" tag acts as a simple loop construct, letting you loop over each item in a sequence. An "if" tag, as you may expect, acts as a logical "if" statement. In this particular case, the tag checks whether the value of the `ordered_warranty` variable evaluates to `True`. If it does, the template system will display everything between the `{% if ordered_warranty %}` and `{% endif %}`. If not, the template system won't display it. The template system also supports `{% else %}` and other various logic statements.

  Each Django template has access to several built-in block tags. In addition, you can write your own tags.

- Finally, the second paragraph of this template has an example of a **filter**. Filters are a way to alter the display of a variable. In this example — `{{ ship_date|date:"F j, Y" }}` — we're passing the `ship_date` variable to the `date` filter, giving the `date` filter an argument `"F j, Y"`. The `date` filter formats dates in a given format, as specified by that argument. Filters are attached using a pipe character (`|`), as a reference to Unix pipes.

  Each Django template has access to several built-in filters. In addition, you can write your own filters.

Using the template system

To use the template system in Python code, just follow these two steps:

- First, create a `Template` object by providing the raw template code as a string. Django also offers a way to create `Template` objects by designating the path to a template file on the filesystem; we'll see that in a bit.
- Then, call the `render()` method of the `Template` object with a given set of variables — the context. This returns a fully rendered template, as a string, with all of the variables and block tags evaluated according to the context.

Creating template objects

The easiest way to create a `Template` object is to instantiate it directly. The `Template` class lives in the `django.template` module, and the constructor takes one argument, the raw template code. Let's dip into the Python interactive interpreter to see how this works in code. (Type `python` at the command line to start the interactive interpreter.) Here's a basic walkthrough:

```
>>> from django.template import Template
>>> t = Template("My name is {{ my_name }}.")
>>> print t
```

If you're following along interactively, you'll see something like this after typing `print t`:

```
<django.template.Template object at 0xb7d5f24c>
```

That `0xb7d5f24c` will be different every time, and it doesn't really matter; it's simply the Python "identity" of the `Template` object.

Interactive interpreter examples

Throughout this book, we'll feature example Python interactive interpreter sessions. You can recognize these examples by spotting the triple greater-than signs (`>>>`), which designate the interpreter's prompt. If you're copying examples from this book, don't copy those greater-than signs.

Multiline statements in the interactive interpreter are padded with three dots (`...`). For example:

```
>>> print """This is a
... string that spans
... three lines."""
This is a
string that spans
three lines.
>>> def my_function(value):
...     print value
>>> my_function('hello')
hello
```

Those three dots at the start of the additional lines are inserted by the Python shell — they're not part of our input. We include them here to be faithful to the actual output of the interpreter. If you copy our examples to follow along, don't copy those dots.

When you create a `Template` object, the template system compiles the raw template code into an internal, optimized form, ready for rendering. But if your template code includes any syntax errors, the call to `Template()` will cause a `TemplateSyntaxError` exception:

```
>>> from django.template import Template
>>> t = Template('{% notatag %} ')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  ...
  django.template.TemplateSyntaxError: Invalid block tag: 'notatag'
```

The system raises a `TemplateSyntaxError` exception for any of the following cases:

- Invalid block tags
- Invalid arguments to valid block tags
- Invalid filters
- Invalid arguments to valid filters
- Invalid template syntax
- Unclosed block tags (for block tags that require closing tags)

Rendering a template

Once you have a `Template` object, you can pass it data by giving it a **context**. A context is simply a set of variables and their associated values. A template uses this to populate its variable tags and evaluate its block tags.

A context is represented in Python by the `Context` class, which lives in the `django.template` module. Its constructor takes one optional argument: a dictionary mapping variable names to variable values. Call the `Template` object's `render()` method with the context to "fill" the template. For example:

```
>>> from django.template import Context, Template
>>> t = Template("My name is {{ name }}.")
>>> c = Context({"name": "Stephane"})
>>> t.render(c)
'My name is Stephane.'
```

Variable names must begin with a letter (A-Z or a-z) and may contain digits, underscores and dots. (Dots are a special case we'll get to in a moment.) Variable names are case sensitive.

Here's an example of template compilation and rendering, using the sample template from the beginning of this chapter:

```
>>> from django.template import Template, Context
>>> raw_template = """<p>Dear {{ person_name }},</p>
...
... <p>Thanks for ordering {{ product }} from {{ company }}. It's scheduled to
... ship on {{ ship_date|date:"F j, Y" }}.</p>
...
... {% if ordered_warranty %}
... <p>Your warranty information will be included in the packaging.</p>
... {% endif %}
...
... <p>Sincerely,<br />{{ company }}</p>"""
>>> t = Template(raw_template)
>>> import datetime
>>> c = Context({'person_name': 'John Smith',
...     'product': 'Super Lawn Mower',
...     'company': 'Outdoor Equipment',
...     'ship_date': datetime.date(2009, 4, 2),
...     'ordered_warranty': True})
>>> t.render(c)
"<p>Dear John Smith,</p>\n\n<p>Thanks for ordering Super Lawn Mower from Outdoor Equipment.
It's scheduled to ship on April 2, 2009.</p>\n\n<p>Your warranty information will be included
in the packaging.</p>\n\n\n<p>Sincerely,<br />Outdoor Equipment</p>"
```

Let's step through this one statement at a time:

- First, we import the classes `Template` and `Context`, which both live in the module `django.template`.
- Next, we save the raw text of our template into the variable `raw_template`. Note that we use a triple quote marks to designate the string, because it wraps over multiple lines; strings designated with single quote marks cannot be wrapped over multiple lines.
- Next, we create a template object `t` by passing `raw_template` to the `Template` class constructor.
- Then we import the `datetime` module from Python's standard library, because we'll need it in the following statement.
- Next, we create a context object `c`. The `Context` constructor takes a Python dictionary mapping variable names to values. Here, for example, we specify that the `person_name` is `'John Smith'`, `product` is `'Super Lawn Mower'`, etc.
- Finally, we call the `render()` method on our template object, passing it the context. This returns the rendered template — that is, it replaces template variables with the actual values of the variables, and it executes any block tags.

  Note that the warranty paragraph was displayed because the `ordered_warranty` variable evaluated to `True`. Also note the date, `April 2, 2009`, which is displayed according to the format string `'F j, Y'`. (We'll explain format strings for the `date` filter shortly.)

  If you're new to Python, you may wonder why this output includes newline characters (`'\n'`) rather than displaying the line breaks. That's happening because of a subtlety in the Python interactive interpreter: The call to `t.render(c)` returns a string, and by default the interactive interpreter displays the *representation* of the string, rather than the printed value of the string. If you want to see the string with line breaks displayed as true line breaks rather than `'\n'` characters, use the `print` statement: `print t.render(c)`.

Those are the fundamentals of using the Django template system — just write a template, create a template object, create a context and call the `render()` method.

Multiple contexts, same template

Once you have a template object, you can render multiple contexts through it. For example:

```
>>> from django.template import Template, Context
>>> t = Template('Hello, {{ name }}')
>>> print t.render(Context({'name': 'John'}))
Hello, John
>>> print t.render(Context({'name': 'Julie'}))
Hello, Julie
>>> print t.render(Context({'name': 'Pat'}))
Hello, Pat
```

Whenever you're using the same template to render multiple contexts like this, it's most efficient to create the `Template` object *once*, then call `render()` on it multiple times. For example:

```
# Bad
for name in ('John', 'Julie', 'Pat'):
    t = Template('Hello, {{ name }}')
    print t.render(Context({'name': name}))

# Good
t = Template('Hello, {{ name }}')
for name in ('John', 'Julie', 'Pat'):
    print t.render(Context({'name': name}))
```

Django's template parsing is quite fast. Behind the scenes, most of the parsing happens via a single call to a short regular expression. This is a stark contrast to XML-based templating engines, which incur the overhead of an XML parser and tend to be orders of magnitude slower than Django's template rendering engine.

Context variable lookup

In the examples so far, we've passed simple values in the template contexts — mostly strings, plus a `datetime.date` example. However, the template system elegantly handles more complex data structures, such as lists, dictionaries and custom objects.

The key to traversing complex data structures in Django templates is the dot (`.`) character. Use a dot to access dictionary keys, attributes, indices or methods of an object.

This is best illustrated with a few examples. First, say you're passing a Python dictionary to a template. To access the values of that dictionary by dictionary key, use a dot:

```
>>> from django.template import Template, Context
>>> person = {'name': 'Sally', 'age': '43'}
>>> t = Template('{{ person.name }} is {{ person.age }} years old.')
>>> c = Context({'person': person})
>>> t.render(c)
'Sally is 43 years old.'
```

Similarly, dots also allow access of object attributes. For example, a Python `datetime.date` object has `year`, `month` and `day` attributes, and you can use a dot to access those attributes in a Django template:

```
>>> from django.template import Template, Context
>>> import datetime
>>> d = datetime.date(1993, 5, 2)
>>> d.year
1993
>>> d.month
5
>>> d.day
2
>>> t = Template('The month is {{ date.month }} and the year is {{ date.year }}.')
>>> c = Context({'date': d})
>>> t.render(c)
'The month is 5 and the year is 1993.'
```

This example uses a custom class:

```
>>> from django.template import Template, Context
>>> class Person(object):
...     def __init__(self, first_name, last_name):
...         self.first_name, self.last_name = first_name, last_name
>>> t = Template('Hello, {{ person.first_name }} {{ person.last_name }}.')
>>> c = Context({'person': Person('John', 'Smith')})
>>> t.render(c)
'Hello, John Smith.'
```

Dots are also used to access list indices. For example:

```
>>> from django.template import Template, Context
>>> t = Template('Item 2 is {{ items.2 }}.')
>>> c = Context({'items': ['apples', 'bananas', 'carrots']})
>>> t.render(c)
'Item 2 is carrots.'
```

Negative list indices are not allowed. For example, the template variable `{{ items.-1 }}` would cause a `TemplateSyntaxError`.

Finally, dots are also used to call methods on objects. For example, each Python string has the methods `upper()` and `isdigit()`, and you can call those in Django templates using the same dot syntax:

```
>>> from django.template import Template, Context
>>> t = Template('{{ var }} -- {{ var.upper }} -- {{ var.isdigit }}')
>>> t.render(Context({'var': 'hello'}))
'hello -- HELLO -- False'
>>> t.render(Context({'var': '123'}))
'123 -- 123 -- True'
```

Note that, in the method calls, you don't include parentheses. Also, it's not possible to pass arguments to the methods; you can only call methods that have no required arguments. (We'll explain this philosophy later in this chapter.)

The dot lookups can be summarized like this: When the template system encounters a dot in a variable name, it tries the following lookups, in this order:

- Dictionary lookup. Example: `foo["bar"]`
- Attribute lookup. Example: `foo.bar`
- Method call. Example: `foo.bar()`
- List-index lookup. Example: `foo[bar]`

The system uses the first lookup type that works. It's short-circuit logic.

Dot lookups can be nested multiple levels deep. For instance, the following example uses `{{ person.name.upper }}`, which translates into a dictionary lookup (`person['name']`), then a method call (`upper()`):

```
>>> from django.template import Template, Context
>>> person = {'name': 'Sally', 'age': '43'}
>>> t = Template('{{ person.name.upper }} is {{ person.age }} years old.')
>>> c = Context({'person': person})
>>> t.render(c)
'SALLY is 43 years old.'
```

A word about method calls

Method calls are slightly more complex than the other lookup types. Here are some things to keep in mind:

- If, during the method lookup, a method raises an exception, the exception will be propagated, unless the exception has an attribute `silent_variable_failure` whose value is `True`. If the exception *does* have a `silent_variable_failure` attribute, the variable will render as an empty string. For example:

```
>>> t = Template("My name is {{ person.first_name }}.")
>>> class PersonClass3:
...     def first_name(self):
...         raise AssertionError, "foo"
>>> p = PersonClass3()
>>> t.render(Context({"person": p}))
Traceback (most recent call last):
...
AssertionError: foo
```

```
>>> class SilentAssertionError(AssertionError):
...     silent_variable_failure = True
>>> class PersonClass4:
...     def first_name(self):
...         raise SilentAssertionError
>>> p = PersonClass4()
>>> t.render(Context({"person": p}))
"My name is ."
```

- A method call will only work if the method has no required arguments. Otherwise, the system will move to the next lookup type (list-index lookup).
- Obviously, some methods have side effects, and it'd be foolish at best, and possibly even a security hole, to allow the template system to access them.

  Say, for instance, you have a `BankAccount` object that has a `delete()` method. The template system shouldn't be allowed to do something like this:

  ```
  I will now delete this valuable data. {{ account.delete }}
  ```

  To prevent this, set a function attribute `alters_data` on the method. The template system won't execute a method if the method has `alters_data=True` set. For example:

  ```
  def delete(self):
      # Delete the account
  delete.alters_data = True
  ```

How invalid variables are handled

By default, if a variable doesn't exist, the template system renders it as an empty string, failing silently. For example:

```
>>> from django.template import Template, Context
>>> t = Template('Your name is {{ name }}.')
>>> t.render(Context())
'Your name is .'
>>> t.render(Context({'var': 'hello'}))
'Your name is .'
>>> t.render(Context({'NAME': 'hello'}))
'Your name is .'
>>> t.render(Context({'Name': 'hello'}))
'Your name is .'
```

The system fails silently rather than raising an exception because it's intended to be resilient to human error. In the real world, it's unacceptable for a Web site to become inaccessible due to a small template syntax error.

Note that it’s possible to change Django’s default behavior in this regard, by tweaking a setting in your Django configuration. We’ll discuss this in Chapter 10, “Extending the template engine.”

Playing with Context objects

Most of the time, you’ll instantiate `Context` objects by passing in a fully-populated dictionary to `Context()`. But you can add and delete items from a `Context` object once it’s been instantiated, too, using standard Python dictionary syntax:

```
>>> from django.template import Context
>>> c = Context({"foo": "bar"})
>>> c['foo']
'bar'
>>> del c['foo']
>>> c['foo']
''
>>> c['newvariable'] = 'hello'
>>> c['newvariable']
'hello'
```

A `Context` object is a stack. That is, you can `push()` and `pop()` it. If you `pop()` too much, it’ll raise `django.template.ContextPopException`:

```
>>> c = Context()
>>> c['foo'] = 'first level'
>>> c.push()
>>> c['foo'] = 'second level'
>>> c['foo']
'second level'
>>> c.pop()
>>> c['foo']
'first level'
>>> c['foo'] = 'overwritten'
>>> c['foo']
'overwritten'
>>> c.pop()
Traceback (most recent call last):
...
django.template.ContextPopException
```

Using a `Context` as a stack comes in handy in some custom template tags, as you’ll see in Chapter 10.

Basic template tags and filters

As we’ve mentioned already, the template system ships with built-in tags and filters. Here’s a rundown of the most common ones.

Appendix 6 includes a full list of all built-in tags and filters, and it's a good idea to familiarize yourself with that list to have an idea of what's possible.

if/else

The `{% if %}` tag evaluates a variable, and if that variable is "true" (i.e., it exists, is not empty, and is not a false boolean value), the system will display everything between `{% if %}` and `{% endif %}`. For example:

```
{% if today_is_weekend %}
    <p>Welcome to the weekend!</p>
{% endif %}
```

An `{% else %}` tag is optional:

```
{% if today_is_weekend %}
    <p>Welcome to the weekend!</p>
{% else %}
    <p>Get back to work.</p>
{% endif %}
```

The `{% if %}` tag accepts `and`, `or` or `not` for testing multiple variables, or to negate a given variable. For example:

```
{% if athlete_list and coach_list %}
    Both athletes and coaches are available.
{% endif %}

{% if not athlete_list %}
    There are no athletes.
{% endif %}

{% if athlete_list or coach_list %}
    There are some athletes or some coaches.
{% endif %}

{% if not athlete_list or coach_list %}
    There are no athletes or there are some coaches (OK, so
    writing English translations of boolean logic sounds
    stupid; it's not our fault).
{% endif %}

{% if athlete_list and not coach_list %}
    There are some athletes and absolutely no coaches.
{% endif %}
```

`{% if %}` tags don't allow `and` and `or` clauses within the same tag, because the order of logic would be ambiguous. For example, this is invalid:

```
{% if athlete_list and coach_list or cheerleader_list %}
```

If you need to combine `and` and `or` to do advanced logic, just use nested `{% if %}` tags. For example:

```
{% if athlete_list %}
    {% if coach_list or cheerleader_list %}
        We have athletes, and either coaches or cheerleaders!
    {% endif %}
{% endif %}
```

Multiple uses of the same logical operator are fine, as long as you use the same operator. For example, this is valid:

```
{% if athlete_list or coach_list or parent_list or teacher_list %}
```

There is no `{% elif %}` tag. Use nested `{% if %}` tags to accomplish the same thing:

```
{% if athlete_list %}
    <p>Here are the athletes: {{ athlete_list }}.</p>
{% else %}
    <p>No athletes are available.</p>
    {% if coach_list %}
        <p>Here are the coaches: {{ coach_list }}.</p>
    {% endif %}
{% endif %}
```

Make sure to close each `{% if %}` with an `{% endif %}`. Otherwise, Django will throw a `TemplateSyntaxError`.

for

The `{% for %}` tag allows you to loop over each item in a sequence. As in Python's `for` statement, the syntax is `for X in Y`, where `Y` is the sequence to loop over and `X` is the name of the variable to use for a particular cycle of the loop. Each time through the loop, the template system will render everything between `{% for %}` and `{% endfor %}`.

For example, to display a list of athletes given a variable `athlete_list`:

```
<ul>
{% for athlete in athlete_list %}
    <li>{{ athlete.name }}</li>
{% endfor %}
</ul>
```

Add `reversed` to the tag to loop over the list in reverse:

```
{% for athlete in athlete_list reversed %}
...
{% endfor %}
```

It's possible to nest `{% for %}` tags:

```
{% for country in countries %}
    <h1>{{ country.name }}</h1>
    <ul>
    {% for city in country.city_list %}
        <li>{{ city }}</li>
    {% endfor %}
    </ul>
{% endfor %}
```

There is no support for "breaking" out of a loop before the loop is finished. If you want to accomplish this, change the variable you're looping over so that it only includes the values you want to loop over. Similarly, there is no support for a "continue" statement that would instruct the loop processor to return immediately to the front of the loop. (See "Philosophies and limitations" later in this chapter for the reasoning behind this design decision.)

The `{% for %}` tag sets a magic `forloop` template variable within the loop. This variable has a few attributes that give you information about the progress of the loop:

- `forloop.counter` is always set to an integer representing the number of times the loop has been entered. This is one-indexed, so the first time through the loop, `forloop.counter` will be set to `1`. Example:

  ```
  {% for item in todo_list %}
      <p>{{ forloop.counter }}: {{ item }}</p>
  {% endfor %}
  ```

- `forloop.counter0` is like `forloop.counter`, except it's zero-indexed. Its value will be set to `0` the first time through the loop.

- `forloop.revcounter` is always set to an integer representing the number of remaining items in the loop. The first time through the loop, `forloop.revcounter` will be set to the total number of items in the sequence you're traversing. The last time through the loop, `forloop.revcounter` will be set to `1`.

- `forloop.revcounter0` is like `forloop.revcounter`, except it's zero-indexed. The first time through the loop, `forloop.revcounter0` will be set to the number of elements in the sequence minus one. The last time through the loop, it will be set to `0`.

- `forloop.first` is a boolean value set to `True` if this is the first time through the loop. This is convenient for special-casing:

```
{% for object in objects %}
    {% if forloop.first %}<li class="first">{% else %}<li>{% endif %}
    {{ object }}
    </li>
{% endfor %}
```

- `forloop.last` is a boolean value set to `True` if this is the last time through the loop. An example use for this would be to put pipe characters between a list of links:

```
{% for link in links %}{{ link }}{% if not forloop.last %} | {% endif %}{% endfor %}
```

- `forloop.parentloop` is a reference to the `forloop` object for the *parent* loop, in case of nested loops. For example:

```
{% for country in countries %}
    <table>
    {% for city in country.city_list %}
        <tr>
        <td>Country #{{ forloop.parentloop.counter }}</td>
        <td>City #{{ forloop.counter }}</td>
        <td>{{ city }}</td>
        </tr>
    {% endfor %}
    </table>
{% endfor %}
```

The magic `forloop` variable is only available within loops. After the template parser has reached `{% endfor %}`, `forloop` disappears.

If your template context already contains a variable called `forloop`, Django will override it within `{% for %}` tags. In other, non-loop parts of the template, your `forloop` will still be available and unchanged. We advise against setting template variables with the name `forloop`, but if you need to do this and want to access your custom `forloop` from within a `{% for %}` tag, you can use `forloop.parentloop`, described above.

ifequal/ifnotequal

The Django template system deliberately is not a full-fledged programming language and, thus, does not allow you to execute arbitrary Python statements. (More on this in “Philosophies and limitations” below.) However, it’s quite a common template requirement to compare two values and display something if they’re equal — and Django provides an `{% ifequal %}` tag for that purpose.

The `{% ifequal %}` tag compares two values and displays everything between `{% ifequal %}` and `{% endifequal %}` if the values are equal.

This example compares the template variables `user` and `currentuser`:

```
{% ifequal user currentuser %}
    <h1>Welcome!</h1>
{% endifequal %}
```

The arguments can be hard-coded strings, with either single or double quotes, so the following is valid:

```
{% ifequal section 'sitenews' %}
    <h1>Site News</h1>
{% endifequal %}

{% ifequal section "community" %}
    <h1>Community</h1>
{% endifequal %}
```

Just like `{% if %}`, the `{% ifequal %}` tag supports an optional `{% else %}`:

```
{% ifequal section 'sitenews' %}
    <h1>Site News</h1>
{% else %}
    <h1>No News Here</h1>
{% endifequal %}
```

Only template variables, strings, integers and decimal numbers are allowed as arguments to `{% ifequal %}`. These are valid examples:

```
{% ifequal variable 1 %}
{% ifequal variable 1.23 %}
{% ifequal variable 'foo' %}
{% ifequal variable "foo" %}
```

Any other types of variables, such as Python dictionaries, lists or booleans, can not be hard-coded in `{% ifequal %}`. These are invalid examples:

```
{% ifequal variable True %}
{% ifequal variable [1, 2, 3] %}
{% ifequal variable {'key': 'value'} %}
```

If you need to test whether something is true or false, use the `{% if %}` tags instead of `{% ifequal %}`.

Comments

Just as in HTML or in a programming language such as Python, the Django template language allows for comments. To designate a comment, use `{# #}`. For example:

```
{# This is a comment #}
```

Comment will not be output when the template is rendered.

A comment cannot span multiple lines. In the following template, the rendered output will look exactly the same as the template (i.e., the comment tag will not be parsed as a comment):

```
This is a {# comment goes here
and spans another line #}
test.
```

Filters

As explained earlier in this chapter, template filters are simple ways of altering the value of variables before they're displayed.

Filters look like this:

```
{{ name|lower }}
```

This displays the value of the `{{ name }}` variable after being filtered through the `lower` filter, which converts text to lowercase. Use a pipe (`|`) to apply a filter.

Filters can be chained — that is, the output of one filter is applied to the next. Here's a common idiom for escaping text contents, then converting line breaks to `<p>` tags:

```
{{ my_text|escape|linebreaks }}
```

Some filters take arguments. A filter argument looks like this:

```
{{ bio|truncatewords:"30" }}
```

This displays the first 30 words of the `bio` variable. Filter arguments always are in double quotes.

Here are a few of the most important filters:

- `addslashes` — Adds a backslash before any backslash, single quote or double quote. This is useful if you're outputting some text into a JavaScript string.
- `date` — Formats a `date` or `datetime` object according to a format string given in the parameter. For example:

  ```
  {{ pub_date|date:"F j, Y" }}
  ```

Format strings are defined in Appendix 6.

- `escape` — Escapes ampersands, quotes and angle brackets in the given string. This is useful for sanitizing user-submitted data and for ensuring data is valid XML or XHTML. Specifically, `escape` makes these conversions:
  - Converts `&` to `&amp;`
  - Converts `<` to `&lt;`
  - Converts `>` to `&gt;`
  - Converts `"` (double quote) to `&quot;`
  - Converts `'` (single quote) to `&#39;`
- `length` — Returns the length of the value. You can use this on a list or a string, or any Python object that knows how to determine its length (i.e., any object that has a `__len__()` method).

Philosophies and limitations

Now that you've gotten a feel for the Django template language, we should point out some of its intentional limitations, along with some philosophies on why it works the way it works.

More than any other component of Web applications, programmer opinions on template systems vary wildly — a statement supported by the fact that Python alone has dozens, if not hundreds, of open-source template-language implementations, each inevitably created because its developer deemed all existing template languages inadequate. (In fact, it is said to be a rite of passage for a Python developer to write his or her own template language! And if you haven't done this yet, consider it. It's a fun exercise.)

With that in mind, the first Django philosophy to point out is that Django doesn't require that you use its template language. Because Django is intended to be a full-stack Web framework that provides all the pieces necessary to be a productive Web developer, many times it's *more convenient* to use Django's template system than other Python template libraries, but it's not a strict requirement in any sense. As we'll see in the section "Using templates in views" below, it's very easy to use another template language with Django — almost as easy as to use Django's template language.

Still, it's clear we have a strong preference for the way Django's template language works. The template system has roots in how Web development is done at World Online and the combined experience of Django's creators. Here are a few of those philosophies:

- **Business logic should be separated from presentation logic.** We see a template system as a tool that controls presentation and presentation-related logic — and that's it. The template system shouldn't support functionality that goes beyond this basic goal.

  For that reason, it's impossible to call Python code directly within Django templates. All "programming" is fundamentally limited to the scope of what template tags can do. It *is* possible to write custom template tags that do arbitrary things, but the out-of-the-box Django template tags intentionally do not allow for arbitrary Python code execution.

- **Syntax should be decoupled from HTML/XML.** Although Django's template system is used primarily to output HTML, it's intended to be just as usable for non-HTML formats, such as plain text. Some other template languages are XML-based, placing all template logic within XML tags or attributes, but Django deliberately avoids this limitation. Requiring valid XML to write templates introduces a world of human mistakes and hard-to-understand error messages, and using an XML engine to parse templates incurs an unacceptable level of overhead in template processing.

- **Designers are assumed to be comfortable with HTML code.** The template system isn't designed so that templates necessarily are displayed nicely in WYSIWYG editors such as Dreamweaver. That is too severe of a limitation and wouldn't allow the syntax to be as nice as it is. Django expects template authors are comfortable editing HTML directly.

- **Designers are assumed not to be Python programmers.** The template system authors recognize that Web page templates are most often written by *designers*, not *programmers*, and therefore should not assume Python knowledge.

  However, the system also intends to accomodate small teams in which the templates *are* created by Python programmers. It offers a way to extend the system's syntax by writing raw Python code. (More on this in Chapter 10.)

- **The goal is not to invent a programming language.** The goal is to offer just enough programming-esque functionality, such as branching and looping, that is essential for making presentation-related decisions.

As a result of these design philosophies, the Django template language has the following limitations:

- **A template cannot set a variable or change the value of a variable.** It's possible to write custom template tags that accomplish these goals (see Chapter 10), but the stock Django template tags do not allow it.
- **A template cannot call raw Python code.** There's no way to "drop into Python mode" or use raw Python constructs. Again, it's possible to write custom template tags to do this, but the stock Django template tags don't allow it.

Using templates in views

We've learned the basics of using the template system; now, let's integrate this into a view. Recall the `current_datetime` view from the previous chapter. Here's what it looked like:

```
from django.http import HttpResponse
import datetime

def current_datetime(request):
    now = datetime.datetime.now()
    html = "<html><body>It is now %s.</body></html>" % now
    return HttpResponse(html)
```

Let's change this view to use Django's template system. At first, you might think to do something like this:

```
from django.template import Template, Context
from django.http import HttpResponse
import datetime

def current_datetime(request):
    now = datetime.datetime.now()
    t = Template("<html><body>It is now {{ current_date }}.</body></html>")
    html = t.render(Context({'current_date': now}))
    return HttpResponse(html)
```

Sure, that uses the template system, but it doesn't solve the problems we pointed out in the introduction of this chapter. Namely, the template is still embedded in the Python code. Let’s fix that by putting the template in a *separate file*, which this view will load.

The simple, “dumb” way to do this would be to save your template somewhere on your filesystem and use Python’s built-in file-opening functionality to read the contents of the template. Here’s what that might look like, assuming the template was saved as the file `/home/djangouser/templates/mytemplate.html`:

```
from django.template import Template, Context
from django.http import HttpResponse
import datetime

def current_datetime(request):
    now = datetime.datetime.now()
    # Simple, "dumb" way of saving templates on the filesystem.
    # This doesn't account for missing files!
    fp = open('/home/djangouser/templates/mytemplate.html')
    t = Template(fp.read())
    fp.close()
    html = t.render(Context({'current_date': now}))
    return HttpResponse(html)
```

This approach, however, is inelegant for these reasons:

- For one, it doesn’t handle the case of a missing file. If the file `mytemplate.html` doesn’t exist or isn’t readable, the `open()` call would raise an `IOError` exception.
- Second, it hard-codes your template location. If you were to use this technique for every view function, you’d be duplicating the template locations. Not to mention that’s a lot of typing!
- Third, it includes a lot of boring boilerplate code. The calls to `open()`, `fp.read()` and `fp.close()` require a lot of typing and not much creativity.

To solve these issues, we’ll use *template loading* and *template directories*.

Template loading

Django provides a convenient and powerful API for loading templates from disk, with the goal of removing redundancy both in your template-loading calls and in your templates themselves.

In order to use this template-loading API, first you’ll need to tell the framework where you store your templates. The place to do this is in your **settings file**.

A Django settings file is the place to put configuration for your Django instance (aka your Django project). It’s a simple Python module with module-level variables, one for each setting.

When you ran `django-admin.py startproject mysite` in Chapter 2, the script created a default settings file for you, aptly named `settings.py`. Have a look at the file’s contents. It contains variables that look like this (though not necessarily in this order):

```
DEBUG = True
TIME_ZONE = 'America/Chicago'
USE_I18N = True
ROOT_URLCONF = 'mysite.urls'
```

This is pretty self-explanatory; the settings and their respective values are simple Python variables. And because the settings file is just a plain Python module, you can do dynamic things such as checking the value of one variable before setting another. (This also means that you should avoid Python syntax errors in your settings file.)

We'll cover settings files in depth later in this book, but for now, have a look at the `TEMPLATE_DIRS` setting. This setting tells Django's template loading mechanism where to look for templates. By default, it's an empty tuple. Pick a directory where you'd like to store your templates, and add it to `TEMPLATE_DIRS`, like so:

```
TEMPLATE_DIRS = (
    '/home/django/mysite/templates',
)
```

A few things to note:

- You can specify any directory you want, as long as the directory and templates within that directory are readable by the user account under which your Web server runs. If you can't think of an obvious place to put your templates, we recommend creating a `templates` directory within your Django project (i.e., within the `mysite` directory you created in Chapter 2, if you've been following along with our examples).

- Don't forget the comma at the end of the template-directory string! Python requires commas within single-element tuples to disambiguate the tuple from a parenthetical statement. This is a common newbie gotcha.

  If you want to avoid this error, you can make `TEMPLATE_DIRS` a list instead of a tuple, because single-element lists don't require a trailing comma:

  ```
  TEMPLATE_DIRS = [
      '/home/django/mysite/templates'
  ]
  ```

  A tuple is slightly more efficient than a list, though, so we recommend using a tuple for your `TEMPLATE_DIRS` setting.

- It's simplest to use absolute paths, i.e. directory paths that start at the root of the filesystem. If you want to be a bit more flexible and decoupled, though, you can take advantage of the fact that Django settings files are just Python code by constructing the contents of `TEMPLATE_DIRS` dynamically. For example:

  ```
  import os.path

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

This example uses the "magic" Python variable `__file__`, which is automatically set to the filename of the Python module in which the code lives.

- If you're on Windows, include your drive letter and use Unix-style forward slashes rather than backslashes. For example:

```
TEMPLATE_DIRS = (
    'C:/www/django/templates',
)
```

With `TEMPLATE_DIRS` set, the next step is to change the view code to use Django's template-loading functionality rather than hard-coding the template paths. Returning to our `current_datetime` view, let's change it like so:

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

def current_datetime(request):
    now = datetime.datetime.now()
    t = get_template('current_datetime.html')
    html = t.render(Context({'current_date': now}))
    return HttpResponse(html)
```

In this example, we're using the function `django.template.loader.get_template()` rather than loading the template from the filesystem manually. The `get_template()` function takes a template name as its argument, figures out where the template lives on the filesystem, opens that file and returns a compiled `Template` object.

If `get_template()` cannot find the template with the given name, it raises a `TemplateDoesNotExist` exception. To see what that looks like, fire up the Django development server again, as in Chapter 3, by running `python manage.py runserver` within your Django project's directory. Then, point your browser at the page that activates the `current_datetime` view (e.g., `http://127.0.0.1:8000/now/`). Assuming your `DEBUG` setting is set to `True` and you haven't yet created a `current_datetime.html` template, you should see a Django error page highlighting the `TemplateDoesNotExist` error.

(We'll have a screenshot here.)

This error page is similar to the one we explained in Chapter 3, with one additional piece of debugging information: a "Template-loader postmortem" section. This section tells you which templates Django tried to load, along with the reason each attempt failed (e.g., "File does not exist"). This information is invaluable when you're trying to debug template-loading errors.

As you can probably tell by looking at the error messages, Django attempted to look for a template by combining the directory in your `TEMPLATE_DIRS` setting with the template name you passed to `get_template()`. So if your `TEMPLATE_DIRS` contained `'/home/django/templates'`, it would look for the file `'/home/django/templates/current_datetime.html'`.

Moving along, create the `current_datetime.html` file within your template directory, using the following template code:

```
<html><body>It is now {{ current_date }}.</body></html>
```

Refresh the page in your Web browser, and you should see the fully rendered page.

render_to_response()

Because it’s such a common idiom to load a template, fill a `Context` and return an `HttpResponse` object with the result of the rendered template, Django provides a shortcut that lets you do those things in one line of code. This shortcut is a function called `render_to_response()`, which lives in the module `django.shortcuts`. Most of the time, you’ll be using `render_to_response()` rather than loading templates and creating `Context` and `HttpResponse` objects manually.

Here’s the ongoing `current_datetime` example rewritten to use `render_to_response()`:

```
from django.shortcuts import render_to_response
import datetime

def current_datetime(request):
    now = datetime.datetime.now()
    return render_to_response('current_datetime.html', {'current_date': now})
```

What a difference! Let’s step through the code changes:

- We no longer have to import `get_template`, `Template`, `Context` or `HttpResponse`. Instead, we import `django.shortcuts.render_to_response`. The `import datetime` remains.
- Within the `current_datetime` function, we still calculate `now`, but the template loading, context creation, template rendering and `HttpResponse` creation is all taken care of by the `render_to_response()` call. Because `render_to_response()` returns an `HttpResponse` object, we can simply `return` that value in the view.

The first argument to `render_to_response()` should be the name of the template to use, relative to your template directory. The second argument, if given, should be a dictionary to use in creating a `Context` for that template. If you don’t provide a second argument, `render_to_response()` will use an empty dictionary.

The locals() trick

Consider our latest incarnation of `current_datetime`:

```
def current_datetime(request):
    now = datetime.datetime.now()
    return render_to_response('current_datetime.html', {'current_date': now})
```

Many times, as in this example, you’ll find yourself calculating some values, storing them in variables (e.g., `now` above) and sending those variables to the template. Particularly lazy programmers would note that it’s slightly redundant to have to give names for temporary variables *and* give names for the template variables. Not only is it redundant; it’s extra typing.

So if you’re one of those lazy programmers and you like keeping code particularly concise, you can take advantage of a built-in

Python function called `locals()`. `locals()` returns a dictionary of all variables defined within the local scope, along with their values. Thus, the above view could be rewritten like so:

```
def current_datetime(request):
    current_date = datetime.datetime.now()
    return render_to_response('current_datetime.html', locals())
```

Here, instead of manually specifying the context dictionary as before, we instead pass the value of `locals()`, which will include all variables defined at that point in the function's execution. As a consequence, we've renamed the `now` variable to `current_date`, because that's the variable name that the template expects. In this example, `locals()` doesn't offer a *huge* improvement, but this technique can save you some typing if you've got several template variables to define — or if you're lazy.

One thing to watch out for when using `locals()` is that it includes *every* local variable, which may comprise more variables than you actually want your template to have access to. In the above example, `locals()` will also include `request`. Whether this matters to you depends on your application.

A final thing to consider is that `locals()` incurs a small bit of overhead, because when you call it, Python has to create the dictionary dynamically. If you specify the context dictionary manually, you avoid this overhead.

Subdirectories in get_template()

It can get unwieldy to store all of your templates in a single directory. You might like to store templates in subdirectories of your template directory, and that's fine. (In fact, we'd recommend it, and some more advanced Django features, such as the generic views system we'll cover in Chapter 9, expect this template layout as a default convention.)

Accomplishing that is easy. In your calls to `get_template()`, just include the subdirectory name and a slash before the template name, like so:

```
t = get_template('dateapp/current_datetime.html')
```

Because `render_to_response()` is a small wrapper around `get_template()`, you can do the same thing with the first argument to `render_to_response()`.

There's no limit to the depth of your subdirectory tree. Feel free to use subdirectories of subdirectories of subdirectories.

Windows users, note: Make sure to use forward slashes rather than backslashes. `get_template()` assumes a Unix-style filename designation.

The include template tag

Now that we've covered the template loading mechanism, we can introduce a built-in template tag that takes advantage of it: `{% include %}`. This tag allows you to include the contents of another template. The argument to the tag should be the name of the template to include, and the template name can be either a variable or a hard-coded (quoted) string, in either single or double quotes.

These two examples include the contents of the template `nav.html`. The examples are equivalent and illustrate that either single or double quotes are allowed:

```
{% include 'nav.html' %}
{% include "nav.html" %}
```

This example includes the contents of the template `includes/nav.html`:

```
{% include 'includes/nav.html' %}
```

This example includes the contents of the template whose name is contained in the variable `template_name`:

```
{% include template_name %}
```

As in `get_template()`, the filename of the template is determined by adding the template directory from `TEMPLATE_DIRS` to the requested template name.

If an included template contains any template code — such as tags or variables — then it will get evaluated with the context of the template that's including it.

If a template with the given name isn't found, Django will do one of two things:

- If your `DEBUG` setting is set to `True`, you'll see the `TemplateDoesNotExist` exception on a Django error page.
- If your `DEBUG` setting is set to `False`, the tag will fail silently, displaying nothing in the place of the tag.

Template inheritance

Our template examples so far have been tiny HTML snippets, but in the real world, you'll be using Django's template system to output entire HTML pages. This leads to a common Web development problem: Across a Web site, how does one reduce the duplication and redundancy of common page areas, such as sitewide navigation?

A classic way of solving this problem is to use server-side includes, directives you can embed within your HTML pages to "include" one Web page inside another. Indeed, Django supports that approach, with the `{% include %}` template tag we described above. But the preferred way of solving this problem with Django is to use a more elegant strategy called **template inheritance**.

In essence, template inheritance lets you build a base "skeleton" template that contains all the common parts of your site and defines "blocks" that child templates can override.

Let's see an example of this by creating a more complete template for our `current_datetime` view, by editing the `current_datetime.html` file:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
<html lang="en">
<head>
    <title>The current time</title>
</head>
```

```
<body>
    <h1>My helpful timestamp site</h1>
    <p>It is now {{ current_date }}.</p>

    <hr>
    <p>Thanks for visiting my site.</p>
</body>
</html>
```

That looks just fine, but what happens when we want to create a template for another view — say, the `hours_ahead` view from Chapter 3? If we want again to make a nice, valid, full HTML template, we'd create something like:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
<html lang="en">
<head>
    <title>Future time</title>
</head>
<body>
    <h1>My helpful timestamp site</h1>
    <p>In {{ hour_offset }} hour(s), it will be {{ next_time }}.</p>

    <hr>
    <p>Thanks for visiting my site.</p>
</body>
</html>
```

Clearly, we've just duplicated a lot of HTML. Imagine if we had a few stylesheets included on every page, maybe a navigation bar, perhaps some JavaScript... We'd end up putting all sorts of redundant HTML into each template.

The server-side include solution to this problem would be to factor out the common bits in both templates and save them in separate template snippets, which would then be included in each template. Perhaps you'd store the top bit of the template in a file called `header.html`:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
<html lang="en">
<head>
```

And perhaps you'd store the bottom bit in a file called `footer.html`:

```
    <hr>
    <p>Thanks for visiting my site.</p>
</body>
</html>
```

With an include-based strategy, headers and footers are easy. But it's the middle ground that's messy. In this example, both pages feature a title — `<h1>My helpful timestamp site</h1>` — but that title can't fit into `header.html` because the `<title>` on both pages is different. If we included the `<h1>` in the header, we'd have to include the `<title>`, which wouldn't allow us to customize it per page. See where this is going?

Django's template inheritance system solves these problems. You can think of it as an "inside out" version of server-side includes. Instead of defining the snippets that are *common*, you define the snippets that are *different*.

The first step is to define a **base template** — a skeleton of your page that **child templates** will later fill in. Here's a base template for our ongoing example:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
<html lang="en">
<head>
    <title>{% block title %}{% endblock %}</title>
</head>
<body>
    <h1>My helpful timestamp site</h1>
    {% block content %}{% endblock %}
    {% block footer %}
    <hr>
    <p>Thanks for visiting my site.</p>
    {% endblock %}
</body>
</html>
```

This template, which we'll call `base.html`, defines a simple HTML skeleton document that we'll use for all the pages on the site. It's the job of child templates to override, or add to, or leave alone the contents of the blocks. (If you're following along at home, save this file to your template directory.)

We're using a template tag here that you haven't seen before — the `{% block %}` tag. All the `{% block %}` tags do is to tell the template engine that a child template may override those portions of the template.

Now that we've got this base template, we can modify our existing `current_datetime.html` template to use it:

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

{% block title %}The current time{% endblock %}

{% block content %}
<p>It is now {{ current_date }}.</p>
{% endblock %}
```

While we're at it, let's create a template for the `hours_ahead` view from Chapter 3. (If you're following along with code, we'll leave it up to you to change `hours_ahead` to use the template system.) Here's what that would look like:

```
{% extends "base.html" %}
{% block title %}Future time{% endblock %}

{% block content %}
<p>In {{ hour_offset }} hour(s), it will be {{ next_time }}.</p>
{% endblock %}
```

Isn't this beautiful? Each template contains only the code that's *unique* to that template. No redundancy needed. If you need to make a sitewide design change, just make the change to `base.html`, and all of the other templates will immediately reflect the change.

Here's how it works:

- When you load the template `current_datetime.html`, the template engine sees the `{% extends %}` tag, noting that this template is a child template. The engine immediately loads the parent template — in this case, `base.html`.

- At that point, the template engine notices the three `{% block %}` tags in `base.html` and replaces those blocks with the contents of the child template. So, the title we've defined in `{% block title %}` will be used, as will the `{% block content %}`.

  Note that since the child template doesn't define the `footer` block, the template system uses the value from the parent template instead. Content within a `{% block %}` tag in a parent template is always used as a fallback.

You can use as many levels of inheritance as needed. One common way of using inheritance is the following three-level approach:

- Create a `base.html` template that holds the main look-and-feel of your site. This is the stuff that rarely, if ever, changes.
- Create a `base_SECTION.html` template for each "section" of your site. For example, `base_photos.html`, `base_forum.html`. These templates all extend `base.html` and include section-specific styles/design.
- Create individual templates for each type of page, such as a forum page or a photo gallery. These templates extend the appropriate section template.

This approach maximizes code reuse and makes it easy to add items to shared areas, such as section-wide navigation.

Here are some tips for working with template inheritance:

- If you use `{% extends %}` in a template, it must be the first template tag in that template. Otherwise, template inheritance won't work.
- Generally, the more `{% block %}` tags in your base templates, the better. Remember, child templates don't have to define all parent blocks, so you can fill in reasonable defaults in a number of blocks, then only define the ones you need in the child templates. It's better to have more hooks than fewer hooks.
- If you find yourself duplicating code in a number of templates, it probably means you should move that code to a `{% block %}` in a parent template.
- If you need to get the content of the block from the parent template, the `{{ block.super }}` variable will do the trick. This is useful if you want to add to the contents of a parent block instead of completely overriding it.
- You may not define multiple `{% block %}` tags with the same name in the same template. This limitation exists because a block tag works in "both" directions. That is, a block tag doesn't just provide a hole to fill — it also defines the content that fills

the hole in the *parent*. If there were two similarly-named `{% block %}` tags in a template, that template's parent wouldn't know which one of the blocks' content to use.

- The template name you pass to `{% extends %}` is loaded using the same method that `get_template()` uses. That is, the template name is appended to your `TEMPLATE_DIRS` setting.
- In most cases, the argument to `{% extends %}` will be a string, but it can also be a variable, if you don't know the name of the parent template until runtime. This lets you do some cool, dynamic stuff.

Exercises

Here are a few exercises that will solidify some of the things you learned in this chapter. (Hint: Even if you think you understood everything, at least give these exercises, and their respective answers, a read. We introduce a couple of new tricks here.)

1. You've got a list of musicians and the genre of music each one plays. This data is stored as a list of dictionaries, which will be hard-coded in your view module. (Usually we'd use a database for this, but we haven't yet covered Django's database layer.) The list looks like this:

```
MUSICIANS = [
    {'name': 'Django Reinhardt', 'genre': 'jazz'},
    {'name': 'Jimi Hendrix',     'genre': 'rock'},
    {'name': 'Louis Armstrong',  'genre': 'jazz'},
    {'name': 'Pete Townsend',    'genre': 'rock'},
    {'name': 'Yanni',            'genre': 'new age'},
    {'name': 'Ella Fitzgerald',  'genre': 'jazz'},
    {'name': 'Wesley Willis',    'genre': 'casio'},
    {'name': 'John Lennon',      'genre': 'rock'},
    {'name': 'Bono',             'genre': 'rock'},
    {'name': 'Garth Brooks',     'genre': 'country'},
    {'name': 'Duke Ellington',   'genre': 'jazz'},
    {'name': 'William Shatner',  'genre': 'spoken word'},
    {'name': 'Madonna',          'genre': 'pop'},
]
```

Write a Django view and corresponding template(s) that display an HTML `<table>` with a row for each musician in this list, in order. Each row should have two columns: the musician's name and the type of music he/she plays.

2. Once you've done that: For all the musicians who play jazz or rock — but *not* the others — bold their names by applying a `style="font-weight: bold;"` to their `<td>` cells.

3. Once you've done that: For all the musicians who have a one-word name — but *not* the others — display an asterisk after their name. Add a footnote to the page that says "* Pretentious." Maintain the `style="font-weight bold;"` from the previous exercise.

4. Given the following three templates, devise a template-inheritance scheme that removes as much redundancy as possible.

Template 1:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
<html lang="en">
<head>
    <link rel="stylesheet" href="default.css" type="text/css">
    <title>My to-do list</title>
</head>
<body>
    <h1 id="top">Latest tasks</h1>
    {% if task_list %}
        <ul>
        {% for task in task_list %}<li>{{ task }}</li>{% endfor %}
        </ul>
    {% else %}
        <p>You have no tasks.</p>
    {% endif %}
    <hr>
    <p><a href="#top">Back to top</a>.</p>
</body>
</html>
```

Template 2:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
<html lang="en">
<head>
    <title>Task: {{ task.title }} | To-do list</title>
    <link rel="stylesheet" href="default.css" type="text/css">
</head>
<body>
    <h1 id="top">{{ task.title }}</h1>
    <p>{{ task.description }}</p>
    <hr>
    <p><a href="#top">Back to top</a>.</p>
</body>
</html>
```

Template 3:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
<html lang="en">
<head>
    <title>Completed tasks | To-do list</title>
    <link rel="stylesheet" href="default.css" type="text/css">
    <script type="text/javascript" src="completed.js">
</head>
```

```
<body>
    <h1 id="top">{{ task.title }}</h1>
    <p>{{ task.description }}</p>
    <hr>
    <p><a href="#top">Back to top</a>.</p>
</body>
</html>
```

Answers to exercises

1. Here's one possible implementation of the view:

   ```
   from django.shortcuts import render_to_response

   MUSICIANS = [
       # ...
   ]

   def musician_list(request):
       return render_to_response('musician_list.html', {'musicians': MUSICIANS})
   ```

   And here's the template:

   ```
   <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
   <html lang="en">
   <head>
       <title>Musician list</title>
   </head>
   <body>
       <table>
       <tr><th>Musician</th><th>Genre</th></tr>
       {% for musician in musicians %}
           <tr>
           <td>{{ musician.name }}</td>
           <td>{{ musician.genre }}</td>
           </tr>
       {% endfor %}
       </table>
   </body>
   </html>
   ```

2. A rather clumsy way to do this would be to use `{% ifequal %}` in the template. The view would stay the same as in the answer to the last exercise, and the template's `{% for %}` loop would change to something like this:

```
{% for musician in musicians %}
    <tr>
    <td {% ifequal musician.genre 'jazz' %}style="font-weight: bold;"{% endifequal %}
        {% ifequal musician.genre 'rock' %}style="font-weight: bold;"{% endifequal %}>
      {{ musician.name }}
    </td>
    <td>{{ musician.genre }}</td>
    </tr>
{% endfor %}
```

This is overly verbose, repetitive and error-prone — and it illustrates an important point. A key to mastering Django's template system is to know *what to pass* to the template. Because templates do not have the full programming-language power of an environment you might be used to, it's more important in a Django environment to do as much *business logic* (as opposed to *presentation logic*) as possible in your view.

In this case, a cleaner way of solving the problem would be to have the view precalculate whether a musician's name is bolded. After all, this is business logic, not presentation logic. The presentation logic dictates *how* the special-case genres should be displayed, not *which* genres are special-cased. This is an important distinction.

Here's one way to code the view:

```
def musician_list(request):
    musicians = []
    for m in MUSICIANS:
        musicians.append({
            'name': m['name'],
            'genre': m['genre'],
            'is_important': m['genre'] in ('rock', 'jazz'),
        })
    return render_to_response('musician_list.html', {'musicians': musicians})
```

And with that view, you could use this template code:

```
{% for musician in musicians %}
    <tr>
    <td{% if musician.is_important %} style="font-weight: bold;"{% endif %}>
      {{ musician.name }}
    </td>
    <td>{{ musician.genre }}</td>
    </tr>
{% endfor %}
```

See how much cleaner that is in the template? Even this is more complex than it usually will be, because usually you'll be dealing with database objects, and database objects can have custom methods (such as `is_important()`). We'll cover database objects in the next chapter.

3. This is a similar problem to the previous exercise, and the solution is also similar. The key is to precalculate whether a musician deserves an asterisk next to his or her name. Because that's business logic, it belongs in the view.

   Here's one possible view implementation:

```
def musician_list(request):
    musicians = []
    for m in MUSICIANS:
        musicians.append({
            'name': m['name'],
            'genre': m['genre'],
            'is_important': m['genre'] in ('rock', 'jazz'),
            'is_pretentious': ' ' not in m['name'],
        })
    return render_to_response('musician_list.html', {'musicians': musicians})
```

   We're using the expression `' ' not in m['name']`, which returns `True` if `m['name']` doesn't include a space. You could also use the `.find()` method, like this:

```
'is_pretentious': m['name'].find(' ') == -1
```

   Note that we're calling this variable `is_pretentious` rather than `has_asterisk`, because the fact that we're using asterisks is a presentation decision.

   With the above view, you could use this template code:

```
{% for musician in musicians %}
    <tr>
    <td{% if musician.is_important %} style="font-weight: bold;"{% endif %}>
      {% if musician.is_pretentious %}* {% endif %}{{ musician.name }}
    </td>
    <td>{{ musician.genre }}</td>
    </tr>
{% endfor %}
```

   Don't forget the "* Pretentious." at the bottom of the template.

   For bonus points, be a perfectionist and only display the "* Pretentious" footnote if there's at least one pretentious musician. Determine the presence of a pretentious musician in the view, like so:

```
def musician_list(request):
    musicians = []
    has_pretentious = False
    for m in MUSICIANS:
        if ' ' not in m['name']:
            has_pretentious = True
        musicians.append({
            'name': m['name'],
            'genre': m['genre'],
            'is_important': m['genre'] in ('rock', 'jazz'),
            'is_pretentious': ' ' not in m['name'],
        })
    return render_to_response('musician_list.html', {
        'musicians': musicians,
        'has_pretentious': has_pretentious,
    })
```

In this implementation, we pass an extra template variable, `has_pretentious`, to the template. Then use it in the template like so:

```
{% if has_pretentious %}* Pretentious{% endif %}
```

4. Here's one way to write the base template:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
    <html lang="en">
    <head>
        <link rel="stylesheet" href="default.css" type="text/css">
        <title>{% block title %}{% endblock %}</title>
        {% block extrahead %}{% endblock %}
    </head>
    <body>
        <h1 id="top">{% block headline %}{% endblock %}</h1>
        {% block content %}{% endblock %}
        <hr>
        <p><a href="#top">Back to top</a>.</p>
    </body>
    </html>
```

And, with that base template, here's how the child templates might look.

Template 1:

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

{% block title %}My to-do list{% endblock %}

{% block headline %}Latest tasks{% endblock %}
{% block content %}
{% if task_list %}
    <ul>
    {% for task in task_list %}<li>{{ task }}</li>{% endfor %}
    </ul>
{% else %}
    <p>You have no tasks.</p>
{% endif %}
{% endblock %}
```

Template 2:

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

{% block title %}Task: {{ task.title }} | To-do list{% endblock %}

{% block headline %}{{ task.title }}{% endblock %}

{% block content %}<p>{{ task.description }}</p>{% endblock %}
```

Template 3:

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

{% block title %}Completed tasks | To-do list{% endblock %}

{% block extrahead %}<script type="text/javascript" src="completed.js">{% endblock %}

{% block headline %}{{ task.title }}{% endblock %}

{% block content %}<p>{{ task.description }}</p>{% endblock %}
```

Note that we like to put an empty line of space between `{% block %}` sections, but that's just our personal style. Any text outside of `{% block %}` tags in child templates will not be rendered.

« previous ◊ table of contents ◊ next »

Chapter 5: Interacting with a database: Models

In Chapter 3, we covered the fundamentals of building dynamic Web sites with Django: setting up views and URLconfs. As we explained, a view is responsible for doing *some arbitrary logic*, then returning a response. In the example, our *arbitrary logic* was to calculate the current date and time.

In modern Web applications, the *arbitrary logic* often involves interacting with a database. Behind the scenes, a **database-driven Web site** connects to a database server, retrieves some data out of it and displays that data, nicely formatted, on a Web page. Or, similarly, the site could provide functionality that lets site visitors populate the database on their own.

Many complex Web sites provide some combination of the two. Amazon.com, for instance, is a great example of a database-driven site. Each product page is essentially an extract of Amazon's product database formatted as HTML, and when you post a customer review, it gets inserted into the database of reviews.

Django is very well-suited for making database-driven Web sites, as it comes with easy yet powerful ways of performing database queries using Python. This chapter explains that functionality — Django's database layer.

(Note: While it's not strictly necessary to know basic database theory and SQL in order to use Django's database layer, it's highly recommended. An introduction to those concepts is out of the scope of this book, but keep reading even if you're a database newbie. You'll probably be able to follow along and grasp concepts based on context.)

The "dumb" way to do database queries in views

Just as the previous chapter detailed a "dumb" way to output HTML within a view (by hard-coding HTML directly within the view), there's a "dumb" way to retrieve data from a database in a view. It's simple: Just use any existing Python library to execute an SQL query and do something with the results.

In this example view, we use the `MySQLdb` library (available at http://sourceforge.net/projects/mysql-python) to connect to a MySQL database, retrieve some records and feed them to a template for display as a Web page:

```
from django.shortcuts import render_to_response
import MySQLdb

def book_list(request):
    db = MySQLdb.connect(user='me', db='mydb', passwd='secret', host='localhost')
    cursor = db.cursor()
    cursor.execute('SELECT name FROM books ORDER BY name')
    names = [row[0] for row in cursor.fetchall()]
    db.close()
    return render_to_response('book_list.html', {'names': names})
```

This approach works, but some problems should jump out at you immediately:

- We're hard-coding the database connection parameters. Ideally, these parameters would be stored in the Django configuration.
- We're having to write a fair bit of boilerplate code: creating a connection, creating a cursor, executing a statement and closing the connection. Ideally, all we'd have to do is specify which results we wanted.
- It ties us to MySQL. If, down the road, we switch from MySQL to PostgreSQL, we'll have to use a different database adapter (e.g., `psycopg` rather than `MySQLdb`), alter the connection parameters and — depending on the nature of the SQL statement — possibly rewrite the SQL. Ideally, the database server we're using would be abstracted, so that a database server change could be made in a single place.

As you might expect, Django's database layer aims to solve these problems. Here's a sneak preview of how the above view can be rewritten using Django's database API:

```
from django.shortcuts import render_to_response
from mysite.books.models import Book

def book_list(request):
    books = Book.objects.order_by('name')
    return render_to_response('book_list.html', {'books': books})
```

We'll explain this code a little later in this chapter. For now, just get a feel for how it looks.

The MTV development pattern

Before we delve into any more code, let's take a moment to consider the overall design of a database-driven Django Web application.

As we've mentioned in previous chapters, Django is designed to encourage loose coupling and strict separation between pieces of an application. If you follow this philosophy, it's easy to make changes to one particular piece of the application without affecting other pieces of the application. In view functions, for instance, we discussed the importance of separating the business logic from the presentation logic by using a template system. With the database layer, we're applying that same philosophy to data-access logic.

Those three pieces together — data-access logic, business logic and presentation logic — comprise a concept that's sometimes called the "Model View Controller" (MVC) pattern of software architecture. In this pattern, "Model" refers to the data-access layer, "View" refers to the part of the system that selects what to display and how to display it, and "Controller" refers to the part of the system that decides which view to use, depending on user input, accessing the model as needed.

Why the acronym?

MVC? MTV? What's the point of these terms?

The goal of explicitly defining patterns such as MVC is mostly to streamline communication among developers. Instead of having to tell your coworkers, "Let's make an abstraction of the data-access, then have a separate layer that handles data display, and let's put a layer in the middle that regulates this," you can take advantage of a shared vocabulary and say, "Let's use the MVC pattern here."

Django follows this MVC pattern closely enough that it can be called an MVC framework. Here's roughly how the M, V and C break down in Django:

- **M**, the data-access portion, is handled by Django's database layer, which is described in this chapter.
- **V**, the portion that selects which data to display and how to display it, is handled by views and templates.
- **C**, the portion that delegates to a view depending on user input, is handled by the framework itself by following your URLconf and calling the appropriate Python function for the given URL.

Because the "C" is handled by the framework itself and most of the excitement in Django happens in models, templates and views, Django has been referred to as an **MTV framework**. In the MTV development pattern,

- "M" stands for model, the data-access layer. This layer contains anything and everything about the data: how to access it, how to validate it, which behaviors it has and the relationships between the data.
- "T" stands for template, the presentation layer. This layer contains presentation-related decisions: how something should be displayed on a Web page or other type of document.
- "V" stands for view, the business-logic layer. This layer contains the logic that access the model and defers to the appropriate template(s). You can think of it as the bridge between models and templates.

If you're familiar with other MVC Web-development frameworks, such as Ruby on Rails, you may consider Django views to be the "controllers" and Django templates to be the "views." This is an unfortunate confusion brought about by differing interpretations of MVC. In Django's interpretation of MVC, the "view" describes the data that gets presented to the user; it's not necessarily just *how* the data looks, but *which* data is presented. In contrast, Ruby on Rails and similar frameworks suggest that the controller's job includes deciding which data gets presented to the user, whereas the view is strictly *how* the data looks, not *which* data is presented.

Neither interpretation is more "correct" than the other. The important thing is to understand the underlying concepts.

Configuring the database

With all of that philosophy in mind, let's start exploring Django's database layer. First, we need to take care of some initial configuration; we need to tell Django which database server to use and how to connect to it.

We'll assume you've set up a database server, activated it and created a database within it (e.g., using a `CREATE DATABASE` statement). SQLite is a special case; in that case, there's no database to create, because SQLite uses standalone files on the filesystem to store its data.

As `TEMPLATE_DIRS` in the previous chapter, database configuration lives in the Django settings file, called `settings.py` by default. Edit that file and look for the database settings:

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

Here's a rundown of each setting.

- `DATABASE_ENGINE` tells Django which database engine to use. If you're using a database with Django, `DATABASE_ENGINE` must be set to one of the following strings:

Setting	Database	Required adapter
`postgresql`	PostgreSQL	`psycopg` version 1.x, http://initd.org/projects/psycopg1
`postgresql_psycopg2`	PostgreSQL	`psycopg` version 2.x, http://initd.org/projects/psycopg2
`mysql`	MySQL	`MySQLdb`, http://sourceforge.net/projects/mysql-python
`sqlite3`	SQLite	No adapter needed if using Python 2.5+. Otherwise, `pysqlite`, http://initd.org/tracker/pysqlite
`ado_mssql`	Microsoft SQL Server	`adodbapi` version 2.0.1+, http://adodbapi.sourceforge.net/
`oracle`	Oracle	`cx_Oracle`, http://www.python.net/crew/atuining/cx_Oracle/

Note that for whichever database backend you use, you'll need to download and install the appropriate database adapter. Each one is available for free on the Web.

- `DATABASE_NAME` tells Django what the name of your database is. If you're using SQLite, specify the full filesystem path to the database file on your filesystem, e.g., `'/home/django/mydata.db'`
- `DATABASE_USER` tells Django which username to use when connecting to your database. If you're using SQLite, leave this blank.
- `DATABASE_PASSWORD` tells Django which password to use when connecting to your database. If you're using SQLite or have an empty password, leave this blank.
- `DATABASE_HOST` tells Django which host to use when connecting to your database. If your database is on the same computer as your Django installation (i.e., localhost), leave this blank. If you're using SQLite, leave this blank.

  MySQL is a special case here. If this value starts with a forward slash (`'/'`) and you're using MySQL, MySQL will connect via a Unix socket to the specified socket. For example:

  ```
  DATABASE_HOST = '/var/run/mysql'
  ```

  If you're using MySQL and this value *doesn't* start with a forward slash, then this value is assumed to be the host.
- `DATABASE_PORT` tells Django which port to use when connecting to your database. If you're using SQLite, leave this blank. Otherwise, if you leave this blank, the underlying database adapter will use whichever port is default for your given database server. In most cases, the default port is fine, so you can leave this blank.

Once you've entered those settings, test your configuration. First, from within the `mysite` project directory you created in Chapter 2, run the command `python manage.py shell`.

You'll notice this starts a Python interactive interpreter. Looks can be deceiving, though! There's an important difference between running the command `python manage.py shell` within your Django project directory and the more generic `python`. The latter is the basic Python shell, but the former tells Django which settings file to use before it starts the shell. This is a key requirement for doing database queries: Django needs to know which settings file to use in order to get your database connection information.

Behind the scenes, `python manage.py shell` sets the environment variable `DJANGO_SETTINGS_MODULE`. We'll cover the subtleties of this later, but for now, just know that you should use `python manage.py shell` whenever you need to drop into the Python interpreter to do Django-specific tinkering.

Once you've entered the shell, type these commands to test your database configuration:

```
>>> from django.db import connection
>>> cursor = connection.cursor()
```

If nothing happens, then your database is configured properly. Otherwise, check the error message for clues about what's wrong. Here are some common errors:

Error message	Solution
You haven't set the DATABASE_ENGINE setting yet.	Set the `DATABASE_ENGINE` setting to something other than an empty string.
Environment variable DJANGO_SETTINGS_MODULE is undefined.	Run the command `python manage.py shell` rather than `python`.
Error loading _____ module: No module named _____.	You haven't installed the appropriate database-specific adapter (e.g. `psycopg` or `MySQLdb`).
_____ isn't an available database backend.	Set your `DATABASE_ENGINE` setting to one of the valid engine settings described above. Perhaps you made a typo?

database _____ does not exist	Change the `DATABASE_NAME` setting to point to a database that exists, or execute the appropriate `CREATE DATABASE` statement in order to create it.
role _____ does not exist	Change `DATABASE_USER` setting to point to a user that exists, or create the user in your database.
could not connect to server	Make sure `DATABASE_HOST` and `DATABASE_PORT` are set correctly, and make sure the server is running.

Your first app

Now that you've verified the connection is working, it's time to create a **Django app** — a bundle of Django code, including models and views, that lives together in a single Python package and represents a full Django application.

It's worth explaining the terminology here, because this tends to trip up beginners. We'd already created a *project*, in Chapter 2, so what's the difference between a *project* and an *app*? The difference is that of configuration vs. code:

- A project is an instance of a certain set of Django apps, plus the configuration for those apps.

  Technically, the only requirement of a project is that it supplies a settings file, which defines the database connection information, the list of installed apps, the `TEMPLATE_DIRS`, etc.

- An app is a portable set of Django functionality, usually including models and views, that lives together in a single Python package.

  For example, Django comes with a number of apps, such as a commenting system and an automatic admin interface. A key thing to note about these apps is that they're portable and reusable across multiple projects.

There are very few hard-and-fast rules about how you fit your Django code into this scheme; it's flexible. If you're building a simple Web site, you may only use a single app. If you're building a complex Web site with several rather unrelated pieces such as an e-commerce system and a message board, you'll probably want to split those into separate apps so that you'll be able to reuse them individually in the future.

Indeed, you don't necessarily need to create apps at all, as evidenced by the example view functions we've created so far in this book. In those cases, we simply created a file called `views.py`, filled it with view functions and pointed our URLconf at those functions. No "apps" were needed.

However, there's one requirement regarding the app convention: If you're using Django's database layer (models), you must create a Django app. Models must live within apps. Thus, in order to start writing our models, we'll need to create a new app.

Within the `mysite` project directory you created in Chapter 2, type this command to create a new app:

```
python manage.py startapp books
```

(Why `books`? That's the sample book app we'll be building together.)

This command does not result in any output, but it will have created a `books` directory within the `mysite` directory. Let's look at the contents of that directory:

```
books/
    __init__.py
    models.py
    views.py
```

These files will contain your models and views for this app.

Have a look at `models.py` and `views.py` in your favorite text editor. Both files are empty, except for an import in `models.py`. This is the blank slate for your Django app.

Defining models in Python

As we discussed above, the "M" in "MTV" stands for "Model." A Django model is a description of the data in your database, represented as Python code. It's your data layout — the equivalent of your SQL `CREATE TABLE` statements — except it's in Python instead of SQL, and it includes more than just database definitions. Django uses a model to execute SQL code behind the scenes and return convenient Python data structures representing the rows in your database tables. Django also uses models to represent higher-level concepts that SQL can't necessarily handle.

If you're familiar with databases, your immediate thought might be, "Isn't it redundant to define data models in Python *and* in SQL?" Django works the way it does for several reasons:

- Introspection requires overhead and is imperfect.

  In order to provide convenient data-access APIs, Django needs to know the database layout *somehow*, and there are two ways of accomplishing this. The first way would be to explicitly describe the data in Python, and the second way would be to introspect the database at runtime to determine the data models.

This second way seems cleaner, because the metadata about your tables only lives in one place, but it introduces a few problems. First, introspecting a database at runtime obviously requires overhead. If the framework had to introspect the database each time it processed a request, or even when the Web server was initialized, this would incur an unacceptable level of overhead. (While some believe that level of overhead is acceptable, Django's developers aim to trim as much framework overhead as possible, and this approach has succeeded in making Django faster than its high-level framework competitors in benchmarks.) Second, some databases, notably older versions of MySQL, do not store sufficient metadata for accurate and complete introspection.

- Writing Python is fun, and keeping everything in Python limits the number of times your brain has to do a "context switch." It helps productivity if you keep yourself in a single programming environment/mentality for as long as possible. Having to write SQL, then Python, then SQL again, is disruptive.
- Having data models stored as code rather than in your database makes it easier to keep your models under version control. This way, you can easily keep track of changes to your data layouts.
- SQL only allows for a certain level of metadata about a data layout. Most database systems, for example, do not provide a specialized data type for representing e-mail addresses or URLs. Django models do. The advantage of higher-level data types is higher productivity and more reusable code.
- SQL is inconsistent across database platforms. If you're distributing a Web application, for example, it's much more pragmatic to distribute a Python module that describes your data layout than separate sets of `CREATE TABLE` statements for MySQL, PostgreSQL and SQLite.

A drawback of this approach, however, is that it's possible for the Python code to get out of sync with what's actually in the database. If you make changes to a Django model, you'll need to make the same changes inside your database to keep your database consistent with the model. We'll detail some strategies for handling this problem later in this chapter.

Finally, we should note that Django includes a utility that can generate models by introspecting an existing database. This is useful for quickly getting up and running with legacy data.

Your first model

As an ongoing example in this chapter and the next chapter, we'll focus on a basic book/author/publisher data layout. We use this as our example because the conceptual relationships between books, authors and publishers are well-known, and this is a common data layout used in introductory SQL textbooks. You're also reading a book, written by authors, produced by a publisher!

We'll suppose the following concepts, fields and relationships:

- An author has a salutation (e.g., Mr. or Mrs.), a first name, a last name, an e-mail address and a headshot photo.
- A publisher has a name, a street address, a city, a state/province, a country and a Web site.
- A book has a title and a publication date. It also has one or more authors (a many-to-many relationship to author) and a single publisher (a one-to-many relationship, aka foreign key, to publisher).

The first step in using this database layout with Django is to express it as Python code. In the `models.py` file that was created by the `startapp` command, enter the following:

```
from django.db import models

class Publisher(models.Model):
    name = models.CharField(maxlength=30)
    address = models.CharField(maxlength=50)
    city = models.CharField(maxlength=60)
    state_province = models.CharField(maxlength=30)
    country = models.CharField(maxlength=50)
    website = models.URLField()

class Author(models.Model):
    salutation = models.CharField(maxlength=10)
    first_name = models.CharField(maxlength=30)
    last_name = models.CharField(maxlength=40)
    email = models.EmailField()
    headshot = models.ImageField(upload_to='/tmp')

class Book(models.Model):
    title = models.CharField(maxlength=100)
    authors = models.ManyToManyField(Author)
    publisher = models.ForeignKey(Publisher)
    publication_date = models.DateField()
```

We will cover model syntax and options throughout this chapter, but let's quickly examine this code to cover the basics. The first thing to notice is that each model is represented by a Python class that is a subclass of `django.db.models.Model`. The parent class, `Model`, contains all the machinery necessary to make these objects capable of interacting with a database — and that leaves our models responsible solely for defining their fields, in a nice and compact syntax. Believe it or not, this is all the code we need to write to have basic data access with Django.

Each model generally corresponds to a single database table, and each attribute on a model generally corresponds to a column in that database table. The attribute name corresponds to the column's name, and the type of field (e.g., `CharField`) corresponds to the database column type (e.g., `varchar`). For example, the `Publisher` model is equivalent to the following table (assuming PostgreSQL `CREATE TABLE` syntax):

```
CREATE TABLE "books_publisher" (
    "id" serial NOT NULL PRIMARY KEY,
    "name" varchar(30) NOT NULL,
    "address" varchar(50) NOT NULL,
    "city" varchar(60) NOT NULL,
    "state_province" varchar(30) NOT NULL,
    "country" varchar(50) NOT NULL,
    "website" varchar(200) NOT NULL
);
```

Indeed, Django can generate that `CREATE TABLE` statement itself, as we'll see in a moment.

The exception to the one-class-per-database-table rule is the case of many-to-many relationships. In our example models, `Book` has a `ManyToManyField` called `authors`. This designates that a book has one or many authors, but the `Book` database table doesn't get an `authors` column. Rather, Django creates an additional table — a many-to-many "join table" — that handles the mapping of books to authors.

Finally, note we haven't explicitly defined a primary key in any of these models. Unless you instruct it otherwise, Django automatically gives every model an integer primary key field called `id`. Each Django model is required to have a single-column primary key.

Installing the model

We've written the code; now, let's create the tables in our database. In order to do that, the first step is to *activate* these models in our Django project. We do that by adding this `books` app to the list of installed apps in the settings file.

Edit the `settings.py` file again, and look for the `INSTALLED_APPS` setting. `INSTALLED_APPS` tells Django which apps are activated for a given project. By default, it looks something like this:

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

Temporarily comment out all four of those strings by putting a hash character (`#`) in front of them. (They're included by default as a common-case convenience, but we'll activate them and discuss them later.) Then, add `'mysite.books'` to the `INSTALLED_APPS` list, so the setting ends up looking like this:

```
INSTALLED_APPS = (
    #'django.contrib.auth',
    #'django.contrib.contenttypes',
    #'django.contrib.sessions',
    #'django.contrib.sites',
    'mysite.books',
)
```

(As we're dealing with a single-element tuple here, don't forget the trailing comma. By the way, this book's authors prefer to put a comma after *every* element of a tuple, regardless of whether the tuple has only a single element. This avoids the issue of forgetting commas, and there's no penalty for using that extra comma.)

`'mysite.books'` refers to the `books` app we're working on. Each app in `INSTALLED_APPS` is represented by its full Python path — that is, the path of packages, separated by dots, leading to the app package.

Now that the Django app has been activated in the settings file, we can create the database tables in our database. First, let's validate the models by running this command:

```
python manage.py validate
```

The `validate` command checks whether your models' syntax and logic are correct. If all is well, you'll see the message `0 errors found`. If you don't, make sure you typed in the model code correctly. The error output should give you helpful information about what was wrong with the code.

Any time you think you have problems with your models, run `python manage.py validate`. It tends to catch all the common model problems.

If your models are valid, run the following command for Django to generate `CREATE TABLE` statements for your models in the `books` app (with colorful syntax highlighting available if you're using Unix):

```
python manage.py sqlall books
```

In this command, `books` is the name of the app. It's what you specified when you ran the command `manage.py startapp`. When you run the command, you should see something like this:

```
BEGIN;
CREATE TABLE "books_publisher" (
    "id" serial NOT NULL PRIMARY KEY,
    "name" varchar(30) NOT NULL,
    "address" varchar(50) NOT NULL,
    "city" varchar(60) NOT NULL,
    "state_province" varchar(30) NOT NULL,
    "country" varchar(50) NOT NULL,
    "website" varchar(200) NOT NULL
);
CREATE TABLE "books_book" (
    "id" serial NOT NULL PRIMARY KEY,
    "title" varchar(100) NOT NULL,
    "publisher_id" integer NOT NULL REFERENCES "books_publisher" ("id"),
    "publication_date" date NOT NULL
);
CREATE TABLE "books_author" (
    "id" serial NOT NULL PRIMARY KEY,
    "salutation" varchar(10) NOT NULL,
    "first_name" varchar(30) NOT NULL,
    "last_name" varchar(40) NOT NULL,
    "email" varchar(75) NOT NULL,
    "headshot" varchar(100) NOT NULL
);
CREATE TABLE "books_book_authors" (
    "id" serial NOT NULL PRIMARY KEY,
    "book_id" integer NOT NULL REFERENCES "books_book" ("id"),
    "author_id" integer NOT NULL REFERENCES "books_author" ("id"),
    UNIQUE ("book_id", "author_id")
);
CREATE INDEX books_book_publisher_id ON "books_book" ("publisher_id");
COMMIT;
```

Note the following:

- Table names are automatically generated by combining the name of the app (`books`) and the lowercase name of the model — `publisher`, `book` and `author`. You can override this behavior, as we'll see later in this chapter.
- As we mentioned above, Django adds a primary key for each table automatically — the `id` fields. You can override this, too.
- By convention, Django appends `"_id"` to the foreign key field name. As you might have guessed, you can override this behavior, too.
- The foreign key relationship is made explicit by a `REFERENCES` statement.
- These `CREATE TABLE` statements are tailored to the database you're using, so database-specific field types such as `auto_increment` (MySQL), `serial` (PostgreSQL), or `integer primary key` (SQLite) are handled for you automatically. The same goes for quoting of column names — e.g., using double quotes or single quotes. This example output is in PostgreSQL syntax.

The `sqlall` command doesn't actually create the tables or otherwise touch your database — it just prints output to the screen so you can see what SQL Django would execute if you asked it. If you wanted to, you could copy and paste this SQL into your database client, or use Unix pipes to pass it directly. However, Django provides an easier way of committing the SQL to the database. Run the `syncdb` command, like so:

```
python manage.py syncdb
```

You'll see something like this:

```
Creating table books_publisher
Creating table books_book
Creating table books_author
Installing index for books.Book model
```

The `syncdb` command is a simple "sync" of your models to your database. It looks at all of the models in each app in your `INSTALLED_APPS` setting, checks the database to see whether the appropriate tables exist yet, and creates the tables if they don't yet exist. Note that `syncdb` does *not* sync changes in models or deletions of models; if you make a change to a model or delete a model, and you want to update the database, `syncdb` will not handle that. (More on this later.)

If you run `python manage.py syncdb` again, nothing happens, because you haven't added any models to the `books` app, or added any apps to `INSTALLED_APPS`. Ergo, it's always safe to run `python manage.py syncdb` — it won't clobber things.

If you're interested, take a moment to dive into your database server's command-line client and see the database tables Django created. You can manually run the command-line client — e.g., `psql` for PostgreSQL — or you can run the command `python manage.py dbshell`, which will figure out which command-line client to run, depending on your `DATABASE_SERVER` setting. The latter is almost always more convenient.

Basic data access

Once you've created a model, Django automatically provides a high-level Python API for working with those models. Try it out by running `python manage.py shell` and typing the following:

```
>>> from books.models import Publisher
>>> p = Publisher(name='Apress', address='2560 Ninth St.',
...     city='Berkeley', state_province='CA', country='U.S.A.',
...     website='http://www.apress.com/')
>>> p.save()
>>> p = Publisher(name="O'Reilly", address='10 Fawcett St.',
...     city='Cambridge', state_province='MA', country='U.S.A.',
...     website='http://www.oreilly.com/')
>>> p.save()
>>> publisher_list = Publisher.objects.all()
>>> publisher_list
[<Publisher: Publisher object>, <Publisher: Publisher object>]
```

In only a few lines of code, this has accomplished quite a bit. The highlights:

- To create an object, just import the appropriate model class and instantiate it by passing in values for each field.
- To save the object to the database, call the `save()` method on the object. Behind the scenes, Django executes an SQL `INSERT` statement here.
- To retrieve objects from the database, use the attribute `Publisher.objects`. Fetch a list of all `Publisher` objects in the database with the statement `Publisher.objects.all()`. Behind the scenes, Django executes an SQL `SELECT` statement here.

Naturally, you can do quite a lot with the Django database API — but first, let's take care of a small annoyance.

Adding model string representations

Above, when we printed out the list of publishers, all we got was this unhelpful display that makes it difficult to tell the `Publisher` objects apart:

```
[<Publisher: Publisher object>, <Publisher: Publisher object>]
```

We can fix this easily by adding a method called `__str__()` to our `Publisher` object. A `__str__()` method tells Python how to display the "string" representation of an object. You can see this in action by adding a `__str__()` method to the three models:

```
class Publisher(models.Model):
    name = models.CharField(maxlength=30)
    address = models.CharField(maxlength=50)
    city = models.CharField(maxlength=60)
    state_province = models.CharField(maxlength=30)
    country = models.CharField(maxlength=50)
    website = models.URLField()

    def __str__(self):
        return self.name

class Author(models.Model):
    salutation = models.CharField(maxlength=10)
    first_name = models.CharField(maxlength=30)
    last_name = models.CharField(maxlength=40)
    email = models.EmailField()
    headshot = models.ImageField(upload_to='/tmp')

    def __str__(self):
        return '%s %s' % (self.first_name, self.last_name)

class Book(models.Model):
    title = models.CharField(maxlength=100)
    authors = models.ManyToManyField(Author)
    publisher = models.ForeignKey(Publisher)
    publication_date = models.DateField()

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

As you can see, a `__str__()` method can do whatever it needs to do in order to return a string representation. Here, the `__str__()` methods for `Publisher` and `Book` simply return the object's name and title, respectively, but the `__str__()` for `Author` is slightly more complex — it pieces together the `first_name` and `last_name` fields. The only requirement for `__str__()` is that it return a string. If `__str__()` doesn't return a string — if it returns, say, an integer — then Python will raise a `TypeError` with a message like `"__str__ returned non-string"`.

For the changes to take effect, exit out of the Python shell and enter it again with `python manage.py shell`. (This is the easiest way to make code changes take effect.) Now, the list of `Publisher` objects is much easier to understand:

```
>>> from books.models import Publisher
>>> publisher_list = Publisher.objects.all()
>>> publisher_list
[<Publisher: Apress>, <Publisher: O'Reilly>]
```

Make sure any model you define has a `__str__()` method — not only for your own convenience when using the interactive interpreter, but also because Django uses the output of `__str__()` in several places when it needs to display objects.

Finally, note that `__str__()` is a good example of adding *behavior* to models. A Django model describes more than the database table layout for an object; it also describes any functionality that object knows how to do. `__str__()` is one example of such functionality — a model knows how to display itself.

Creating and modifying objects

This chapter is not yet finished.

« previous ◊ table of contents ◊ next »

Chapter 6: The Django admin site

There's one part of Web development we've always hated: writing administration interfaces. Developing the parts of the site that the general public sees is always different and interesting, but the bits that the administrators use to modify the site are always the same. You've got to deal with authenticating users, display and handle forms, deal with tricky validation issues... It's boring, and it's repetitive.

Django's approach to this boring, repetitive task? Do it all for you — in just a couple of lines of code, no less.

One of the oldest and most powerful parts of Django is the automatic admin interface. It hooks off of metadata in your model to provide a powerful and production-ready interface that content producers can immediately use to start adding content to the site.

Activating the admin interface

We think the admin interface is the coolest part of Django — and most Djangonauts agree — but since not everyone actually needs it, it's an optional piece. That means there are three steps you'll need to follow to activate the admin interface:

1. Add admin metadata to your models.

   Not all models can (or should) be editable by admin users, so you need to "mark" models that should have an admin interface. You do that be adding an inner `Admin` class to your model (alongside the `Meta` class, if you have one). So, to add an admin interface to our `Book` model from the previous chapter:

```
class Book(models.Model):
    title = models.CharField(maxlength=100)
    authors = models.ManyToManyField(Author)
    publisher = models.ForeignKey(Publisher)
    publication_date = models.DateField()

    class Admin:
        pass
```

   The `Admin` declaration flags the class as having an admin interface. There are a number of options that you can put beneath `Admin`, but for now we're sticking with all the defaults, so we put `pass` in there to signify to Python that the `Admin` class is empty.

   If you're following this example with your own code, it's probably a good idea to add `Admin` declarations to the `Publisher` and `Author` classes at this point.

2. Install the admin models. Simply add `"django.contrib.admin"` to your `INSTALLED_APPS` setting and run `python manage.py syncdb` to install the extra tables the admin uses.

Note

When you first ran `syncdb`, you were probably asked about creating a superuser. If you didn't that time, you'll need to run `django/contrib/auth/bin/create_superuser.py` to create an admin user. Otherwise you won't be able to log into the admin interface.

3. Add the URL pattern to your `urls.py`. If you're still using the one created by `startproject`, the admin URL pattern should be already there, but commented out. Either way, your URL patterns should look like:

```
from django.conf.urls.defaults import *

urlpatterns = patterns('',
    (r'^admin/', include('django.contrib.admin.urls')),
)
```

That's it. Now run `python manage.py runserver` to start the development server; you'll see something like:

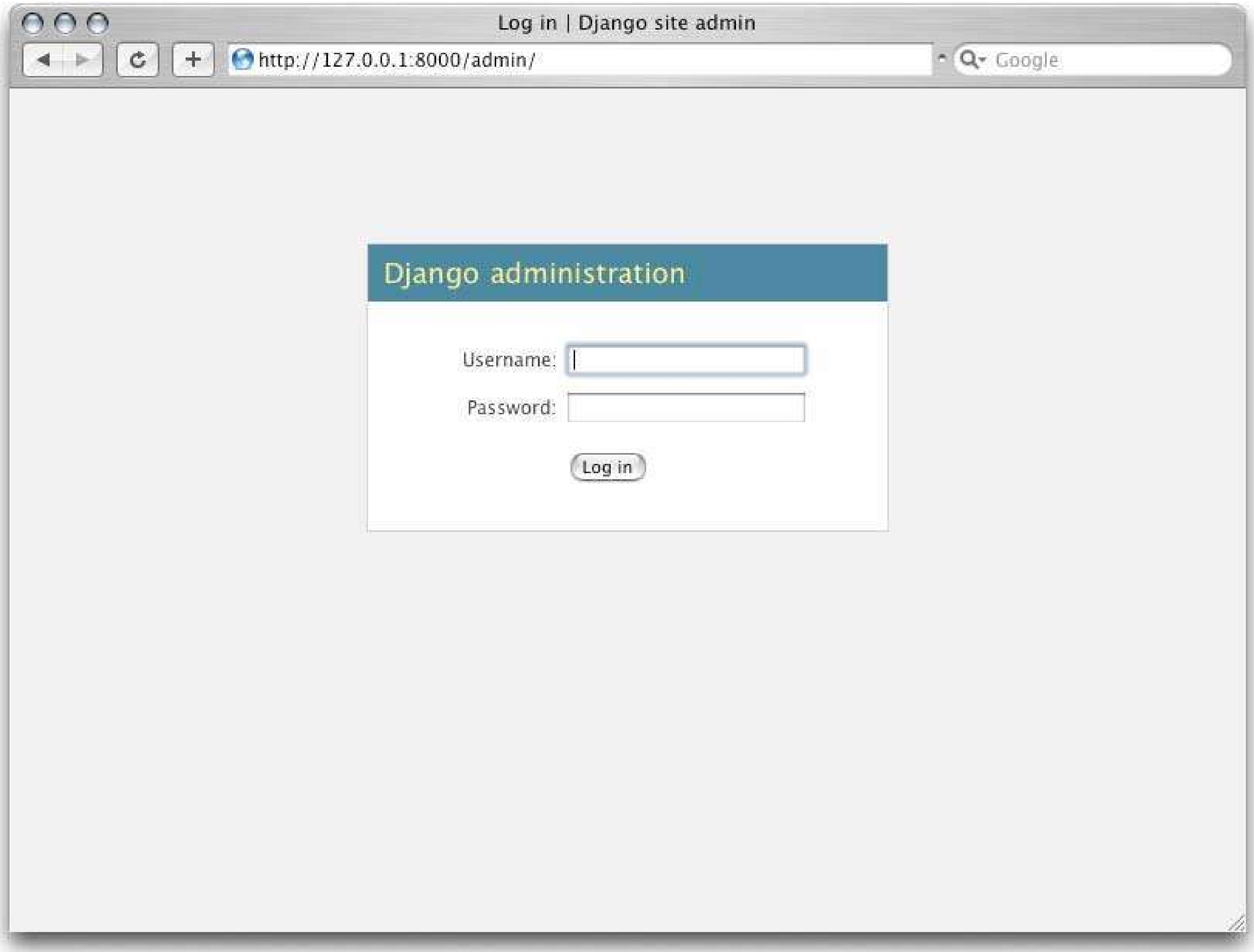
```
Validating models...
0 errors found.

Django version 0.96-pre, using settings 'ch6.settings'
Development server is running at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

Now you can visit the URL given to you by Django (http://127.0.0.1:8000/admin/ in the example above), log in, and play around.

Using the admin interface

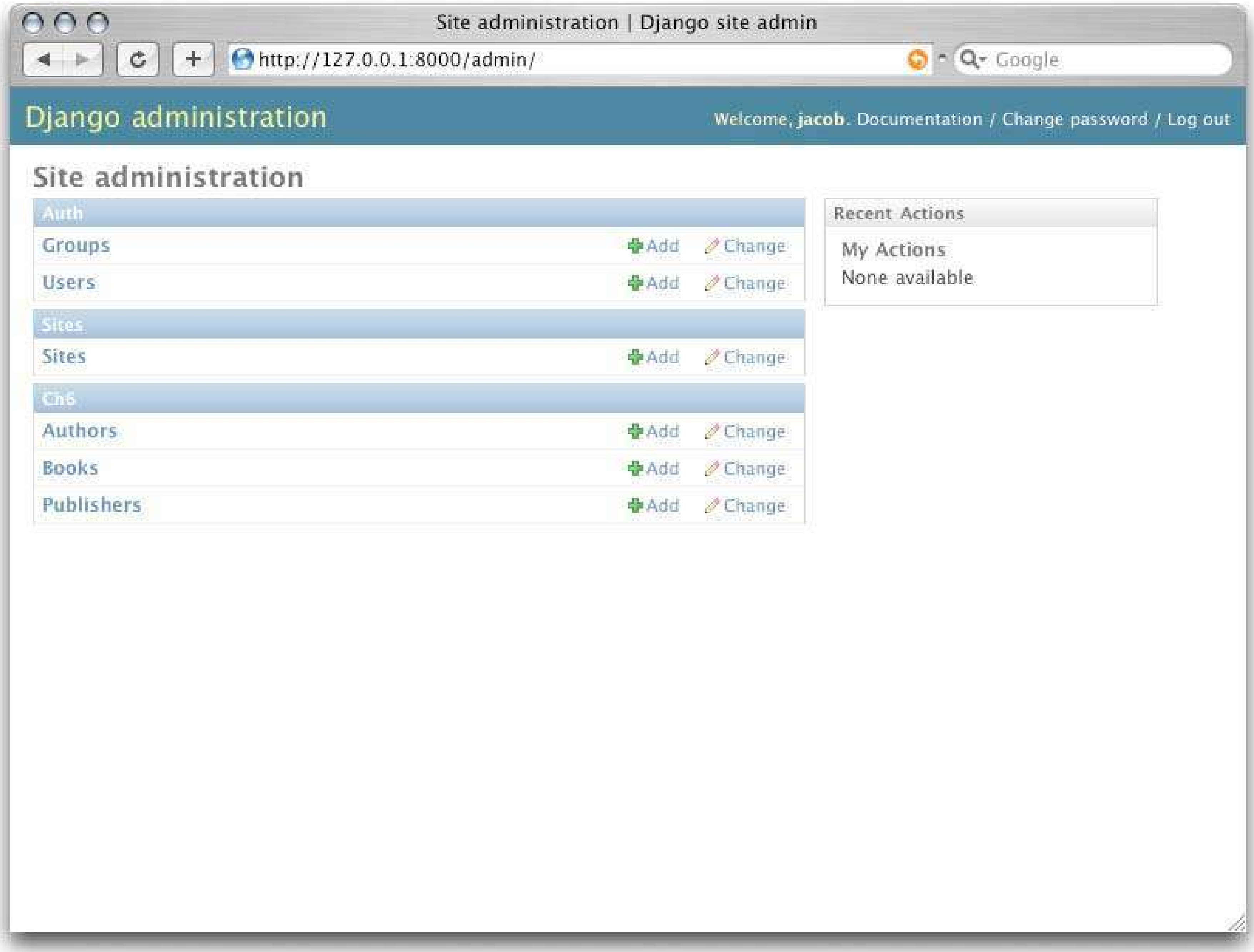
The admin interface is designed to be used by non-technical users, and as such should be pretty self-explanatory. Nevertheless, a few notes about the features of the admin are in order.

The first thing you'll see is a login screen:



You'll use the username and password you set up when you first ran `syncdb` here. Once you've logged in, you'll see that you can manage users, groups, and permissions in the admin; see more on that below.

Each object given an `Admin` declaration shows up on the main index page. Links to add and change objects lead to two pages we refer to as object "change lists" and "edit forms":


Site administration | Django site admin
http://127.0.0.1:8000/admin/
Google

Django administration

Welcome, **jacob**. Documentation / Change password / Log out

Site administration

Auth		
Groups	Add	Change
Users	Add	Change

Sites		
Sites	Add	Change

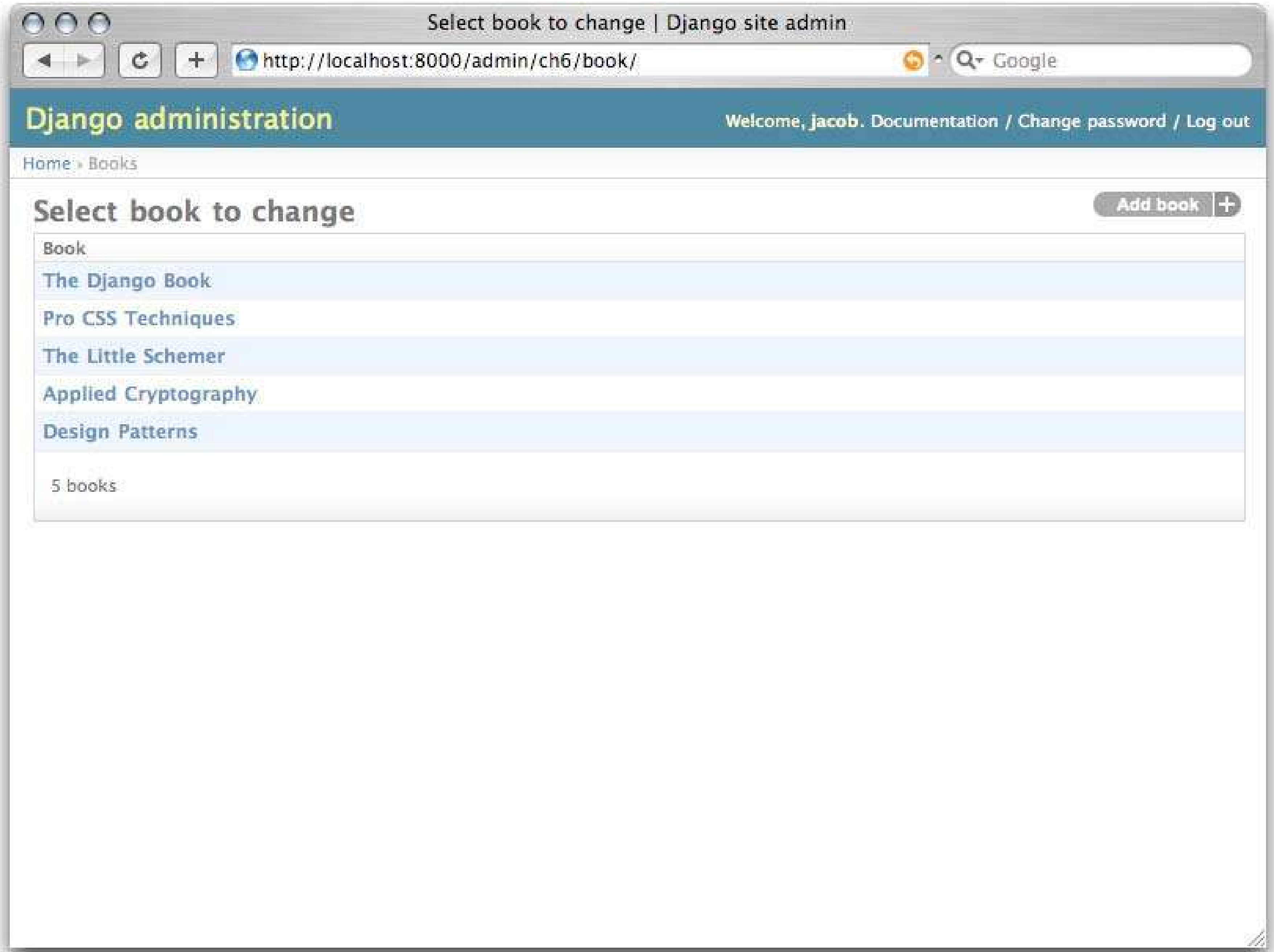
Ch6		
Authors	Add	Change
Books	Add	Change
Publishers	Add	Change

Recent Actions

My Actions

None available

Change lists are essentially index pages of objects in the system:


Select book to change | Django site admin
http://localhost:8000/admin/ch6/book/
Google
Django administration
Welcome, jacob. Documentation / Change password / Log out
Home › Books
Add book

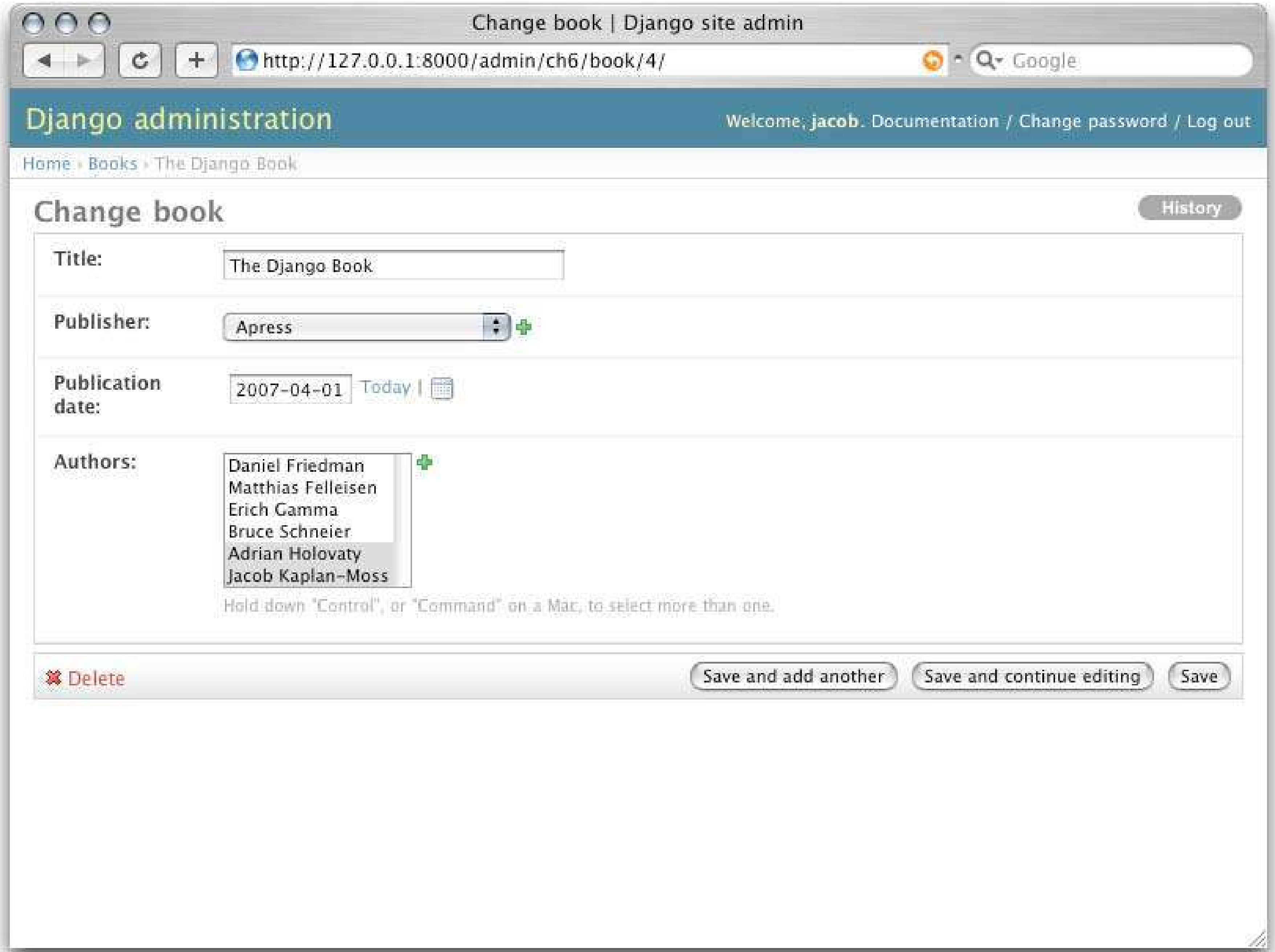
Select book to change

Book
The Django Book
Pro CSS Techniques
The Little Schemer
Applied Cryptography
Design Patterns

5 books

There are a number of options that can control which fields appear on these lists and the appearance of extra features like date drill downs, search fields, and filter interfaces. There's more about these features below.

Edit forms are used to edit existing objects and create new ones. Each field defined in your model appears here, and you'll notice that fields of different types get different widgets (i.e. date/time fields have calendar controls; foreign keys use a select box, etc.):


Change book | Django site admin
http://127.0.0.1:8000/admin/ch6/book/4/
Google
Django administration
Welcome, jacob. Documentation / Change password / Log out
Home › Books › The Django Book
History
Today |
Delete
Save and add another
Save and continue editing
Save

Change book

Title: The Django Book

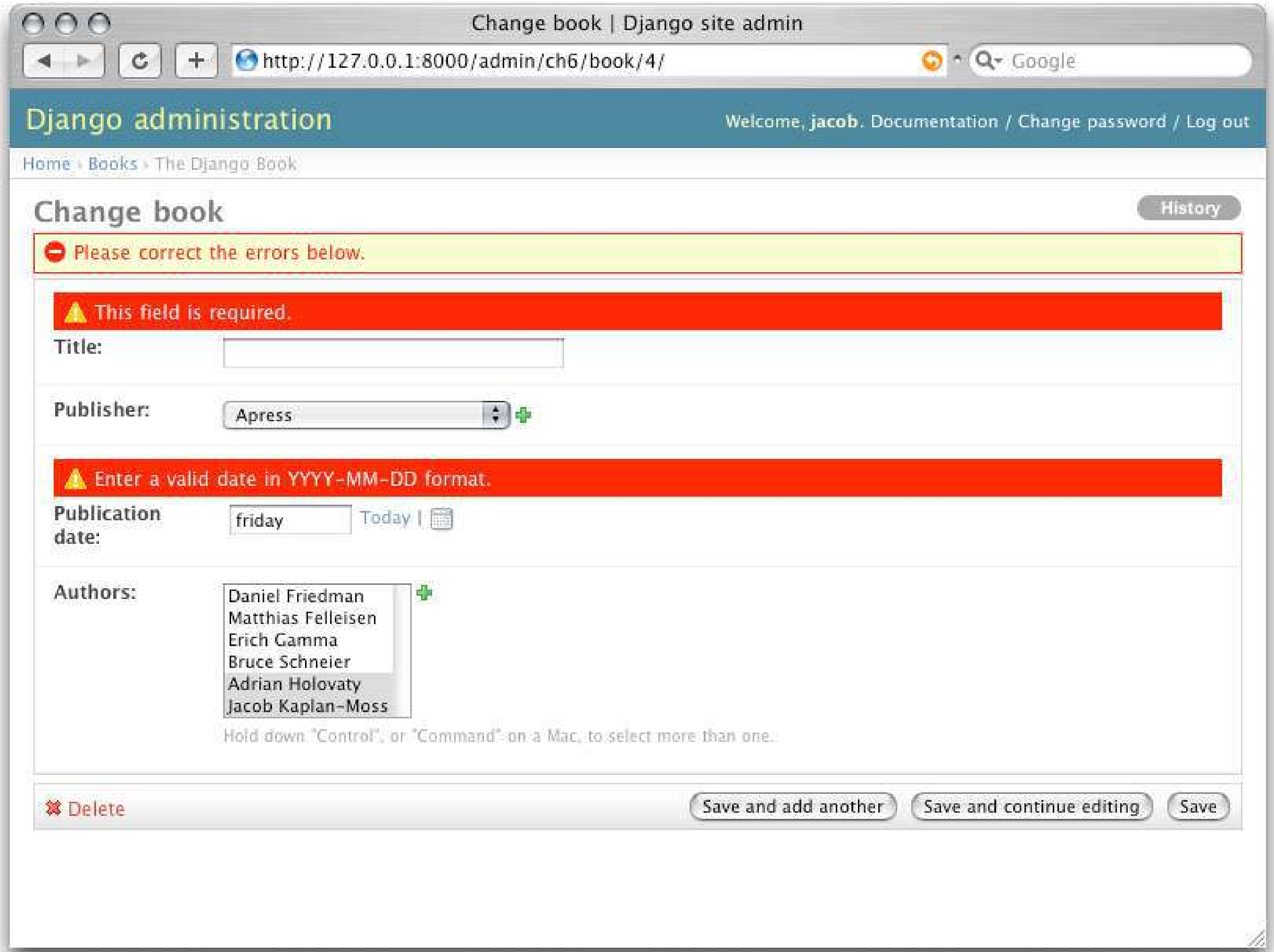
Publisher: Apress

Publication date: 2007-04-01

Authors: Daniel Friedman, Matthias Felleisen, Erich Gamma, Bruce Schneier, Adrian Holovaty, Jacob Kaplan-Moss

Hold down "Control", or "Command" on a Mac, to select more than one.

You'll notice that the admin also handles input validation for you; try leaving a required field blank, or putting an invalid time into a time field and you'll see those errors when you try to save:


Change book | Django site admin
http://127.0.0.1:8000/admin/ch6/book/4/
Google
Django administration
Welcome, jacob. Documentation / Change password / Log out
Home › Books › The Django Book
History
Today |
Delete
Save and add another
Save and continue editing
Save

Change book

Please correct the errors below.

This field is required.

Title:

Publisher: Apress

Enter a valid date in YYYY-MM-DD format.

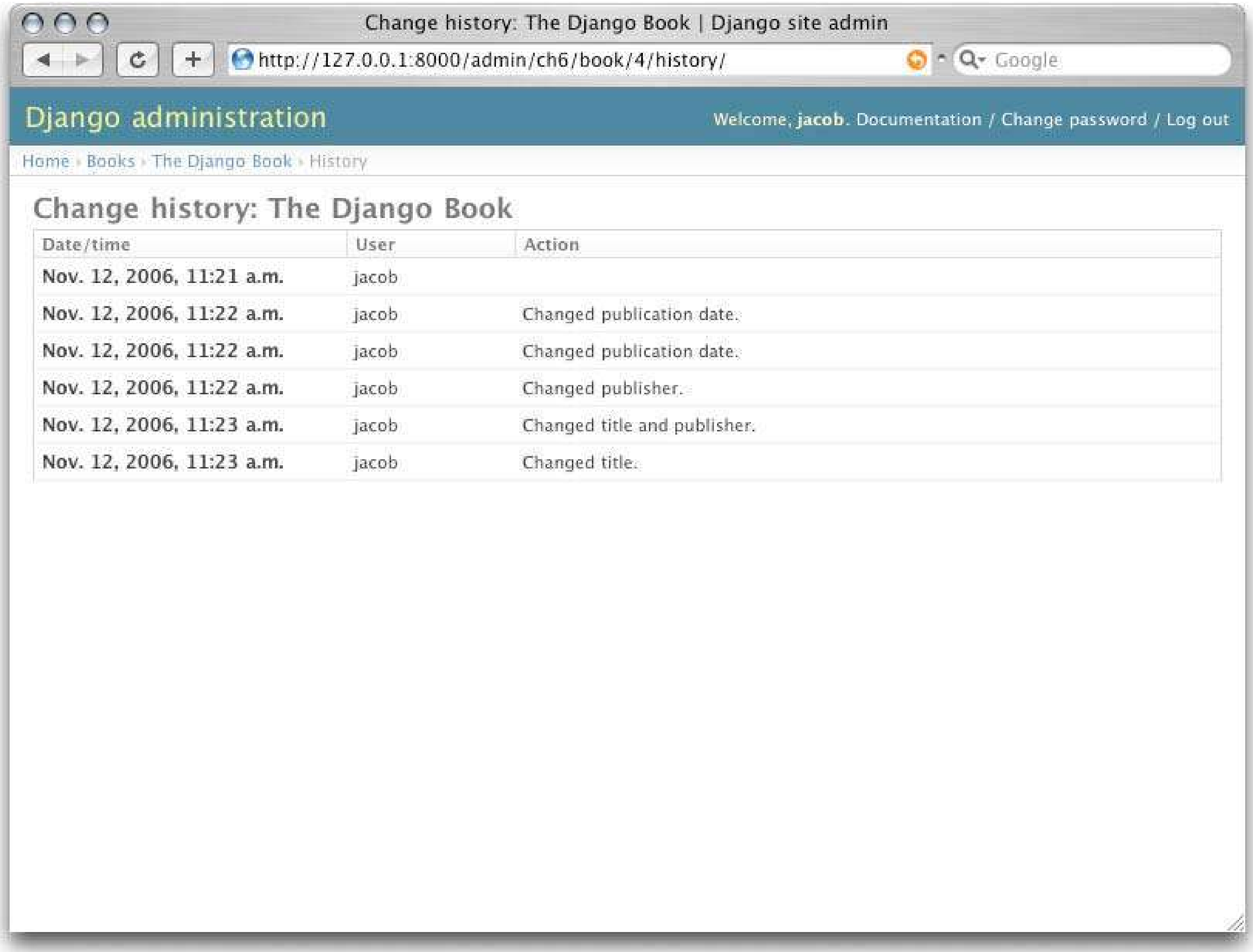
Publication date: friday

Authors: Daniel Friedman, Matthias Felleisen, Erich Gamma, Bruce Schneier, Adrian Holovaty, Jacob Kaplan-Moss

Hold down "Control", or "Command" on a Mac, to select more than one.

This validation actually is done with a powerful validation framework which is discussed in Chapter 7.

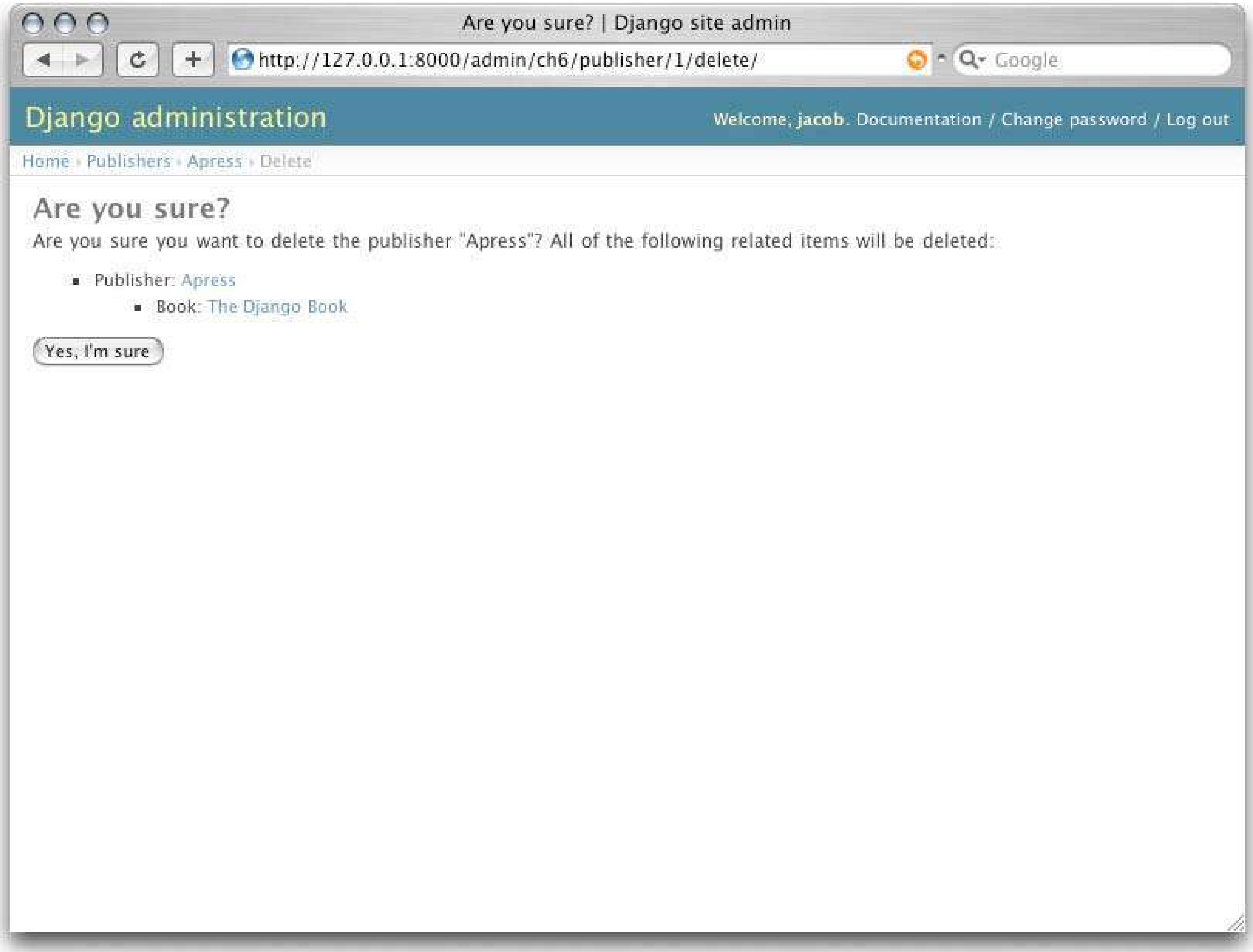
When editing an existing object, you'll notice a "history" link in the upper-right. Every change made through the admin is logged, and you can examine this log by clicking the history button:


Change history: The Django Book | Django site admin
http://127.0.0.1:8000/admin/ch6/book/4/history/
Google
Django administration
Welcome, jacob. Documentation / Change password / Log out
Home › Books › The Django Book › History

Change history: The Django Book

Date/time	User	Action
Nov. 12, 2006, 11:21 a.m.	jacob	
Nov. 12, 2006, 11:22 a.m.	jacob	Changed publication date.
Nov. 12, 2006, 11:22 a.m.	jacob	Changed publication date.
Nov. 12, 2006, 11:22 a.m.	jacob	Changed publisher.
Nov. 12, 2006, 11:23 a.m.	jacob	Changed title and publisher.
Nov. 12, 2006, 11:23 a.m.	jacob	Changed title.

Deletions in the admin cascade. That is, when deleting an existing object, you'll see that the admin asks you to confirm the delete action to avoid costly mistakes. What might not be instantly obvious is that this page will show you all the related objects that will be deleted as well:


Are you sure? | Django site admin
http://127.0.0.1:8000/admin/ch6/publisher/1/delete/
Google
Django administration
Welcome, jacob. Documentation / Change password / Log out
Home › Publishers › Apress › Delete
Yes, I'm sure

Are you sure?

Are you sure you want to delete the publisher "Apress"? All of the following related items will be deleted:

- Publisher: Apress
  - Book: The Django Book

Users, groups, and permissions

Since you're logged in as a superuser, you have access to create, edit, and delete any object. However, the admin has a user permissions system that you can use to give other users access only to the portions of the admin they need.

You edit these users and permissions through the admin just like any other object; the link to the User and Group models is there on the admin index along with all the objects you've defined yourself.

User objects have the standard username, password, email, and real name fields you might expect, along with a set of fields that define what the user is allowed to do in the admin. First, there's a set of three flags:

- The "is active" flag controls whether the user is active at all. If this flag is off, the user has no access to any URLs that require login.
- The "is staff" flag controls whether the user is allowed to log into the admin (i.e. is considered a "staff member" in your organization). Since this same user system can be used to control access to public (i.e. non-admin) sites (see Chapter 12), this flag differentiates between public users and administrators.
- The "is superuser" flag gives the user full unfettered access to every item in the admin; regular permissions are ignored.

For "normal" admin users — active, non-superuser staff members — the access they are granted depends on a set of assigned permissions. Each object editable through the admin has three permissions: a "create" permission, an "edit" permission, and a "delete" permission. Assigning permissions to a user grants the user access to do what is described by those permissions.

Note

Notice that access to edit users and permissions is also controlled by this permission system. If you give a user permission to edit users, she will be able to edit her own permissions, which might not be what you want!

You can also assign users to groups. A group is simply a set of permissions to apply to all members of that group. Groups are extremely useful for granting a large number of users identical permissions.

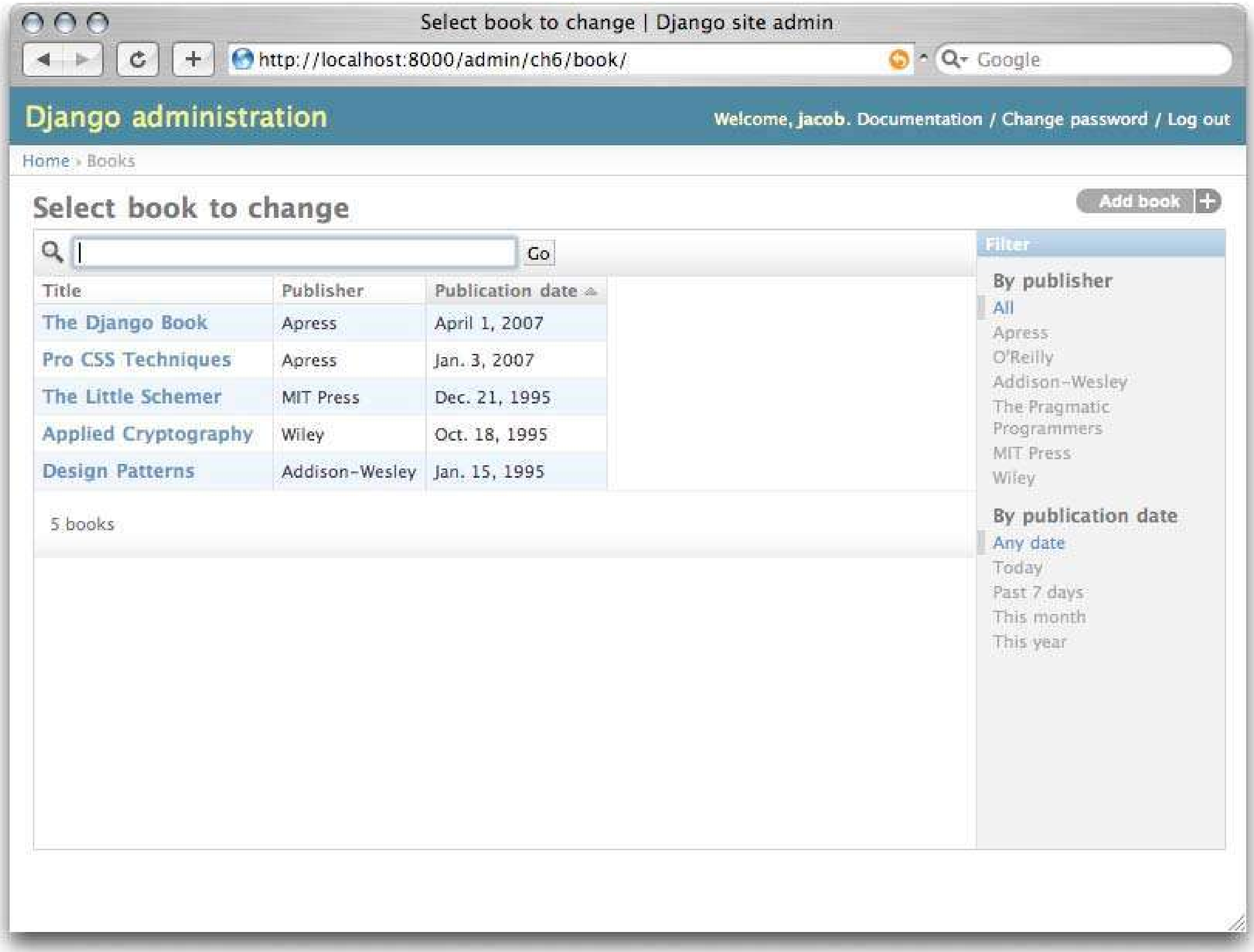
Customizing the admin interface

There are a number of ways to customize the way the admin interface looks and behaves. We'll cover just a few of them below as they relate to our `` Book`` model, but Chapter 12 covers customizing the admin interface in detail.

As it stands now, the change list for our books show only the string representation of the model we added to its `__str__`. This works fine for just a few books, but if we had hundreds or thousands of books, it would be very hard to locate a single needle in the haystack. However, we can easily add some display, searching, and filtering functions to this interface. Change the `Admin` declaration to:

```
class Book(models.Model):
    title = models.CharField(maxlength=100)
    authors = models.ManyToManyField(Author)
    publisher = models.ForeignKey(Publisher)
    publication_date = models.DateField()

    class Admin:
        list_display   = ('title', 'publisher', 'publication_date')
        list_filter    = ('publisher', 'publication_date')
        ordering       = ('-publication_date',)
        search_fields  = ('title',)
```

These four lines of code dramatically change our list interface:


Select book to change | Django site admin
http://localhost:8000/admin/ch6/book/
Google
Django administration
Welcome, jacob. Documentation / Change password / Log out
Home › Books
Add book
Go
Filter

Select book to change

Title	Publisher	Publication date
The Django Book	Apress	April 1, 2007
Pro CSS Techniques	Apress	Jan. 3, 2007
The Little Schemer	MIT Press	Dec. 21, 1995
Applied Cryptography	Wiley	Oct. 18, 1995
Design Patterns	Addison-Wesley	Jan. 15, 1995

5 books

By publisher
- All
- Apress
- O'Reilly
- Addison-Wesley
- The Pragmatic Programmers
- MIT Press
- Wiley

By publication date
- Any date
- Today
- Past 7 days
- This month
- This year

Each of those lines instructed the admin to construct a different piece of this interface:

- The `ordering` option controls what order the objects are presented in the admin. It's simply a list of fields to order the results by; prefixing a field with a minus sign reverses the given order. So in this example, we're ordering by publication date, most recent first.
- The `list_display` option controls which columns appear in the change list table. By default, only a single column with the object's string representation appears; here we've changed that to show the title, publisher, and publication date.
- The `list_filter` option creates the filtering bar on the right side of the list. We've allowed filtering either by date (which allows you to see only books published in the last week, month, etc.), and by publisher.

  You can instruct the admin to filter by any field, but foreign keys or any field with a `choices` attribute set work best.
- Finally, the `search_fields` option creates a field that allows text searches. This allows searches by the `title` field (so you could type "Django" to show all books with "Django" in the title).

Using these options — and the other ones described in Chapter 12 — you can with only a few lines of code make a very powerful, production-ready interface for data editing.

Customize the admin look and feel

Clearly, having "Django administration" at the top of each admin page is ridiculous. It's just placeholder text.

That's easy to change, though, using Django's template system. The Django admin is powered by Django itself, and its interfaces use Django's own template system. (How meta!)

Open your settings file (`mysite/settings.py`, remember) and look at the `TEMPLATE_DIRS` setting. `TEMPLATE_DIRS` is a tuple of filesystem directories to check when loading Django templates. It's a search path.

By default, `TEMPLATE_DIRS` is empty. So, let's add a line to it, to tell Django where our templates live:

```
TEMPLATE_DIRS = (
    "/home/mytemplates", # Change this to your own directory.
)
```

Note

Make sure to include the trailing comma there — Python uses it to distinguish between single-element tuples and parenthesized expressions.

Now copy the template `admin/base_site.html` from within the default Django admin template directory (`django/contrib/admin/templates`) into an `admin` subdirectory of whichever directory you're using in `TEMPLATE_DIRS`. For example, if your `TEMPLATE_DIRS` includes `"/home/mytemplates"`, as above, then copy `django/contrib/admin/templates/admin/base_site.html` to `/home/mytemplates/admin/base_site.html`. Don't forget that `admin` subdirectory.

Then, just edit the new `admin/base_site.html` file to replace the generic Django text with your own site's name as you see fit.

Note that any of Django's default admin templates can be overridden. To override a template, just do the same thing you did with `base_site.html` — copy it from the default directory into your custom directory and make changes to the copy.

Astute readers wonder how, if `TEMPLATE_DIRS` was empty by default, Django was finding the default admin templates? The answer is that, by default, Django automatically looks for templates within a `templates/` subdirectory in each app package as a fallback. See "Template loaders" in Chapter 10 for more information about how this works.

Customize the admin index page

On a similar note, you might want to customize the look and feel of the Django admin index page.

By default, it displays all available apps, according to your `INSTALLED_APPS` setting, sorted by the name of the application. You might, however, want to change this order to make it easier to find the apps you're looking for. After all, the index is probably the most important page of the admin, so it should be easy to use.

The template to customize is `admin/index.html`. (Remember to copy `admin/base_site.html` to your custom template directory as in the previous example.) Edit the file, and you'll see it uses a template tag called `{% get_admin_app_list as app_list %}`. That's the magic that retrieves every installed Django app. Instead of using that, you can hard-code links to object-specific admin pages in whatever way you think is best. If hard-coding links doesn't appeal to you, you can also see Chapter 10 for details on implementing your own template tags.

Django offers another shortcut in this department. Run the command `python manage.py adminindex <app>` to get a chunk of template code for inclusion in the admin index template. It's a useful starting point.

For full details on customizing the look and feel of the Django admin site in general, see Chapter 12.

When and why to use the admin interface

We think Django's admin interface is pretty spectacular. In fact, we'd call it one of Django's "killer features". However, we often get asked questions about "use cases" for the admin — when do *we* use it, and why? Over the years, we've discovered a number of patterns for using the admin interface that we think might be helpful.

Obviously, it's extremely useful for editing data (fancy that). If you have any sort of data entry tasks, the admin simply can't be beat. We suspect that the vast majority of readers of this book will have a whole host of data entry tasks.

Django's admin especially shines when non-technical users need to be able to enter data; that's the original genesis of the feature. At the newspaper where Django was first developed, development of a typical online feature — a special report on water quality in the municipal supply, say — goes something like this:

- The reporter responsible for the story meets with one of the developers and goes over the available data.
- The developer designs a model around this data, and then opens up the admin interface to the reporter.
- While the reporter enters data into Django, the programmer can focus on developing the publicly-accessible interface (the fun part!)

In other works, the raison d'être of Django's admin is facilitating the simultaneous work of content producers and programmers.

However, beyond the obvious data-entry tasks, we find the admin useful in a few other cases:

- Inspecting data models: the first thing we do when we've defined a new model is to call it up in the admin and enter some dummy data. This is usually when we find any data modeling errors; having a graphical interface to a model quickly reveals those mistakes.
- Managing acquired data: there's little actual data entry associated with a site like chicagocrime.org since most of the data comes from an automated source. However, when problems with the automatically acquired data crop up, it's very useful to be able to go in and edit that data easily.

What's next?

So far we've created a few models and configured a top-notch interface for editing that data. In the next chapter, we'll move onto the real "meat and potatoes" of Web development: form creation and processing.

So grab another cup of your favorite beverage and let's get started.

« previous ◊ table of contents ◊ next »

Chapter 8: Advanced views and URLconfs

In Chapter 3, we explained the basics of Django view functions and URLconfs. This chapter goes into more detail about advanced functionality in those two pieces of the framework.

URLconf tricks

Streamlining function imports

Consider this URLconf, which builds on the example in Chapter 3:

```
from django.conf.urls.defaults import *
from mysite.views import current_datetime, hours_ahead, hours_behind, now_in_chicago,
now_in_london

urlpatterns = patterns('',
    (r'^now/$', current_datetime),
    (r'^now/plus(\d{1,2})hours/$', hours_ahead),
    (r'^now/minus(\d{1,2})hours/$', hours_behind),
    (r'^now/in_chicago/$', now_in_chicago),
    (r'^now/in_london/$', now_in_london),
)
```

As explained in Chapter 3, each entry in the URLconf includes its associated view function, passed directly as a function object. This means it's necessary to import the view functions at the top of the module.

But as a Django application grows in complexity, its URLconf grows, too, and keeping those imports can be tedious to manage. (For each new view function, you've got to remember to import it, and the import statement tends to get overly long if you use this approach.) It's possible to avoid that tedium by importing the `views` module itself. This example URLconf is equivalent to the previous one:

```
from django.conf.urls.defaults import *
from mysite import views

urlpatterns = patterns('',
    (r'^now/$', views.current_datetime),
    (r'^now/plus(\d{1,2})hours/$', views.hours_ahead),
    (r'^now/minus(\d{1,2})hours/$', views.hours_behind),
    (r'^now/in_chicago/$', views.now_in_chicago),
    (r'^now/in_london/$', views.now_in_london),
)
```

Django offers another way of specifying the view function for a particular pattern in the URLconf: You can pass a string containing the module name and function name rather than the function object itself. Continuing the ongoing example:

```
from django.conf.urls.defaults import *

urlpatterns = patterns('',
    (r'^now/$', 'mysite.views.current_datetime'),
    (r'^now/plus(\d{1,2})hours/$', 'mysite.views.hours_ahead'),
    (r'^now/minus(\d{1,2})hours/$', 'mysite.views.hours_behind'),
    (r'^now/in_chicago/$', 'mysite.views.now_in_chicago'),
    (r'^now/in_london/$', 'mysite.views.now_in_london'),
)
```

Using this technique, it's no longer necessary to import the view functions; Django automatically imports the appropriate view function the first time it's needed, according to the string describing the name and path of the view function.

A further shortcut you can take when using the string technique is to factor out a common "view prefix." In our URLconf example, each of the view strings starts with `'mysite.views'`, which is redundant to type. We can factor out that common prefix and pass it as the first argument to `patterns()`, like this:

```
from django.conf.urls.defaults import *
```

```
urlpatterns = patterns('mysite.views',
    (r'^now/$', 'current_datetime'),
    (r'^now/plus(\d{1,2})hours/$', 'hours_ahead'),
    (r'^now/minus(\d{1,2})hours/$', 'hours_behind'),
    (r'^now/in_chicago/$', 'now_in_chicago'),
    (r'^now/in_london/$', 'now_in_london'),
)
```

Note that you don't put a trailing dot (`"."`) in the prefix, nor do you put a leading dot in the view strings. Django puts that in automatically.

With these two approaches in mind, which is better? It really depends on your personal coding style and needs.

Advantages of the string approach are:

- It's more compact, because it doesn't require you to import the view functions.
- It results in more readable and manageable URLconfs if your view functions are spread across several different Python modules.

Advantages of the function object approach are:

- It allows for easy "wrapping" of view functions. See "Wrapping view functions" later in this chapter.
- It's more "Pythonic" — that is, it's more in line with Python traditions, such as passing functions as objects.

Both approaches are valid, and you can even mix them within the same URLconf. The choice is yours.

Multiple view prefixes

In practice, if you use the string technique, you'll probably end up mixing views to the point where the views in your URLconf won't have a common prefix. However, you can still take advantage of the view prefix shortcut to remove duplication. Just add multiple `patterns()` objects together, like this:

Old:

```
from django.conf.urls.defaults import *

urlpatterns = patterns('',
    (r'^/?$', 'mysite.views.archive_index'),
    (r'^(\d{4})/([a-z]{3})/$', 'mysite.views.archive_month'),
    (r'^tag/(\w+)/$', 'weblog.views.tag'),
)
```

New:

```
from django.conf.urls.defaults import *

urlpatterns = patterns('mysite.views',
    (r'^/?$', 'archive_index'),
    (r'^(\d{4})/([a-z]{3})/$','archive_month'),
)

urlpatterns += patterns('weblog.views',
    (r'^tag/(\w+)/$', 'tag'),
)
```

All the framework cares about is that there's a module-level variable called `urlpatterns`. This variable can be constructed dynamically, as we do in this example.

Named groups

In all of our URLconf examples so far, we've used simple, *non-named* regular-expression groups — i.e., we put parentheses around parts of the URL we wanted to capture, and Django passes that captured text to the view function as a positional argument. In more advanced usage, it's possible to use *named* regular-expression groups to capture URL bits and pass them as *keyword* arguments to a view.

Keyword arguments vs. positional arguments

A Python function can be called using keyword arguments or positional arguments — and, in some cases, both at the same time. In a keyword argument call, you specify the names of the arguments along with the values you're passing. In a positional argument call, you simply pass the arguments without explicitly specifying which argument matches which value; the association is implicit in the arguments' order.

For example, consider this simple function:

```
def sell(item, price, quantity):
    print "Selling %s unit(s) of %s at %s" % (quantity, item, price)
```

To call it with positional arguments, you specify the arguments in the order in which they're listed in the function definition:

```
sell('Socks', '$2.50', 6)
```

To call it with keyword arguments, you specify the names of the arguments along with the values. The following statements are equivalent:

```
sell(item='Socks', price='$2.50', quantity=6)
sell(item='Socks', quantity=6, price='$2.50')
sell(price='$2.50', item='Socks', quantity=6)
sell(price='$2.50', quantity=6, item='Socks')
sell(quantity=6, item='Socks', price='$2.50')
sell(quantity=6, price='$2.50', item='Socks')
```

In Python regular expressions, the syntax for named regular-expression groups is `(?P<name>pattern)`, where `name` is the name of the group and `pattern` is some pattern to match.

Here's an example URLconf that uses non-named groups:

```
from django.conf.urls.defaults import *
from mysite import views

urlpatterns = patterns('',
    (r'^articles/(\d{4})/$', views.year_archive),
    (r'^articles/(\d{4})/(\d{2})/$', views.month_archive),
)
```

Here's the same URLconf, rewritten to use named groups:

```
from django.conf.urls.defaults import *
from mysite import views

urlpatterns = patterns('',
    (r'^articles/(?P<year>\d{4})/$', views.year_archive),
    (r'^articles/(?P<year>\d{4})/(?P<month>\d{2})/$', views.month_archive),
)
```

This accomplishes exactly the same thing as the previous example, with one subtle difference: The captured values are passed to view functions as keyword arguments rather than positional arguments.

For example, with non-named groups, a request to `/articles/2006/03/` would result in a function call equivalent to this:

```
month_archive(request, '2006', '03')
```

With named groups, though, the same request would result in this function call:

```
month_archive(request, year='2006', month='03')
```

In practice, using named groups makes your URLconfs slightly more explicit and less prone to argument-order bugs — and you can reorder the arguments in your views' function definitions. Following the above example, if we wanted to change the URLs to include the month *before* the year, and we were using non-named groups, we'd have to remember to change the order of arguments in the `month_archive` view. If we were using named groups, changing the order of the captured parameters in the URL would have no effect on the view.

Of course, the benefits of named groups come at the cost of brevity; some developers find the named-group syntax ugly and too verbose.

The matching/grouping algorithm

If you use both named and non-named groups in the same pattern in your URLconf, you should be aware of how Django treats this special case. Here's the algorithm the URLconf parser follows, with respect to named groups vs. non-named groups in a regular expression:

- If there are any named arguments, it will use those, ignoring non-named arguments.
- Otherwise, it will pass all non-named arguments as positional arguments.
- In both cases, it will pass any extra keyword arguments as keyword arguments. See “Passing extra options to view functions” below.

Passing extra options to view functions

Sometimes you’ll find yourself writing view functions that are quite similar, with only a few small differences. For example, say you’ve got two views whose contents are identical except for the template they use:

```
# urls.py

from django.conf.urls.defaults import *
from mysite import views

urlpatterns = patterns('',
    (r'^foo/$', views.foo_view),
    (r'^bar/$', views.bar_view),
)

# views.py

from django.shortcuts import render_to_response
from mysite.models import MyModel

def foo_view(request):
    m_list = MyModel.objects.filter(is_new=True)
    return render_to_response('template1.html', {'m_list': m_list})

def bar_view(request):
    m_list = MyModel.objects.filter(is_new=True)
    return render_to_response('template2.html', {'m_list': m_list})
```

We’re repeating ourselves in this code, and that’s inelegant. At first, you may think to remove the redundancy by using the same view for both URLs, putting parenthesis around the URL to capture it, and checking the URL within the view to determine the template, like so:

```
# urls.py

from django.conf.urls.defaults import *
from mysite import views

urlpatterns = patterns('',
    (r'^(foo)/$', views.foobar_view),
    (r'^(bar)/$', views.foobar_view),
)

# views.py

from django.shortcuts import render_to_response
from mysite.models import MyModel

def foobar_view(request, url):
    m_list = MyModel.objects.filter(is_new=True)
    if url == 'foo':
        template_name = 'template1.html'
    elif url == 'bar':
        template_name = 'template2.html'
    return render_to_response(template_name, {'m_list': m_list})
```

The problem with that solution, though, is that it couples your URLs to your code. If you decide to rename `/foo/` to `/fooey/`, you’ll have to remember to change the view code.

The elegant solution involves a feature called extra URLconf options. Each pattern in a URLconf may include a third item — a dictionary of keyword arguments to pass to the view function.

With this in mind, we can rewrite our ongoing example like this:

```
# urls.py

from django.conf.urls.defaults import *
from mysite import views

urlpatterns = patterns('',
    (r'^foo/$', views.foobar_view, {'template_name': 'template1.html'}),
    (r'^bar/$', views.foobar_view, {'template_name': 'template2.html'}),
)

# views.py

from django.shortcuts import render_to_response
from mysite.models import MyModel

def foobar_view(request, template_name):
    m_list = MyModel.objects.filter(is_new=True)
    return render_to_response(template_name, {'m_list': m_list})
```

As you can see, the URLconf in this example specifies `template_name` in the URLconf. The view function treats it as just another parameter.

This extra URLconf options technique is a nice way of sending additional information to your view functions with minimal fuss. As such, it's used by a couple of Django's bundled applications, most notably its generic views system, which we'll cover in Chapter 9.

Here are a couple of ideas on how you can use the extra URLconf options technique in your own projects:

Faking captured URLconf values

Say you've got a set of views that match a pattern, along with another URL that doesn't fit the pattern but whose view logic is the same. In this case, you can "fake" the capturing of URL values by using extra URLconf options to handle that extra URL with the same view.

For example, you might have an application that displays some data for a particular day, with URLs such as this:

```
/mydata/jan/01/
/mydata/jan/02/
/mydata/jan/03/
# ...
/mydata/dec/30/
/mydata/dec/31/
```

This is simple enough to deal with; you can capture those in a URLconf like this (using named group syntax):

```
urlpatterns = patterns('',
    (r'^mydata/(?P<month>\w{3})/(?P<day>\d\d)/$', views.my_view),
)
```

And the view function signature would look like this:

```
def my_view(request, month, day):
    # ....
```

This is straightforward — it's nothing we haven't seen before. The trick comes in when you want to add another URL that uses `my_view` but whose URL doesn't include a `month` and/or `day`.

For example, you might want to add another URL, `/mydata/birthday/`, which would be equivalent to `/mydata/jan/06/`. We can take advantage of extra URLconf options like so:

```
urlpatterns = patterns('',
    (r'^mydata/birthday/$', views.my_view, {'month': 'jan', 'day': '06'}),
    (r'^mydata/(?P<month>\w{3})/(?P<day>\d\d)/$', views.my_view),
)
```

The cool thing here is that we don't have to change our view function at all. The view function only cares that it *gets* `month` and `day` parameters — it doesn't matter whether they come from the URL capturing itself or extra parameters.

Making a view generic

It's good programming practice to "factor out" commonalities in code. For example, with these two Python functions:

```
def say_hello(person_name):
    print 'Hello, %s' % person_name
```

```
def say_goodbye(person_name):
    print 'Goodbye, %s' % person_name
```

...we can factor out the greeting to make it a parameter:

```
def greet(person_name, greeting):
    print '%s, %s' % (greeting, person_name)
```

You can apply this same philosophy to your Django views by using extra URLconf parameters.

With this in mind, you can start making higher-level abstractions of your views. Instead of thinking to yourself, “This view displays a list of `Event` objects,” and “That view displays a list of `BlogEntry` objects,” realize they’re both specific cases of “A view that displays a list of objects, where the type of object is variable.”

Take this code, for example:

```
# urls.py

from django.conf.urls.defaults import *
from mysite import views

urlpatterns = patterns('',
    (r'^events/$', views.event_list),
    (r'^blog/entries/$', views.entry_list),
)

# views.py

from django.shortcuts import render_to_response
from mysite.models import Event, BlogEntry

def event_list(request):
    obj_list = Event.objects.all()
    return render_to_response('mysite/event_list.html', {'event_list': obj_list})

def entry_list(request):
    obj_list = BlogEntry.objects.all()
    return render_to_response('mysite/blogentry_list.html', {'entry_list': obj_list})
```

The two views do essentially the same thing: they display a list of objects. So let’s factor out the type of object they’re displaying:

```
# urls.py

from django.conf.urls.defaults import *
from mysite import models, views

urlpatterns = patterns('',
    (r'^events/$', views.object_list, {'model': models.Event}),
    (r'^blog/entries/$', views.object_list, {'model': models.BlogEntry}),
)

# views.py

from django.shortcuts import render_to_response

def object_list(request, model):
    obj_list = model.objects.all()
    template_name = 'mysite/%s_list.html' % model.__name__.lower()
    return render_to_response(template_name, {'object_list': obj_list})
```

With those small changes, we suddenly have a reusable, model-agnostic view! From now on, any time we need a view that lists a set of objects, we can simply reuse this `object_list` view rather than writing view code. Here are a couple of notes about what we did:

- We’re passing the model classes directly, as the `model` parameter. The dictionary of extra URLconf options can pass any type of Python object — not just strings.
- The `model.objects.all()` line is an example of *duck typing*: “If it walks like a duck and talks like a duck, we can treat it like a duck.” Note the code doesn’t know what type of object `model` is; the only requirement is that `model` have an `objects` attribute, which in turn has an `all()` method.

- We're using `model.__name__.lower()` in determining the template name. Every Python class has a `__name__` attribute that returns the class name. This feature is useful at times like these, when we don't know the type of class until runtime.

  For example, the `BlogEntry` class' `__name__` is the string `'BlogEntry'`.

- In a slight difference between this example and the previous example, we're passing the generic variable name `object_list` to the template. We could easily change this variable name to be `blogentry_list` or `event_list`, but we've left that as an exercise for the reader.

Because database-driven Web sites have several common patterns, Django comes with a set of "generic views" that use this exact technique to save you time. We'll cover Django's built-in generic views in the next chapter.

Giving a view configuration options

If you're distributing a Django application, chances are that your users will want some degree of configuration. In this case, it's a good idea to add hooks to your views for any configuration options you think people may want to change. You can use extra URLconf parameters for this purpose.

A common bit of an application to make configurable is the template name:

```
def my_view(request, template_name):
    var = do_something()
    return render_to_response(template_name, {'var': var})
```

Precedence of captured values vs. extra options

When there's a conflict, extra URLconf parameters get precedence over captured parameters. In other words, if your URLconf captures a named-group variable and an extra URLconf parameter includes a variable with the same name, the extra URLconf parameter value will be used.

For example, consider this URLconf:

```
from django.conf.urls.defaults import *

urlpatterns = patterns('',
    (r'^mydata/(?P<id>\d+)/$', views.my_view, {'id': 3}),
)
```

Here, both the regular expression and the extra dictionary include an `id`. The hard-coded `id` gets precedence. That means any request — e.g., `/mydata/2/` or `/mydata/432432/` — will be treated as if `id` is set to `3`, regardless of the value captured in the URL.

Astute readers will note that in this case, it's a waste of time and typing to capture the `id` in the regular expression, because its value will always be overridden by the dictionary's value. Those astute readers would be correct. We bring this up only to help you avoid making the mistake.

Using default view arguments

Another convenient trick is to specify default parameters for a view's arguments. This tells the view which value to use for a parameter by default if none is specified.

For example:

```
# urls.py

from django.conf.urls.defaults import *

urlpatterns = patterns('',
    (r'^blog/$', views.page),
    (r'^blog/page(?P<num>\d+)/$', views.page),
)

# views.py

def page(request, num="1"):
    # Output the appropriate page of blog entries, according to num.
    # ...
```

Here, both URL patterns point to the same view — `views.page` — but the first pattern doesn't capture anything from the URL. If the first pattern matches, the `page()` function will use its default argument for `num`, `"1"`. If the second pattern matches, `page()` will use whatever `num` value was captured by the regex.

It's common to use this technique in conjunction with configuration options, as explained above. This example makes a slight improvement to the example in the Giving a view configuration options section by providing a default value for `template_name`:

```
def my_view(request, template_name='mysite/my_view.html'):
    var = do_something()
    return render_to_response(template_name, {'var': var})
```

Special-casing views

Sometimes you'll have a pattern in your URLconf that handles a large set of URLs but you'll need to special-case one of them. In this case, take advantage of the linear way a URLconf is processed and put the special case first.

For example, the "add an object" pages in Django's admin site are represented by this URLconf line:

```
urlpatterns = patterns('',
    # ...
    ('^([^/]+)/([^/]+)/add/$', 'django.contrib.admin.views.main.add_stage'),
    # ...
)
```

This matches URLs such as `/myblog/entries/add/` and `/auth/groups/add/`. However, the "add" page for a user object (`/auth/user/add/`) is a special case — it doesn't display all of the form fields, it displays two password fields, etc. We *could* solve this by special-casing in the view, like so:

```
def add_stage(request, app_label, model_name):
    if app_label == 'auth' and model_name == 'user':
        # do special-case code
    else:
        # do normal code
```

..but that's inelegant for a reason we've touched on multiple times in this chapter: it puts URL logic in the view. As a more elegant solution, we can take advantage of the fact that URLconfs are processed in order from top to bottom:

```
urlpatterns = patterns('',
    # ...
    ('^auth/user/add/$', 'django.contrib.admin.views.auth.user_add_stage'),
    ('^([^/]+)/([^/]+)/add/$', 'django.contrib.admin.views.main.add_stage'),
    # ...
)
```

With this in place, a request to `/auth/user/add/` will be handled by the `user_add_stage` view. Although that URL matches the second pattern, it matches the top one first. (This is short-circuit logic.)

Notes on capturing text in URLs

Each captured argument is sent to the view as a plain Python string, regardless of what sort of match the regular expression makes. For example, in this URLconf line:

```
(r'^articles/(?P<year>\d{4})/$', views.year_archive),
```

..the `year` argument to `views.year_archive()` will be a string, not an integer, even though the `\d{4}` will only match integer strings.

This is important to keep in mind when you're writing view code. Many built-in Python functions are fussy (and rightfully so) about accepting only objects of a certain type. A common error is to attempt to create a `datetime.date` object with string values instead of integer values:

```
>>> import datetime
>>> datetime.date('1993', '7', '9')
Traceback (most recent call last):
    ...
TypeError: an integer is required
>>> datetime.date(1993, 7, 9)
datetime.date(1993, 7, 9)
```

Translated to a URLconf and view, the error looks like this:

```
# urls.py

from django.conf.urls.defaults import *
```

```
urlpatterns = patterns('',
    (r'^articles/(\d{4})/(\d{2})/(\d{2})/$', views.day_archive),
)

# views.py

import datetime

def day_archive(request, year, month, day)
    # The following statement raises a TypeError!
    date = datetime.date(year, month, day)
```

Instead, `day_archive()` can be written correctly like this:

```
def day_archive(request, year, month, day)
    date = datetime.date(int(year), int(month), int(day))
```

Note that `int()` itself raises a `ValueError` when you pass it a string that is not comprised solely of digits, but we're avoiding that error in this case because the regular expression in our URLconf has ensured that only strings containing digits are passed to the view function.

What the URLconf searches against

When a request comes in, Django tries to match the URLconf patterns against the requested URL, as a normal Python string (not as a Unicode string). This does not include GET or POST parameters, or the domain name. It also does not include the leading slash, because every URL has a leading slash.

For example, in a request to `http://www.example.com/myapp/`, Django will try to match `myapp/`.

In a request to `http://www.example.com/myapp/?page=3`, Django will try to match `myapp/`.

The request method — e.g., `POST`, `GET`, `HEAD` — is *not* taken into account when traversing the URLconf. In other words, all request methods will be routed to the same function for the same URL. It's the responsibility of a view function to perform branching based on request method.

Including other URLconfs

At any point, your URLconf can "include" other URLconf modules. This essentially "roots" a set of URLs below other ones.

For example, this URLconf includes other URLconfs:

```
from django.conf.urls.defaults import *

urlpatterns = patterns('',
    (r'^weblog/', include('mysite.blog.urls')),
    (r'^photos/', include('mysite.photos.urls')),
    (r'^about/$', 'mysite.views.about'),
)
```

There's an important gotcha here: The regular expressions in this example that point to an `include()` do *not* have a `$` (end-of-string match character) but *do* include a trailing slash. Whenever Django encounters `include()`, it chops off whatever part of the URL matched up to that point and sends the remaining string to the included URLconf for further processing.

Continuing this example, here's the URLconf `mysite.blog.urls`:

```
from django.conf.urls.defaults import *

urlpatterns = patterns('',
    (r'^(\d\d\d\d)/$', 'mysite.blog.views.year_detail'),
    (r'^(\d\d\d\d)/(\d\d)/$', 'mysite.blog.views.month_detail'),
)
```

With these two URLconfs, here's how a few sample requests would be handled:

- `/weblog/2007/` — In the first URLconf, the pattern `r'^weblog/'` matches. Because it is an `include()`, Django strips all the matching text, which is `'weblog/'` in this case. The remaining part of the URL is `2007/`, which matches the first line in the `mysite.blog.urls` URLconf.
- `/weblog//2007/` — In the first URLconf, the pattern `r'^weblog/'` matches. Because it is an `include()`, Django strips all the matching text, which is `'weblog/'` in this case. The remaining part of the URL is `/2007/` (with a leading slash), which does not match any of the lines in the `mysite.blog.urls` URLconf.

- `/about/` — Matches the view `mysite.views.about` in the first URLconf. This demonstrates that you can mix `include()` patterns with non-`include()` patterns.

How captured parameters work with `include()`

An included URLconf receives any captured parameters from parent URLconfs. For example:

```
# root urls.py

from django.conf.urls.defaults import *

urlpatterns = patterns('',
    (r'^(?P<username>\w+)/blog/', include('foo.urls.blog')),
)

# foo/urls/blog.py

from django.conf.urls.defaults import *

urlpatterns = patterns('',
    (r'^$', 'foo.views.blog_index'),
    (r'^archive/$', 'foo.views.blog_archive'),
)
```

In this example, the captured `username` variable is passed to the included URLconf and, hence, to *every* view function within that URLconf.

Note that the captured parameters will *always* be passed to *every* line in the included URLconf, regardless of whether the line's view actually accepts those parameters as valid. For this reason, this technique is only useful if you're certain that every view in the the included URLconf accepts the parameters you're passing.

How extra URLconf options work with `include()`

Similarly, you can pass extra URLconf options to `include()`, just as you can pass extra URLconf options to a normal view — as a dictionary. When you do this, *each* line in the included URLconf will be passed the extra options.

For example, these two URLconf sets are functionally identical:

Set one:

```
# urls.py

from django.conf.urls.defaults import *

urlpatterns = patterns('',
    (r'^blog/', include('inner'), {'blogid': 3}),
)

# inner.py

from django.conf.urls.defaults import *

urlpatterns = patterns('',
    (r'^archive/$', 'mysite.views.archive'),
    (r'^about/$', 'mysite.views.about'),
    (r'^rss/$', 'mysite.views.rss'),
)
```

Set two:

```
# urls.py

from django.conf.urls.defaults import *

urlpatterns = patterns('',
    (r'^blog/', include('inner')),
)

# inner.py

from django.conf.urls.defaults import *
```

```
urlpatterns = patterns('',
    (r'^archive/$', 'mysite.views.archive', {'blogid': 3}),
    (r'^about/$', 'mysite.views.about', {'blogid': 3}),
    (r'^rss/$', 'mysite.views.rss', {'blogid': 3}),
)
```

Note that extra options will *always* be passed to *every* line in the included URLconf, regardless of whether the line’s view actually accepts those options as valid. For this reason, this technique is only useful if you’re certain that every view in the the included URLconf accepts the extra options you’re passing.

View tricks

This chapter is not yet finished. What else would you like to see? Leave a comment on this paragraph and let us know! Interaction is cool.

« previous ◊ table of contents ◊ next »

The Django Book

Chapter 9: Generic views

Here again is a recurring theme of this book: at its worst, web development is boring and monotonous.

So far we've covered how Django tries to take away some of that monotony at the model and template layers, but web developers also experience this boredom at the view level.

Django's **generic views** were to developed to ease that pain. They take certain common idioms and patterns in view development and abstract them so that you can quickly write common views of onto data without having to write too much code.

In fact, nearly every view example in the preceding chapters could be re-written with the help of generic views.

Django contains generic views to do the following:

- Perform common "simple" tasks: redirect to a different page, and render a given template.
- Display list and detail pages for a single object. For example, the Django documentation index (http://www.djangoproject.com/documentation/) and individual document pages are built this way. The crime index and list of crimes by type views from Chapter 5 could easily be re-written to use generic views; we'll do so below.
- Present date-based objects in year/month/day archive pages, associated detail and "latest" pages. The Django weblog's (http://www.djangoproject.com/weblog/) year, month, and day archives are built with these, as are ljworld.com's news archives, and a whole host of others.
- Allow users to create, update, and delete objects — with or without authorization.

Taken together, these views provide easy interfaces to perform the most common tasks developers encounter.

Using generic views

All of these views are used by creating configuration dictionaries in your URLconf files and passing those dictionaries as the third member of the URLconf tuple for a given pattern.

For example, here's the URLconf for the simple weblog app that drives the blog on djangoproject.com:

```
from django.conf.urls.defaults import *
from django_website.apps.blog.models import Entry

info_dict = {
    'queryset': Entry.objects.all(),
    'date_field': 'pub_date',
}

urlpatterns = patterns('django.views.generic.date_based',
   (r'^(?P<year>\d{4})/(?P<month>[a-z]{3})/(?P<day>\w{1,2})/(?P<slug>[-\w]+)/$',
'object_detail', dict(info_dict, slug_field='slug')),
   (r'^(?P<year>\d{4})/(?P<month>[a-z]{3})/(?P<day>\w{1,2})/$',
'archive_day',   info_dict),
   (r'^(?P<year>\d{4})/(?P<month>[a-z]{3})/$',
'archive_month', info_dict),
   (r'^(?P<year>\d{4})/$',
'archive_year',  info_dict),
   (r'^/?$',
'archive_index', info_dict),
)
```

As you can see, this URLconf defines a few options in `info_dict`. `'queryset'` gives the generic view a `QuerySet` of objects to use (in this case, all of the `Entry` objects) and tells the generic view which model is being used. The remaining arguments to each generic view are taken from the named captures in the URLconf.

This is really all the "view" code for Django's weblog! The only thing that's left is writing a template.

Documentation of each generic view follows, along with a list of all keyword arguments that a generic view expects. Remember that as in the example above, arguments may either come from the URL pattern (as `month`, `day`, `year`, etc. do above) or from the additional-information dictionary (as for `queryset`, `date_field`, etc.).

Most generic views require the `queryset` key, which is a `QuerySet` instance; see the database API reference in Appendix 3 for more information about `QuerySet` objects.

Most views also take an optional `extra_context` dictionary that you can use to pass any auxiliary information you wish to the view. The values in the `extra_context` dictionary can be either functions (or other callables) or other objects. Functions are evaluated just before they are passed to the template.

"Simple" generic views

The `django.views.generic.simple` module contains simple views to handle a couple of common cases: rendering a template when no view logic is needed, and issuing a redirect.

Rendering a template

The function `django.views.generic.simple.direct_to_template` renders a given template, passing it a `{{ params }}` template variable, which is a dictionary of the parameters captured in the URL.

Example

Given the following URL patterns:

```
urlpatterns = patterns('django.views.generic.simple',
    (r'^foo/$',             'direct_to_template', {'template': 'foo_index.html'}),
    (r'^foo/(?P<id>\d+)/$', 'direct_to_template', {'template': 'foo_detail.html'}),
)
```

a request to `/foo/` would render the template `foo_index.html`, and a request to `/foo/15/` would render the `foo_detail.html` with a context variable `{{ params.id }}` that is set to `15`.

Required arguments

`template`
The full name of a template to use.

Redirecting to another URL

`django.views.generic.simple.redirect_to` redirects to another URL. The given URL may contain dictionary-style string formatting, which will be interpolated against the parameters captured in the URL.

If the given URL is `None`, Django will return an HTTP 410 (Gone) message.

Example

This example redirects from `/foo/<id>/` to `/bar/<id>/`:

```
urlpatterns = patterns('django.views.generic.simple',
    ('^foo/(?p<id>\d+)/$', 'redirect_to', {'url': '/bar/%(id)s/'}),
)
```

This example returns a 410 HTTP error for requests to `/bar/`:

```
urlpatterns = patterns('django.views.generic.simple',
    ('^bar/$', 'redirect_to', {'url': None}),
)
```

Required arguments

`url`
The URL to redirect to, as a string. Or `None` to return a 410 ("gone") HTTP response.

More complex generic views

Although the simple generic views certainly are useful, the real power in Django's generic views comes from the more complex views that allow you to build common CRUD (Create/Retrieve/Update/Delete) pages with a minimum amount of code.

These views break down into a few different types:

- List/detail views, which provide flat lists of objects and individual object detail pages (for example, a list of places and individual place information pages).
- Date-based views, which provide year/month/day drill-down pages of date-centric information.
- Create/update/delete views, which allow you to quickly create views to create, modify, or delete objects.

Common optional arguments

Most of these views take a large number of optional arguments that can control various bits of behavior. Many of these arguments may be given to any of these views, so many of the views below refer back to this list of optional arguments:

`allow_empty`
A boolean specifying whether to display the page if no objects are available. If this is `False` and no objects are available, the view will raise a 404 instead of displaying an empty page. By default, this is `False`.

`context_processors`
A list of template-context processors to apply to the view's template. See Chapter 10 for information on template context processors.

`extra_context`
A dictionary of values to add to the template context. By default, this is an empty dictionary. If a value in the dictionary is callable, the generic view will call it just before rendering the template.

`mimetype`
The MIME type to use for the resulting document. Defaults to the value of the `DEFAULT_MIME_TYPE` setting.

`template_loader`
The template loader to use when loading the template. By default, it's `django.template.loader`. See Chapter 10 for information on template loaders.

`template_name`
The full name of a template to use in rendering the page. This lets you override the default template name derived from the `QuerySet`.

`template_object_name`
Designates the name of the template variable to use in the template context. By default, this is `'object'`. Views list list more than one objec will append `'_list'` to the value of this parameter.

List/detail generic views

The list-detail generic views (in the module `django.views.generic.list_detail`) handles the common case of displaying a list of items at one view, and individual "detail" views of those items at another.

For the examples in the rest of this chapter, we'll be working with the simple book/author/publisher objects from chapters 5 and 6:

```
class Publisher(models.Model):
    name = models.CharField(maxlength=30)
    address = models.CharField(maxlength=50)
    city = models.CharField(maxlength=60)
    state_province = models.CharField(maxlength=30)
    country = models.CharField(maxlength=50)
    website = models.URLField()

class Author(models.Model):
    salutation = models.CharField(maxlength=10)
    first_name = models.CharField(maxlength=30)
    last_name = models.CharField(maxlength=40)
    email = models.EmailField()
    headshot = models.ImageField()

class Book(models.ModelField):
    title = models.CharField(maxlength=100)
    authors = models.ManyToManyField(Author)
    publisher = models.ForeignKey(Publisher)
    publication_date = models.DateField()
```

We'll also be working with a URL module; if you're following along, you can start with an skeleton URL config in `bookstore.urls`:

```
from django.conf.urls.defaults import *
from django.views.generic import list_detail, date_based, create_update
from bookstore.models import Publisher, Author, Book

urlpatterns = patterns('',
    # We'll add URL patterns here.
)
```

We'll build this up with generic views as we go.

Lists of objects

The view `django.views.generic.list_detail.object_list` is used to create a page representing a list of objects.

Example

We can use the `object_list` view to show a simple list of all authors in the bookstore. First, we'll need to construct a info dictionary for the generic view. Add the following to the top of the `bookstore/urls.py` file:

```
author_list_info = {
    'queryset' :   Author.objects.all(),
    'allow_empty': True,
}
```

Then, we need to register this view at a certain URL. We can do that by adding this URL config piece (inside the `patterns` directive):

```
(r'authors/$', list_detail.object_list, author_list_info)
```

From there, we just need to make a template for this generic view to render. Since we didn't provide the `template_name` parameter (see below), Django will guess the name of the template; here it'll use `bookstore/author_list.html`. See below for more details on how this "guess" is made.

Required arguments

`queryset`
A `QuerySet` of objects to list

Optional arguments

`paginate_by`
An integer specifying how many objects should be displayed per page. If this is given, the view will paginate objects with `paginate_by` objects per page. The view will expect either a `page` query string parameter (via `GET`) containing a zero-indexed page number, or a `page` variable specified in the URLconf. See "Notes on pagination" below.

Additionally, this view may take any of these common arguments described above:

- `allow_empty`
- `context_processors`
- `extra_context`
- `mimetype`
- `template_loader`
- `template_name`
- `template_object_name`

Template name

If `template_name` isn't specified, this view will use the template `<app_label>/<model_name>_list.html` by default. Both the app label and the model name are derived from the `queryset` parameter: the app label is the name of the app that the model is defined in, and the model name is the lower-cased version of the name of the model class.

So, if we passed `Author.objects.all()` as the `queryset`, the app label would be `bookstore` and the model name would be `author`. This means the default template would be `bookstore/author_list.html`.

Template context

In addition to `extra_context`, the template's context will contain:

`object_list`
The list of objects. This variable's name depends on the `template_object_name` parameter, which is `'object'` by default. If `template_object_name` is `'foo'`, this variable's name will be `foo_list`.

`is_paginated`
A boolean representing whether the results are paginated. Specifically, this is set to `False` if the number of available objects is less than or equal to `paginate_by`.

If the results are paginated, the context will contain these extra variables:

`results_per_page`
The number of objects per page. (Same as the `paginate_by` parameter.)

`has_next`

A boolean representing whether there's a next page.

has_previous
A boolean representing whether there's a previous page.

page
The current page number, as an integer. This is 1-based.

next
The next page number, as an integer. If there's no next page, this will still be an integer representing the theoretical next-page number. This is 1-based.

previous
The previous page number, as an integer. This is 1-based.

pages
The total number of pages, as an integer.

hits
The total number of objects across *all* pages, not just this page.

A note on pagination:

If paginate_by is specified, Django will paginate the results. You can specify the page number in the URL in one of two ways:

- Use the page parameter in the URLconf. For example, this is what your URLconf might look like:

```
(r'^objects/page(?P<page>[0-9]+)/$', 'object_list', dict(info_dict))
```

- Pass the page number via the page query-string parameter. For example, a URL would look like this:

> /objects/?page=3

In both cases, page is 1-based, not 0-based, so the first page would be represented as page 1.

Detail views

The django.views.generic.list_detail.object_detail gives a "detail" view of a single object.

Example

Extending the example above, we could make a detail view for a given author. Given an info dict like this:

```
author_detail_info = {
    "queryset" : Author.objects.all(),
    "template_object_name" : "author",
}
```

We could use a urlpattern like:

```
(r'^authors/(?P<object_id>\d+)/$', list_detail.object_detail, author_detail_info),
```

to show details about a given book, rendered in the bookstore/author_detail.html template. In that template, the Author object itself would be put into the {{ author }} variable.

Required arguments

queryset`
A QuerySet that will be searched for the object.

Either:

object_id
The value of the primary-key field for the object.

or:

slug
The slug of the given object. If you pass this field, then the slug_field argument (below) is also required.

Optional arguments

`slug_field`
The name of the field on the object containing the slug. This is required if you are using the `slug` argument, but must be absent if you're using the `object_id` argument.

`template_name_field`

The name of a field on the object whose value is the template name to use. This lets you store template names in your data.

In other words, if your object has a field `'the_template'` that contains a string `'foo.html'`, and you set `template_name_field` to `'the_template'`, then the generic view for this object will use the template `'foo.html'`.

It's a bit of a brain-bender, but it's useful in some cases.

This view may also take these common arguments (documented above):

- `context_processors`
- `extra_context`
- `mimetype`
- `template_loader`
- `template_name`
- `template_object_name`

Template name

If `template_name` and `template_name_field` aren't specified, this view will use the template `<app_label>/<model_name>_detail.html` by default.

Template context

In addition to `extra_context`, the template's context will be:

`object`
The object. This variable's name depends on the `template_object_name` parameter, which is `'object'` by default. If `template_object_name` is `'foo'`, this variable's name will be `foo`.

Date-based generic views

Date-based generic views are generally used to provide a set of "archive" pages for dated material. Think year/month/day archives for a newspaper, or a blog like the official Django blog described at the beginning of this chapter.

For the examples, we'll be using the `Book` object from above, and build up a way to browse books by year, month, and day published. Notice that for each of these views, we have to tell Django the name of the date field we want to key off of. We have to provide this information since models could contain multiple date or datetime fields.

Into the future…

By default, these views ignore objects with dates in the future.

This means that if you try to visit an archive page in the future, Django will automatically show a 404 ("not found") error, even if there are objects published that day.

Thus, you can publish post-dated objects that don't appear publically until after their publication date.

However, for different types of date-based objects this isn't appropriate (for example, a calendar of upcoming events). For these views, setting the `allow_future` option to `True` will make the future objects appear (and allow users to visit "future" archive pages).

Archive index

The `django.views.generic.date_based.archive_index` view provides a top-level index page showing the "latest" objects, by date.

Example

A typical publisher probably wants to highlight recently-published books. We can use the `archive_index` view for this common task. Here's a info dict:

```
book_info = {
    "queryset"   : Book.objects.all(),
    "date_field" : "publication_date"
}
```

And the corresponding urlconf piece (which roots this index at the bottom level of wherever it's included):

```
(r'^books/$', date_based.archive_index, book_info),
```

Required arguments

`date_field`:
The name of the `DateField` or `DateTimeField` in the `QuerySet`'s model that the date-based archive should use to determine the objects on the page.

`queryset`
A `QuerySet` of objects for which the archive serves.

Optional arguments

`allow_future`
A boolean specifying whether to include "future" objects on this page, as described in the note above.

`num_latest`
The number of latest objects to send to the template context. By default, it's 15.

This view may also take these common arguments (documented above):

- `allow_empty`
- `context_processors`
- `extra_context`
- `mimetype`
- `template_loader`
- `template_name`

Template name

If `template_name` isn't specified, this view will use the template `<app_label>/<model_name>_archive.html` by default.

Template context

In addition to `extra_context`, the template's context will be:

`date_list`

A list of `datetime.date` objects representing all years that have objects available according to `queryset`. These are ordered in reverse.

For example, if you have blog entries from 2003 through 2006, this list will contain four `datetime.date` objects: one for each of those years.

`latest`
The `num_latest` objects in the system, ordered descending by `date_field`. For example, if `num_latest` is `10`, then `latest` will be a list of the latest 10 objects in `queryset`.

Year archives

The `django.views.generic.date_based.archive_year` view provides a yearly archive page showing all available months in a given year.

Example

Contuting on with our example, we'll want to add a way to view all the books published in a given year. We can keep using the `book_info` dictionary from the above example, but this time we'll wire it up to the `archive_year` view:

```
(r'^books/(?P<year>\d{4})/?$', date_based.archive_year, book_info),
```

Since there are likely many, many books published each year, we won't display them on this page, just a list of years in which books are available. Conveniently for us, this is what Django does by default; to change it we could use the `make_object_list` argument; see below.

Required arguments

`date_field`
As above.

`queryset`
A `QuerySet` of objects for which the archive serves.

`year`
The four-digit year for which the archive serves (usually taken from URL parameters).

Optional arguments

`make_object_list`
A boolean specifying whether to retrieve the full list of objects for this year and pass those to the template. If `True`, this list of objects will be made available to the template as `object_list`. (The name `object_list` may be different; see the information about `object_list` in the "Template context" section below.) By default, this is `False`.

`allow_future`
A boolean specifying whether to include "future" objects on this page, as described in the note above.

This view may also take these common arguments (documented above):

- `allow_empty`
- `context_processors`
- `extra_context`
- `mimetype`
- `template_loader`
- `template_name`
- `template_object_name`

Template name

If `template_name` isn't specified, this view will use the template `<app_label>/<model_name>_archive_year.html` by default.

Template context

In addition to `extra_context`, the template's context will be:

`date_list`
A list of `datetime.date` objects representing all months that have objects available in the given year, according to `queryset`, in ascending order.

`year`
The given year, as a four-character string.

`object_list`

If the `make_object_list` parameter is `True`, this will be set to a list of objects available for the given year, ordered by the date field. This variable's name depends on the `template_object_name` parameter, which is `'object'` by default. If `template_object_name` is `'foo'`, this variable's name will be `foo_list`.

If `make_object_list` is `False`, `object_list` will be passed to the template as an empty list.

Monthly archives

The `django.views.generic.date_based.archive_month` views provides a monthly archive page showing all objects in a given month.

Example

Continuing on with our example, creating month views should look mighty familiar:

```
(r'^(?P<year>\d{4})/(?P<month>[a-z]{3})/$', date_based.archive_month, book_info),
```

Required arguments

`year`
The four-digit year for which the archive serves (a string).

`month`
The month for which the archive serves, formatted according to the `month_format` argument.

queryset
A QuerySet of objects for which the archive serves.

date_field
The name of the DateField or DateTimeField in the QuerySet's model that the date-based archive should use to determine the objects on the page.

Optional arguments

month_format
A format string that regulates what format the month parameter uses. This should be in the syntax accepted by Python's time.strftime. (See Python's strftime docs at http://www.python.org/doc/current/lib/module-time.html#l2h-1941) It's set to "%b" by default, which is a three-letter month abbreviation (i.e. "jan", "feb", etc.). To change it to use numbers, use "%m".

allow_future
A boolean specifying whether to include "future" objects on this page, as described in the note above.

This view may also take these common arguments (documented above):

- allow_empty
- context_processors
- extra_context
- mimetype
- template_loader
- template_name
- template_object_name

Template name

If template_name isn't specified, this view will use the template <app_label>/<model_name>_archive_month.html by default.

Template context

In addition to extra_context, the template's context will be:

month
A datetime.date object representing the given month.

next_month
A datetime.date object representing the first day of the next month. If the next month is in the future, this will be None.

previous_month
A datetime.date object representing the first day of the previous month. Unlike next_month, this will never be None.

object_list
A list of objects available for the given month. This variable's name depends on the template_object_name parameter, which is 'object' by default. If template_object_name is 'foo', this variable's name will be foo_list.

Week archives

The django.views.generic.date_based.archive_week view shows all objects in a given week.

Note

Django believes that weeks start on Sunday, for the perfectly arbitrary reason that Python does, too.

Example

Are you starting to see a pattern here yet?

```
(r'^(?P<year>\d{4})/(?P<week>\d{2})/$', date_based.archive_week, book_info),
```

Required arguments

year
The four-digit year for which the archive serves (a string).

week
The week of the year for which the archive serves (a string).

queryset
A `QuerySet` of objects for which the archive serves.

date_field
The name of the `DateField` or `DateTimeField` in the `QuerySet`'s model that the date-based archive should use to determine the objects on the page.

Optional arguments

allow_future
A boolean specifying whether to include "future" objects on this page, as described in the note above.

This view may also take these common arguments (documented above):

- `allow_empty`
- `context_processors`
- `extra_context`
- `mimetype`
- `template_loader`
- `template_name`
- `template_object_name`

Template name

If `template_name` isn't specified, this view will use the template `<app_label>/<model_name>_archive_week.html` by default.

Template context

In addition to `extra_context`, the template's context will be:

week
A `datetime.date` object representing the first day of the given week.

object_list
A list of objects available for the given week. This variable's name depends on the `template_object_name` parameter, which is `'object'` by default. If `template_object_name` is `'foo'`, this variable's name will be `foo_list`.

Day archives

The `django.views.generic.date_based.archive_day` view provides a page showing all objects in a given day.

Example

Keep on keepin' on:

```
(r'^(?P<year>\d{4})/(?P<month>[a-z]{3})/(?P<day>\d{2})/$', date_based.archive_day, book_info),
```

Required arguments

year
The four-digit year for which the archive serves (a string).

month
The month for which the archive serves, formatted according to the `month_format` argument.

day
The day for which the archive serves, formatted according to the `day_format` argument.

queryset
A `QuerySet` of objects for which the archive serves.

date_field
The name of the `DateField` or `DateTimeField` in the `QuerySet`'s model that the date-based archive should use to determine the objects on the page.

Optional arguments

month_format
A format string that regulates what format the `month` parameter uses. See the detailed explanation above.

day_format
Like `month_format`, but for the `day` parameter. It defaults to `"%d"` (day of the month as a decimal number, 01-31).

`allow_future`
A boolean specifying whether to include “future” objects on this page, as described in the note above.

This view may also take these common arguments (documented above):

- `allow_empty`
- `context_processors`
- `extra_context`
- `mimetype`
- `template_loader`
- `template_name`
- `template_object_name`

Template name

If `template_name` isn’t specified, this view will use the template `<app_label>/<model_name>_archive_day.html` by default.

Template context

In addition to `extra_context`, the template’s context will be:

`day`
A `datetime.date` object representing the given day.

`next_day`
A `datetime.date` object representing the next day. If the next day is in the future, this will be `None`.

`previous_day`
A `datetime.date` object representing the given day. Unlike `next_day`, this will never be `None`.

`object_list`
A list of objects available for the given day. This variable’s name depends on the `template_object_name` parameter, which is `'object'` by default. If `template_object_name` is `'foo'`, this variable’s name will be `foo_list`.

Archive for today

The `django.views.generic.date_based.archive_today` view shows all objects for *today*. This is exactly the same as `archive_day`, except the `year`/`month`/`day` arguments are not used, and today’s date is used instead.

Date-based detail pages

The `django.views.generic.date_based.object_detail` view shows a page representing an individual object. This differs from the `object_detail` page in their respective URLs; the `object_detail` view uses URLs like `/entries/<slug>/`, while this one uses URLs like `/entries/2006/aug/27/<slug>/`.

Note

If you’re using date-based detail pages with slugs in the URLs, you probably also want to use the `unique_for_date` option on the slug field to validate that slugs aren’t duplicated in a single day. See Appendix 2 for details on `unique_for_date`.

Example

This one differs (slightly) from all the other examples in that we need to either provide an object ID or a slug so that Django can look up the object in question.

Since the object we’re using doesn’t have a slug field, we’ll use the slightly uglyier ID-based URLs. In practice we’d prefer to use a slug field, but in the interest of simplicity we’ll let it go.

We’ll add the following to the URLconf:

```
(r'^(?P<year>\d{4})/(?P<month>[a-z]{3})/(?P<day>\d{2})/(?P<object_id>[\w-]+)/$', date_based.
object_detail, book_info),
```

Required arguments

`year`
The object’s four-digit year (a string).

`month`

The object's month , formatted according to the `month_format` argument.

`day`
The object's day , formatted according to the `day_format` argument.

`queryset`
A `QuerySet` that contains the object.

`date_field`
The name of the `DateField` or `DateTimeField` in the `QuerySet`'s model that the generic view should use to look up the object according to `year`, `month` and `day`.

Either:

`object_id`
The value of the primary-key field for the object.

or:

`slug`
The slug of the given object. If you pass this field, then the `slug_field` argument (below) is also required.

Optional arguments

`allow_future`
A boolean specifying whether to include "future" objects on this page, as described in the note above.

`day_format`
Like `month_format`, but for the `day` parameter. It defaults to `"%d"` (day of the month as a decimal number, 01-31).

`month_format`
A format string that regulates what format the `month` parameter uses. See the detailed explanation above.

`slug_field`
The name of the field on the object containing the slug. This is required if you are using the `slug` argument, but must be absent if you're using the `object_id` argument.

`template_name_field`

The name of a field on the object whose value is the template name to use. This lets you store template names in the data. In other words, if your object has a field `'the_template'` that contains a string `'foo.html'`, and you set `template_name_field` to `'the_template'`, then the generic view for this object will use the template `'foo.html'`.

It's a bit of a brain-bender, but it's useful in some cases.

This view may also take these common arguments (documented above):

- `context_processors`
- `extra_context`
- `mimetype`
- `template_loader`
- `template_name`
- `template_object_name`

Template name

If `template_name` isn't specified, this view will use the template `<app_label>/<model_name>_detail.html` by default.

Template context

In addition to `extra_context`, the template's context will be:

`object`
The object. This variable's name depends on the `template_object_name` parameter, which is `'object'` by default. If `template_object_name` is `'foo'`, this variable's name will be `foo`.

Create/update/delete generic views

Note

These views will change slightly when Django's revised form architecture (currently under development as `django.newforms`) is finalized. This section will be updated accordingly.

The `django.views.generic.create_update` module contains a set of functions for creating, editing and deleting objects.

Create object view

The `django.views.generic.create_update.create_object` view displays a form for creating an object, redisplays the form with validation errors (if there are any) and saves the object. This uses the automatic manipulators that come with Django models.

These views all present forms if accessed with a `GET` and perform the requested action (create/update/delete) if accessed via `POST` .

Note that these views all have a very rough idea of security. Although they take a `login_required` attribute which if given will restrict access to logged-in users, that's as far as it goes. They won't, for example, check that the user editing an object is the same user that created it, nor will they validate any sort of permissions.

Much of the time, however, those features can be accomplished by writing a small wrapper around the generic view; see "extending generic views", below, for more about this topic.

Example

If we wanted to allow users to create new books in our database, we could do something like this:

```
(r'^books/create/$', create_update.create_object, {'model' : Book}),
```

Required arguments

`model`
The Django model of the object that the form will create.

> **Note**
>
> Notice that this view takes the *model* to be created, not a `QuerySet` (as all the list/detail/date-based views above do).

Optional arguments

`post_save_redirect`
A URL to which the view will redirect after saving the object. By default, it's `object.get_absolute_url()`.

`post_save_redirect`
May contain dictionary string formatting, which will be interpolated against the object's field attributes. For example, you could use `post_save_redirect="/polls/%(slug)s/"`.

`login_required`

A boolean that designates whether a user must be logged in, in order to see the page and save changes. This hooks into the Django authentication system. By default, this is `False`.

If this is `True`, and a non-logged-in user attempts to visit this page or save the form, Django will redirect the request to `/accounts/login/`.

This view may also take these common arguments (documented above):

- `context_processors`
- `extra_context`
- `template_loader`
- `template_name`

Template name

If `template_name` isn't specified, this view will use the template `<app_label>/<model_name>_form.html` by default.

Template context

In addition to `extra_context`, the template's context will be:

`form`:

A `FormWrapper` instance representing the form for editing the object. This lets you refer to form fields easily in the template system.

For example, if the model has two fields, `name` and `address`:

```
<form action="" method="post">
<p><label for="id_name">Name:</label> {{ form.name }}</p>
<p><label for="id_address">Address:</label> {{ form.address }}</p>
</form>
```

See Chapter 7 for more information about working with forms.

Update object view

The `django.views.generic.create_update.update_object` view is almost identical to the create-object view above, but this one allows the editing of an existing object instead of the creation of a new one.

Example

Following the above example, we could provide an edit interface for a single book with this URLconf snippet:

```
(r'^books/edit/(?P<object_id>\d+)/$', create_update.update_object, {'model' : Book}),
```

Required arguments

`model`
The Django model the form will be editing.

Either:

`object_id`
The value of the primary-key field for the object.

or:

`slug`
The slug of the given object. If you pass this field, then the `slug_field` argument (below) is also required.

Optional arguments

`slug_field`
The name of the field on the object containing the slug. This is required if you are using the `slug` argument, but must be absent if you're using the `object_id` argument.

Additionally, this view takes all same optional arguments as the creation view (above), plus the `template_object_name` common argument.

Template name

This view uses the same default template name (`<app_label>/<model_name>_form.html`) as the creation view.

Template context

In addition to `extra_context`, the template's context will be:

`form`:
A `FormWrapper` instance representing the form for editing the object. See the create object (above) for more about this value.

`object`:
The original object being edited (this variable may be named differently if you've provided the `template_object_name` argument).

Delete object view

The `django.views.generic.create_update.delete_object` view is also very similar to the other two.

If this view is fetched with `GET`, it will display a confirmation page (i.e. "do you really want to delete this object?"). If the view is submitted with `POST`, the object will be deleted without confirmation.

All the arguments are the same as for the update object view, as is the context; the template name for this view is `<app_label>/<model_name>_confirm_delete.html`

Extending generic views

There's no question that using generic views can speed up development substantially. In most projects, however, there comes a moment when the generic views no longer suffice. Indeed, the most common question asked by new Django developers is about how to make generic views handle a wider array of situations.

Luckily, in nearly every one of these cases, there are ways to simply extend generic views to handle a larger array of use cases. These situations usually fall into a couple of patterns:

Adding extra context

Often you simply need to present some extra information than that provided by the generic view. For example, think of showing a list of all publishers on a book's detail page; the `object_detail` generic view provides the book to the context, but it seems there's no way to get a list of publishers in that template.

But there is: all generic views take an extra optional parameter `extra_context`. This is a dictionary of extra objects which will be added to the template's context. So, to provide the list of publishers in the book detail view, we'd use an info dict like this:

```
book_info = {
    "queryset"   : Book.objects.all(),
    "date_field" : "publication_date",
    "extra_context" : {
        "publisher_list" : Publisher.objects.all(),
    }
}
```

This would populate a `{{ publisher_list }}` variable in the template context. This pattern can be used to pass any information down into the template for the generic view; it's very handy.

More complex filtering with wrapper functions

Another common need is to filter down the objects given in a list page by some key in the URL. For example, let's look at providing an interface to browse books by title. We'd like to provide URLs of the form `/books/by-title/a/`, `/books/by-title/b/`, etc. — one list page for each letter of the alphabet.

The problem seems to be that the generic view has no concept of reading variables from the URL; if we wired a URL pattern matching those URLs up to the `object_list` view, we'd get twenty-six pages displaying all the books. Although we *could* write twenty-six different info dicts (each with a different `queryset` argument), that's just silly. The right technique involves writing a simple "wrapper" function around the generic view.

In our alphabetic-browsing example, we'd start by adding a small bit to the URLconf:

```
from bookstore.views import browse_alphabetically

urlpatterns = patterns('',
    # ...
    (r'^books/by-title/([a-z])/$', browse_alphabetically)
)
```

As you can see, this wires the set of URLs to the `browse_alphabetically` function, so let's take a look at how that function could be written:

```
from bookstore.models. import Book
from django.views.generic import list_detail

def browse_alphabetically(request, letter):
    return list_detail.object_list(
        request,
        queryset = Book.objects.filter(title__istartswith=letter),
        template_name = "bookstore/browse_alphabetically.html",
        extra_context = {
            'letter' : letter,
        }
    )
```

That's it!

This works because there's really nothing special about generic views — they're just Python functions. Like any view function, generic views expect a certain set of arguments and return `HttpResponse` objects. Thus, it's incredibly easy to wrap a small function around a generic view that does additional work before — or after; see below — handing things off to the generic view.

Note

Notice that in the above example we've passed the current letter being display in the `extra_context`. This is usually a good idea in wrappers of this nature; it lets the template know which letter is currently being browsed.

Also (while we're on the topic of templates) notice that we've passed in a custom template name. Without that, it would try to use the same template as a "vanilla" `object_list`, which could conflict with other generic views.

Performing extra work

The last common pattern we'll look at involves doing some extra work before or after calling the generic view.

Imagine we had a `last_accessed` field on our `Author` object that we were using to keep track of the last time a anybody looked at that author. The generic `object_detail` view, of course, wouldn't know anything about this field, but once again we could easily write a custom view to keep that field updated.

First, we'd need to modify the author detail bit in the URLconf to point to a custom view:

```
from bookstore.views import author_detail

urlpatterns = patterns('',
    #...
    (r'^authors/(?P<author_id>d+)/$', author_detail),
)
```

Then we'd write our wrapper function:

```
import datetime
from bookstore.models import Author
from django.views.generic import list_detail
from django.shortcuts import get_object_or_404

def author_detail(request, author_id):
    # Look up the Author (and raise a 404 if she's not found)
    author = get_object_or_404(Author, pk=author_id)

    # Record the last accessed date
    author.last_accessed = datetime.datetime.now()
    author.save()

    # Show the detail page
    return list_detail.object_detail(
        request,
        queryset = Author.objects.all(),
        object_id = author_id,
    )
```

Note

This code won't actually work unless you add the `last_accessed` field to your `Author` model.

We can use a similar idiom to alter the response returned by the generic view. If we wanted to provide a downloadable plain-text version of the list of authors, we could use a view like this:

```
def author_list_plaintext(request):
    response = list_detail.object_list(
        queryset = Author.objects.all(),
        mimetype = "text/plain",
        template_name = "bookstore/author_list.txt"
    )
    response["Content-Disposition"] = "attachment; filename=authors.txt"
    return response
```

This works because the generic views return simple `HttpResponse` objects which can be treated like dictionaries to set HTTP headers. This `Content-Disposition` business, by the way, instructs the browser to download and save the page instead of displaying it in the browser.

What's next?

Until now, we've treated the template engine as a mostly static tool you can use to render your content. It's true that most of the time you'll just treat it in that way, but the template engine is actually quite extensible.

In the next chapter we'll delve deep into the inner workings of Django's templates, showing all the cool ways it can be extended.

Onward, comrades!

« previous ◊ table of contents ◊ next »

Chapter 10: Inside the template engine

Most of your interaction with Django's template language will probably be in the role of a template author. This chapter delves much deeper into the guts of Django's template system; read on if you need to extend the template system, or if you're just curious about how it works internally.

If you're looking to use the Django template system as part of another application — i.e., without the rest of the framework — make sure to read the configuration section later in this document.

Basics

A **template** is a text document, or a normal Python string, that is marked-up using the Django template language. A template can contain **block tags** or **variables**.

A **block tag** is a symbol within a template that does something.

This definition is deliberately vague. For example, a block tag can output content, serve as a control structure (an "if" statement or "for" loop), grab content from a database or enable access to other template tags.

Block tags are surrounded by `{%` and `%}`:

```
{% if is_logged_in %}
  Thanks for logging in!
{% else %}
  Please log in.
{% endif %}
```

A **variable** is a symbol within a template that outputs a value.

Variable tags are surrounded by `{{` and `}}`:

```
My first name is {{ first_name }}. My last name is {{ last_name }}.
```

A **context** is a "name" -> "value" mapping (similar to a Python dictionary) that is passed to a template.

A template **renders** a context by replacing the variable "holes" with values from the context and executing all block tags.

Using the template object

At its lowest level, using the template system in Python is a two-step process:

- First, you compile the raw template code into a `Template` object.
- Then, you call the `render()` method of the `Template` object with a given context.

Compiling a string

The easiest way to create a `Template` object is by instantiating it directly. The constructor takes one argument — the raw template code:

```
>>> from django.template import Template
>>> t = Template("My name is {{ my_name }}.")
>>> print t
<django.template.Template object at 0x1150c70>
```

Behind the scenes

The system only parses your raw template code once — when you create the `Template` object. From then on, it's stored internally as a "node" structure for performance.

Even the parsing itself is quite fast. Most of the parsing happens via a single call to a single, short, regular expression.

Rendering a context

Once you have a compiled `Template` object, you can render a context — or multiple contexts — with it. The `Context` constructor takes one (optional) argument: a dictionary mapping variable names to variable values.

Call the `Template` object's `render()` method with the context to "fill" the template:

```
>>> from django.template import Context, Template
>>> t = Template("My name is {{ my_name }}.")

>>> c = Context({"my_name": "Adrian"})
>>> t.render(c)
"My name is Adrian."

>>> c = Context({"my_name": "Dolores"})
>>> t.render(c)
"My name is Dolores."
```

Variable names must consist of any letter (A-Z), any digit (0-9), an underscore or a dot.

Dots have a special meaning in template rendering. A dot in a variable name signifies **lookup**. Specifically, when the template system encounters a dot in a variable name, it tries a number of possible options. For example, the variable `{{ foo.bar }}` could expand to any of the following:

- Dictionary lookup: `foo["bar"]`
- Attribute lookup: `foo.bar`
- Method call: `foo.bar()`
- List-index lookup: `foo[bar]`

The template system uses the first lookup type that works; it's short-circuit logic.

Here are a few examples:

```
>>> from django.template import Context, Template
>>> t = Template("My name is {{ person.first_name }}.")

>>> d = {"person": {"first_name": "Joe", "last_name": "Johnson"}}
>>> t.render(Context(d))
"My name is Joe."

>>> class Person:
...     def __init__(self, first_name, last_name):
...         self.first_name, self.last_name = first_name, last_name
...
>>> p = Person("Ron", "Nasty")
>>> t.render(Context({"person": p}))
"My name is Ron."

>>> class Person2:
...     def first_name(self):
...         return "Samantha"
...
>>> p = Person2()
>>> t.render(Context({"person": p}))
"My name is Samantha."

>>> t = Template("The first stooge in the list is {{ stooges.0 }}.")
>>> c = Context({"stooges": ["Larry", "Curly", "Moe"]})
>>> t.render(c)
"The first stooge in the list is Larry."
```

Method lookups are slightly more complex than the other lookup types. Here are some things to keep in mind:

- If, during the method lookup, a method raises an exception, the exception will be propagated unless the exception has an attribute `silent_variable_failure` whose value is `True`.

  If the exception *does* have such an attribute, the variable will render as an empty string.

  For example:

  ```
  >>> t = Template("My name is {{ person.first_name }}.")
  ```

```
>>> class Person3:
...     def first_name(self):
...         raise AssertionError("foo")
...
>>> p = Person3()
>>> t.render(Context({"person": p}))
Traceback (most recent call last):
...
AssertionError: foo

>>> class SilentAssertionError(AssertionError):
...     silent_variable_failure = True
...
>>> class Person4:
...     def first_name(self):
...         raise SilentAssertionError("foo")
...
>>> p = PersonClass4()
>>> t.render(Context({"person": p}))
"My name is ."
```

Note that `django.core.exceptions.ObjectDoesNotExist`, which is the base class for all Django database API `DoesNotExist` exceptions, has `silent_variable_failure = True`. So if you're using Django templates with Django model objects, any `DoesNotExist` exception will fail silently.

- A method call will only work if the method has no required arguments. Otherwise, the system will move to the next lookup type (list-index lookup).

- Obviously, some methods have side effects, and it'd be either foolish or a security hole to allow the template system to access them.

  A good example is the `delete()` method on each Django model object. The template system shouldn't be allowed to do something like this:

  ```
  I will now delete this valuable data. {{ data.delete }}
  ```

  To prevent this, set a function attribute `alters_data` on the method. The template system won't execute a method if the method has `alters_data=True` set:

  ```
  def sensitive_function(self):
      self.database_record.delete()
  sensitive_function.alters_data = True
  ```

  The dynamically-generated `delete()` and `save()` methods on Django model objects get `alters_data=True` automatically, for example.

How invalid variables are handled

Generally, if a variable doesn't exist, the template system inserts the value of the `TEMPLATE_STRING_IF_INVALID` setting, which is set to the empty string by default.

Filters that are applied to an invalid variable will only be applied if `TEMPLATE_STRING_IF_INVALID` is set to its default value. If `TEMPLATE_STRING_IF_INVALID` is set to any other value, variable filters will be ignored.

This behavior is slightly different for the `if`, `for` and `regroup` template tags. If an invalid variable is provided to one of these template tags, the variable will be interpreted as `None`. Filters are always applied to invalid variables within these template tags.

Playing with Context objects

Most of the time, you'll instantiate `Context` objects by passing in a fully-populated dictionary to `Context()`. But you can add and delete items from a `Context` object once it's been instantiated, too, using standard dictionary syntax:

```
>>> c = Context({"foo": "bar"})
>>> c['foo']
'bar'
>>> del c['foo']
>>> c['foo']
''
>>> c['newvariable'] = 'hello'
>>> c['newvariable']
'hello'
```

Furthermore, a Context object acts like a stack. That is, you can push() and pop() additional contexts onto the stack. All setting operations happen to the top-most context on the stack, but get operations search the stack (top-down) until a value is found.

If you pop() too much, you'll get a django.template.ContextPopException.

Here's an example of how these multiple levels might work:

```
# Create a new blank context and set a simple value:
>>> c = Context()
>>> c['foo'] = 'first level'

# Push a new context onto the stack:
>>> c.push()
>>> c['foo'] = 'second level'

# The value of "foo" is now what we set at the second level:
>>> c['foo']
'second level'

# After popping a layer off, the old value is still there:
>>> c.pop()
>>> c['foo']
'first level'

# If we don't push() again, we'll overwrite existing values:
>>> c['foo'] = 'overwritten'
>>> c['foo']
'overwritten'

# There's only one context on the stack, so pop()ing will fail:
>>> c.pop()
Traceback (most recent call last):
...
django.template.ContextPopException
```

Using a Context as a stack comes in handy in some custom template tags, as you'll see below.

RequestContext and context processors

Django comes with a special Context class, django.template.RequestContext, that acts slightly differently than the normal django.template.Context. The first difference is that takes an HttpRequest object (see Chapter XXX) as its first argument:

```
c = RequestContext(request, {
    'foo': 'bar',
}
```

The second difference is that it automatically populates the context with a few variables, according to your TEMPLATE_CONTEXT_PROCESSORS setting.

The TEMPLATE_CONTEXT_PROCESSORS setting is a tuple of callables called **context processors** that take a request object as their argument and return a dictionary of items to be merged into the context. By default, TEMPLATE_CONTEXT_PROCESSORS is set to:

```
("django.core.context_processors.auth",
 "django.core.context_processors.debug",
 "django.core.context_processors.i18n")
```

Each processor is applied in order. That is, if one processor adds a variable to the context and a second processor adds a variable with the same name, the second will override the first. The default processors are explained below.

Also, you can give RequestContext a list of additional processors, using the optional, third argument, processors. In this example, the RequestContext instance gets a ip_address variable:

```
def ip_address_processor(request):
    return {'ip_address': request.META['REMOTE_ADDR']}

def some_view(request):
    # ...
    return RequestContext(request, {
        'foo': 'bar',
    }, processors=[ip_address_processor])
```

Here's what each of the default processors does:

django.core.context_processors.auth

If `TEMPLATE_CONTEXT_PROCESSORS` contains this processor, every `RequestContext` will contain these three variables:

user
A `djangol.contrib.auth.models.User` instance representing the currently logged-in user (or an `AnonymousUser` instance, if the client isn't logged in).

messages

A list of messages (as strings) for the currently logged-in user. Behind the scenes, this calls `request.user.get_and_delete_messages()` for every request. That method collects the user's messages and deletes them from the database.

Note that messages are set with `user.add_message()`.

perms
An instance of `django.core.context_processors.PermWrapper`, representing the permissions that the currently logged-in user has.

See Chapter XXX for more on users, permissions, and messages.

django.core.context_processors.debug

This processor pushed debugging information down to the template layer. If it is enabled, it will only actually operate if:

- the `DEBUG` setting is `True`, and
- the request came from an IP address in the `INTERNAL_IPS` setting.

If those conditions *are* met, the following variables will be set:

debug
Set to `True`; you can use this in templates to test whether you're in `DEBUG` mode.

sql_queries
A list of `{'sql': ..., 'time': ...}` dictionaries, representing every SQL query that has happened so far during the request and how long it took. The list is in order by query.

django.core.context_processors.i18n

If this processor is enabled this processor, every `RequestContext` will contain these two variables:

LANGUAGES
The value of the `LANGUAGES` setting.

LANGUAGE_CODE
`request.LANGUAGE_CODE`, if it exists. Otherwise, the value of the `LANGUAGE_CODE` setting

Appendix XXX has more information about these two settings.

django.core.context_processors.request

If enabled, every `RequestContext` will contain a variable `request`, which is the current `HttpRequest` object. Note that this processor is not enabled by default; you'll have to activate it.

Loading templates

Generally, you'll store templates in files on your filesystem (or in other places if you've written custom template loaders) rather than using the low-level `Template` API yourself.

Django searches for template directories in a number of places, depending on your template-loader settings (see "Loader types" below), but the most basic way of specifying template directories is by using the `TEMPLATE_DIRS` setting.

This should be set to a list or tuple of strings that contain full paths to your template directory(ies):

```
TEMPLATE_DIRS = (
    "/home/html/templates/lawrence.com",
    "/home/html/templates/default",
)
```

Your templates can go anywhere you want, as long as the directories and templates are readable by the Web server. They can have any extension you want, such as `.html` or `.txt`, or they can have no extension at all.

Note that these paths should use Unix-style forward slashes, even on Windows.

The Python API

Django has two ways to load templates from files:

`django.template.loader.get_template(template_name)`
`get_template` returns the compiled template (a `Template` object) for the template with the given name. If the template doesn't exist, it raises `django.template.TemplateDoesNotExist`.

`django.template.loader.select_template(template_name_list)`
`select_template` is just like `get_template`, except it takes a list of template names. Of the list, it returns the first template that exists.

For example, if you call `get_template('story_detail.html')` and have the above `TEMPLATE_DIRS` setting, here are the files Django will look for, in order:

- `/home/html/templates/lawrence.com/story_detail.html`
- `/home/html/templates/default/story_detail.html`

If you call `select_template(['story_253_detail.html', 'story_detail.html'])`, here's what Django will look for:

- `/home/html/templates/lawrence.com/story_253_detail.html`
- `/home/html/templates/default/story_253_detail.html`
- `/home/html/templates/lawrence.com/story_detail.html`
- `/home/html/templates/default/story_detail.html`

When Django finds a template that exists, it stops looking.

Tip

You can use `select_template()` for super-flexible "templatability." For example, if you've written a news story and want some stories to have custom templates, use something like `select_template(['story_%s_detail.html' % story.id, 'story_detail.html'])`. That'll allow you to use a custom template for an individual story, with a fallback template for stories that don't have custom templates.

Using subdirectories

It's possible — and preferable — to organize templates in subdirectories of the template directory. The convention is to make a subdirectory for each Django app, with subdirectories within those subdirectories as needed.

Do this for your own sanity. Storing all templates in the root level of a single directory gets messy.

To load a template that's within a subdirectory, just use a slash, like so:

```
get_template('news/story_detail.html')
```

Using the same `TEMPLATE_DIRS` setting from above, this example `get_template()` call will attempt to load the following templates:

- `/home/html/templates/lawrence.com/news/story_detail.html`
- `/home/html/templates/default/news/story_detail.html`

Again, use UNIX-style forward slashes, even on Windows.

Template loaders

By default, Django loads templates from the filesystem, but Django comes with a few other **template loaders** which know how to load templates from other sources.

Some of these other loaders are disabled by default, but you can activate them by editing your `TEMPLATE_LOADERS` setting. `TEMPLATE_LOADERS` should be a tuple of strings, where each string represents a template loader. These template loaders ship with Django:

`django.template.loaders.filesystem.load_template_source`

Loads templates from the filesystem, according to `TEMPLATE_DIRS`.

This loader is enabled by default.

`django.template.loaders.app_directories.load_template_source`

Loads templates from Django apps on the filesystem. For each app in `INSTALLED_APPS`, the loader looks for a `templates` subdirectory. If the directory exists, Django looks for templates in there.

This means you can store templates with your individual apps. This also makes it easy to distribute Django apps with default templates.

For example, if `INSTALLED_APPS` contains `('myproject.polls', 'myproject.music')`, then `get_template('foo.html')` will look for templates in in this order:

- `/path/to/myproject/polls/templates/foo.html`
- `/path/to/myproject/music/templates/foo.html`

Note that the loader performs an optimization when it is first imported: It caches a list of which `INSTALLED_APPS` packages have a `templates` subdirectory.

This loader is enabled by default.

`django.template.loaders.eggs.load_template_source`

Just like `app_directories` above, but it loads templates from Python eggs rather than from the filesystem.

This loader is disabled by default; you'll need to enable it if you're using eggs to distribute your app.

Django uses the template loaders in order according to the `TEMPLATE_LOADERS` setting. It uses each loader until a loader finds a match.

Extending the template system

Although the Django template language comes with several default tags and filters, you might want to write your own, and it's easy to do.

First, create a `templatetags` package in the appropriate Django app's package. It should be on the same level as `models.py`, `views.py`, etc. For example:

```
polls/
    models.py
    templatetags/
    views.py
```

Add two files to the `templatetags` package: an `__init__.py` file (to indicate to Python that this is a module containing Python code) and a file that will contain your custom tag/filter definitions.

The name of the latter file is the name you'll use to load the tags later. For example, if your custom tags/filters are in a file called `poll_extras.py`, you'd do the following in a template:

```
{% load poll_extras %}
```

The `{% load %}` tag looks at your `INSTALLED_APPS` setting and only allows the loading of template libraries within installed Django apps. This is a security feature: It allows you to host Python code for many template libraries on a single computer without enabling access to all of them for every Django installation.

If you write a template library that isn't tied to any particular models/views, it's perfectly OK to have a Django app package that only contains a `templatetags` package.

There's no limit on how many modules you put in the `templatetags` package. Just keep in mind that a `{% load %}` statement will load tags/filters for the given Python module name, not the name of the app.

Once you've created that Python module, you'll just have to write a bit of Python code, depending on whether you're writing filters or tags.

To be a valid tag library, the module contain a module-level variable named `register` that is a `template.Library` instance, in which all the tags and filters are registered. So, near the top of your module, put the following:

```
from django import template

register = template.Library()
```

Behind the scenes

For a ton of examples, read the source code for Django’s default filters and tags. They’re in `django/template/defaultfilters.py` and `django/template/defaulttags.py`, respectively.

The apps in `django.contrib` also contain numerous examples.

Writing custom template filters

Custom filters are just Python functions that take one or two arguments:

- The value of the variable (input).
- The value of the argument, which this can have a default value, or be left out altogether.

For example, in the filter `{{ var|foo:"bar" }}`, the filter `foo` would be passed the variable `var` and the argument `"bar"`.

Filter functions should always return something. They shouldn’t raise exceptions and should fail silently. If there’s an error, they should return either the original input or an empty string — whichever makes more sense.

Here’s an example filter definition:

```
def cut(value, arg):
    "Removes all values of arg from the given string"
    return value.replace(arg, '')
```

And here’s an example of how that filter would be used:

```
{{ somevariable|cut:"0" }}
```

Most filters don’t take arguments. In this case, just leave the argument out of your function:

```
def lower(value): # Only one argument.
    "Converts a string into all lowercase"
    return value.lower()
```

When you’ve written your filter definition, you need to register it with your `Library` instance, to make it available to Django’s template language:

```
register.filter('cut', cut)
register.filter('lower', lower)
```

The `Library.filter()` method takes two arguments:

1. The name of the filter (a string).
2. The compilation function (a Python function, not the name of the function).

If you’re using Python 2.4 or above, you can use `register.filter()` as a decorator instead:

```
@register.filter(name='cut')
def cut(value, arg):
    return value.replace(arg, '')

@register.filter
def lower(value):
    return value.lower()
```

If you leave off the `name` argument, as in the second example above, Django will use the function’s name as the filter name.

Writing custom template tags

Tags are more complex than filters, because tags can do nearly anything.

A quick overview

Above, this chapter describes how the template system works in a two-step process: compiling and rendering. To define a custom template tag, you need to tell Django how to manage both steps when it gets to your tag.

When Django compiles a template, it splits the raw template text into ''nodes''. Each node is an instance of `django.template.Node` and has a `render()` method. Thus, a compiled template is simply a list of `Node` objects.

When you call `render()` on a compiled template, the template calls `render()` on each `Node` in its node list, with the given context. The results are all concatenated together to form the output of the template.

Thus, to define a custom template tag, you specify how the raw template tag is converted into a `Node` (the compilation function), and what the node's `render()` method does.

Writing the compilation function

For each template tag the template parser encounters, it calls a Python function with the tag contents and the parser object itself. This function is responsible for returning a `Node` instance based on the contents of the tag.

For example, let's write a template tag, `{% current_time %}`, that displays the current date/time, formatted according to a parameter given in the tag, in `strftime` syntax (see http://www.python.org/doc/current/lib/module-time.html#l2h-1941). It's a good idea to decide the tag syntax before anything else. In our case, let's say the tag should be used like this:

```
<p>The time is {% current_time "%Y-%m-%d %I:%M %p" %}.</p>
```

Note

Yes, this template tag is redundant; Django's default `{% now %}` tag does the same task with simpler syntax. This one's just for an example.

The parser for this function should grab the parameter and create a `Node` object:

```
from django import template

def do_current_time(parser, token):
    try:
        # split_contents() knows not to split quoted strings.
        tag_name, format_string = token.split_contents()
    except ValueError:
        raise template.TemplateSyntaxError("%r tag requires a single argument" % token.
contents[0])
    return CurrentTimeNode(format_string[1:-1])
```

There's actually a lot going here:

- `parser` is the template parser object. We don't need it in this example.
- `token.contents` is a string of the raw contents of the tag. In our example, it's `'current_time "%Y-%m-%d %I:%M %p"'`.
- The `token.split_contents()` method separates the arguments on spaces while keeping quoted strings together. The more straightforward `token.contents.split()` wouldn't be as robust, as it would naively split on *all* spaces, including those within quoted strings. It's a good idea to always use `token.split_contents()`.
- This function is responsible for raising `django.template.TemplateSyntaxError`, with helpful messages, for any syntax error.
- Don't hard-code the tag's name in your error messages, because that couples the tag's name to your function. `token.contents.split()[0]` will ''always'' be the name of your tag — even when the tag has no arguments.
- The function returns a `CurrentTimeNode` (which we'll create below) containing everything the node needs to know about this tag. In this case, it just passes the argument — `"%Y-%m-%d %I:%M %p"`. The leading and trailing quotes from the template tag are removed with `format_string[1:-1]`.
- Template tag compilation functions **must** return a `Node` subclass; any other return value is an error.
- The parsing is very low-level. We've experimented with writing small frameworks on top of this parsing system (using techniques such as EBNF grammars) but those experiments made the template engine too slow. Low level is fast.

Writing the template node

The second step in writing custom tags is to define a `Node` subclass that has a `render()` method. Continuing the above example, we need to define `CurrentTimeNode`:

```
import datetime

class CurrentTimeNode(template.Node):

    def __init__(self, format_string):
        self.format_string = format_string

    def render(self, context):
        return datetime.datetime.now().strftime(self.format_string)
```

These two functions (`__init__` and `render`) map directly to the two steps in template processing (compilation and rendering). Thus, the initialization function only needs to store the format string for later use, and the `render()` function does the real work.

Like template filters, these rendering functions should fail silently instead of raising errors. The only time that template tags are allowed to raise errors is at compilation time.

Registering the tag

Finally, you need to register the tag with your module's `Library` instance, as explained in "Writing custom template filters" above:

```
register.tag('current_time', do_current_time)
```

The `tag()` method takes two arguments:

1. The name of the template tag (string). If this is left out, the name of the compilation function will be used.
2. The compilation function.

As with filter registration, it is also possible to use this as a decorator in Python 2.4 and above:

```
@register.tag(name="current_time")
def do_current_time(parser, token):
    # ...

@register.tag
def shout(parser, token):
    # ...
```

If you leave off the `name` argument, as in the second example above, Django will use the function's name as the tag name.

Setting a variable in the context

The above example simply output a value. Often it's useful to set template variables instead of outputting values. That way, template authors can simply use the values that your template tags create.

To set a variable in the context, just use dictionary assignment on the context object in the `render()` method. Here's an updated version of `CurrentTimeNode` that sets a template variable `current_time` instead of outputting it:

```
class CurrentTimeNode2(template.Node):

    def __init__(self, format_string):
        self.format_string = format_string

    def render(self, context):
        context['current_time'] = datetime.datetime.now().strftime(self.format_string)
        return ''
```

Note that `render()` returns the empty string; `render()` should always return string output, so if all the template tag does is set a variable, `render()` should return an empty string.

Here's how you'd use this new version of the tag:

```
{% current_time "%Y-%M-%d %I:%M %p" %}
<p>The time is {{ current_time }}.</p>
```

But, there's a problem with `CurrentTimeNode2`: the variable name `current_time` is hard-coded. This means you'll need to make sure your template doesn't use `{{ current_time }}` anywhere else, because the `{% current_time %}` will blindly overwrite that variable's value.

A cleaner solution is to make the template tag specify the name of the output variable, like so:

```
{% get_current_time "%Y-%M-%d %I:%M %p" as my_current_time %}
<p>The current time is {{ my_current_time }}.</p>
```

To do that, you'll need to refactor both the compilation function and the `Node` class, like so:

```
import re
```

```
class CurrentTimeNode3(template.Node):

    def __init__(self, format_string, var_name):
        self.format_string = format_string
        self.var_name = var_name

    def render(self, context):
        context[self.var_name] = datetime.datetime.now().strftime(self.format_string)
        return ''

def do_current_time(parser, token):
    # This version uses a regular expression to parse tag contents.
    try:
        # Splitting by None == splitting by spaces.
        tag_name, arg = token.contents.split(None, 1)
    except ValueError:
        raise template.TemplateSyntaxError("%r tag requires arguments" % token.contents[0])

    m = re.search(r'(.*?) as (\w+)', arg)
    if m:
        format_string, var_name = m.groups()
    else:
        raise template.TemplateSyntaxError("%r tag had invalid arguments" % tag_name)

    if not (format_string[0] == format_string[-1] and format_string[0] in ('"', "'")):
        raise template.TemplateSyntaxError("%r tag's argument should be in quotes" % tag_name)

    return CurrentTimeNode3(format_string[1:-1], var_name)
```

Now, `do_current_time()` grabs the format string and the variable name, passing both to `CurrentTimeNode3`.

Parsing until another block tag

Template tags can work as blocks containing other tags. For example, the standard `{% comment %}` tag hides everything until `{% endcomment %}`.

To create a template tag like this, use `parser.parse()` in your compilation function.

Here's how the standard `{% comment %}` tag is implemented:

```
def do_comment(parser, token):
    nodelist = parser.parse(('endcomment',))
    parser.delete_first_token()
    return CommentNode()

class CommentNode(template.Node):
    def render(self, context):
        return ''
```

`parser.parse()` takes a tuple of names of block tags to parse until. It returns an instance of `django.template.NodeList`, which is a list of all `Node` objects that the parser encountered *before* it encountered any of the tags named in the tuple.

So in the above example, `nodelist` is a list of all nodes between the `{% comment %}` and `{% endcomment %}`, not counting `{% comment %}` and `{% endcomment %}` themselves.

After `parser.parse()` is called, the parser hasn't yet "consumed" the `{% endcomment %}` tag, so the code needs to explicitly call `parser.delete_first_token()` to prevent that tag from being processed twice.

Then, `CommentNode.render()` simply returns an empty string. Anything between `{% comment %}` and `{% endcomment %}` is ignored.

Parsing until another block tag and saving contents

In the previous example, `do_comment()` discarded everything between `{% comment %}` and `{% endcomment %}`. Instead of doing that, it's possible to do something with the code between block tags.

For example, here's a custom template tag, `{% upper %}`, that capitalizes everything between itself and `{% endupper %}`:

```
{% upper %}
    This will appear in uppercase, {{ your_name }}.
{% endupper %}
```

As in the previous example, we'll use `parser.parse()`. This time, we pass the resulting `nodelist` to the `Node`:

```
@register.tag
def do_upper(parser, token):
    nodelist = parser.parse(('endupper',))
    parser.delete_first_token()
    return UpperNode(nodelist)

class UpperNode(template.Node):

    def __init__(self, nodelist):
        self.nodelist = nodelist

    def render(self, context):
        output = self.nodelist.render(context)
        return output.upper()
```

The only new concept here is the `self.nodelist.render(context)` in `UpperNode.render()`.

For more examples of complex rendering, see the source code for `{% if %}`, `{% for %}`, `{% ifequal %}` and `{% ifchanged %}`. They live in `django/template/defaulttags.py`.

Shortcut for simple tags

Many template tags take a single argument — a string or a template variable reference — and return a string after doing some processing based solely on the input argument and some external information. For example, the `current_time` tag we wrote above is of this variety: we give it a format string, it returns the time as a string.

To ease the creation of the types of tags, Django provides a helper function, `simple_tag`. This function, which is a method of `django.template.Library`, takes a function that accepts one argument, wraps it in a `render` function and the other necessary bits mentioned above and registers it with the template system.

Our earlier `current_time` function could thus be written like this:

```
def current_time(format_string):
    return datetime.datetime.now().strftime(format_string)

register.simple_tag(current_time)
```

In Python 2.4, the decorator syntax also works:

```
@register.simple_tag
def current_time(token):
    ...
```

A couple of things to notice about the `simple_tag` helper function:

- Only the (single) argument is passed into our function.
- Checking for the required number of arguments has already been done by the time our function is called, so we don't need to do that.
- The quotes around the argument (if any) have already been stripped away, so we just receive a plain string.

Inclusion tags

Another common type of template tag is the type that displays some data by rendering *another* template.

For example, Django's admin interface uses custom template tags to display the buttons along the bottom of the "add/change" form pages. Those buttons always look the same, but the link targets change depending on the object being edited. They're a perfect case for using a small template that is filled with details from the current object.

These sorts of tags are called **inclusion tags**.

Writing inclusion tags is probably best demonstrated by example. Let's write a tag that outputs a list of choices for a simple multiple-choice `Poll` object. We'll use the tag like this:

```
{% show_results poll %}
```

..and the output will be something like this:

```
<ul>
```

```
  <li>First choice</li>
  <li>Second choice</li>
  <li>Third choice</li>
</ul>
```

First, we define the function that takes the argument and produces a dictionary of data for the result. Notice that we only need to return a dictionary, not anything more complex. This will be used as the context for the template fragment:

```
def show_results(poll):
    choices = poll.choice_set.all()
    return {'choices': choices}
```

Next, we create the template used to render the tag's output. Following our example, the template is very simple:

```
<ul>
{% for choice in choices %}
    <li> {{ choice }} </li>
{% endfor %}
</ul>
```

Finally, we create and register the inclusion tag by calling the `inclusion_tag()` method on a `Library` object.

Following our example, if the above template is in a file called `polls/result_snippet.html`, we'd register the tag like this:

```
register.inclusion_tag('polls/result_snippet.html')(show_results)
```

As always, Python 2.4 decorator syntax works as well, so we could have instead written:

```
@register.inclusion_tag('results.html')
def show_results(poll):
    ...
```

Sometimes, your inclusion tags need access to the context in the parent template.

To solve this, Django provides a `takes_context` option for inclusion tags. If you specify `takes_context` in creating a template tag, the tag will have no required arguments, and the underlying Python function will have one argument — the template context as of when the tag was called.

For example, say you're writing an inclusion tag that will always be used in a context that contains `home_link` and `home_title` variables that point back to the main page. Here's what the Python function would look like:

```
@register.inclusion_tag('link.html', takes_context=True)
def jump_link(context):
    return {
        'link': context['home_link'],
        'title': context['home_title'],
    }
```

Note

The first parameter to the function *must* be called `context`.

The template `link.html` might contain:

```
Jump directly to <a href="{{ link }}">{{ title }}</a>.
```

Then, any time you want to use that custom tag, load its library and call it without any arguments, like so:

```
{% jump_link %}
```

Note that when you're using `takes_context=True`, there's no need to pass arguments to the template tag. It automatically gets access to the context.

Writing custom template loaders

Django's built-in template loaders will usually cover all your template-loading needs, but it's pretty easy to write your own if you need special loading logic.

A template loader — that is, each entry in the `TEMPLATE_LOADERS` settings — is expected to be a callable with this interface:

```
load_template_source(template_name, template_dirs=None)
```

The `template_name` argument is the name of the template to load (as passed to `loader.get_template()` or `loader.select_template()`), and `template_dirs` is an optional list of directories to search instead of `TEMPLATE_DIRS`.

If a loader is able to successfully load a template, it should return a tuple: `(template_source, template_path)`. Here, `template_source` is the template string which will be compiled by the template engine, and `template_path` is the path the template was loaded from. That path might be shown to the user for debugging purposes, so it should quickly identify where the template was loaded from.

If the loader is unable to load a template, it should raise `django.template.TemplateDoesNotExist`.

Each loader function should also have an `is_usable` function attribute. This is a boolean that informs the template engine whether or not this loader is available in the current Python installation.

For example, the eggs loader (which is capable of loading templates from Python eggs) sets `is_usable` to `False` if the `pkg_resources` module isn't installed, because `pkg_resources` is necessary to read data from eggs.

An example should help clarify all of this. Here's a template loader function that can load templates from a ZIP file. It uses a custom setting, `TEMPLATE_ZIP_FILES` as a search path instead of `TEMPLATE_DIRS`, and expects each item on that path to be a ZIP file containing templates:

```
import zipfile
from django.conf import settings
from django.template import TemplateDoesNotExist

def load_template_source(template_name, template_dirs=None):
    """Template loader that loads templates from a ZIP file."""

    # Lookup ZIP file list from settings if it's not already given.
    if template_zipfiles is None:
        template_zipfiles = getattr(settings, "TEMPLATE_ZIP_FILES", [])

    # Try each ZIP file in TEMPLATE_ZIP_FILES.
    for fname in template_zipfiles:
        try:
            z = zipfile.ZipFile(fname)
            source = z.read(template_name)
        except (IOError, KeyError):
            continue

        # We found a template, so return the source.
        template_path = "%s:%s" % (fname, template_name)
        return (source, template_path)

    # If we reach here, the template couldn't be loaded
    raise TemplateDoesNotExist(template_name)

# This loader is always usable (since zipfile is a Python standard library function)
load_template_source.is_usable = True
```

The only step left if we wanted to use this loader is to add it to the `TEMPLATE_LOADERS` setting. If we put this code in a module called `myproject.zip_loader`, then we'd add `myproject.zip_loader.load_template_source` to `TEMPLATE_LOADERS`.

Using the built-in template reference

Django's admin interface includes a complete reference of all template tags and filters available for a given site. It's designed to be a tool that Django programmers give to template developers. To see it, go to your admin interface and click the "Documentation" link in the upper right of the page.

The reference is divided into 4 sections: tags, filters, models, and views.

The **tags** and **filters** sections describe all the built-in tags (in fact, the tag and filter references below come directly from those pages) as well as any custom tag or filter libraries available.

The **views** page is the most valuable. Each URL in your site has a separate entry here, and clicking on a URL will show you:

- The name of the view function that generates that view.
- A short description of what the view does.

- The **context**, or a list of variables available in the view's template.
- The name of the template or templates that are used for that view.

Each view documentation page also has a bookmarklet that you can use to jump from any page to the documentation page for that view.

Because Django-powered sites usually use database objects, the **models** section of the documentation page describes each type of object in the system along with all the fields available on that object.

Taken together, the documentation pages should tell you every tag, filter, variable and object available to you in a given template.

Configuring the template system in standalone mode

Note

This section is only of interest to people trying to use the template system as an output component in another application. If you are using the template system as part of a Django application, nothing here applies to you.

Normally, Django will load all the configuration information it needs from its own default configuration file, combined with the settings in the module given in the `DJANGO_SETTINGS_MODULE` environment variable. But if you're using the template system independently of the rest of Django, the environment variable approach isn't very convenient, because you probably want to configure the template system in line with the rest of your application rather than dealing with settings files and pointing to them via environment variables.

To solve this problem, you need to use the manual configuration option described in Appendix XXX.

Simply import the appropriate pieces of the template system and then, *before* you call any of the template functions, call `django.conf.settings.configure()` with any settings you wish to specify.

You might want to consider setting at least `TEMPLATE_DIRS` (if you are going to use template loaders), `DEFAULT_CHARSET` (although the default of `utf-8` is probably fine) and `TEMPLATE_DEBUG`. All available settings are described in the Chapter XXX, and any setting starting with *TEMPLATE_* is of obvious interest.

« previous ◊ table of contents ◊ next »

Chapter 11: Generating non-HTML content

Usually when we talk about developing web sites, we're talking about producing some flavor of HTML. Of course, there's a lot more to the web than HTML, though; we use the web to distribute all kinds of content, not just HTML.

Until this point, we've focused just on the common case of HTML production, but in this chapter we'll take a detour and look at using Django to produce other types of content.

Django has convenient built-in tools that you can use to produce some common non-HTML content:

- RSS/Atom syndication feeds.
- Sitemaps — consumed by Google, Yahoo and Microsoft's search engines.
- JSON and XML serialized representations of models (usually used for AJAX functions).

We'll cover each of those tools a little later on, but first, some basics.

The basics

Remember this from Chapter 3?

> A view function, or view for short, is simply a Python function that takes a Web request and returns a Web response. This response can be the HTML contents of a Web page, or a redirect, or a 404 error, or an XML document, or an image…or anything, really.

More formally, a Django view function *must*:

- Accept an `HttpRequest` instance as its first argument, and
- return an `HttpResponse` instance.

The key to returning non-HTML content from a view lies in the `HttpResponse` class, and specifically the `mimetype` constructor argument. By tweaking the mime-type, we can indicate to the browser that we've returned an object of a different type.

For a very simple example, let's look at a view that returns a PNG image. To keep things simple, we'll just read the file off the disk:

```
from django.http import HttpResponse

def my_image(request):
    image_data = open("/path/to/my/image.png", "rb").read()
    return HttpResponse(image_data, mimetype="image/png")
```

That's it! If you replace the image path in the `open()` call with a path to a real image, you can use this very simple view to serve an image, and the browser will display it correctly.

The other important thing to keep in mind is that `HttpResponse` objects implement Python's standard file API. This means that you can pass in an `HttpResponse` instance to any place Python (or a third-party library) expects a file.

For an example of how that works, let's take a look at producing CSV with Django.

Producing CSV

CSV is a simple data format usually used by spreadsheet software. It's basically a series of table rows, with each cell in the row separated by commas (CSV stands for "Comma Separated Values"). For example, here's a list of the number of "unruly" airline passengers over the last 10 years, as compiled by the FAA:

```
Year,Unruly Airline Passengers
1995,146
1996,184
1997,235
1998,200
1999,226
2000,251
2001,299
```

```
2002,273
2003,281
2004,304
2005,203
```

Note

See http://www.faa.gov/data_statistics/passengers_cargo/unruly_passengers/ for the source of this data.

Unfortunately, CSV It's not a format that's ever been formally defined; different pieces of software produce and consume different variants of CSV, making it a bit tricky to use. Luckily, Python comes with a standard CSV library, `csv`, that is pretty much bulletproof.

The key to using this library with Django is that the `csv` module's CSV-creation capability acts on file-like objects, and Django's `HttpResponse` objects are file-like objects:

```
import csv
from django.http import HttpResponse

# Number of unruly passengers each year 1995 - 2005
UNRULY_PASSENGERS = [146,184,235,200,226,251,299,273,281,304,203]

def unruly_passengers_csv(request):
    # Create the HttpResponse object with the appropriate CSV header.
    response = HttpResponse(mimetype='text/csv')
    response['Content-Disposition'] = 'attachment; filename=unruly.csv'

    # Create the CSV writer using the HttpResponse as the "file"
    writer = csv.writer(response)
    writer.writerow(['Year', 'Unruly Airline Passengers'])
    for (year, num) in zip(range(1995, 2006), UNRULY_PASSENGERS):
        writer.writerow([year, num])

    return response
```

The code and comments should be pretty clear, but a few things deserve a mention:

- The response is given the `text/csv` mime-type. This tells browsers that the document is a CSV file, rather than an HTML file.
- The response gets an additional `Content-Disposition` header, which contains the name of the CSV file. This header (well, the "attachment" part) will instruct the browser to prompt for a location to save the file (instead of just displaying it). This filename is arbitrary; call it whatever you want. It'll be used by browsers in the "Save as..." dialogue
- Hooking into the CSV-generation API is easy: Just pass `response` as the first argument to `csv.writer`. The `csv.writer` function expects a file-like object, and `HttpResponse` objects fit the bill.
- For each row in your CSV file, call `writer.writerow`, passing it an iterable object such as a list or tuple.
- The CSV module takes care of quoting for you, so you don't have to worry about escaping strings with quotes or commas in them. Just pass information to `writerow()`, and it'll do the right thing.

You'll usually repeat this pattern — create an `HttpResponse` response object (with a special mime-type), pass it to something expecting a file, then return the response — any time you generate non-HTML content.

Let's look at a few more examples:

Generating PDFs

PDF (Portable Document Format) is a format developed by Adobe that's used to represent printable documents, complete with pixel-perfect formatting, embedded fonts, and 2D vector graphics. You can think of a PDF document as the digital equivalent of a printed document; indeed, PDFs are usually used when you need to give a document to someone else to print.

You can easily generate PDFs with Python and Django thanks to the excellent excellent open-source ReportLab library (http://www.reportlab.org/rl_toolkit.html).

The advantage of generating PDF files dynamically is that you can create customized PDFs for different purposes — say, for different users or different pieces of content.

For example, we used Django and ReportLab at KUSports.com to generate customized, printer-ready NCAA tournament brackets for people participating in a March Madness (college basketball) contest.

Installing ReportLab

Before you do any PDF generation, however, you'll need to install ReportLab. It's usually pretty simple: just download and install the library from http://www.reportlab.org/downloads.html.

The user guide (not coincidentally, a PDF file) at http://www.reportlab.org/rsrc/userguide.pdf has additional help on installation.

Note

If you're using a modern Linux distribution, you might want to check your package management utility before installing ReportLab by hand; most package repositories have added ReportLab.

For example, if you're using the (excellent) Ubuntu distribution, a simple `aptitude install python-reportlab` will do the trick nicely.

Test your installation by importing it in the Python interactive interpreter:

```
>>> import reportlab
```

If that command doesn't raise any errors, the installation worked.

Writing your view

Again, key to generating PDFs dynamically with Django is that the ReportLab API acts on file-like objects, and Django's `HttpResponse` objects are file-like objects.

Here's a "Hello World" example:

```
from reportlab.pdfgen import canvas
from django.http import HttpResponse

def hello_pdf(request):
    # Create the HttpResponse object with the appropriate PDF headers.
    response = HttpResponse(mimetype='application/pdf')
    response['Content-Disposition'] = 'attachment; filename=hello.pdf'

    # Create the PDF object, using the response object as its "file."
    p = canvas.Canvas(response)

    # Draw things on the PDF. Here's where the PDF generation happens.
    # See the ReportLab documentation for the full list of functionality.
    p.drawString(100, 100, "Hello world.")

    # Close the PDF object cleanly, and we're done.
    p.showPage()
    p.save()
    return response
```

Like above, a few notes are in order:

- Here we use the `application/pdf` mime-type. This tells browsers that the document is a PDF file, rather than an HTML file. If you leave this off, browsers will probably interpret the output as HTML, which will result in scary gobbledygook in the browser window.
- Hooking into the ReportLab API is easy: Just pass `response` as the first argument to `canvas.Canvas`. The `Canvas` class expects a file-like object, and `HttpResponse` objects fit the bill.
- All subsequent PDF-generation methods are called on the PDF object (in this case, `p`) — not on `response`.
- Finally, it's important to call `showPage()` and `save()` on the PDF file (or else you'll end up with a corrupted PDF file).

Complex PDFs

If you're creating a complex PDF document with ReportLab, consider using the `cStringIO` library as a temporary holding place for your PDF file. The `cStringIO` library provides a file-like object interface that is particularly efficient (much more so than the naive `HttpResponse`-as-file implementation).

Here's the above "Hello World" example rewritten to use `cStringIO`:

```
from cStringIO import StringIO
from reportlab.pdfgen import canvas
from django.http import HttpResponse

def hello_pdf(request):
    # Create the HttpResponse object with the appropriate PDF headers.
    response = HttpResponse(mimetype='application/pdf')
    response['Content-Disposition'] = 'attachment; filename=hello.pdf'

    buffer = StringIO()

    # Create the PDF object, using the StringIO object as its "file."
    p = canvas.Canvas(buffer)

    # Draw things on the PDF. Here's where the PDF generation happens.
    # See the ReportLab documentation for the full list of functionality.
    p.drawString(100, 100, "Hello world.")

    # Close the PDF object cleanly.
    p.showPage()
    p.save()

    # Get the value of the StringIO buffer and write it to the response.
    response.write(buffer.getvalue())
    return response
```

Other possibilities

There's a whole world of other types of content you can generate in Python. Here are a few more ideas, and some pointers to libraries you could use to implement them:

- **Generating ZIP files**: Python's standard library ships with the `zipfile` module, which can both read and write compressed ZIP files. You could use it to provide on-demand archives of a bunch of files, or perhaps compress large documents when requested. You could similarly produce TAR files using the standard library `tarfile` module.

- **Dynamic image generation**: the Python Imaging Library (http://www.pythonware.com/products/pil/) is a fantastic toolkit for producing images (PNG, JPEG, GIF, and a whole lot more). You could use it to automatically scale down images into thumbnails, composite multiple images into a single frame, or even do web-based image processing.

- **Plots and charts**: there are a number of incredibly powerful Python plotting and charting libraries you could use to produce on-demand maps, charts, plots, and graphs. We can't possibly list them all, so here are a couple of the highlights:
  - `matplotlib` (http://matplotlib.sourceforge.net/), which can be used to produce the type of high-quality plots usually generated with MatLab or Mathematica.
  - `pygraphviz` (https://networkx.lanl.gov/wiki/pygraphviz), an interface to the Graphviz graph layout toolkit (http://graphviz.org/), used for generating structured diagrams of graphs and networks.

In general, any Python library capable of writing to a file can be hooked into Django; the possibilities really are endless.

Now that we've looked at the basics of generating non-HTML content, let's step up a level of abstraction. Django ships with some pretty nifty built-in tools for generating some common types of non-HTML content.

The syndication feed framework

Django comes with a high-level syndication-feed-generating framework that makes creating RSS and Atom feeds easy.

What's RSS? What's Atom?

RSS and Atom are both XML-based formats you can use to provide automatically updating "feeds" of your site's content. Read more about RSS at http://www.whatisrss.com/, and more about Atom at http://www.atomenabled.org/.

To create any syndication feed, all you have to do is write a short Python class. You can create as many feeds as you want.

Django also comes with a lower-level feed-generating API. Use this if you want to generate feeds outside of a Web context, or in some other lower-level way.

The high-level framework

Overview

The high-level feed-generating framework is a view that's hooked to `/feeds/` by default. Django uses the remainder of the URL (everything after `/feeds/`) to determine which feed to output.

To create a feed, just write a `Feed` class and point to it in your URLconf (see Chapters 3 and 8 fore more about URLconfs).

Initialization

To activate syndication feeds on your Django site, add this line to your URLconf:

```
(r'^feeds/(?P<url>.*)/$', 'django.contrib.syndication.views.feed', {'feed_dict': feeds}),
```

This tells Django to use the RSS framework to handle all URLs starting with `"feeds/"`. (You can change that `"feeds/"` prefix to fit your own needs.)

This URLconf line has an extra argument: `{'feed_dict': feeds}`. Use this extra argument to pass the syndication framework the feeds that should be published under that URL.

Specifically, `feed_dict` should be a dictionary that maps a feed's slug (short URL label) to its `Feed` class.

You can define the `feed_dict` in the URLconf itself. Here's a full example URLconf:

```
from django.conf.urls.defaults import *
from myproject.feeds import LatestEntries, LatestEntriesByCategory

feeds = {
    'latest': LatestEntries,
    'categories': LatestEntriesByCategory,
}

urlpatterns = patterns('',
    # ...
    (r'^feeds/(?P<url>.*)/$', 'django.contrib.syndication.views.feed',
        {'feed_dict': feeds}),
    # ...
)
```

The above example registers two feeds:

- The feed represented by `LatestEntries` will live at `feeds/latest/`.
- The feed represented by `LatestEntriesByCategory` will live at `feeds/categories/`.

Once that's set up, you just need to define the `Feed` classes themselves.

Feed classes

A `Feed` class is a simple Python class that represents a syndication feed. A feed can be simple (e.g., a "site news" feed, or a basic feed displaying the latest entries of a blog) or more complex (e.g., a feed displaying all the blog entries in a particular category, where the category is variable).

`Feed` classes must subclass `django.contrib.syndication.feeds.Feed`. They can live anywhere in your code tree.

A simple example

This simple example, taken from chicagocrime.org, describes a feed of the latest five news items:

```
from django.contrib.syndication.feeds import Feed
from chicagocrime.models import NewsItem

class LatestEntries(Feed):
    title = "Chicagocrime.org site news"
    link = "/sitenews/"
    description = "Updates on changes and additions to chicagocrime.org."

    def items(self):
        return NewsItem.objects.order_by('-pub_date')[:5]
```

The important things to notice here:

- The class subclasses `django.contrib.syndication.feeds.Feed`.
- `title`, `link` and `description` correspond to the standard RSS `<title>`, `<link>` and `<description>` elements, respectively.
- `items()` is simply a method that returns a list of objects that should be included in the feed as `<item>` elements. Although this example returns `NewsItem` objects using Django's database API, `items()` doesn't have to return model instances.

  You do get a few bits of functionality "for free" by using Django models, but `items()` can return any type of object you want.

There's just one more step. In an RSS feed, each `<item>` has a `<title>`, `<link>` and `<description>`. We need to tell the framework what data to put into those elements.

- To specify the contents of `<title>` and `<description>`, create Django templates (see Chapter 4) called `feeds/latest_title.html` and `feeds/latest_description.html`, where `latest` is the `slug` specified in the URLconf for the given feed.

  Note that the `.html` extension is required.

  The RSS system renders that template for each item, passing it two template context variables:

  `obj`

  The current object (one of whichever objects you returned in `items()`).

  `site`

  A `django.models.core.sites.Site` object representing the current site. This is useful for `{{ site.domain }}` or `{{ site.name }}`.

  If you don't create a template for either the title or description, the framework will use the template `"{{ obj }}"` by default — that is, the normal string representation of the object.

  You can also change the names of these two templates by specifying `title_template` and `description_template` as attributes of your `Feed` class.

- To specify the contents of `<link>`, you have two options. For each item in `items()`, Django first tries executing a `get_absolute_url()` method on that object. If that method doesn't exist, it tries calling a method `item_link()` in the `Feed` class, passing it a single parameter, `item`, which is the object itself.

  Both `get_absolute_url()` and `item_link()` should return the item's URL as a normal Python string.

- For the `LatestEntries` example above, we could have very simple feed templates. `latest_title.html` contains:

  ```
  {{ obj.title }}
  ```

  and `latest_description.html` contains:

  ```
  {{ obj.description }}
  ```

  It's almost *too* easy...

A complex example

The framework also supports more complex feeds, via parameters.

For example, chicagocrime.org offers an RSS feed of recent crimes for every police beat in Chicago. It'd be silly to create a separate `Feed` class for each police beat; that would violate the DRY (Don't Repeat Yourself) principle and would couple data to programming logic.

Instead, the syndication framework lets you make generic feeds that output items based on information in the feed's URL.

On chicagocrime.org, the police-beat feeds are accessible via URLs like this:

- `/rss/beats/0613/` — Returns recent crimes for beat 0613.
- `/rss/beats/1424/` — Returns recent crimes for beat 1424.

The slug here is `"beats"`. The syndication framework sees the extra URL bits after the slug — `0613` and `1424` — and gives you a hook to tell it what those URL bits mean, and how they should influence which items get published in the feed.

An example makes this clear. Here's the code for these beat-specific feeds:

```
from django.core.exceptions import ObjectDoesNotExist

class BeatFeed(Feed):
    def get_object(self, bits):
        # In case of "/rss/beats/0613/foo/bar/baz/", or other such
        # clutter, check that bits has only one member.
        if len(bits) != 1:
            raise ObjectDoesNotExist
        return Beat.objects.get(beat__exact=bits[0])
```

```
    def title(self, obj):
        return "Chicagocrime.org: Crimes for beat %s" % obj.beat

    def link(self, obj):
        return obj.get_absolute_url()

    def description(self, obj):
        return "Crimes recently reported in police beat %s" % obj.beat

    def items(self, obj):
        crimes =  Crime.objects.filter(beat__id__exact=obj.id)
        return crimes.order_by('-crime_date')[:30]
```

Here's the basic algorithm the RSS framework follows, given this class and a request to the URL `/rss/beats/0613/`:

1. The framework gets the URL `/rss/beats/0613/` and notices there's an extra bit of URL after the slug. It splits that remaining string by the slash character (`"/"`) and calls the `Feed` class' `get_object()` method, passing it the bits.

   In this case, bits is `['0613']`. For a request to `/rss/beats/0613/foo/bar/`, bits would be `['0613', 'foo', 'bar']`.

2. `get_object()` is responsible for retrieving the given beat, from the given `bits`.

   In this case, it uses the Django database API to retrieve the beat. Note that `get_object()` should raise `django.core.exceptions.ObjectDoesNotExist` if given invalid parameters. There's no `try`/`except` around the `Beat.objects.get()` call, because it's not necessary; that function raises `Beat.DoesNotExist` on failure, and `Beat.DoesNotExist` is a subclass of `ObjectDoesNotExist`. Raising `ObjectDoesNotExist` in `get_object()` tells Django to produce a 404 error for that request.

3. To generate the feed's `<title>`, `<link>` and `<description>`, Django uses the `title()`, `link()` and `description()` methods. In the previous example, they were simple string class attributes, but this example illustrates that they can be either strings *or* methods. For each of `title`, `link` and `description`, Django follows this algorithm:

   1. First, it tries to call a method, passing the `obj` argument, where `obj` is the object returned by `get_object()`.
   2. Failing that, it tries to call a method with no arguments.
   3. Failing that, it uses the class attribute.

4. Finally, note that `items()` in this example also takes the `obj` argument. The algorithm for `items` is the same as described in the previous step — first, it tries `items(obj)`, then `items()`, then finally an `items` class attribute (which should be a list).

Full documentation on all the methods and attributes of `Feed` classes is always available from the official Django documentation; see http://www.djangoproject.com/documentation/syndication/.

Specifying the type of feed

By default, feeds produced in by framework use RSS 2.0.

To change that, add a `feed_type` attribute to your `Feed` class:

```
from django.utils.feedgenerator import Atom1Feed

class MyFeed(Feed):
    feed_type = Atom1Feed
```

Note that you set `feed_type` to a class object, not an instance. Currently available feed types are:

Feed class	Format
`django.utils.feedgenerator.Rss201rev2Feed`	RSS 2.01 (default).
`django.utils.feedgenerator.RssUserland091Feed`	RSS 0.91.
`django.utils.feedgenerator.Atom1Feed`	Atom 1.0.

Enclosures

To specify enclosures, such as those used in creating podcast feeds, use the `item_enclosure_url`, `item_enclosure_length` and `item_enclosure_mime_type` hooks. For example:

```
from myproject.models import Song

class MyFeedWithEnclosures(MyFeed):
    title = "Example feed with enclosures"
    link = "/feeds/example-with-enclosures/"
    def items(self):
        return Song.objects.all()[:30]

    def item_enclosure_url(self, item):
        return item.song_url

    def item_enclosure_length(self, item):
        return item.song_length

    item_enclosure_mime_type = "audio/mpeg"
```

This assumes, of course, you've created a `Song` object with `song_url` and `song_length` (i.e. the size in bytes) fields.

Language

Feeds created by the syndication framework automatically include the appropriate `<language>` tag (RSS 2.0) or `xml:lang` attribute (Atom). This comes directly from your `LANGUAGE_CODE` setting.

URLs

The `link` method/attribute can return either an absolute URL (e.g. `"/blog/"`) or a URL with the fully-qualified domain and protocol (e.g. `"http://www.example.com/blog/"`). If `link` doesn't return the domain, the syndication framework will insert the domain of the current site, according to your `SITE_ID` setting.

Atom feeds require a `<link rel="self">` that defines the feed's current location. The syndication framework populates this automatically, using the domain of the current site according to the `SITE_ID` setting.

Publishing Atom and RSS feeds in tandem

Some developers like to make available both Atom *and* RSS versions of their feeds. That's easy to do with Django: Just create a subclass of your `feed` class and set the `feed_type` to something different. Then update your URLconf to add the extra versions.

Here's a full example:

```
from django.contrib.syndication.feeds import Feed
from chicagocrime.models import NewsItem
from django.utils.feedgenerator import Atom1Feed

class RssSiteNewsFeed(Feed):
    title = "Chicagocrime.org site news"
    link = "/sitenews/"
    description = "Updates on changes and additions to chicagocrime.org."

    def items(self):
        return NewsItem.objects.order_by('-pub_date')[:5]

class AtomSiteNewsFeed(RssSiteNewsFeed):
    feed_type = Atom1Feed
```

And the accompanying URLconf:

```
from django.conf.urls.defaults import *
from myproject.feeds import RssSiteNewsFeed, AtomSiteNewsFeed

feeds = {
    'rss': RssSiteNewsFeed,
    'atom': AtomSiteNewsFeed,
}

urlpatterns = patterns('',
    # ...
    (r'^feeds/(?P<url>.*)/$', 'django.contrib.syndication.views.feed',
        {'feed_dict': feeds}),
    # ...
)
```

The sitemap framework

Django also comes with a high-level Sitemap generating framework that's similar to the syndication framework.

A Sitemap is an XML file on your Web site that tells search-engine indexers how frequently your pages change and how "important" certain pages are in relation to other pages on your site. This information helps search engines index your site.

For more on Sitemaps, see http://www.sitemaps.org/.

The Django sitemap framework automates the creation of this XML file by letting you express this information in Python code. To create a sitemap, you just need to write a `Sitemap` class and point to it in your URLconf.

Installation

To install the sitemap app, follow these steps:

1. Add `'django.contrib.sitemaps'` to your `INSTALLED_APPS` setting.
2. Make sure `'django.template.loaders.app_directories.load_template_source'` is in your `TEMPLATE_LOADERS` setting. It's in there by default, so you'll only need to change this if you've changed that setting.
3. Make sure you've installed the sites framework (see Chapter 15).

Note

The sitemap application doesn't install any database tables. The only reason it needs to go into `INSTALLED_APPS` is so that the `load_template_source` template loader can find the default templates.

Initialization

To activate sitemap generation on your Django site, add this line to your URLconf:

```
(r'^sitemap.xml$', 'django.contrib.sitemaps.views.sitemap', {'sitemaps': sitemaps})
```

This tells Django to build a sitemap when a client accesses `/sitemap.xml`.

The name of the sitemap file is not important, but the location is. Search engines will only index links in your sitemap for the current URL level and below. For instance, if `sitemap.xml` lives in your root directory, it may reference any URL in your site. However, if your sitemap lives at `/content/sitemap.xml`, it may only reference URLs that begin with `/content/`.

The sitemap view takes an extra, required argument: `{'sitemaps': sitemaps}`. `sitemaps` should be a dictionary that maps a short section label (e.g., `blog` or `news`) to its `Sitemap` class (e.g., `BlogSitemap` or `NewsSitemap`). It may also map to an *instance* of a `Sitemap` class (e.g., `BlogSitemap(some_var)`).

Sitemap classes

A `Sitemap` class is a simple Python class that represents a "section" of entries in your sitemap. For example, one `Sitemap` class could represent all the entries of your weblog, while another could represent all of the events in your events calendar.

In the simplest case, all these sections get lumped together into one `sitemap.xml`, but it's also possible to use the framework to generate a sitemap index that references individual sitemap files, one per section. (See below.)

`Sitemap` classes must subclass `django.contrib.sitemaps.Sitemap`. They can live anywhere in your code tree.

For example, let's assume you have a blog system, with an `Entry` model, and you want your sitemap to include all the links to your individual blog entries. Here's how your sitemap class might look:

```
from django.contrib.sitemaps import Sitemap
from mysite.blog.models import Entry

class BlogSitemap(Sitemap):
    changefreq = "never"
    priority = 0.5

    def items(self):
        return Entry.objects.filter(is_draft=False)

    def lastmod(self, obj):
        return obj.pub_date
```

After looking at the syndication framework this should look pretty familiar:

- `changefreq` and `priority` are class attributes corresponding to `<changefreq>` and `<priority>` elements, respectively. They can be made callable as functions, as `lastmod` was in the example.
- `items()` is simply a method that returns a list of objects. The objects returned will get passed to any callable methods corresponding to a sitemap property (`location`, `lastmod`, `changefreq`, and `priority`).

- `lastmod` should return a Python `datetime` object.
- There is no `location` method in this example, but you can provide it in order to specify the URL for your object. By default, `location()` calls `get_absolute_url()` on each object and returns the result.

`Sitemap` methods/attributes

Like `Feed` classes, `Sitemap` members can be either methods or attributes; see the steps under "A complex example", above, for more about how this works.

A `Sitemap` class can define the following methods/attributes:

`items` (required)
Provides list of objects. The framework doesn't care what *type* of objects they are; all that matters is that these objects get passed to the `location()`, `lastmod()`, `changefreq()` and `priority()` methods.

`location` (optional)

Gives he absolute URL for a given object,

Here, "absolute URL" means a URL that doesn't include the protocol or domain. Examples:

- Good: `'/foo/bar/'`
- Bad: `'example.com/foo/bar/'`
- Bad: `'http://example.com/foo/bar/'`

If `location` isn't provided, the framework will call the `get_absolute_url()` method on each object as returned by `items()`.

`lastmod` (optional)
The object's "last modification" date, as a Python `datetime` object.

`changefreq` (optional)

How often the object changes. Possible values (as given by the Sitemaps spec) are:

- `'always'`
- `'hourly'`
- `'daily'`
- `'weekly'`
- `'monthly'`
- `'yearly'`
- `'never'`

`priority` (optional)
A suggested indexing priority, between `0.0` and `1.0`. The default priority of a page is `0.5`; see the sitemaps.org documentation for more about how `priority` works.

Shortcuts

The sitemap framework provides a couple convenience classes for common cases:

`FlatPageSitemap`

The `django.contrib.sitemaps.FlatPageSitemap` class looks at all flat pages defined for the current site and creates an entry in the sitemap. These entries include only the `location` attribute — not `lastmod`, `changefreq` or `priority`.

See Chapter 15 for more about flat pages.

`GenericSitemap`

The `GenericSitemap` class works with any generic views (see Chapter 9) you already have.

To use it, create an instance, passing in the same `info_dict` you pass to the generic views. The only requirement is that the dictionary have a `queryset` entry. It may also have a `date_field` entry that specifies a date field for objects retrieved from the `queryset`. This will be used for the `lastmod` attribute in the generated sitemap. You may also pass `priority` and `changefreq` keyword arguments to the `GenericSitemap` constructor to specify these attributes for all URLs.

Here's an example of a URLconf using both `FlatPageSitemap` and `GenericSiteMap` (with the hypothetical `Entry` object from above):

```
from django.conf.urls.defaults import *
from django.contrib.sitemaps import FlatPageSitemap, GenericSitemap
from mysite.blog.models import Entry

info_dict = {
    'queryset': Entry.objects.all(),
    'date_field': 'pub_date',
}

sitemaps = {
    'flatpages': FlatPageSitemap,
    'blog': GenericSitemap(info_dict, priority=0.6),
}

urlpatterns = patterns('',
    # some generic view using info_dict
    # ...

    # the sitemap
    (r'^sitemap.xml$', 'django.contrib.sitemaps.views.sitemap', {'sitemaps': sitemaps})
)
```

Creating a sitemap index

The sitemap framework also has the ability to create a sitemap index that references individual sitemap files, one per each section defined in your `sitemaps` dictionary. The only differences in usage are:

- You use two views in your URLconf: `django.contrib.sitemaps.views.index` and `django.contrib.sitemaps.views.sitemap`.
- The `django.contrib.sitemaps.views.sitemap` view should take a `section` keyword argument.

Here is what the relevant URLconf lines would look like for the example above:

```
(r'^sitemap.xml$', 'django.contrib.sitemaps.views.index', {'sitemaps': sitemaps})
(r'^sitemap-(?P<section>.+).xml$', 'django.contrib.sitemaps.views.sitemap', {'sitemaps':
sitemaps})
```

This will automatically generate a `sitemap.xml` file that references both `sitemap-flatpages.xml` and `sitemap-blog.xml`. The `Sitemap` classes and the `sitemaps` dictionary don't change at all.

Pinging Google

You may want to "ping" Google when your sitemap changes, to let it know to reindex your site. The framework provides a function to do just that: `django.contrib.sitemaps.ping_google()`.

Note

At the time this book was written, only Google responded to sitemap pings. However, it's quite likely that Yahoo and/or Microsoft will soon support these pings as well.

At that time, we'll likely change the name of `ping_google()` to something like `ping_search_engines()`, so make sure to check the latest sitemap documentation at http://www.djangoproject.com/documentation/sitemaps/.

`ping_google()` takes an optional argument, `sitemap_url`, which should be the absolute URL of your site's sitemap (e.g., `'/sitemap.xml'`). If this argument isn't provided, `ping_google()` will attempt to figure out your sitemap by performing a reverse looking in your URLconf.

`ping_google()` raises the exception `django.contrib.sitemaps.SitemapNotFound` if it cannot determine your sitemap URL.

One useful way to call `ping_google()` is from a model's `save()` method:

```
from django.contrib.sitemaps import ping_google

class Entry(models.Model):
    # ...
    def save(self):
        super(Entry, self).save()
        try:
            ping_google()
        except Exception:
            # Bare 'except' because we could get a variety
            # of HTTP-related exceptions.
            pass
```

A more efficient solution, however, would be to call `ping_google()` from a `cron` script, or some other scheduled task. The function makes an HTTP request to Google’s servers, so you may not want to introduce that network overhead each time you call `save()`.

What’s next?

Next, we’ll continue to dig deeper into all the nifty built-in tools Django gives you. Chapter 12 looks at all the tools you need to provide user-customized sites: sessions, users, and authentication.

Onwards!

« previous ◊ table of contents ◊ next »

Chapter 12: Sessions, users, and registration

It's time for a confession: we've been deliberately ignoring an incredibly important aspect of web development prior to this point. So far, we've thought of the traffic visiting our sites as some faceless, anonymous mass hurtling itself against our carefully designed pages.

This isn't true, of course; the browsers hitting our sites have real humans behind them (some of the time, at least). That's a big thing to ignore: the Internet is at its best when it serves to connect *people*, not machines. If we're going to develop truly compelling sites, eventually we're going to have to deal with the bodies behind the browsers.

Unfortunately, it's not all that easy. HTTP is designed to be **stateless**; that is, each and every request happens in a vacuum. There's no persistence between one request and the next, and we can't count on any aspects of a request (IP address, user-agent, etc.) to consistently indicate successive requests from the same person.

Browser developers long ago recognized that HTTP's statelessness poses a huge problem for web developers, and thus **cookies** were born. A cookie is a small piece of information that browsers store on behalf of web servers; every time a browser requests a page from a certain server, it gives back the cookie that it initially received.

Cookies

Let's take a look how this might work. When you open your browser and type in `google.com`, your browser sends an HTTP request to Google that starts something like this:

```
GET / HTTP/1.1
Host: google.com
...
```

When Google replies, the HTTP response looks something like:

```
HTTP/1.1 200 OK
Content-Type: text/html
Set-Cookie: PREF=ID=5b14f22bdaf1e81c:TM=1167000671:LM=1167000671;
            expires=Sun, 17-Jan-2038 19:14:07 GMT;
            path=/; domain=.google.com
Server: GWS/2.1
...
```

Notice the `Set-Cookie` header. Your browser will store that cookie value (`PREF=ID=5b14f22bdaf1e81c:TM=1167000671:LM=1167000671`) and serve it back to Google every time you access the site. So the next time you access Google, your browser is going to send a request like this:

```
GET / HTTP/1.1
Host: google.com
Cookie: PREF=ID=5b14f22bdaf1e81c:TM=1167000671:LM=1167000671
...
```

Google then can use that `Cookie` value to know that you're the same person who accessed the site earlier. This value might, for example, be a key into a database that stores user information; Google could (and does) use it to display your name on the page.

Getting and setting cookies

When dealing with persistence in Django, most of the time you'll want to use the higher-level session and/or user frameworks discussed a little later on. However, we'll pause and look at how to read and write cookies in Django anyway. It should help you understand how the rest of the pieces discussed in the chapter actually work, and it'll come in handy if you ever need to play with cookies directly.

Reading cookies that are already set is incredibly simple: every request object has a `COOKIES` object that acts like a dictionary; you can use it to read any cookies that the browser has sent to the view:

```
def show_color(request):
    if "favorite_color" in request.COOKIES:
        return HttpResponse("Your favorite color is %s" % \
            request.COOKIES["favorite_color"])
    else:
        return HttpResponse("You don't have a favorite color.")
```

Writing cookies is slightly more complicated; you need to use the `set_cookie()` method on a `HttpResponse` object. Here's an example that sets the `favorite_color` cookie based on a `GET` parameter:

```
def set_color(request):
    if "favorite_color" in request.GET:

        # Create an HttpResponse object...
        response = HttpResponse("Your favorite color is now %s" % \
            request.GET["favorite_color"])

        # ... and set a cookie on the response
        response.set_cookie("favorite_color",
                            request.GET["favorite_color"])

    else:
        return HttpResponse("You didn't give a favorite color.")
```

You can also pass a number of optional arguments to `request.set_cookie()` that control aspects of the cookie:

Parameter	Default	Description
`max_age`	`None`	Age (in seconds) that the cookie should last. If `None`, the cookie will last only until the browser is closed.
`expires`	`None`	The actual date/time when the cookie should expire. Needs to be in the format `"Wdy, DD-Mth-YY HH:MM:SS GMT"`. If given, this overrides the `max_age` parameter.
`path`	`"/"`	The path prefix that this cookie is valid for. Browsers will only pass the cookie back to pages below this path prefix, so you can use this to prevent cookies from being sent to other sections of your site. This is especially useful when you don't control the top level of your site's domain.
`domain`	`None`	The domain that this cookie is valid for. You can use this to set a cross-domain cookie. For example, domain=".example.com" will set a cookie that is readable by the domains `www.example.com`, `www2.example.com` and `an.other.sub.domain.example.com`. If set to `None`, a cookie will only be readable by the domain that set it.
`secure`	`False`	If set to `True`, this instructs the browser to only return this cookie to pages accessed over HTTPS.

The mixed blessing of cookies

You might notice a number of potential problems with the way cookies work. Let's look at some of the more important ones:

- Cookies are essentially voluntary; browser don't guarantee storage of cookies. In fact, every browser on the planet will let you control your browser's policy for accepting cookies. If you want to see just how vital cookies are to the web, try turning on your browser's "prompt to accept every cookie" option. Even a big blue monster would fill up on all those cookies!

  This means, of course, that cookies are the definition of unreliability; developers should check that a user actually accepts cookies before relying on them.

  More importantly, you should *never* store important data in cookies. The web is filled with horror stories of developers who've stored unrecoverable information in browser cookies only to have that data purged by the browser for one reason or another.

- Cookies are not in any way secure. Because HTTP data is sent in cleartext, cookies are extremely vulnerable to snooping attacks. That is, an attacker snooping on the wire can intercept a cookie and read it. This means you should never store sensitive information in a cookie.

  There's an even more insidious attack known as a "man in the middle" attack, wherein an attacker intercepts a cookie and uses it to pose as another user. Chapter 20 discusses attacks of this nature in depth, as well as ways to prevent it.

- Cookies aren't even secure from their intended recipients. Most browsers provide easy ways to edit the content of individual cookies, and resourceful users can always use tools like mechanize to construct HTTP requests by hand.

  So you can't store data in cookies that might be sensitive to tampering. The canonical mistake in this scenario is storing something like `IsLoggedIn=1` in a cookie when a user logs in. You'd be amazed at the number of sites that make mistakes of this nature; it takes only a second to fool these sites' "security" systems.

Django's session framework

With all of these limitations and potential security holes, it's obvious that cookies and persistent sessions are another of those "pain points" in web development. Of course, Django's goal is to be an effective painkiller, so Django comes with a session framework designed to smooth over these difficulties for you.

This session framework lets you store and retrieve arbitrary data on a per-site-visitor basis. It stores data on the server side and abstracts the sending and receiving of cookies. Cookies use only a hashed session ID — not the data itself — thus protecting you from most of the common cookie problems.

Enabling sessions

Sessions are implemented via a piece of middleware (see Chapter 16) and a Django model. To enable sessions, you'll need to:

1. Edit your `MIDDLEWARE_CLASSES` setting and make sure `MIDDLEWARE_CLASSES` contains `'django.contrib.sessions.middleware.SessionMiddleware'`.
2. Make sure `'django.contrib.sessions'` is in your `INSTALLED_APPS` setting (and run `manage.py syncdb` if you have to add it).

The default skeleton settings created by `startproject` has both of these bits already installed, so unless you've removed them, you probably don't have to change anything to get sessions to work.

If you don't want to use sessions, you might want to remove the `SessionMiddleware` line from `MIDDLEWARE_CLASSES` and `'django.contrib.sessions'` from your `INSTALLED_APPS`. It'll only save you a very small amount of overhead, but every little bit counts.

Using sessions in views

When `SessionMiddleware` is activated, each `HttpRequest` object — the first argument to any Django view function — will have a `session` attribute, which is a dictionary-like object. You can read it and write to it in the same way you'd use a normal dictionary. For example, in a view you could do stuff like this:

```
# Set a session value:
request.session["fav_color"] = "blue"

# Get a session value -- this could be called in a different view,
# or many requests later (or both):
fav_color = request.session["fav_color"]

# Clear an item from the session:
del request.session["fav_color"]

# Check if the session has a given key:
if "fav_color" in request.session:
    ...
```

You can also use other mapping methods like `keys()` and `items()` on `request.session`.

There are a couple of simple rules for using Django's sessions effectively:

- Use normal Python strings as dictionary keys on `request.session` (as opposed to integers, objects, etc.). This is more of a convention than a hard-and-fast rule, but it's worth following.
- Session dictionary keys that begin with an underscore are reserved for internal use by Django. In practice the framework only uses a very small number of underscore-prefixed session variables, but unless you know what they all are (and are willing to keep up with any changes in Django itself), staying away from underscore prefixes will keep Django from interfering with your app.
- Don't override `request.session` with a new object, and don't access or set its attributes. Use it like a Python dictionary.

Let's take a look at a few quick examples. This simplistic view sets a `has_commented` variable to `True` after a user posts a comment. It doesn't let a user post a comment more than once:

```
def post_comment(request, new_comment):
    if request.session.get('has_commented', False):
        return HttpResponse("You've already commented.")
    c = comments.Comment(comment=new_comment)
    c.save()
    request.session['has_commented'] = True
    return HttpResponse('Thanks for your comment!')
```

This simplistic view logs in a "member" of the site:

```
def login(request):
    m = members.get_object(username__exact=request.POST['username'])
    if m.password == request.POST['password']:
        request.session['member_id'] = m.id
        return HttpResponse("You're logged in.")
    else:
        return HttpResponse("Your username and password didn't match.")
```

And this one logs a member out, according to `login()` above:

```
def logout(request):
    try:
        del request.session['member_id']
    except KeyError:
        pass
    return HttpResponse("You're logged out.")
```

Note

In practice, this is a lousy way of logging users in. The authentication framework discussed below handles this for you in a much more robust and useful manner; these examples are just here to provide easily understood examples.

Setting test cookies

As mentioned above, you can't rely on every browser accepting cookies. So, as a convenience, Django provides an easy way to test whether the user's browser accepts cookies. You just need to call `request.session.set_test_cookie()` in a view, and check `request.session.test_cookie_worked()` in a subsequent view — not in the same view call.

This awkward split between `set_test_cookie()` and `test_cookie_worked()` is necessary due to the way cookies work. When you set a cookie, you can't actually tell whether a browser accepted it until the browser's next request.

It's good practice to use `delete_test_cookie()` to clean up after yourself. Do this after you've verified that the test cookie worked.

Here's a typical usage example:

```
def login(request):

    # If we submitted the form...
    if request.method == 'POST':

        # Check that the test cookie worked (we set it below):
        if request.session.test_cookie_worked():

            # The test cookie worked, so delete it.
            request.session.delete_test_cookie()

            # In practice, we'd need some logic to check username/password
            # here, but since this is an example...
            return HttpResponse("You're logged in.")

        # The test cookie failed, so display an error message. If this
        # was a real site we'd want to display a more friendly message.
        else:
            return HttpResponse("Please enable cookies and try again.")

    # If we didn't post, send the test cookie along with the login form.
    request.session.set_test_cookie()
    return render_to_response('foo/login_form.html')
```

Note

Again, the built-in login and logout functions handle this check for you.

Using sessions outside of views

Internally, each session is just a normal Django model defined in `django.contrib.sessions.models`. Because it's a normal model, you can access sessions using the normal Django database API:

```
>>> from django.contrib.sessions.models import Session
>>> s = Session.objects.get_object(pk='2b1189a188b44ad18c35e113ac6ceead')
>>> s.expire_date
datetime.datetime(2005, 8, 20, 13, 35, 12)
```

You'll need to call `get_decoded()` to get the actual session data. This is necessary because the dictionary is stored in an encoded format:

```
>>> s.session_data
'KGRwMQpTJ19hdXRoX3VzZXJfaWQnCnAyCkkxCnMuMTExY2ZjODI2Yj...'
>>> s.get_decoded()
{'user_id': 42}
```

When sessions are saved

By default, Django only saves to the session database when the session has been modified — that is if any of its dictionary values have been assigned or deleted:

```
# Session is modified.
request.session['foo'] = 'bar'

# Session is modified.
del request.session['foo']

# Session is modified.
request.session['foo'] = {}

# Gotcha: Session is NOT modified, because this alters
# request.session['foo'] instead of request.session.
request.session['foo']['bar'] = 'baz'
```

To change this default behavior, set the `SESSION_SAVE_EVERY_REQUEST` setting to `True`. If `SESSION_SAVE_EVERY_REQUEST` is `True`, Django will save the session to the database on every single request, even if it wasn't changed.

Note that the session cookie is only sent when a session has been created or modified. If `SESSION_SAVE_EVERY_REQUEST` is `True`, the session cookie will be sent on every request.

Similarly, the `expires` part of a session cookie is updated each time the session cookie is sent.

Browser-length sessions vs. persistent sessions

You might have noticed above that the cookie Google sent use contained `expires=Sun, 17-Jan-2038 19:14:07 GMT;`. Cookies can optionally contain an expiration date which advises the browser on when to remove the cookie. If a cookie doesn't contain an expiration value, the browser will expire it when the user closes her browser window. You can control the session framework's behavior in this regard with the `SESSION_EXPIRE_AT_BROWSER_CLOSE` setting.

By default, `SESSION_EXPIRE_AT_BROWSER_CLOSE` is set to `False`, which means session cookies will be stored in users' browsers for `SESSION_COOKIE_AGE` seconds (which defaults to 2 weeks — 1209600 seconds). Use this if you don't want people to have to log in every time they open a browser.

If `SESSION_EXPIRE_AT_BROWSER_CLOSE` is set to `True`, Django will use browser-length cookies.

Other session settings

Besides the settings already mentioned, there are a few other settings that influence how Django's session framework uses cookies:

Setting	Explanation	Default
`SESSION_COOKIE_DOMAIN`	The domain to use for session cookies. Set this to a string such as `".lawrence.com"` for cross-domain cookies, or use `None` for a standard cookie.	`None`
`SESSION_COOKIE_NAME`	The name of the cookie to use for sessions. This can be any string.	`"sessionid"`
`SESSION_COOKIE_SECURE`	Whether to use a "secure" cookie for the session cookie. If this is set to `True`, the cookie will be marked as "secure," which means that browsers will ensure that the cookie is only sent via HTTPS.	`False`

Technical details

For the curious, here are a few technical notes about the inner workings of the session framework:

- The session dictionary accepts any **pickleable** Python object. See the documentation for Python's built-in `pickle` module for more information about how this works.

- Session data is stored in a database table named `django_session`.
- Session data is ready "lazily": if you never access `request.session`, Django won't hit that database table.
- Django only sends a cookie if it needs to. If you don't set any session data, it won't send a session cookie (unless `SESSION_SAVE_EVERY_REQUEST` is set to `True`).
- The Django sessions framework is entirely, and solely, cookie-based. It does not fall back to putting session IDs in URLs as a last resort, as some other tools (PHP, JSP) do.

  This is an intentional design decision. Putting sessions in URLs don't just make URLs ugly, they makes your site vulnerable to a certain form of session-ID theft via the "Referer" header.

If you're still curious, the source is pretty straightforward; look in `django.contrib.sessions` for the streight dope.

Users and authentication

So now we're halfway to linking browsers directly to Real People. Sessions give us a way of persisting data between multiple browser requests; the second part of the equation is using those sessions to for user login. Of course, we can't just trust that users are who they say they are, so we'll need to authenticate them along the way.

Naturally, Django provides tools to handle this common task (and many others). Django's user authentication system handles user accounts, groups, permissions and cookie-based user sessions. This system is often referred to as an "auth/auth" system — authentication and authorization. That name recognizes that dealing with users is often a two step process; we need to

1. verify (**authenticate**) that a user is who she claims to be (usually by checking a username and password against a database of users), and then
2. verify that the user is **authorized** to perform some given operation (usually checking against a table of permissions).

Following these needs, Django's auth/auth system consists of a number of parts:

- Users
- Permissions: binary (yes/no) flags designating whether a user may perform a certain task.
- Groups: a generic way of applying labels and permissions to more than one user.
- Messages: a simple way to queue and display system messages to users.
- Profiles: a mechanism to extend the user object with custom fields.

If you've used the admin tool (Chapter 6), you've already seen many of these tools, and if you've edited users or groups in the admin you've actually been editing data in the auth system's database tables.

Installation

Like the session tools, authentication support is bundled as a Django application in `django.contrib` which needs to be installed. Like the session system it's also installed by default, but if you've removed it you'll need to follow these steps to install it:

1. Make sure the session framework is installed (see above). Keeping track of users obviously requires cookies, and thus builds on the session framework.
2. Put `'django.contrib.auth'` in your `INSTALLED_APPS` setting and run `manage.py syncdb`.
3. Make sure that `'django.contrib.auth.middleware.AuthenticationMiddleware'` is in your `MIDDLEWARE_CLASSES` setting — *after* `SessionMiddleware`.

With that installation out of the way, we're ready to deal with users in view functions. The main interface you'll use to access users within a view is `request.user`; this is an object that represents the currently logged-in user. If the user isn't logged in, this will instead be an `AnonymousUser` object (see below for more details).

You can easily tell if a user is logged in with the `is_authenticated()` method:

```
if request.user.is_authenticated():
    # Do something for authenticated users.
else:
    # Do something for anonymous users.
```

Using users

Once you've got ahold of a user — often from `request.user`, but possibly through one of the other methods discussed below — you've got a number of fields methods available on that object. `AnonymousUser` objects emulate *some* of these fields and methods, but not all of them, so you should always check `user.is_authenticated()` before assuming you're dealing with a bona-fide user object.

Fields on `User`objects

Field	Description
`username`	Required. 30 characters or fewer. Alphanumeric characters only (letters, digits and underscores).
`first_name`	Optional. 30 characters or fewer.
`last_name`	Optional. 30 characters or fewer.
`email`	Optional. E-mail address.
`password`	Required. A hash of, and metadata about, the password (Django doesn't store the raw password). See the "Passwords" section below for more about this value
`is_staff`	Boolean. Designates whether this user can access the admin site.
`is_active`	Boolean. Designates whether this account can be used to log in. Set this flag to `False` instead of deleting accounts.
`is_superuser`	Boolean. Designates that this user has all permissions without explicitly assigning them.
`last_login`	A datetime of the user's last login. Is set to the current date/time by default.
`date_joined`	A datetime designating when the account was created. Is set to the current date/time by default when the account is created.

Methods on `User`objects

Method	Description
`is_authenticated()`	Always returns `True` for "real" `User` objects. This is a way to tell if the user has been authenticated. This does not imply any permissions, and doesn't check if the user is active - it only indicates that the user has sucessfully authenticated.
`is_anonymous()`	Returns `True` only for `AnonymousUser` objects (and `False` for "real" `User` objects). Generally, you should prefer using `is_authenticated()` to this method.
`get_full_name()`	Returns the `first_name` plus the `last_name`, with a space in between.
`set_password(passwd)`	Sets the user's password to the given raw string, taking care of the password hashing. This doesn't actually save the `User` object.
`check_password(passwd)`	Returns `True` if the given raw string is the correct password for the user. This takes care of the password hashing in making the comparison.
`get_group_permissions()`	Returns a list of permission strings that the user has through the groups she belongs to.
`get_all_permissions()`	Returns a list of permission strings that the user has, both through group and user permissions.
`has_perm(perm)`	Returns `True` if the user has the specified permission, where perm is in the format `"package.codename"`. If the user is inactive, this method will always return `False`.
`has_perms(perm_list)`	Returns `True` if the user has *all* of the specified permissions, If the user is inactive, this method will always return `False`.
`has_module_perms(appname)`	Returns `True` if the user has any permissions in the given `appname` If the user is inactive, this method will always return `False`.
`get_and_delete_messages()`	Returns a list of `Message` objects in the user's queue and deletes the messages from the queue.
`email_user(subj, msg)`	Sends an e-mail to the user. This email is sent from the `DEFAULT_FROM_EMAIL` setting. You can also pass a third argument, `from_email`, to override the from address on the email.
`get_profile()`	Returns a site-specific profile for this user; see the section on profiles, below, for more on this method

Finally, `User` objects have two many-to-many fields: `groups` and `permissions`. `User` objects can access their related objects in the same way as any other many-to-many field:

```
# Set a users groups:
myuser.groups = group_list

# Add a user to some groups:
myuser.groups.add(group1, group2,...)

# Remove a user from some groups:
myuser.groups.remove(group1, group2,...)

# Remove a user from all groups:
```

```
myuser.groups.clear()

# Permissions work the same way
myuser.permissions = permission_list
myuser.permissions.add(permission1, permission2, ...)
myuser.permissions.remove(permission1, permission2, ...)
myuser.permissions.clear()
```

Logging in and out

Django provides built-in view functions for handling logging in and out (and a few other nifty tricks), but before we get to those let's take a look at how to log users in and out "by hand". Django provides two functions to perform these actions in `django.contrib.auth`: `authenticate()` and `login()`.

To authenticate a given username and password, use `authenticate()`. It takes two keyword arguments, `username` and `password` , and it returns a `User` object if the password is valid for the given username. If the password is invalid, `authenticate()` returns `None`:

```
>>> from django.contrib import auth authenticate
>>> user = auth.authenticate(username='john', password='secret')
>>> if user is not None:
...     print "Correct!"
... else:
...     print "Oops, that's wrong!"
Oops, that's wrong!
```

To log a user in, in a view, use `login()`. It takes an `HttpRequest` object and a `User` object and saves the user's ID in the session, using Django's session framework.

This example shows how you might use both `authenticate()` and `login()` within a view function:

```
from django.contrib import auth

def login(request):
    username = request.POST['username']
    password = request.POST['password']
    user = auth.authenticate(username=username, password=password)
    if user is not None and user.is_active:
        # Correct password, and the user is marked "active"
        auth.login(request, user)
        # Redirect to a success page.
        return HttpResponseRedirect("/account/loggedin/")
    else:
        # Show an error page
        return HttpResponseRedirect("/account/invalid/")
```

To log out a user who has been logged, use `django.contrib.auth.logout()` within your view. It takes an `HttpRequest` object and has no return value:

```
from django.contrib import auth

def logout(request):
    auth.logout(request)
    # Redirect to a success page.
    return HttpResponseRedirect("/account/loggedout/")
```

Note that `logout()` doesn't throw any errors if the user wasn't logged in.

Logging in and out, the easy way

In practice, you'll usually not need to write your own login/logout functions; the auth system comes with a set of views for generically handling logging in and out.

The first step in using the authentication views is to wire 'em up in your URLconf. You'll need to add this snippet:

```
from django.contrib.auth.views import login, logout

urlpatterns = patterns('',
    # existing patterns here...
    (r'^accounts/login/$',  login)
    (r'^accounts/logout/$', logout)
)
```

`/accounts/login/` and `/accounts/logout/` are the default URLs that Django uses for these views, but you can put them anywhere you like with a little effort.

By default, the `login` view renders a template at `registration/login.html` (you can change this template name by passing an extra view argument `template_name`). This form needs to contain a `username` and a `password` field. A simple template might look like:

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

{% block content %}

  {% if form.errors %}
    <p class="error">Sorry, that's not a valid username or password</p>
  {% endif %}

  <form action='.' method='post'>
    <label for="username">User name:</label>
    <input type="text" name="username" value="" id="username">
    <label for="password">Password:</label>
    <input type="password" name="password" value="" id="password">

    <input type="submit" value="login" />
    <input type="hidden" name="next" value="{{ next }}" />
  <form action='.' method='post'>

{% endblock %}
```

If the user successfully logs in, she'll be redirected to `/accounts/profile/` by default. You can override this by providing a hidden field called `next` with the URL to redirect to after logging in. You can also pass this value as a `GET` parameter to the login view and it'll be automatically added to the context as a variable called `next` that you can insert into that hidden field.

The log out view works a little differently; by default it renders a template at `registration/logged_out.html` (which usually contains a "you've successfully logged out" message). However, you can call the view with an extra argument, `next_page`, which will instruct the view to redirect after a log out.

Limiting access to logged-in users

Of course, the reason we're going through all this trouble is so that we can limit access to parts of our site.

The simple, raw way to limit access to pages is to check `request.user.is_authenticated()` and redirect to a login page:

```
from django.http import HttpResponseRedirect

def my_view(request):
    if not request.user.is_authenticated():
        return HttpResponseRedirect('/login/?next=%s' % request.path)
    # ...
```

Or perhaps display an error message:

```
def my_view(request):
    if not request.user.is_authenticated():
        return render_to_response('myapp/login_error.html')
    # ...
```

As a shortcut, you can use the convenient `login_required` decorator:

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

@login_required
def my_view(request):
    # ...
```

`login_required` does the following:

- If the user isn't logged in, redirect to `/accounts/login/`, passing the current absolute URL in the query string as `next`. For example: `/accounts/login/?next=/polls/3/`.
- If the user is logged in, execute the view normally. The view code can then assume that the user is logged in.

Note

If you're into programming patterns, note that this decorator and the ones discussed below are examples of the "Guard" pattern. Aren't patterns fun?

Limiting access to users that pass a test

Limiting access based on certain permissions or some other test, or providing a different location for the log-in view works essentially the same way.

The raw way is to run your test on `request.user` in the view directly. For example, this view checks to make sure the user is logged in and has the permission `polls.can_vote` (see below for more about how permissions work):

```
def vote(request):
    if request.user.is_authenticated() and request.user.has_perm('polls.can_vote')):
        # vote here
    else:
        return HttpResponse("You can't vote in this poll.")
```

Again, Django provides a shortcut. This one is called `user_passes_test` which is actually a **decorator factory**: it takes arguments and generates a specialized decorator for your particular situation. For example:

```
def user_can_vote(user):
    return user.is_authenticated() and user.has_perm("polls.can_vote")

@user_passes_text(user_can_vote, login_url="/login/")
def vote(request):
    # Code here can assume a logged in user with the correct permission.
    ...
```

`user_passes_test` takes one required argument: a callable that takes a `User` object and returns `True` if the user is allowed to view the page. Note that `user_passes_test` does not automatically check that the `User` is authenticated; you should do that yourself.

In this example we're also showing the second optional argument, `login_url`, which lets you specify the URL for your login page (`/accounts/login/` by default).

Since it's a relatively common task to check whether a user has a particular permission, Django provides a shortcut for that case: the `permission_required()` decorator. Using this decorator, the earlier example can be written as:

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

@permission_required('polls.can_vote', login_url="/login/")
def vote(request):
    # ...
```

Note that `permission_required()` also takes an optional `login_url` parameter which also defaults to `'/accounts/login/'`.

Limiting access to generic views

One of the most frequently asked questions on the Django-users list deals with limiting access to a generic view. To pull this off, you'll need to write a thin wrapper around the view, and point your URLconf to your wrapper instead of the generic view itself:

```
from dango.contrib.auth.decorators import login_required
from django.views.generic.date_based import object_detail

@login_required
def limited_object_detail(*args, **kwargs):
    return object_detail(*args, **kwargs)
```

You can, of course, replace `login_required` with any of the other limiting decorators.

Managing users, permissions, and groups

The easiest way by far to manage the auth system is through the admin. Chapter 6 discusses how to use Django's admin to edit users and control their permissions and access, and most of the time you'll just use that interface.

However, there are low-level APIs you can delve into when you need absolute control.

Creating users

The basic way to create users is to use the `create_user` helper function:

```
>>> from django.contrib.auth.models import User
>>> user = User.objects.create_user(username='john',
...                                 email='jlennon@beatles.com',
...                                 password='glass onion')
```

At this point, `user` is a `User` instance ready to be saved to the database. You can continue to change its attributes before saving, too:

```
>>> user.is_staff = True
>>> user.save()
```

Changing passwords

You can change a password with `set_password()`:

```
>>> user = User.objects.get(username='john')
>>> user.set_password('goo goo goo joob')
>>> user.save()
```

Don't set the `password` attribute directly unless you know what you're doing; the password is actually stored as a **salted hash** and thus can't be edited directly.

More formally, the `password` attribute of a `User` object is a string in this format:

```
hashtype$salt$hash
```

That's a hash type, the salt and the hash itself, separated by the dollar-sign character.

`hashtype` is either `sha1` (default) or `md5` — the algorithm used to perform a one-way hash of the password. Salt is a random string used to salt the raw password to create the hash.

For example:

```
sha1$a1976$a36cc8cbf81742a8fb52e221aaeab48ed7f58ab4
```

The `User.set_password()` and `User.check_password()` functions handle the setting and checking of these values behind the scenes.

Is that some kind of drug?

No, a **salted hash** has nothing to do with marijuana; it's actually a common way to securly store passwords. A **hash** is a one-way crytographic function; that is, you can easily compute the hash of a given value, but it's nearly impossible to take a hash and reconstruct the original value.

If we stored passwords as plain text, anyone who got their hands on the password database would instantly know everyone's password. Storing passwords as hashes reduces the value of a compromised database.

However, an attacker with the password database could still run a **brute force** attack, hashing millions of passwords and comparing those hashes against the stored values. This might take some time, but less than you think — computers are incredibly fast.

Worse, there are publically available **rainbow tables** — databases of precomputed hashes of millions of passwords. With a rainbow table, an attacker can break most passwords in seconds.

Adding a **salt** — basically an initial random value — to the stored hash adds another layer of difficulty. Since the salt will differ from password to password, salts also prevent the use of a rainbow table, thus forcing attackers to fall back on a brute force attack — itself made more difficult by the extra entropy added to the hash by the salt.

While salted hashes aren't absolutely the most secure way of storing passwords, they're a good middle ground between security and convience.

Handling registration

We can use these low-level tools to create views that allow users to sign up. Nearly every developer wants to implement registration differently, so Django leaves writing a registration view up to you; luckily, it's pretty easy.

At its simplest, we could provide a small view that prompts for the required user information and creates those users. Django provides a built-in form you can use for this purpose, which we'll use in this example:

```
from django import oldforms as forms
from django.http import HttpResponseRedirect
from django.shortcuts import render_to_response
from django.contrib.auth.forms import UserCreationForm

def register(request):
    form = UserCreationForm()

    if request.method == 'POST':
        data = request.POST.copy()
        errors = form.get_validation_errors(data)
        if not errors:
            new_user = form.save()
            return HttpResponseRedirect("/accounts/created/")
    else:
        data, errors = {}, {}

    return render_to_response("registration/register.html", {
        'form' : forms.FormWrapper(form, data, errors)
    })
```

This assumes a template named `registration/register.html`; here's an example of what that template might look like:

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

{% block title %}Create an account{% endblock %}

{% block content %}
      <h1>Create an account</h1>
      <form action="." method="post">
        {% if form.error_dict %}
          <p class="error">Please correct the errors below.</p>
        {% endif %}

        {% if form.username.errors %}
          {{ form.username.html_error_list }}
        {% endif %}
        <label for="id_username">Username:</label> {{ form.username }}

        {% if form.password1.errors %}
          {{ form.password1.html_error_list }}
        {% endif %}
        <label for="id_password1">Password: {{ form.password1 }}

        {% if form.password2.errors %}
          {{ form.password2.html_error_list }}
        {% endif %}
        <label for="id_password2">Password (again): {{ form.password2 }}

        <input type="submit" value="Create the account" />
      </label>
{% endblock %}
```

Using authentication data in templates

The currently logged-in user and his/her permissions are made available in the template context when you use `RequestContext` (see Chapter 10).

Note

Technically, these variables are only made available in the template context if you use `RequestContext` *and* your `TEMPLATE_CONTEXT_PROCESSORS` setting contains `"django.core.context_processors.auth"`, which is default. Again, see Chapter 10 for the full skinny.

When using `RequestContext`, the current user — either a `User` instance or an``AnonymousUser`` instance — is stored in the template variable `{{ user }}`:

```
{% if user.is_authenticated %}
  <p>Welcome, {{ user.username }}. Thanks for logging in.</p>
{% else %}
  <p>Welcome, new user. Please log in.</p>
{% endif %}
```

This user's permissions are stored in the template variable `{{ perms }}`. This is is a template-friendly proxy of to a couple of permission methods. See the section on permissions, below, for more about what these methods map to.

There are two ways you can use this `perms` object. You can use something like `{{ perms.polls }}` to check if the user has *any* permissions some given app, or you can use something like `{{ perms.polls.can_vote }}` to check if the user has a specific permission.

Thus, you can check permissions in template `{% if %}` statements:

```
{% if perms.polls %}
  <p>You have permission to do something in the polls app.</p>
  {% if perms.polls.can_vote %}
    <p>You can vote!</p>
  {% endif %}
{% else %}
  <p>You don't have permission to do anything in the polls app.</p>
{% endif %}
```

The other bits: permissions, groups, messages, and profiles

There's a few other bits of the authentication framework that we've only dealt with in passing. Let's take a closer look at them:

Permissions

Permissions are a simple way to "mark" users and groups as being able to perform some action. It's usually used by the Django admin site, but you can easily use it in your own code.

The Django admin site uses permissions as follows:

- Access to view the "add" form and add an object is limited to users with the "add" permission for that type of object.
- Access to view the change list, view the "change" form and change an object is limited to users with the "change" permission for that type of object.
- Access to delete an object is limited to users with the "delete" permission for that type of object.

Permissions are set globally per type of object, not per specific object instance. For example, it's possible to say "Mary may change news stories," but it's not currently possible to say "Mary may change news stories, but only the ones she created herself" or "Mary may only change news stories that have a certain status, publication date or ID."

These three basic permissions — add, create and delete — are automatically created for each Django model that has a `class Admin`. Behind the scenes, these permissions are added to the `auth_permission` database table when you run `manage.py syncdb`.

These permissions will be of the form `"<app>.<action>_<object_name>"`. That is, if you've got a `polls` app with a `Choice` model, you'll get permissions named `"polls.add_choice"`, `"polls.change_choice"` and `"polls.delete_choice"`.

Note that if your model doesn't have `class Admin` set when you run `syncdb`, the permissions won't be created. If you initialize your database and add `class Admin` to models after the fact, you'll need to run `syncdb` again to create any missing permissions for your installed apps.

You can also create custom permissions for a given model object, using the `permissions` attribute on `Meta`. This example model creates three custom permissions:

```
class USCitizen(models.Model):
    # ...
    class Meta:
        permissions = (
            # Permission identifier     human-readable permission name
            ("can_drive",               "Can drive"),
            ("can_vote",                "Can vote in elections"),
            ("can_drink",               "Can drink alcohol"),
        )
```

This only creates those extra permissions when you run `syncdb`; it's up to you to check for these permissions in your views (see above).

Just like users, permissions are implemented in a Django model that lives in `django.contrib.auth.models`; this means that you can use Django's database API to interact directly with permissions if you like.

Groups

Groups are a generic way of categorizing users so you can apply permissions, or some other label, to those users. A user can belong to any number of groups.

A user in a group automatically has the permissions granted to that group. For example, if the group `Site editors` has the permission `can_edit_home_page`, any user in that group will have that permission.

Beyond permissions, groups are a convenient way to categorize users to give them some label, or extended functionality. For example, you could create a group `'Special users'`, and you could write code that could, say, give them access to a members-only portion of your site, or send them members-only e-mail messages.

Like users, the easiest way to manage groups in through the admin. However, groups are also just Django models that live in `django.contrib.auth.models`, so once again you can always use Django's database APIs to deal with groups at a low level.

Messages

The message system is a lightweight way to queue messages for given users. A message is associated with a `User`. There's no concept of expiration or time stamps.

Messages are used by the Django admin after successful actions. For example, when you create an object, you'll notice a "The object was created successfully" message at the top of the admin page.

You can use the same API to queue and display messages in your own app. The API is simple:

- To create a new message, use `user.message_set.create(message='message_text')`.
- To retrieve/delete messages, use `user_obj.get_and_delete_messages()`, which returns a list of `Message` objects in the user's queue (if any) and deletes the messages from the queue.

In this example view, the system saves a message for the user after creating a playlist:

```
def create_playlist(request, songs):
    # Create the playlist with the given songs.
    # ...
    request.user.message_set.create(
        message="Your playlist was added successfully."
    )
    return render_to_response("playlists/create.html",
        context_instance=RequestContext(request))
```

When you use `RequestContext`, the currently logged-in user and his/her messages are made available in the template context as the template variable `{{ messages }}`. Here's an example of template code that displays messages:

```
{% if messages %}
<ul>
    {% for message in messages %}
    <li>{{ message }}</li>
    {% endfor %}
</ul>
{% endif %}
```

Note that `RequestContext` calls `get_and_delete_messages` behind the scenes, so any messages will be deleted even if you don't display them.

Finally, note that this messages framework only works with users in the user database. To send messages to anonymous users, use the session framework directly.

Profiles

The final piece of the puzzle is the profile system. To understand what profiles are all about, let's first look at the problem:

In a nutshell, many sites need to store more user information than is available on the standard `User` object. To compound the problem, most sites will have different "extra" fields. Thus, Django provides a lightweight way of defining a "profile" object that's linked to a given user; this profile object can differ from project to project, and can even handle different profiles for different sites served from the same database.

The first step in creating a profile is to define a model that holds the profile information. The only requirement Django places on this model is that it have a unique `ForeignKey` to the `User` model; this field must be named `user` Other that that, you can use any other fields you like. Here's a strictly arbitrary profile model:

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

class MySiteProfile(models.Model):
    # This is the only required field
    user = models.ForeignKey(User, unique=True)

    # The rest is completely up to you...
    favorite_band = models.CharField(maxlength=100, blank=True)
    favorite_cheese = models.CharField(maxlength=100, blank=True)
    lucky_number = models.IntegerField()
```

Next, you'll need to tell Django where to look for this profile object. You do that by setting the `AUTH_PROFILE_MODULE` setting to the identifier for your model. So, if your model lives in an app called `myapp`, you'd put this in your settings file:

```
AUTH_PROFILE_MODULE = "myapp.mysiteprofile"
```

Once that's done, you can access a user's profile by calling `user.get_profile()`. This function will raise a `SiteProfileNotAvailable` exception if `AUTH_PROFILE_MODULE` isn't defined, and it also might raise a `DoesNotExist` exception if the user doesn't have a profile already (you'll usually catch that exception and create a new profile at that time).

Wrapping up

Yes, the session and authorization system is a lot to absorb. Most of the time you won't need all the features described in this chapter, but when you need to allow complex interactions between users, it's good to have all that power available.

In the next chapter, we'll take a look at a piece of Django that builds on top of this session/user system: the comments app. It allows you to easily attach comments — from anonymous or authenticated users — to arbitrary objects.

Onwards and upwards!

« previous ◊ table of contents ◊ next »

The Django Book

Chapter 14: Caching

Static websites, in which simple files are served directly to the Web, scale like crazy. But a fundamental tradeoff in dynamic Web sites is, well, they're dynamic. Each time a user requests a page, the Web server makes all sorts of calculations — from database queries to template rendering to business logic — to create the page that your site's visitor sees. From a processing-overhead perspective, this is a lot more expensive.

For most Web applications, this overhead isn't a big deal. Most Web applications aren't washingtonpost.com or slashdot.org; they're simply small- to medium-sized sites with so-so traffic. But for medium- to high-traffic sites, it's essential to cut as much overhead as possible.

That's where caching comes in.

To cache something is to save the result of an expensive calculation so that you don't have to perform the calculation next time. Here's some pseudocode explaining how this would work for a dynamically generated Web page:

```
given a URL, try finding that page in the cache
if the page is in the cache:
    return the cached page
else:
    generate the page
    save the generated page in the cache (for next time)
    return the generated page
```

Django comes with a robust cache system that lets you save dynamic pages so they don't have to be calculated for each request. For convenience, Django offers different levels of cache granularity. You can cache the output of specific views, you can cache only the pieces that are difficult to produce, or you can cache your entire site.

Django also works well with "upstream" caches, such as Squid (http://www.squid-cache.org/) and browser-based caches. These are the types of caches that you don't directly control but to which you can provide hints (via HTTP headers) about which parts of your site should be cached, and how.

Setting up the cache

The cache system requires a small amount of setup. Namely, you have to tell it where your cached data should live — whether in a database, on the filesystem or directly in memory. This is an important decision that affects your cache's performance; yes, some cache types are faster than others. In-memory caching will generally be much faster than filesystem or database caching, because the former lacks the overhead of hitting the filesystem or database.

Your cache preference goes in the `CACHE_BACKEND` setting in your settings file. If you use caching and do not specify `CACHE_BACKEND`, Django will use `simple:///` by default. Here's an explanation of all available values for `CACHE_BACKEND`.

Memcached

By far the fastest, most efficient type of cache available to Django, Memcached is an entirely memory-based cache framework originally developed to handle high loads at LiveJournal.com and subsequently open-sourced by Danga Interactive(http://www.danga.com). It's used by sites such as Slashdot and Wikipedia to reduce database access and dramatically increase site performance.

Memcached is available for free at http://danga.com/memcached/ . It runs as a daemon and is allotted a specified amount of RAM. Its primary feature is to provide an interface — a *super-lightning-fast* interface — for adding, retrieving and deleting arbitrary data in the cache. All data is stored directly in memory, so there's no overhead of database or filesystem usage.

After installing Memcached itself, you'll need to install the Memcached Python bindings, which are not bundled with Django directly. These bindings are in a single Python module, memcache.py, available at http://www.djangoproject.com/thirdparty/python-memcached/ .

To use Memcached with Django, set `CACHE_BACKEND` to `memcached://ip:port/`, where `ip` is the IP address of the Memcached daemon and `port` is the port on which Memcached is running.

In this example, Memcached is running on localhost (127.0.0.1) port 11211:

```
CACHE_BACKEND = 'memcached://127.0.0.1:11211/'
```

One excellent feature of Memcached is its ability to share cache over multiple servers. This means you can run Memcached daemons on multiple machines, and the program will treat the group of machines as a *single* cache, without the need to duplicate cache values on each machine. To take advantage of this feature with Django, include all server addresses in `CACHE_BACKEND`, separated by semicolons.

In this example, the cache is shared over Memcached instances running on the IP addresses 172.19.26.240 and 172.19.26.242, both on port 11211:

```
CACHE_BACKEND = 'memcached://172.19.26.240:11211;172.19.26.242:11211/'
```

In this example, the cache is shared over Memcached instances running on the IP addresses 172.19.26.240 (port 11211), 172.19.26.242 (port 11212) and 172.19.26.244 (port 11213):

```
CACHE_BACKEND = 'memcached://172.19.26.240:11211;172.19.26.242:11212;172.19.26.244:11213/'
```

A final point about Memcached is that memory-based caching has one important disadvantage. Because the cached data is stored only in memory, the data will be lost if your server crashes. Clearly, memory isn't intended for permanent data storage, so don't rely on memory-based caching as your only data storage. Without a doubt, *none* of the Django caching backends should be used for permanent storage — they're all intended to be solutions for caching, not storage — but we point this out here because memory-based caching is particularly temporary.

Database caching

To use a database table as your cache backend, create a cache table in your database and point Django's cache system at that table.

First, create a cache table by running this command:

```
python manage.py createcachetable [cache_table_name]
```

..where `[cache_table_name]` is the name of the database table to create. This name can be whatever you want, as long as it's a valid table name that's not already being used in your database. This command creates a single table in your database that is in the proper format Django's database-cache system expects.

Once you've created that database table, set your `CACHE_BACKEND` setting to `"db://tablename"`, where `tablename` is the name of the database table. In this example, the cache table's name is `my_cache_table`:

```
CACHE_BACKEND = 'db://my_cache_table'
```

The database caching backend uses the same database as specified in your settings file. You can't use a different database backend for your cache table.

Filesystem caching

To store cached items on a filesystem, use the `"file://"` cache type for `CACHE_BACKEND`, specifying the directory on your filesystem that should store the cached data.

For example, to store cached data in `/var/tmp/django_cache`, use this setting:

```
CACHE_BACKEND = 'file:///var/tmp/django_cache'
```

Note that there are three forward slashes toward the beginning of that example. The first two are for `file://`, and the third is the first character of the directory path, `/var/tmp/django_cache`. If you're on Windows, put the drive letter after the `file://`, like so:: `file://c:/foo/bar`.

The directory path should be absolute — that is, it should start at the root of your filesystem. It doesn't matter whether you put a slash at the end of the setting.

Make sure the directory pointed-to by this setting exists and is readable and writable by the system user under which your Web server runs. Continuing the above example, if your server runs as the user `apache`, make sure the directory `/var/tmp/django_cache` exists and is readable and writable by the user `apache`.

Each cache value will be stored as a separate file whose contents are the cache data saved in a serialized ("pickled") format, using Python's `pickle` module. Each file's name is the cache key, escaped for safe filesystem use.

Local-memory caching

If you want the speed advantages of in-memory caching but don't have the capability of running Memcached, consider the local-memory cache backend. This cache is multi-process and thread-safe, but isn't as efficient as Memcached due to its simplistic locking and memory allocation strategies.

To use it, set `CACHE_BACKEND` to `'locmem:///'`. For example:

```
CACHE_BACKEND = 'locmem:///'
```

Simple caching (for development)

A simple, single-process memory cache is available as `'simple:///'`. This merely saves cached data in process, which means it should only be used in development or testing environments. For example:

```
CACHE_BACKEND = 'simple:///'
```

Dummy caching (for development)

Finally, Django comes with a “dummy” cache that doesn’t actually cache — it just implements the cache interface without doing anything.

This is useful if you have a production site that uses heavy-duty caching in various places but a development/test environment on which you don’t want to cache. In that case, set `CACHE_BACKEND` to `'dummy:///'` in the settings file for your development environment. As a result, your development environment won’t use caching and your production environment still will. For example:

```
CACHE_BACKEND = 'dummy:///'
```

CACHE_BACKEND arguments

Each cache backend may take arguments. They’re given in query-string style on the `CACHE_BACKEND` setting. Valid arguments are:

-

Chapter 15: Other contributed sub-frameworks

One of the many strengths of Python is its "batteries included" philosophy; when you install Python, it comes with a large "standard library" of commonly used modules that you can start using immediately, without having to download anything else. Django aims to follow this philosophy, as it includes its own standard library of add-ons useful for common Web development tasks. This chapter covers that collection of add-ons.

About the standard library

Django's standard library lives in the package `django.contrib`. Within, each subpackage is a separate piece of add-on functionality. These pieces are not necessarily related, but some `django.contrib` subpackages may require other ones.

There's no hard requirement for the types of functionality in `django.contrib`. Some of the packages include models (and, hence, require you to install their database tables into your database), but others consist solely of middleware or template tags.

The single characteristic the `django.contrib` packages have in common is this: If you were to remove the `django.contrib` package entirely, you could still use Django's fundamentals with no problems. When the developers of Django add new functionality to the framework, they use this rule of thumb in deciding whether the new functionality should live in `django.contrib` or elsewhere.

`django.contrib` consists of these packages:

- `admin` — The automatic admin site. See Chapter 6.
- `auth` — Django's authentication framework. See Chapter 12.
- `comments` — A comments application. See Chapter 13.
- `contenttypes` — A framework for hooking into "types" of content, where each installed Django model is a separate content type. See "Content types" below.
- `csrf` — Protection against Cross Site Request Forgeries. See "CSRF protection" below.
- `flatpages` — A framework for managing simple "flat" HTML content in a database. See "Flatpages" below.
- `formtools` — A set of high-level abstractions for Django forms. See "Form tools" below.
- `humanize` — A set of Django template filters useful for adding a "human touch" to data. See "Humanizing data" below.
- `markup` — A set of Django template filters that implement a number of common markup languages. See "Markup filters" below.
- `redirects` — A framework for managing redirects. See "Redirects" below.
- `sessions` — Django's session framework. See Chapter 12.
- `sitemaps` — A framework for generating sitemap XML files. See "Sitemaps" below.
- `sites` — A framework that lets you operate multiple Web sites off of the same database and Django installation. See "Sites" below.
- `syndication` — A framework for generating syndication feeds in RSS and Atom. See "Syndication feeds" below.

The rest of this chapter goes into detail about each `django.contrib` package that hasn't yet been covered in this book.

Sites

Django's "sites" system is a generic framework that lets you operate multiple Web sites off of the same database and Django project. As this is an abstract concept, it can be tricky to understand — so we'll start with a couple of examples.

Example 1: Reusing data on multiple sites

As we explained in Chapter 1, the Django-powered sites LJWorld.com and Lawrence.com are operated by the same news organization — the Lawrence Journal-World newspaper in Lawrence, Kansas. LJWorld.com focuses on news, while Lawrence.com focuses on local entertainment. But sometimes editors want to publish an article on *both* sites.

The brain-dead way of solving the problem would be to use a separate database for each site, and to require site producers to publish the same story twice: once for LJWorld.com and again for Lawrence.com. But that's inefficient for site producers, and it's redundant to store multiple copies of the same story in the database.

The better solution is simple: Both sites use the same article database, and an article is associated with one or more sites via a many-to-many relationship. The Django sites framework provides the database table to which articles can be related. It's a hook for associating data with one or more "sites."

Example 2: Storing your site name/domain in one place

LJWorld.com and Lawrence.com both have e-mail alert functionality, which lets readers sign up to get notifications when news happens. It's pretty basic: A reader signs up on a Web form, and he immediately gets an e-mail saying, "Thanks for your subscription."

It'd be inefficient and redundant to implement this signup-processing code twice, so the sites use the same code behind the scenes. But the "thank you for signing up" notice needs to be different for each site. By using `Site` objects, we can abstract the "thank you" notice to use the values of the current site's `name` (e.g., `'LJWorld.com'`) and `domain` (e.g., `'www.ljworld.com'`).

The Django sites framework provides a place for you to store the `name` and `domain` for each site in your Django project, which means you can reuse those values in a generic way.

Using the sites framework

The sites framework is more of a series of conventions than a framework. The whole thing is based on two simple concepts:

- The `Site` model, found in `django.contrib.sites`, has `domain` and `name` fields.
- The `SITE_ID` setting specifies the database ID of the `Site` object associated with that particular settings file.

How you use these two concepts is up to you, but Django uses them in a couple of ways automatically via simple conventions.

To install the sites app, follow these steps:

1. Add `'django.contrib.sites'` to your `INSTALLED_APPS`.
2. Run the command `manage.py syncdb` to install the `django_site` table into your database.
3. Add one or more `Site` objects, either through the Django admin site or via the Python API. Create a `Site` object for each site/domain that this Django project powers.
4. Define the `SITE_ID` setting in each of your settings files. This value should be the database ID of the `Site` object for the site powered by that settings file.

Things you can do with the sites framework

Reusing data on multiple sites

To reuse data on multiple sites, as explained in "Example 1," just create a `ManyToManyField` to `Site` in your models. For example:

```
from django.db import models
from django.contrib.sites.models import Site

class Article(models.Model):
    headline = models.CharField(maxlength=200)
    # ...
    sites = models.ManyToManyField(Site)
```

That's the necessary infrastructure you need in order to associate articles with multiple sites in your database. With that in place, you can reuse the same Django view code for multiple sites. Continuing the `Article` example, here's what an `article_detail` view might look like:

```
from django.conf import settings

def article_detail(request, article_id):
    try:
        a = Article.objects.get(id=article_id, sites__id=settings.SITE_ID)
    except Article.DoesNotExist:
        raise Http404
    # ...
```

This view function is reusable because it checks the article's site dynamically, according to the value of the `SITE_ID` setting.

For example, say LJWorld.com's settings file has a `SITE_ID` set to `1` and Lawrence.com's settings file has a `SITE_ID` set to `2`. If this view is called when LJWorld.com's settings file is active, then it will limit the article lookup to articles in which the list of sites includes LJWorld.com.

Associating content with a single site

Similarly, you can associate a model to the `Site` model in a many-to-one relationship, using `ForeignKey`.

For example, if an article is only allowed on a single site, you'd use a model like this:

```
from django.db import models
from django.contrib.sites.models import Site
```

```
class Article(models.Model):
    headline = models.CharField(maxlength=200)
    # ...
    site = models.ForeignKey(Site)
```

This has the same benefits as described in the last section.

Hooking into the current site from views

On a lower level, you can use the sites framework in your Django views to do particular things based on what site in which the view is being called. For example:

```
from django.conf import settings

def my_view(request):
    if settings.SITE_ID == 3:
        # Do something.
    else:
        # Do something else.
```

Of course, it's ugly to hard-code the site IDs like that. This sort of hard-coding is best for hackish fixes that you need done quickly. A slightly cleaner way of accomplishing the same thing is to check the current site's domain:

```
from django.conf import settings
from django.contrib.sites.models import Site

def my_view(request):
    current_site = Site.objects.get(id=settings.SITE_ID)
    if current_site.domain == 'foo.com':
        # Do something
    else:
        # Do something else.
```

The idiom of retrieving the `Site` object for the value of `settings.SITE_ID` is quite common, so the `Site` model's manager (`Site.objects`) has a `get_current()` method. This example is equivalent to the previous one:

```
from django.contrib.sites.models import Site

def my_view(request):
    current_site = Site.objects.get_current()
    if current_site.domain == 'foo.com':
        # Do something
    else:
        # Do something else.
```

Note that in this final example, you don't have to import `django.conf.settings`.

Getting the current domain for display

For a DRY (Don't Repeat Yourself) approach to storing your site's name and domain name, as explained in “Example 2,” just reference the `name` and `domain` of the current `Site` object. For example:

```
from django.contrib.sites.models import Site
from django.core.mail import send_mail

def register_for_newsletter(request):
    # Check form values, etc., and subscribe the user.
    # ...
    current_site = Site.objects.get_current()
    send_mail('Thanks for subscribing to %s alerts' % current_site.name,
        'Thanks for your subscription. We appreciate it.\n\n-The %s team.' % current_site.
name,
        'editor@%s' % current_site.domain,
        [user_email])
    # ...
```

Continuing our ongoing example of LJWorld.com and Lawrence.com: On Lawrence.com, this e-mail has the subject line “Thanks for subscribing to lawrence.com alerts.” On LJWorld.com, the e-mail has the subject “Thanks for subscribing to LJWorld.com alerts.” This same site-specific behavior is done in the e-mail's message body.

Note that an even more flexible (but more heavyweight) way of doing this would be to use Django's template system. Assuming Lawrence.com and LJWorld.com have different template directories (`TEMPLATE_DIRS`), you could simply delegate to the template system like so:

```
from django.core.mail import send_mail
from django.template import loader, Context

def register_for_newsletter(request):
    # Check form values, etc., and subscribe the user.
    # ...
    subject = loader.get_template('alerts/subject.txt').render(Context({}))
    message = loader.get_template('alerts/message.txt').render(Context({}))
    send_mail(subject, message, 'do-not-reply@example.com', [user_email])
    # ...
```

In this case, you'd have to create `subject.txt` and `message.txt` templates in both the LJWorld.com and Lawrence.com template directories. That gives you more flexibility, but it's also more complex.

It's a good idea to exploit the `Site` objects as much as possible, to remove unneeded complexity and redundancy.

Getting the current domain for full URLs

Django's `get_absolute_url()` convention is nice for getting your objects' URL without the domain name, but in some cases you might want to display the full URL — with `http://` and the domain and everything — for an object. To do this, you can use the sites framework. A simple example:

```
>>> from django.contrib.sites.models import Site
>>> obj = MyModel.objects.get(id=3)
>>> obj.get_absolute_url()
'/mymodel/objects/3/'
>>> Site.objects.get_current().domain
'example.com'
>>> 'http://%s%s' % (Site.objects.get_current().domain, obj.get_absolute_url())
'http://example.com/mymodel/objects/3/'
```

The `CurrentSiteManager`

If `Site`s play a key role in your application, consider using the helpful `CurrentSiteManager` in your model(s). It's a model manager (see Chapter 5) that automatically filters its queries to include only objects associated with the current `Site`.

Use `CurrentSiteManager` by adding it to your model explicitly. For example:

```
from django.db import models
from django.contrib.sites.models import Site
from django.contrib.sites.managers import CurrentSiteManager

class Photo(models.Model):
    photo = models.FileField(upload_to='/home/photos')
    photographer_name = models.CharField(maxlength=100)
    pub_date = models.DateField()
    site = models.ForeignKey(Site)
    objects = models.Manager()
    on_site = CurrentSiteManager()
```

With this model, `Photo.objects.all()` will return all `Photo` objects in the database, but `Photo.on_site.all()` will return only the `Photo` objects associated with the current site, according to the `SITE_ID` setting.

In other words, these two statements are equivalent:

```
Photo.objects.filter(site=settings.SITE_ID)
Photo.on_site.all()
```

How did `CurrentSiteManager` know which field of `Photo` was the `Site`? It defaults to looking for a field called `site`. If your model has a `ForeignKey` or `ManyToManyField` called something *other* than `site`, you need to explicitly pass that as the parameter to `CurrentSiteManager`. The following model, which has a field called `publish_on`, demonstrates this:

```
from django.db import models
from django.contrib.sites.models import Site
from django.contrib.sites.managers import CurrentSiteManager
```

```
class Photo(models.Model):
    photo = models.FileField(upload_to='/home/photos')
    photographer_name = models.CharField(maxlength=100)
    pub_date = models.DateField()
    publish_on = models.ForeignKey(Site)
    objects = models.Manager()
    on_site = CurrentSiteManager('publish_on')
```

If you attempt to use `CurrentSiteManager` and pass a field name that doesn't exist, Django will raise a `ValueError`.

Finally, note that you'll probably want to keep a normal (non-site-specific) `Manager` on your model, even if you use `CurrentSiteManager`. As explained in Chapter 5, if you define a manager manually, then Django won't create the automatic `objects = models.Manager()` manager for you. Also, note that certain parts of Django — namely, the Django admin site and generic views — use whichever manager is defined *first* in the model, so if you want your admin site to have access to all objects (not just site-specific ones), put `objects = models.Manager()` in your model, before you define `CurrentSiteManager`.

How Django uses the sites framework

Although it's not required that you use the sites framework, it's strongly encouraged, because Django takes advantage of it in a few places. Even if your Django installation is powering only a single site, you should take the two seconds to create the site object with your `domain` and `name`, and point to its ID in your `SITE_ID` setting.

Here's how Django uses the sites framework:

- In the redirects framework (see "Redirects" below), each redirect object is associated with a particular site. When Django searches for a redirect, it takes into account the current `SITE_ID`.
- In the comments framework (see Chapter 13), each comment is associated with a particular site. When a comment is posted, its `site` is set to the current `SITE_ID`, and when comments are listed via the appropriate template tag, only the comments for the current site are displayed.
- In the flatpages framework (see "Flatpages" below), each flatpage is associated with a particular site. When a flatpage is created, you specify its `site`, and the flatpage middleware checks the current `SITE_ID` in retrieving flatpages to display.
- In the syndication framework (see "Syndication feeds" below), the templates for `title` and `description` automatically have access to a variable `{{{ site }}}`, which is the `Site` object representing the current site. Also, the hook for providing item URLs will use the `domain` from the current `Site` object if you don't specify a fully-qualified domain.
- In the authentication framework (see Chapter 12), the `django.contrib.auth.views.login` view passes the current `Site` name to the template as `{{{ site_name }}}`.
- The shortcut view (see Chapter XX) uses the domain of the current `Site` object when calculating an object's URL.

Flatpages

Often times, you'll have a database-driven Web application up and running, but you'll need to add a couple "one-off" static pages, such as an "About" page or a "Privacy Policy" page. It'd be possible to use a standard Web server such as Apache to serve these files as flat HTML files, but that introduces an extra level of complexity into your application, because then you have to worry about configuring Apache, you've got to set up access for your team to edit those files, and you can't take advantage of Django's template system to style the pages.

The solution to this problem is Django's "flatpages" app, which lives in the package `django.contrib.flatpages`. This app lets you manage such "one-off" pages via Django's admin site, and it lets you specify templates for them using Django's template system. It uses Django models behind the scenes, which means it stores the pages in a database, just like the rest of your data, and you can access flatpages with the standard Django database API.

Flatpages are keyed by their URL and site. When you create a flatpage, you specify which URL it's associated with, along with which site(s) it's on. (For more on sites, see the "Sites" section above.)

Using flatpages

To install the flatpages app, follow these steps:

1. Add `'django.contrib.flatpages'` to your `INSTALLED_APPS`.
2. Add `'django.contrib.flatpages.middleware.FlatpageFallbackMiddleware'` to your `MIDDLEWARE_CLASSES` setting.
3. Run the command `manage.py syncdb` to install the two required tables into your database.

How it works

The flatpages app creates two tables in your database: `django_flatpage` and `django_flatpage_sites`. `django_flatpage` is a lookup table that simply maps a URL to a title and bunch of text content. `django_flatpage_sites` is a many-to-many table that associates a flatpage with one or more sites.

The app comes with a single `FlatPage` model, defined in `django/contrib/flatpages/models.py`. It looks like this:

```
from django.db import models
from django.contrib.sites.models import Site
```

```
class FlatPage(models.Model):
    url = models.CharField(maxlength=100)
    title = models.CharField(maxlength=200)
    content = models.TextField()
    enable_comments = models.BooleanField()
    template_name = models.CharField(maxlength=70, blank=True)
    registration_required = models.BooleanField()
    sites = models.ManyToManyField(Site)
```

Let's cover these fields one at a time:

- `url` — The URL at which this flatpage lives, excluding the domain name but including the leading slash. Example: `'/about/contact/`.
- `title` — The title of the flatpage. The framework doesn't do anything special with this. It's your responsibility to display it in your template.
- `content` — The content of the flatpage, i.e., the HTML of the page. The framework doesn't do anything special with this. It's your responsibility to display it in the template.
- `enable_comments` — Whether to enable comments on this flatpage. The framework doesn't do anything special with this. You can check this value in your template and display a comment form if needed.
- `template_name` — The name of the template to use for rendering this flatpage. This is optional. If it's not given, the framework will use the template `flatpages/default.html`.
- `registration_required` — Whether registration is required for viewing this flatpage. This integrates with Django's authentication/user framework, which was explained in Chapter 12.
- `sites` — The sites that this flatpage lives on. This integrates with Django's sites framework, which was explained in the "Sites" section above.

You can create flatpages through either the Django admin interface or the Django database API. For more, see "How to add, change and delete flatpages" below.

Once you've created flatpages, the `FlatpageFallbackMiddleware` does all of the work. Each time any Django application raises a 404 error, this middleware checks the flatpages database for the requested URL as a last resort. Specifically, it checks for a flatpage with the given URL with a site ID that corresponds to the `SITE_ID` setting.

If it finds a match, it loads the flatpage's template, or `flatpages/default.html` if the flatpage has not specified a custom template. It passes that template a single context variable, `flatpage`, which is the flatpage object. It uses `RequestContext` in rendering the template.

If it doesn't find a match, the request continues to be processed as usual.

Note that this middleware only gets activated for 404s — not for 500s or responses of any other status code. Also note that the order of `MIDDLEWARE_CLASSES` matters. Generally, you can put `FlatpageFallbackMiddleware` at the end of the list, because it's a last resort.

How to add, change and delete flatpages

Via the admin interface

If you've activated the automatic Django admin interface, you should see a "Flatpages" section on the admin index page. Edit flatpages as you edit any other object in the system.

Via the Python API

As described above, flatpages are represented by a standard Django model that lives in `django/contrib/flatpages/models.py`. Hence, you can access flatpage objects via the Django database API. For example:

```
>>> from django.contrib.flatpages.models import FlatPage
>>> from django.contrib.sites.models import Site
>>> fp = FlatPage(
...     url='/about/',
...     title='About',
...     content='<p>About this site...</p>',
...     enable_comments=False,
...     template_name='',
...     registration_required=False,
... )
>>> fp.save()
>>> fp.sites.add(Site.objects.get(id=1))
>>> FlatPage.objects.get(url='/about/')
<FlatPage: /about/ -- About>
```

Flatpage templates

By default, flatpages are rendered via the template `flatpages/default.html`, but you can override that for a particular flatpage.

Creating the `flatpages/default.html` template is your responsibility. In your template directory, just create a `flatpages` directory containing a file `default.html`.

Flatpage templates are passed a single context variable, `flatpage`, which is the flatpage object.

Here's a sample `flatpages/default.html` template:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN"
    "http://www.w3.org/TR/REC-html40/loose.dtd">
<html>
<head>
<title>{{ flatpage.title }}</title>
</head>
<body>
{{ flatpage.content }}
</body>
</html>
```

Redirects

Django's redirects framework lets you manage redirects easily by storing them in a database and treating them as any other Django model object. For example, you can use the redirects framework to tell Django, "redirect any request to `/music/` to `/sections/arts/music/`."

Using the redirects framework

To install the redirects app, follow these steps:

1. Add `'django.contrib.redirects'` to your `INSTALLED_APPS`.
2. Add `'django.contrib.redirects.middleware.RedirectFallbackMiddleware'` to your `MIDDLEWARE_CLASSES` setting.
3. Run the command `manage.py syncdb` to install the single required table into your database.

How it works

`manage.py syncdb` creates a `django_redirect` table in your database. This is a simple lookup table with `site_id`, `old_path` and `new_path` fields.

You can create redirects through either the Django admin interface or the Django database API. For more, see "How to add, change and delete redirects" below.

Once you've created redirects, the `RedirectFallbackMiddleware` does all of the work. Each time any Django application raises a 404 error, this middleware checks the redirects database for the requested URL as a last resort. Specifically, it checks for a redirect with the given `old_path` with a site ID that corresponds to the `SITE_ID` setting. (See "Sites" above for more on `SITE_ID` and the sites framework.) Then, it follows these steps:

- If it finds a match, and `new_path` is not empty, it redirects to `new_path`.
- If it finds a match, and `new_path` is empty, it sends a 410 ("Gone") HTTP header and empty (content-less) response.
- If it doesn't find a match, the request continues to be processed as usual.

The middleware only gets activated for 404s — not for 500s or responses of any other status code.

Note that the order of `MIDDLEWARE_CLASSES` matters. Generally, you can put `RedirectFallbackMiddleware` at the end of the list, because it's a last resort.

How to add, change and delete redirects

Via the admin interface

If you've activated the automatic Django admin interface, you should see a "Redirects" section on the admin index page. Edit redirects as you edit any other object in the system.

Via the Python API

Redirects are represented by a standard Django model that lives in `django/contrib/redirects/models.py`. Hence, you can access redirect objects via the Django database API. For example:

```
>>> from django.contrib.redirects.models import Redirect
>>> from django.contrib.sites.models import Site
>>> red = Redirect(
...     site=Site.objects.get(id=1),
...     old_path='/music/',
...     new_path='/sections/arts/music/',
... )
>>> red.save()
>>> Redirect.objects.get(old_path='/music/')
<Redirect: /music/ ---> /sections/arts/music/>
```

CSRF protection

The `django.contrib.csrf` package provides easy-to-use protection against Cross-Site Request Forgeries (CSRF).

CSRF explained

CSRF, also known as "session riding," is a Web-site security exploit. It happens when a malicious Web site tricks a user into unknowingly loading a URL from a site at which they're already authenticated — hence, taking advantage of their authenticated status. This can be a bit tricky to understand at first, so we've included two examples here:

A simple example

Say you're logged into a webmail account at `example.com`. Say this webmail site has a "Log out" button that points to the URL `example.com/logout` — that is, the only action you need to take in order to log out is to visit the page `example.com/logout`.

A malicious site can coerce you to visit the URL `example.com/logout` by including that URL as a hidden `<iframe>` on its own (malicious) page. Thus, if you're logged into the `example.com` webmail account and visit the malicious page that has an `<iframe>` to `example.com/logout`, the act of visiting the malicious page will log you out from `example.com`.

Clearly, being logged out of a webmail site against your will is not a terrifying breach of security, but this same type of exploit can happen to *any* site that "trusts" users — such as bank sites or e-commerce sites.

A more complex example

In previous example, `example.com` was partially at fault because it allowed a state change (i.e., logging yourself out) to be requested via the HTTP `GET` method. It's much better practice to require an HTTP `POST` for any request that changes state on the server. But even Web sites that require `POST` for state-changing actions are vulnerable to CSRF.

Say `example.com` has upgraded its "Log out" functionality so that it's a `<form>` button that is requested via `POST` to the URL `example.com/logout`. Furthermore, the log-out `<form>` includes this hidden field:

```
<input type="hidden" name="confirm" value="true" />
```

This ensures that a simple `POST` to the URL `example.com/logout` won't perform the logging out; in order for a user to log out, the user must request `example.com/logout` via `POST` *and* send the `confirm` `POST` variable with a value of `'true'`.

Well, despite the extra security, this arrangement can still be exploited by CSRF; the malicious page just needs to do a little more work. Instead of loading the `example.com/logout` page in an `<iframe>`, it can call that URL via `POST` using JavaScript, passing the `confirm=true` variable.

Prevention

How, then, can your site protect itself from this exploit?

The first step is to make sure all `GET` requests are free of side effects. That way, if a malicious site includes one of your pages as an `<iframe>`, it won't have a negative effect.

That leaves `POST` requests. The second step, then, is to give each `POST` `<form>` a hidden field whose value is secret and is generated from the user's session ID. Then, when processing the form on the server side, check for that secret field and raise an error if it doesn't validate.

This is exactly what Django's CSRF prevention layer does.

Using the CSRF middleware

The `django.csrf` package contains only one module: `middleware.py`. This module contains a Django middleware class, `CsrfMiddleware`, which implements the CSRF protection.

To use it, add `'django.contrib.csrf.middleware.CsrfMiddleware'` to the `MIDDLEWARE_CLASSES` setting in your settings file. This middleware needs to process the response *after* `SessionMiddleware`, so `CsrfMiddleware` must appear *before* `SessionMiddleware` in the list. Also, it must process the response before the response gets compressed or otherwise mangled, so `CsrfMiddleware` must come after `GZipMiddleware`.

Once you've added that to your `MIDDLEWARE_CLASSES` setting, you're done. That's all you need to do.

How it works

In case you're interested, here's how `CsrfMiddleware` works. It does these two things:

1. It modifies outgoing requests by adding a hidden form field to all `POST` forms, with the name `csrfmiddlewaretoken` and a value that is a hash of the session ID plus a secret. The middleware does *not* modify the response if there's no session ID set, so the performance penalty is negligible for requests that don't use sessions.
2. On all incoming `POST` requests that have the session cookie set, it checks that the `csrfmiddlewaretoken` is present and correct. If it isn't, the user will get a 403 `HTTP` error. The contents of the 403 error page are the message "Cross Site Request Forgery detected. Request aborted."

This ensures that only forms originating from your Web site can be used to POST data back.

This middleware deliberately only targets HTTP `POST` requests (and the corresponding POST forms). As we explained above, `GET` requests ought never to have side effects; ensuring this is your own responsibility.

`POST` requests that are not accompanied by a session cookie are not protected, but they don't *need* to be protected, because a malicious Web site could make these kind of requests anyway.

To avoid altering non-textual requests, the middleware checks the response's `Content-Type` header before modifying it. Only pages that are served as `text/html` or `application/xml+xhtml` are modified.

Limitations

`CsrfMiddleware` requires Django's session framework to work. (See Chapter 12 for more on sessions.) If you've using a custom session or authentication framework that manually manages session cookies, this middleware will not help you.

If your app creates HTML pages and forms in some unusual way — e.g., if it sends fragments of HTML in JavaScript `document.write` statements — you might bypass the filter that adds the hidden field to the form. In this case, the form submission would always fail. (This would happen because the `CsrfMiddleware` uses a regular expression to add the `csrfmiddlewaretoken` field to your HTML before the page is sent to the client, and the regular expression sometimes cannot handle wacky HTML.) If you suspect this might be happening, just view source in your Web browser to see whether the `csrfmiddlewaretoken` was inserted into your `<form>`.

For more CSRF information and examples, visit http://en.wikipedia.org/wiki/Csrf

Content types

This section hasn't been written yet.

Form tools

This section hasn't been written yet.

Humanizing data

This section hasn't been written yet.

Markup filters

This section hasn't been written yet.

Syndication feeds

This section hasn't been written yet.

« previous ◊ table of contents ◊ next »

Chapter 16: Middleware

On occasion, you'll need to run a piece of code on each and every request that Django handles. This code might need to modify the request before the view handles it, or maybe log information about the request for debugging purposes, etc.

Django's **middleware** framework is essentially a set of hooks into Django's request/response processing. It's a light, low-level "plugin" system for globally altering Django's input and/or output.

Each middleware component is responsible for doing some specific function. If you're reading this book linearly — sorry, postmodernists — you'll have already seen middleware a number of times:

- All of the nifty session and user tools that we looked at in Chapter 12 are made possible by a few small pieces of middleware (more specifically, the middleware makes `request.session` and `request.user` available to you in views).
- The site-wide cache discussed in Chapter 12 is actually just a piece of middleware that short-circuits the call to your view function if the response for that view has already been cached.
- The `flatpages`, `redirects` and `csrf` contributed apps from Chapter 15 all do their magic through the use of middleware components

This chapters dives deeper into exactly what middleware is and how it works, and explains how you can write your own middleware.

What's middleware?

Middleware is actually incredible simple. A middleware component is simply a Python class that conforms to a certain API — duck typing strikes again! Before diving into the formal aspects of what that API is, let's look at a very simple example.

High-traffic sites often need to deploy Django behind a load balancing proxy (see Chapter 21). This can cause a few small complications, one of which is that every request's remote IP (`request.META["REMOTE_IP"]`) will be that of the load balancer, not the actual IP making the request. Load balancers deal with this by setting a special header, `X-Forwarded-For`, to the actual requesting IP address.

So here's a small bit of middleware that lets sites running behind a proxy still see the correct IP address in `request.META["REMOTE_IP"]`:

```
class SetRemoteAddrFromForwardedFor(object):

    def process_request(self, request):
        try:
            real_ip = request.META['HTTP_X_FORWARDED_FOR']
        except KeyError:
            pass
        else:
            # HTTP_X_FORWARDED_FOR can be a comma-separated list of IPs.
            # Take just the first one.
            real_ip = real_ip.split(",")[0]
            request.META['REMOTE_ADDR'] = real_ip
```

If this is installed (see below), every request's `X-Forwarded-For` value will be automatically inserted into `request.META['REMOTE_ADDR']`. Simple, isn't it?

In fact, this is a common enough need that this piece of middleware is a built-in part of Django; it lives in `django.middleware.http`, and you can read a bit more about it below.

Installing middleware

The linear readers in the crowd are probably old hands at this already; many of the examples in the previous few chapters will only work if you've already figured out how to enable middleware. However, for completeness — and for the benefit of Julio Cortázar fans who've torn all the pages out of this book, shuffled them, and are now reading them in random order — let's break it down.

To activate a middleware component, add it to the `MIDDLEWARE_CLASSES` list in your settings module. In `MIDDLEWARE_CLASSES`, each middleware component is represented by a string: the full Python path to the middleware's class name. For example, here's the default `MIDDLEWARE_CLASSES` created by `django-admin.py startproject`:

```
MIDDLEWARE_CLASSES = (
    'django.middleware.common.CommonMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.middleware.doc.XViewMiddleware',
)
```

A Django installation doesn't require any middleware — e.g., `MIDDLEWARE_CLASSES` can be empty, if you'd like — but it's strongly suggested that you use `CommonMiddleware`.

The order is significant. On the request and view phases, Django applies middleware in the order given in `MIDDLEWARE_CLASSES`, and on the response and exception phases, Django applies middleware in reverse order. That is, Django treats `MIDDLEWARE_CLASSES` as a sort of "wrapper" around the view function: on the request, it walks down the list to the view, and on the response it walks back up.

Middleware methods

Now that we know what middleware is and how to install it, let's take a look at all the available methods that middleware classes may define.

Initializer: `__init__(self)`

If middleware classes define an initializer (i.e. an `__init__` method), it should take no arguments (beyond the standard `self`).

For performance reasons, middleware classes are only instantiated **once** in long-running server processes; this means that you can't count on `__init__` getting called every time a request runs, only once at server startup.

Middleware classes may also use initialization time to remove themselves from being installed. If an initializer raises `django.exceptions.MiddlewareNotUsed`, Django will remove that piece of middleware from the middleware stack. You might use this to check for some piece of software that the middleware class depends on, or whether the server is running in debug mode, or any other sort of environmental situation that might make you want to disable the middleware.

Request pre-processor: `process_request(self, request)`

This method gets called as soon as the request as been received, and before the URL has been resolved to determine which view to run. It's passed the `HttpRequest` object, which you may modify at will.

`process_request()` should return either `None` or an `HttpResponse` object. If it returns `None`, Django will continue processing this request, executing any other middleware and then the appropriate view.

If a request middleware returns an `HttpResponse` object, Django won't bother calling *any* other middleware (of any type) or the appropriate view; it'll return that `HttpResponse`.

View pre-processor: `process_view(self, request, view, args, kwargs)`

This method gets called after the request middleware has run, and after the URL has been resolved into a view, but before that view has actually been called.

The arguments passed to this view are:

Argument	Explanation
`request`	The `HttpRequest` object.
`view`	The Python function that Django will call to handle this request. This is the actual function object itself, not the name of the function as a string.
`args`	The list of positional arguments that will be passed to the view, not including the `request` argument (which is always the first argument to a view).
`kwargs`	The dictionary of keyword arguments that will be passed to the view.

Just like `process_request()`, `process_view()` should return either `None` or an `HttpResponse` object. If it returns `None`, Django will continue processing this request, executing any other view middleware and then the appropriate view.

If any view middleware returns an `HttpResponse` object, Django won't bother calling any other middleware or the appropriate view; it'll return that response.

Response post-processor: `process_response(self, request, response)`

This method gets called after the view function has already been called and the response has been generation. This is where middleware can modify the output of a response; output compression (see below) is one obvious use for response middleware.

The parameters should be pretty self-explanatory — `request` is the request object, and `response` is the response object returned from the view.

Unlike the request and view middleware methods which may return `None`, `process_response()` *must* return an `HttpResponse` object. That response could be the original one passed into the function (possibly modified), or a brand new one.

Exception post-processor: `process_exception(self, request, exception)`

This method only gets called if something goes wrong and a view raises an uncaught exception, not including `Http404` exceptions. You can use this hook to send error notifications, dump post-mortem information to a log, or even try to recover from the error automatically.

The parameters to this function are the same `request` object we've been dealing with all along, and `exception`, which is the actual `Exception` object raised by the view function.

`process_exception()` may return an `HttpResponse` which will be used as the response shown to the browser, or it may return `None` to continue with Django's built-in exception handling.

Examples

Django ships with a number of middleware classes — discussed below — that make good examples; reading the code for them should give you a good feel for the power of middleware.

You can also find a number of community-contributed examples on Django's wiki: http://code.djangoproject.com/wiki/ContributedMiddleware.

Built-in middleware

Django comes with some built-in middleware to deal with common problems.

Authentication support middleware

Middleware class: `django.contrib.auth.middleware.AuthenticationMiddleware`

Enables authentication support. Technically, this middleware adds the `request.user` attribute, representing the currently-logged-in user, to every incoming `HttpRequest` object.

See Chapter 15 for the complete details.

"Common" middleware

Middleware class: `django.middleware.common.CommonMiddleware`.

Adds a few conveniences for perfectionists:

- Forbids access to user agents in the `DISALLOWED_USER_AGENTS` setting, which should be a list of strings.
- Performs URL rewriting based on the `APPEND_SLASH` and `PREPEND_WWW` settings. If `APPEND_SLASH` is `True`, URLs that lack a trailing slash will be redirected to the same URL with a trailing slash, unless the last component in the path contains a period. So `foo.com/bar` is redirected to `foo.com/bar/`, but `foo.com/bar/file.txt` is passed through unchanged.

  If `PREPEND_WWW` is `True`, URLs that lack a leading "www." will be redirected to the same URL with a leading "www."

  Both of these options are meant to normalize URLs. The philosophy is that each URL should exist in one, and only one, place. Technically a URL `foo.com/bar` is distinct from `foo.com/bar/` — a search-engine indexer would treat them as separate URLs — so it's best practice to normalize URLs.
- Handles ETags based on the `USE_ETAGS` setting. If `USE_ETAGS` is set to `True`, Django will calculate an ETag for each request by MD5-hashing the page content, and it'll take care of sending `Not Modified` responses, if appropriate.

Compression middleware

Middleware class: `django.middleware.gzip.GZipMiddleware`

If enabled, this middleware will automatically compress content for browsers that understand gzip compression (all modern browsers).

This can greatly reduce the amount of bandwidth a web server consumes at the expense of processing time. We usually prefer speed over bandwidth, but if you'd like to take the opposite side of this trade-off, just enable this middleware.

Conditional `GET` middleware

Middleware class: `django.middleware.http.ConditionalGetMiddleware`

If enabled, provides support for conditional `GET` operations. If the response has a `ETag` or `Last-Modified` header, and the request has `If-None-Match` or `If-Modified-Since`, the response is replaced by an 304 ("Not modified") response.

Also removes the content from any response to a HEAD request and sets the `Date` and `Content-Length` response-headers for all requests.

Reverse proxy support (`X-Forwarded-For` middleware)

Middleware class: `django.middleware.http.SetRemoteAddrFromForwardedFor`

This is the example we looked at above. It sets `request.META['REMOTE_ADDR']` based on `request.META['HTTP_X_FORWARDED_FOR']`, if the latter is set. This is useful if you're sitting behind a reverse proxy that causes each request's `REMOTE_ADDR` to be set to `127.0.0.1`.

Danger, Will Robinson!

This does **not** validate `HTTP_X_FORWARDED_FOR`.

If you're not behind a reverse proxy that sets `HTTP_X_FORWARDED_FOR` automatically, do not use this middleware. Anybody can spoof the value of `HTTP_X_FORWARDED_FOR`, and because this sets `REMOTE_ADDR` based on `HTTP_X_FORWARDED_FOR`, that means anybody can fake their IP address.

Only use this middlware when you can absolutely trust the value of `HTTP_X_FORWARDED_FOR`.

Session support middleware

Middleware class: `django.contrib.sessions.middleware.SessionMiddleware`.

Enables session support; see Chapter 15 for details.

Site-wide cache middleware

Middleware class: `django.middleware.cache.CacheMiddleware`.

If this is enabled, each Django-powered page will be cached. This is discussed in detail in Chapter 14.

Transaction middleware

Middleware class: `django.middleware.transaction.TransactionMiddleware`

Binds a database `COMMIT` or `ROLLBACK` to the request/response phase. If a view function runs successfully, a `COMMIT` is done. If it fails with an exception, a `ROLLBACK` is done.

The order of this middleware in the stack is important: middleware modules running outside of it run with commit-on-save - the default Django behavior. Middleware modules running inside it (coming later in the stack) will be under the same transaction control as the view functions.

See XXX for more about information about database transactions.

"X-View" middleware

Middleware class: `django.middleware.doc.XViewMiddleware`

Sends custom `X-View` HTTP headers to HEAD requests that come from IP addresses defined in the `INTERNAL_IPS` setting. This is used by Django's automatic documentation system.

« previous ◊ table of contents ◊ next »

Chapter 17: Integrating with legacy databases and applications

Although Django is best suited for developing projects from scratch — so-called "green-field" development — it's possible to integrate the framework into legacy databases and applications. This chapter explains a few integration strategies.

Integrating with a legacy database

Django's database layer generates SQL schemas from Python code — but in the case of a legacy database, you already have the SQL schemas. In that case, you'll need to write models for your existing database tables. (For performance reasons, Django's database layer does not support on-the-fly object-relational mapping by introspecting the database at run time; in order to use the database API, you're required to write model code.) Fortunately, Django comes with a utility that can generate model code by reading your database table layouts. This utility is called `manage.py inspectdb`.

Using `inspectdb`

The `inspectdb` utility introspects the database pointed to by your settings file, determines the Django model representation of your tables and prints the Python model code to standard output. Here's a walkthrough of a typical legacy-database process from scratch; the only things it assumes are that Django is installed and that you have a legacy database.

1. Create a Django project by running `django-admin.py startproject mysite` (where `mysite` is your project's name). We'll use `mysite` as the project name in this example.
2. Edit the settings file in that project, `mysite/settings.py`, to tell Django what your database connection parameters are, and what the name of the database is. Specifically, you'll want to specify the `DATABASE_NAME`, `DATABASE_ENGINE`, `DATABASE_USER`, `DATABASE_PASSWORD`, `DATABASE_HOST` and `DATABASE_PORT` settings.
3. Create a Django app within your project by running `python mysite/manage.py startapp myapp` (where `myapp` is your app's name). We'll use `myapp` as the project name here.
4. Run the command `python mysite/manage.py inspectdb`. This will examine the tables in the `DATABASE_NAME` database and print the model class for each table. Take a look at the output to get an idea of what `inspectdb` can do.
5. Save that output to the `models.py` file within your app by using standard shell output redirection:

   ```
   python mysite/manage.py inspectdb > mysite/myapp/models.py
   ```
6. Edit the `mysite/myapp/models.py` file to clean up the generated models and make whatever customizations you need to make. We'll give some hints for this in the next section.

Cleaning up generated models

As you might expect, the database introspection isn't perfect, and you'll need to do some light cleanup of the resulting model code. Here are a few pointers for dealing with the generated models:

1. Each database table is converted to a model class — i.e., there is a one-to-one mapping between database tables and model classes. This means that you'll need to refactor the models for any many-to-many join tables into `ManyToManyField` objects.
2. Each generated model has an attribute for every field — including `id` primary-key fields. However, recall that Django automatically adds an `id` primary-key field if a model doesn't have a primary key. Thus, if you're particularly anal, you'll want to remove any lines that look like this, because they're redundant:

   ```
   id = models.IntegerField(primary_key=True)
   ```
3. Each field's type (e.g., `CharField`, `DateField`) is determined by looking at the database column type (e.g., `VARCHAR`, `DATE`). If `inspectdb` cannot map a column's type to a model field type, it will use `TextField` and will insert the Python comment `'This field type is a guess.'` next to the field in the generated model. Keep an eye out for that, and change the field type accordingly if needed.
4. If a database column name is a Python reserved word (such as `pass`, `class` or `for`), `inspectdb` will append `'_field'` to the attribute name. For example, if a table has a column `for`, the generated model will have a field `for_field`, with the `db_column` attribute set to `'for'`. `inspectdb` will insert the Python comment `'Field renamed because it was a Python reserved word.'` next to the field.
5. If your database contains tables that refer to other tables (as most databases do), you might need to rearrange the order of the generated models so that models that refer to other models are ordered properly. For example, if model `Foo` has a `ForeignKey` to model `Bar`, model `Bar` should be defined before model `Foo`.

6. `inspectdb` detects primary keys for PostgreSQL, MySQL and SQLite. That is, it inserts `primary_key=True` where appropriate. For other databases, you'll need to insert `primary_key=True` for at least one field in each model, because Django models are required to have a `primary_key=True` field.

7. Foreign-key detection only works with PostgreSQL and with certain types of MySQL tables. In other cases, foreign-key fields will be generated as `IntegerField``s, assuming the foreign-key column was an ``INT` column.

More

What else would you like to see in this chapter? What problems/questions do you have with integrating Django with legacy databases/applications? Leave a comment on this paragraph and let us know.

« previous ◊ table of contents ◊ next »

Chapter 18: Extending Django's admin interface

Chapter 6 introduced Django's admin interface, and now it's time to circle back and take a closer look.

As we've said a few times before, the admin is one of Django's "killer features," and most Django developers quickly fall in love with all its timesaving features. It follows naturally, then, that eventually most Django developers look to customize or extend the admin.

The last few sections of Chapter 6 talk about some simple ways to customize certain parts of the admin interface. It's probably a good idea to go read back over that material; it covers some simple ways to customize the admin change lists and edit forms, as well as an easy way to "re-brand" the admin to match your site.

Chapter 6 also discusses when and why you'd want to use the admin interface, and since that material makes a good jumping-off point for the rest of this chapter, we'll reproduce it here:

> Obviously, [the admin is] extremely useful for editing data (fancy that). If you have any sort of data entry tasks, the admin simply can't be beat. We suspect that the vast majority of readers of this book will have a whole host of data entry tasks.
>
> Django's admin especially shines when non-technical users need to be able to enter data; that's the original genesis of the feature. At the newspaper where Django was first developed, development of a typical online feature —a special report on water quality in the municipal supply, say —goes something like this:
>
> - The reporter responsible for the story meets with one of the developers and goes over the available data.
> - The developer designs a model around this data, and then opens up the admin interface to the reporter.
> - While the reporter enters data into Django, the programmer can focus on developing the publicly-accessible interface (the fun part!)
>
>   In other works, the raison d'être of Django's admin is facilitating the simultaneous work of content producers and programmers.
>
> However, beyond the obvious data-entry tasks, we find the admin useful in a few other cases:

> - Inspecting data models: the first thing we do when we've defined a new model is to call it up in the admin and enter some dummy data. This is usually when we find any data modeling errors; having a graphical interface to a model quickly reveals those mistakes.
> - Managing acquired data: there's little actual data entry associated with a site like chicagocrime.org since most of the data comes from an automated source. However, when problems with the automatically acquired data crop up, it's very useful to be able to go in and edit that data easily.

Django's admin handles these common cases with little or no customization. As with most design trade-offs, though, handling these common cases so well means that Django's admin doesn't handle some other modes of editing very well at all.

We'll talk about the cases that Django's admin *isn't* designed to cover a bit later on, but first, a brief digression on philosophy:

The Zen of Admin

At it's core, Django's admin is designed for a single activity:

> Trusted users editing structured content.

Yes, extremely simple — but in that simplicity lies a whole host of suppositions that the admin takes as given. The entire philosophy of Django's admin follows directly from these assumptions, so let's dig into the subtext of this phrase:

"Trusted users ..."

The admin is designed to be used by people who you, the developer, **trust**. This doesn't just mean "people who have been authenticated;" it means that Django assumes that your content editors can be trusted to do the right thing.

This means that there's no "approval" process for editing content — if you trust your users, nobody needs to approve of their edits. It also means that the permission system, while powerful, has no support for limiting access on a per-object basis. If you trust someone to edit their own stories, you trust them not to edit anyone else's without permission.

".. editing ..."

The primary purpose of Django's admin is to let people edit stuff. This seems obvious at first, but again has some subtle and powerful repercussions.

For instance, although the admin is quite useful for reviewing data (see above), it's not designed with that purpose as a goal: note the lack of a "can view" permission (see Chapter 12). Django assumes that if people are allowed to view content in the admin, they're also allowed to edit it.

Another more important note is the lack of anything even remotely approaching "workflow." If some given tasks requires a series of steps, there's no support for enforcing that they be done in any particular order. Django's admin focuses on **editing**, not on activities surrounding that editing. This avoidance of workflow also stems from the principle of trust: the admin's philosophy is that workflow is a personnel issue, not one to be implemented in code.

Finally, note the lack of aggregation in the admin. That is, there's no support for displaying totals, averages, etc. Again, the admin is for editing — it's expected that you'll write custom views for all the rest.

"...structured content"

As with the rest of Django, the admin wants you to work with structured data. Thus, the admin only supports editing data stored in Django models; for anything else, you'll need custom views.

Full stop

It should be clear by now that Django's admin does *not* try to be all things to all people; instead we choose to focus tightly on one thing, and do that thing extremely well.

When it comes to extending Django's admin, much of that same philosophy holds (note that "extensibility" shows up nowhere in our goals). Because custom Django views can do *anything* — and because they can easily be visually integrated into the admin (see below) — the built-in opportunities for customizing the admin are somewhat limited by design.

Customizing admin templates

Out of the box, you've got a number of tools for customizing the built-in admin templates which we'll go over below, but for tasks beyond that — anything requiring custom workflow or granular permissions, for example — you'll need to read the section on custom admin views at the end of this chapter.

For now, though, let's look at some quick ways of customizing the appearance (and, to some extent, behavior) of the admin. Chapter 6 covers a few of the most common tasks — "re-branding" the Django admin (for those pointy-haired bosses who hate blue) and providing a custom admin form.

Past that point, the goal usually involves changing some of the templates for a particular item. Each of the admin views — the change lists, edit forms, delete confirmation pages, and history views — has an associated template which can be overridden in a number of ways.

First, you can override the template globally. The admin view looks for templates using the standard template loading mechanism, so if you create templates in one of your template directories, Django will load those instead of the default admin templates bundles with Django.

These global templates are:

View	Base template name
Change list	`admin/change_list.html`
Add/edit form	`admin/change_form.html`
Delete confirmation	`admin/delete_confirmation.html`
Object history	`admin/object_history.html`

However, most of the time you'll want to change the template just for a single object or app (not globally). Thus, each admin view looks for model- and app-specific templates first. Those views look for templates in this order:

- `admin/<app_label>/<object_name>/<template>.html`

- `admin/<app_label>/<template>.html`
- `admin/<template>.html`

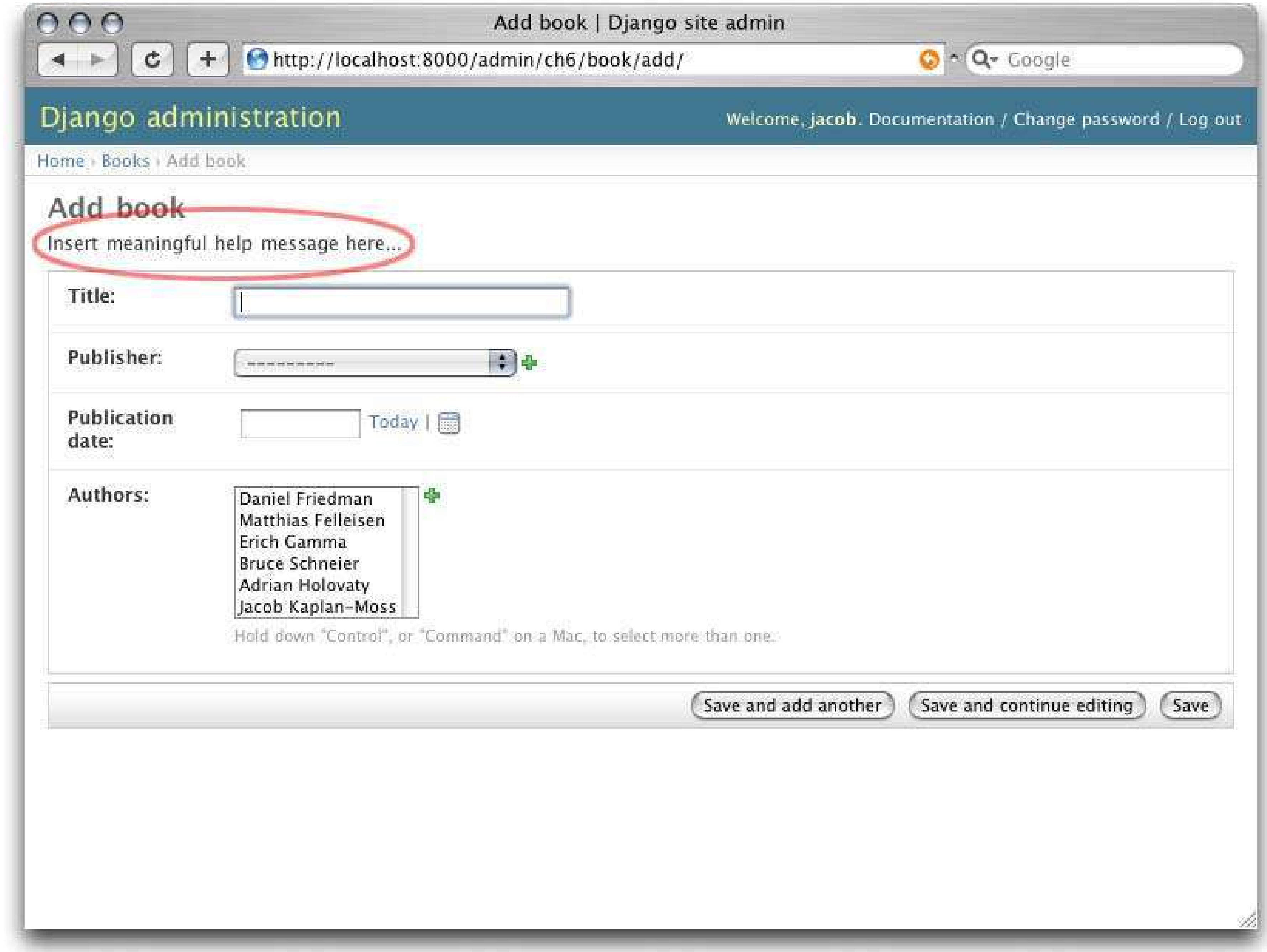
For example, the add/edit form view for a `Book` model in the `bookstore` app (i.e. the example from Chapter 6) looks for templates in this order:

- `admin/bookstore/book/change_form.html`
- `admin/bookstore/change_form.html`
- `admin/change_form.html`

Custom model templates

Most of the time, you'll usually want to use the first template to create a model-specific template; this is usually best done by extending the base template and adding information to one of the blocks defined in that template.

For example, let's say we wanted to add a little bit of help text to the top of that book page. Maybe something like this:


Add book | Django site admin
http://localhost:8000/admin/ch6/book/add/
Google
Django administration
Welcome, jacob. Documentation / Change password / Log out
Home › Books › Add book
Add book
Insert meaningful help message here...
Title:
Publisher:

Publication date:
Today |
Authors:
Daniel Friedman
Matthias Felleisen
Erich Gamma
Bruce Schneier
Adrian Holovaty
Jacob Kaplan-Moss
Hold down "Control", or "Command" on a Mac, to select more than one.
Save and add another
Save and continue editing
Save

This is pretty easy to do: simple create a template called admin/bookstore/book/change_form.html` , and insert this code:

```
{% extends "admin/change_form.html" %}
{% block form_top %}
  <p>Insert meaningful help message here...</p>
{% endblock %}
```

All these templates define a number of blocks you can override. As with most programs, the best documentation is the code, so we encourage you to look through the admin templates (they like in `django/contrib/admin/templates/`) for the most up-to-date information.

Custom JavaScript

A common use for these custom model templates involves adding custom JavaScript to admin pages — perhaps to implement some special widget or client-side behavior.

Luckily, that couldn't be easier. Each admin template defines a `{% block extrahead %}` which you can use to put extra content in to the `<head>` element. For example, if you wanted to include jQuery in one of your admin history, it's as simple as:

```
{% extends "admin/object_history.html" %}

{% block extrahead %}
    <script src="http://media.example.com/javascript/jquery.js" type="text/javascript"></script>
    <script type="text/javascript">

        // code to actually use jQuery here...

    </script>
{% endblock %}
```

(I'm not sure why you'd need jQuery on the object history page, but of course this example applies to any of the admin templates.)

You can use this technique to include any sort of extra JavaScript widgets you might need.

Custom admin views

At this point, anyone looking to add custom *behavior* to Django's admin is probably starting to get a bit frustrated. "All you've talked about is how to change the the admin *visually*," they'll cry, "but how do I change the way the admin *works*?"

Well, cry no more, for here comes the answer.

The first thing to understand is that **it's not magic**. That is, nothing the admin does is "special" in any way — the admin is just a set of views (they live in `django.contrib.admin.views`) that manipulate data just like any other view.

Sure, there's quite a bit of code in there; it has to deal with all the various options, field types, and settings that influence model behavior. Still, when you realize that the admin is just a set of views, adding custom admin views becomes easier to understand.

By way of example, let's add a "publisher report" view to our book app from Chapter 6. We'll build an admin view that shows the list of books broken down by publisher — a pretty typical example of a custom admin "report" view you might need to build.

First, we'll wire up a view in our URLconf. We need to insert this line:

```
(r'^admin/bookstore/report/$', 'bookstore.admin_views.report'),
```

before the line including the admin views. A bare-bones URLconf might look like:

```
from django.conf.urls.defaults import *

urlpatterns = patterns('',
    (r'^admin/bookstore/report/$', 'bookstore.admin_views.report'),
    (r'^admin/', include('django.contrib.admin.urls')),
)
```

Why put the custom view before the admin inclusion? Well, recall that Django processes URL patterns in order. Because the admin URLs match nearly anything that falls under the inclusion point, if we reverse the order of those lines Django will find a built-in admin view for that pattern, which of course won't work. In this particular case, it'll try to load a change list for a `Report` model in the `bookstore` app, which doesn't exist.

Now let's write our view. For the sake of simplicity, we'll just load all books into the context and let the template handle the grouping with the `{% regroup %}` tag. Create a file `bookstore/admin_views.py` with this code:

```
from bookstore.models import Book
from django.template import RequestContext
from django.shortcuts import render_to_response
from django.contrib.admin.views.decorators import staff_member_required

@staff_member_required
def report(request):
    return render_to_response(
        "admin/bookstore/report.html",
        {'book_list' : Book.objects.all()},
        RequestContext(request, {}),
    )
```

Because we left the grouping up to the template, this view is pretty simple. However, there are some subtle bits here worth making explicit:

- We use the `staff_member_required` decorator from `django.contrib.admin.views.decorators`. This is the similar to the `login_required` decorator discussed in Chapter 12, but this one also checks that the given user is marked as a "staff" member, and thus is allowed access to the admin.

  This decorator protects all the built-in admin views, and thus makes the authentication logic for your view match the rest of the admin.

- We render a template located under `admin/`. While this isn't strictly required, it's considered good practice to keep all your admin templates grouped in an `admin` directory. We've also put the template in a directory named `bookstore` after our app — also a best practice.

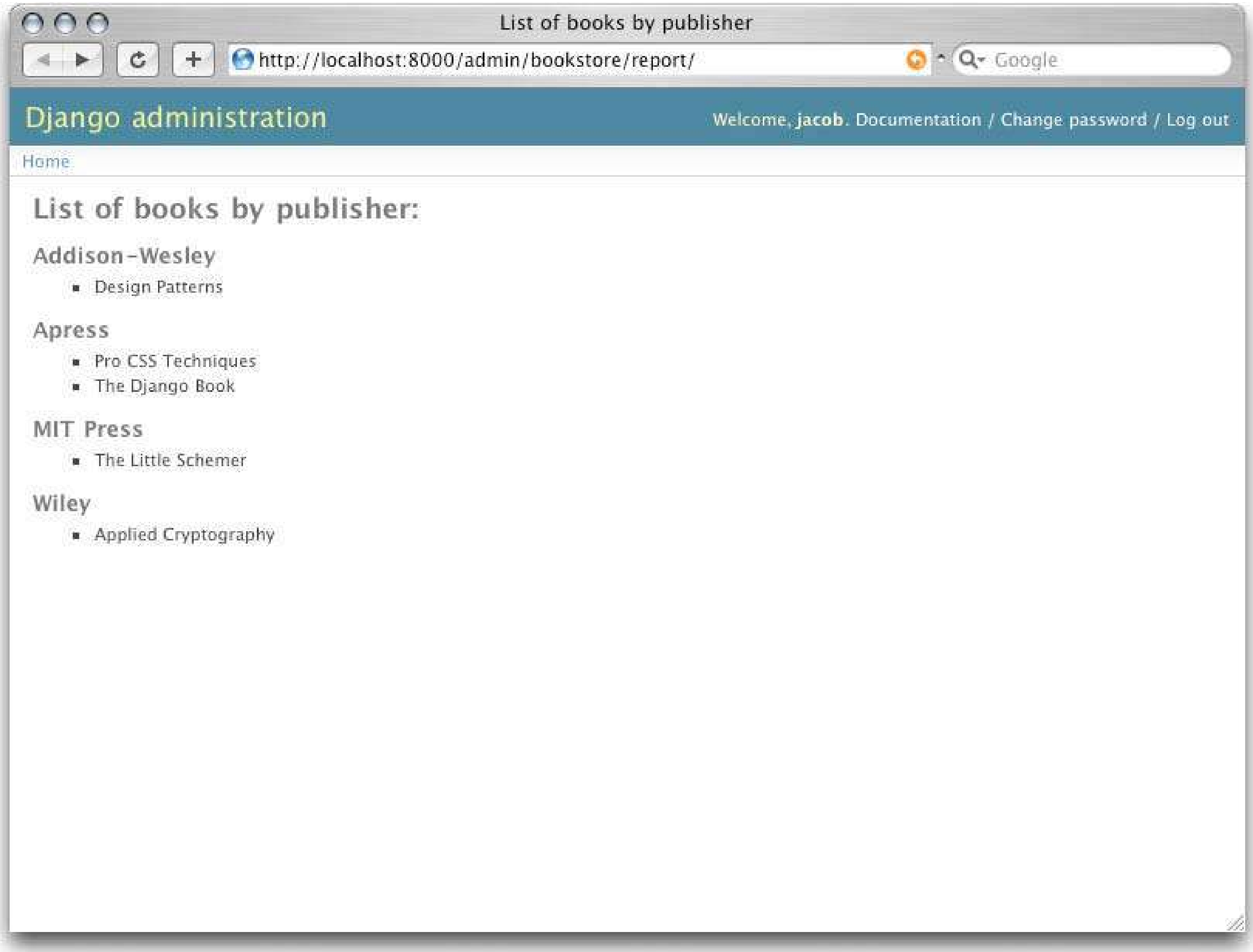
- We use `RequestContext` as the third parameter (`context_instance`) to `render_to_response`. This ensures that information about the current user is available to the template.

  See Chapter 10 for more about `RequestContext`.

Finally, we'll make a template for this view. We'll extend the built-in admin templates to make this view visually appear to be part of the admin:

```
{% extends "admin/base_site.html" %}

{% block title %}List of books by publisher{% endblock %}

{% block content %}
<div id="content-main">
  <h1>List of books by publisher:</h1>
  {% regroup book_list|dictsort:"publisher.name" by publisher as books_by_publisher %}
  {% for publisher in books_by_publisher %}
    <h3>{{ publisher.grouper }}</h3>
    <ul>
      {% for book in publisher.list|dictsort:"title" %}
            <li>{{ book }}</li>
      {% endfor %}
    </ul>
  {% endfor %}
</div>
{% endblock %}
```

By extending `admin/base_site.html` we get the look and feel of the Django admin "for free." Here's what the end result looks like:


List of books by publisher
http://localhost:8000/admin/bookstore/report/
Google
Django administration
Welcome, jacob. Documentation / Change password / Log out
Home

List of books by publisher:

Addison-Wesley

- Design Patterns

Apress

- Pro CSS Techniques
- The Django Book

MIT Press

- The Little Schemer

Wiley

- Applied Cryptography

Where do you want to admin today?

You can use this technique to add anything you can dream of to the admin. Remember that these so-called “custom admin views” are really just normal Django views; you can use all the techniques you learn in the rest of this book to provide as complex an admin as you need.

We'll close out this chapter with some ideas for custom admin views:

Overriding built-in views

At times the default admin views just don't cut it. You can easily swap in your own custom view for any stage of the admin; just let your URL shadow the built-in admin one.

For example, we could replace the built-in "create" view for a book with a form that lets the user simply enter an ISBN We could then look up the book's information from http://isbn.nu/ and create the object automatically.

The code for such a view is left as an exercise to the reader, but the important part is this URLconf snippet:

```
(r'^admin/bookstore/book/add/$', 'bookstore.admin_views.add_by_isbn'),
```

If this bit comes before the admin URLs in your URLconf, the `add_by_isbn` view will completely replace the standard admin view.

We could follow a similar tact to replace a delete confirmation page, the edit stage, or any other part of the admin.

Contribute!

This section is not yet complete. Are there other types of custom admin views you'd like covered? Leave a comment on this paragraph and let us know!

Chapter 19: Internationalization

Django has full support for internationalization of text in code and templates. Here's how it works.

Overview

The goal of internationalization is to allow a single Web application to offer its content and functionality in multiple languages.

You, the Django developer, can accomplish this goal by adding a minimal amount of hooks to your Python code and templates. These hooks are called **translation strings**. They tell Django: "This text should be translated into the end user's language, if a translation for this text is available in that language."

Django takes care of using these hooks to translate Web apps, on the fly, according to users' language preferences.

Essentially, Django does two things:

- It lets developers and template authors specify which parts of their apps should be translatable.
- It uses these hooks to translate Web apps for particular users according to their language preferences.

How to internationalize your app: in three steps

1. Embed translation strings in your Python code and templates.
2. Get translations for those strings, in whichever languages you want to support.
3. Activate the locale middleware in your Django settings.

Behind the scenes

Django's translation machinery uses the standard `gettext` module that comes with Python.

If you don't need internationalization

Django's internationalization hooks are on by default, and that means there's a bit of i18n-related overhead in certain places of the framework. If you don't use internationalization, you should take the two seconds to set `USE_I18N = False` in your settings file. If `USE_I18N` is set to `False`, then Django will make some optimizations so as not to load the internationalization machinery.

You'll probably also want to remove `'django.core.context_processors.i18n'` from your `TEMPLATE_CONTEXT_PROCESSORS` setting.

How to specify translation strings

Translation strings specify "This text should be translated." These strings can appear in your Python code and templates. It's your responsibility to mark translatable strings; the system can only translate strings it knows about.

In Python code

Standard translation

Specify a translation string by using the function `_()`. (Yes, the name of the function is the "underscore" character.) This function is available globally in any Python module; you don't have to import it.

In this example, the text `"Welcome to my site."` is marked as a translation string:

```
def my_view(request):
    output = _("Welcome to my site.")
    return HttpResponse(output)
```

The function `django.utils.translation.gettext()` is identical to `_()`. This example is identical to the previous one:

```
from django.utils.translation import gettext
def my_view(request):
    output = gettext("Welcome to my site.")
    return HttpResponse(output)
```

Translation works on computed values. This example is identical to the previous two:

```
def my_view(request):
    words = ['Welcome', 'to', 'my', 'site.']
    output = _(' '.join(words))
    return HttpResponse(output)
```

Translation works on variables. Again, here's an identical example:

```
def my_view(request):
    sentence = 'Welcome to my site.'
    output = _(sentence)
    return HttpResponse(output)
```

(The caveat with using variables or computed values, as in the previous two examples, is that Django's translation-string-detecting utility, `make-messages.py`, won't be able to find these strings. More on `make-messages` later.)

The strings you pass to `_()` or `gettext()` can take placeholders, specified with Python's standard named-string interpolation syntax. Example:

```
def my_view(request, n):
    output = _('%(name)s is my name.') % {'name': n}
    return HttpResponse(output)
```

This technique lets language-specific translations reorder the placeholder text. For example, an English translation may be `"Adrian is my name."`, while a Spanish translation may be `"Me llamo Adrian."` — with the placeholder (the name) placed after the translated text instead of before it.

For this reason, you should use named-string interpolation (e.g., `%(name)s`) instead of positional interpolation (e.g., `%s` or `%d`). If you used positional interpolation, translations wouldn't be able to reorder placeholder text.

Marking strings as no-op

Use the function `django.utils.translation.gettext_noop()` to mark a string as a translation string without translating it. The string is later translated from a variable.

Use this if you have constant strings that should be stored in the source language because they are exchanged over systems or users — such as strings in a database — but should be translated at the last possible point in time, such as when the string is presented to the user.

Lazy translation

Use the function `django.utils.translation.gettext_lazy()` to translate strings lazily — when the value is accessed rather than when the `gettext_lazy()` function is called.

For example, to translate a model's `help_text`, do the following:

```
from django.utils.translation import gettext_lazy

class MyThing(models.Model):
    name = models.CharField(help_text=gettext_lazy('This is the help text'))
```

In this example, `gettext_lazy()` stores a lazy reference to the string — not the actual translation. The translation itself will be done when the string is used in a string context, such as template rendering on the Django admin site.

If you don't like the verbose name `gettext_lazy`, you can just alias it as `_` (underscore), like so:

```
from django.utils.translation import gettext_lazy as _

class MyThing(models.Model):
    name = models.CharField(help_text=_('This is the help text'))
```

Always use lazy translations in Django models. And it's a good idea to add translations for the field names and table names, too. This means writing explicit `verbose_name` and `verbose_name_plural` options in the `Meta` class, though:

```
from django.utils.translation import gettext_lazy as _

class MyThing(models.Model):
    name = models.CharField(_('name'), help_text=_('This is the help text'))
    class Meta:
        verbose_name = _('my thing')
        verbose_name_plural = _('mythings')
```

Pluralization

Use the function `django.utils.translation.ngettext()` to specify pluralized messages. Example:

```
from django.utils.translation import ngettext
def hello_world(request, count):
    page = ngettext('there is %(count)d object', 'there are %(count)d objects', count) % {
        'count': count,
    }
    return HttpResponse(page)
```

`ngettext` takes three arguments: the singular translation string, the plural translation string and the number of objects (which is passed to the translation languages as the `count` variable).

In template code

Using translations in Django templates uses two template tags and a slightly different syntax than in Python code. To give your template access to these tags, put `{% load i18n %}` toward the top of your template.

The `{% trans %}` template tag translates a constant string or a variable content:

```
<title>{% trans "This is the title." %}</title>
```

If you only want to mark a value for translation, but translate it later from a variable, use the `noop` option:

```
<title>{% trans "value" noop %}</title>
```

It's not possible to use template variables in `{% trans %}` — only constant strings, in single or double quotes, are allowed. If your translations require variables (placeholders), use `{% blocktrans %}`. Example:

```
{% blocktrans %}This will have {{ value }} inside.{% endblocktrans %}
```

To translate a template expression — say, using template filters — you need to bind the expression to a local variable for use within the translation block:

```
{% blocktrans with value|filter as myvar %}
This will have {{ myvar }} inside.
{% endblocktrans %}
```

If you need to bind more than one expression inside a `blocktrans` tag, separate the pieces with `and`:

```
{% blocktrans with book|title as book_t and author|title as author_t %}
This is {{ book_t }} by {{ author_t }}
{% endblocktrans %}
```

To pluralize, specify both the singular and plural forms with the `{% plural %}` tag, which appears within `{% blocktrans %}` and `{% endblocktrans %}`. Example:

```
{% blocktrans count list|count as counter %}
There is only one {{ name }} object.
{% plural %}
There are {{ counter }} {{ name }} objects.
{% endblocktrans %}
```

Internally, all block and inline translations use the appropriate `gettext` / `ngettext` call.

Each `RequestContext` has access to two translation-specific variables:

- `LANGUAGES` is a list of tuples in which the first element is the language code and the second is the language name (in that language).
- `LANGUAGE_CODE` is the current user's preferred language, as a string. Example: `en-us`. (See "How language preference is discovered", below.)
- `LANGUAGE_BIDI` is the current language's direction. If True, it's a right-to-left language, e.g: Hebrew, Arabic. If False it's a

left-to-right language, e.g: English, French, German etc.

If you don't use the `RequestContext` extension, you can get those values with three tags:

```
{% get_current_language as LANGUAGE_CODE %}
{% get_available_languages as LANGUAGES %}
{% get_current_language_bidi as LANGUAGE_BIDI %}
```

These tags also require a `{% load i18n %}`.

Translation hooks are also available within any template block tag that accepts constant strings. In those cases, just use `_()` syntax to specify a translation string. Example:

```
{% some_special_tag _("Page not found") value|yesno:_("yes,no") %}
```

In this case, both the tag and the filter will see the already-translated string, so they don't need to be aware of translations.

How to create language files

Once you've tagged your strings for later translation, you need to write (or obtain) the language translations themselves. Here's how that works.

Message files

The first step is to create a **message file** for a new language. A message file is a plain-text file, representing a single language, that contains all available translation strings and how they should be represented in the given language. Message files have a `.po` file extension.

Django comes with a tool, `bin/make-messages.py`, that automates the creation and upkeep of these files.

To create or update a message file, run this command:

```
bin/make-messages.py -l de
```

...where `de` is the language code for the message file you want to create. The language code, in this case, is in locale format. For example, it's `pt_BR` for Brazilian and `de_AT` for Austrian German.

The script should be run from one of three places:

- The root `django` directory (not a Subversion checkout, but the one that is linked-to via `$PYTHONPATH` or is located somewhere on that path).
- The root directory of your Django project.
- The root directory of your Django app.

The script runs over the entire Django source tree and pulls out all strings marked for translation. It creates (or updates) a message file in the directory `conf/locale`. In the `de` example, the file will be `conf/locale/de/LC_MESSAGES/django.po`.

If run over your project source tree or your application source tree, it will do the same, but the location of the locale directory is `locale/LANG/LC_MESSAGES` (note the missing `conf` prefix).

No gettext?

If you don't have the `gettext` utilities installed, `make-messages.py` will create empty files. If that's the case, either install the `gettext` utilities or just copy the English message file (`conf/locale/en/LC_MESSAGES/django.po`) and use it as a starting point; it's just an empty translation file.

The format of `.po` files is straightforward. Each `.po` file contains a small bit of metadata, such as the translation maintainer's contact information, but the bulk of the file is a list of **messages** — simple mappings between translation strings and the actual translated text for the particular language.

For example, if your Django app contained a translation string for the text `"Welcome to my site."`, like so:

```
_("Welcome to my site.")
```

...then `make-messages.py` will have created a `.po` file containing the following snippet — a message:

```
#: path/to/python/module.py:23
msgid "Welcome to my site."
```

```
msgstr ""
```

A quick explanation:

- `msgid` is the translation string, which appears in the source. Don't change it.
- `msgstr` is where you put the language-specific translation. It starts out empty, so it's your responsibility to change it. Make sure you keep the quotes around your translation.
- As a convenience, each message includes the filename and line number from which the translation string was gleaned.

Long messages are a special case. There, the first string directly after the `msgstr` (or `msgid`) is an empty string. Then the content itself will be written over the next few lines as one string per line. Those strings are directly concatenated. Don't forget trailing spaces within the strings; otherwise, they'll be tacked together without whitespace!

Mind your charset

When creating a `.po` file with your favorite text editor, first edit the charset line (search for `"CHARSET"`) and set it to the charset you'll be using to edit the content. Generally, utf-8 should work for most languages, but `gettext` should handle any charset you throw at it.

To reexamine all source code and templates for new translation strings and update all message files for **all** languages, run this:

```
make-messages.py -a
```

Compiling message files

After you create your message file — and each time you make changes to it — you'll need to compile it into a more efficient form, for use by `gettext`. Do this with the `bin/compile-messages.py` utility.

This tool runs over all available `.po` files and creates `.mo` files, which are binary files optimized for use by `gettext`. In the same directory from which you ran `make-messages.py`, run `compile-messages.py` like this:

```
bin/compile-messages.py
```

That's it. Your translations are ready for use.

How Django discovers language preference

Once you've prepared your translations — or, if you just want to use the translations that come with Django — you'll just need to activate translation for your app.

Behind the scenes, Django has a very flexible model of deciding which language should be used — installation-wide, for a particular user, or both.

To set an installation-wide language preference, set `LANGUAGE_CODE` in your settings file. Django uses this language as the default translation — the final attempt if no other translator finds a translation.

If all you want to do is run Django with your native language, and a language file is available for your language, all you need to do is set `LANGUAGE_CODE`.

If you want to let each individual user specify which language he or she prefers, use `LocaleMiddleware`. `LocaleMiddleware` enables language selection based on data from the request. It customizes content for each user.

To use `LocaleMiddleware`, add `'django.middleware.locale.LocaleMiddleware'` to your `MIDDLEWARE_CLASSES` setting. Because middleware order matters, you should follow these guidelines:

- Make sure it's one of the first middlewares installed.
- It should come after `SessionMiddleware`, because `LocaleMiddleware` makes use of session data.
- If you use `CacheMiddleware`, put `LocaleMiddleware` after it.

For example, your `MIDDLEWARE_CLASSES` might look like this:

```
MIDDLEWARE_CLASSES = (
   'django.contrib.sessions.middleware.SessionMiddleware',
   'django.middleware.locale.LocaleMiddleware',
   'django.middleware.common.CommonMiddleware',
)
```

`LocaleMiddleware` tries to determine the user's language preference by following this algorithm:

- First, it looks for a `django_language` key in the the current user's session.
- Failing that, it looks for a cookie called `django_language`.
- Failing that, it looks at the `Accept-Language` HTTP header. This header is sent by your browser and tells the server which language(s) you prefer, in order by priority. Django tries each language in the header until it finds one with available translations.
- Failing that, it uses the global `LANGUAGE_CODE` setting.

Notes:

- In each of these places, the language preference is expected to be in the standard language format, as a string. For example, Brazilian is `pt-br`.
- If a base language is available but the sublanguage specified is not, Django uses the base language. For example, if a user specifies `de-at` (Austrian German) but Django only has `de` available, Django uses `de`.
- Only languages listed in the `LANGUAGES` setting can be selected. If you want to restrict the language selection to a subset of provided languages (because your application doesn't provide all those languages), set `LANGUAGES` to a list of languages. For example:

  ```
  LANGUAGES = (
    ('de', _('German')),
    ('en', _('English')),
  )
  ```

  This example restricts languages that are available for automatic selection to German and English (and any sublanguage, like de-ch or en-us).

- If you define a custom `LANGUAGES` setting, as explained in the previous bullet, it's OK to mark the languages as translation strings — but use a "dummy" `gettext()` function, not the one in `django.utils.translation`. You should *never* import `django.utils.translation` from within your settings file, because that module in itself depends on the settings, and that would cause a circular import.

  The solution is to use a "dummy" `gettext()` function. Here's a sample settings file:

  ```
  gettext = lambda s: s

  LANGUAGES = (
      ('de', gettext('German')),
      ('en', gettext('English')),
  )
  ```

  With this arrangement, `make-messages.py` will still find and mark these strings for translation, but the translation won't happen at runtime — so you'll have to remember to wrap the languages in the *real* `gettext()` in any code that uses `LANGUAGES` at runtime.

- The `LocaleMiddleware` can only select languages for which there is a Django-provided base translation. If you want to provide translations for your application that aren't already in the set of translations in Django's source tree, you'll want to provide at least basic translations for that language. For example, Django uses technical message IDs to translate date formats and time formats — so you will need at least those translations for the system to work correctly.

  A good starting point is to copy the English `.po` file and to translate at least the technical messages — maybe the validator messages, too.

  Technical message IDs are easily recognized; they're all upper case. You don't translate the message ID as with other messages, you provide the correct local variant on the provided English value. For example, with `DATETIME_FORMAT` (or `DATE_FORMAT` or `TIME_FORMAT`), this would be the format string that you want to use in your language. The format is identical to the format strings used by the `now` template tag.

Once `LocaleMiddleware` determines the user's preference, it makes this preference available as `request.LANGUAGE_CODE` for each request object. Feel free to read this value in your view code. Here's a simple example:

```
def hello_world(request, count):
    if request.LANGUAGE_CODE == 'de-at':
        return HttpResponse("You prefer to read Austrian German.")
    else:
        return HttpResponse("You prefer to read another language.")
```

Note that, with static (middleware-less) translation, the language is in `settings.LANGUAGE_CODE`, while with dynamic (middleware) translation, it's in `request.LANGUAGE_CODE`.

The `set_language` redirect view

As a convenience, Django comes with a view, `django.views.i18n.set_language`, that sets a user's language preference and redirects back to the previous page.

Activate this view by adding the following line to your URLconf:

```
(r'^i18n/', include('django.conf.urls.i18n')),
```

(Note that this example makes the view available at `/i18n/setlang/`.)

The view expects to be called via the `GET` method, with a `language` parameter set in the query string. If session support is enabled, the view saves the language choice in the user's session. Otherwise, it saves the language choice in a `django_language` cookie.

After setting the language choice, Django redirects the user, following this algorithm:

- Django looks for a `next` parameter in the query string.
- If that doesn't exist, or is empty, Django tries the URL in the `Referer` header.
- If that's empty — say, if a user's browser suppresses that header — then the user will be redirected to `/` (the site root) as a fallback.

Here's example HTML template code:

```
<form action="/i18n/setlang/" method="get">
<input name="next" type="hidden" value="/next/page/" />
<select name="language">
{% for lang in LANGUAGES %}
<option value="{{ lang.0 }}">{{ lang.1 }}</option>
{% endfor %}
</select>
<input type="submit" value="Go" />
</form>
```

Using translations in your own projects

Django looks for translations by following this algorithm:

- First, it looks for a `locale` directory in the application directory of the view that's being called. If it finds a translation for the selected language, the translation will be installed.
- Next, it looks for a `locale` directory in the project directory. If it finds a translation, the translation will be installed.
- Finally, it checks the base translation in `django/conf/locale`.

This way, you can write applications that include their own translations, and you can override base translations in your project path. Or, you can just build a big project out of several apps and put all translations into one big project message file. The choice is yours.

Note

If you're using manually configured settings, the `locale` directory in the project directory will not be examined, since Django loses the ability to work out the location of the project directory. (Django normally uses the location of the settings file to determine this, and a settings file doesn't exist if you're manually configuring your settings.)

All message file repositories are structured the same way. They are:

- `$APPPATH/locale/<language>/LC_MESSAGES/django.(po|mo)`
- `$PROJECTPATH/locale/<language>/LC_MESSAGES/django.(po|mo)`
- All paths listed in `LOCALE_PATHS` in your settings file are searched in that order for `<language>/LC_MESSAGES/django.(po|mo)`
- `$PYTHONPATH/django/conf/locale/<language>/LC_MESSAGES/django.(po|mo)`

To create message files, you use the same `make-messages.py` tool as with the Django message files. You only need to be in the right place — in the directory where either the `conf/locale` (in case of the source tree) or the `locale/` (in case of app messages or project messages) directory are located. And you use the same `compile-messages.py` to produce the binary `django.mo` files that are used by `gettext`.

Application message files are a bit complicated to discover — they need the `LocaleMiddleware`. If you don't use the middleware, only the Django message files and project message files will be processed.

Finally, you should give some thought to the structure of your translation files. If your applications need to be delivered to other users and will be used in other projects, you might want to use app-specific translations. But using app-specific translations and project translations could produce weird problems with `make-messages`: `make-messages` will traverse all directories below the current path and so might put message IDs into the project message file that are already in application message files.

The easiest way out is to store applications that are not part of the project (and so carry their own translations) outside the project tree. That way, `make-messages` on the project level will only translate strings that are connected to your explicit project and not strings that are distributed independently.

Translations and JavaScript

Adding translations to JavaScript poses some problems:

- JavaScript code doesn't have access to a `gettext` implementation.
- JavaScript code doesn't have access to .po or .mo files; they need to be delivered by the server.
- The translation catalogs for JavaScript should be kept as small as possible.

Django provides an integrated solution for these problems: It passes the translations into JavaScript, so you can call `gettext`, etc., from within JavaScript.

The `javascript_catalog` view

The main solution to these problems is the `javascript_catalog` view, which sends out a JavaScript code library with functions that mimic the `gettext` interface, plus an array of translation strings. Those translation strings are taken from the application, project or Django core, according to what you specify in either the {{{info_dict}}} or the URL.

You hook it up like this:

```
js_info_dict = {
    'packages': ('your.app.package',),
}

urlpatterns = patterns('',
    (r'^jsi18n/$', 'django.views.i18n.javascript_catalog', js_info_dict),
)
```

Each string in `packages` should be in Python dotted-package syntax (the same format as the strings in `INSTALLED_APPS`) and should refer to a package that contains a `locale` directory. If you specify multiple packages, all those catalogs are merged into one catalog. This is useful if you have JavaScript that uses strings from different applications.

You can make the view dynamic by putting the packages into the URL pattern:

```
urlpatterns = patterns('',
    (r'^jsi18n/(?P<packages>\S+?)/$, 'django.views.i18n.javascript_catalog'),
)
```

With this, you specify the packages as a list of package names delimited by '+' signs in the URL. This is especially useful if your pages use code from different apps and this changes often and you don't want to pull in one big catalog file. As a security measure, these values can only be either `django.conf` or any package from the `INSTALLED_APPS` setting.

Using the JavaScript translation catalog

To use the catalog, just pull in the dynamically generated script like this:

```
<script type="text/javascript" src="/path/to/jsi18n/"></script>
```

This is how the admin fetches the translation catalog from the server. When the catalog is loaded, your JavaScript code can use the standard `gettext` interface to access it:

```
document.write(gettext('this is to be translated'));
```

There even is a `ngettext` interface and a string interpolation function:

```
d = {
    count: 10
};
s = interpolate(ngettext('this is %(count)s object', 'this are %(count)s objects', d.count),
d);
```

The `interpolate` function supports both positional interpolation and named interpolation. So the above could have been written as:

```
s = interpolate(ngettext('this is %s object', 'this are %s objects', 11), [11]);
```

The interpolation syntax is borrowed from Python. You shouldn't go over the top with string interpolation, though: this is still JavaScript, so the code will have to do repeated regular-expression substitutions. This isn't as fast as string interpolation in Python, so keep it to those cases where you really need it (for example, in conjunction with `ngettext` to produce proper pluralizations).

Creating JavaScript translation catalogs

You create and update the translation catalogs the same way as the other Django translation catalogs — with the {{{make-messages.py}}} tool. The only difference is you need to provide a `-d djangojs` parameter, like this:

```
make-messages.py -d djangojs -l de
```

This would create or update the translation catalog for JavaScript for German. After updating translation catalogs, just run `compile-messages.py` the same way as you do with normal Django translation catalogs.

Notes for users familiar with `gettext`

If you know `gettext`, you might note these specialities in the way Django does translation:

- The string domain is `django` or `djangojs`. The string domain is used to differentiate between different programs that store their data in a common message-file library (usually `/usr/share/locale/`). The `django` domain is used for python and template translation strings and is loaded into the global translation catalogs. The `djangojs` domain is only used for JavaScript translation catalogs to make sure that those are as small as possible.
- Django only uses `gettext` and `gettext_noop`. That's because Django always uses `DEFAULT_CHARSET` strings internally. There isn't much use in using `ugettext`, because you'll always need to produce utf-8 anyway.
- Django doesn't use `xgettext` alone. It uses Python wrappers around `xgettext` and `msgfmt`. That's mostly for convenience.

« previous ◊ table of contents ◊ next »

Security

The internet can be a scary place.

In the past few years, internet horror stories have been in the news almost continuously. We've seen viruses spread with amazing speed, swarms of compromised computers wielded as weapons, a never-ending arms race against spammers, and many, many reports of identify theft from compromised web sites.

As good web developers, it's our duty to do what we can to combat these forces of darkness. Every web developer needs to treat security as a fundamental aspect of web programming. Unfortunately, it turns out that security is *hard* — attackers only need to find a single vulnerability, but defenders have to protect every single one.

Django attempts to mitigate this difficulty. It's designed to automatically protect you from many of the common security mistakes that new (and even experienced) web developers make. Still, it's important to understand what these problems are, how Django protects you, and — most importantly — the steps you can take to make your code even more secure.

First, though, an important disclaimer: we're in no way experts in this realm, and so we won't try to explain each vulnerability in a comprehensive manner. Instead, we'll give a short synopsis of security problems as they apply to Django.

The theme of web security

If you learn only one thing from this chapter, let it be this:

> Never —under any circumstances—trust data from the browser.

You *never* know who's on the other side of that HTTP connection. It might be one of your users, but it just as easily could be a cracker or script kiddie looking for an opening.

Any data of any nature that comes from the browser needs to be treated with a healthy dose of paranoia. This includes data that's both "in band" — i.e. submitted from web forms — and "out of band" — i.e. HTTP headers, cookies, and other request info. It's trivial to spoof the request metadata that browsers usually add automatically.

Every one of the vulnerabilities discussed in this chapter stems directly from trusting data that comes over the wire and then failing to sanitize that data before using it. You should make it a general practice to continuously ask, "where does this data come from?".

SQL injection

SQL injection is a common exploit in which an attacker alters Web-page parameters (such as `GET`/`POST` data or URLs) to insert arbitrary SQL snippets that a naive Web application executes in its database directly. It's probably the most dangerous — and unfortunately one of the most common — vulnerabilities in the wild.

This vulnerability most commonly crops up when constructing SQL "by hand" from user input. For example, imagine writing a function to gather a list of a contact info from a contact search page. To prevent spammers from reading every single email in our system, we'll force the user to type in someone's username before we provide their email address:

```
def user_contacts(request):
    user = request.GET['username']
    sql = "SELECT * FROM user_contacts WHERE username = '%s';" % username
    # execute the SQL here...
```

Note

In this example, and all similar "don't do this" examples that follow, we've deliberately left out most of the code needed to make the functions actually work. We don't want this code to work if someone accidentally takes it out of context.

Though at first this doesn't look dangerous, it really is.

First, our attempt at protecting our entire email list will fail with a cleverly constructed query. Think about what happens if an attacker types `"' OR 'a'='a"` into the query box. In that case, the query that the string interpolation will construct will be:

```
SELECT * FROM user_contacts WHERE username = '' OR 'a' = 'a';
```

Because we allowed unsecured SQL into the string, the attacker's added `OR` clause ensures that every single row is returned.

However, that's the *least* scary attack. Imagine what will happen if the attacker submits `"'; DELETE FROM user_contacts WHERE 'a' = 'a`". We'll end up with this complete query:

```
SELECT * FROM user_contacts WHERE username = ''; DELETE FROM user_contacts WHERE 'a' = 'a';
```

Yikes! Where'd our contact list go?

The solution

Although this problem is insidious and sometimes hard to spot, the solution is simple: *never* trust user-submitted data, and *always* escape it when passing it into SQL.

The Django database API does this for you. It automatically escapes all special SQL parameters, according to the quoting conventions of the database server you're using (e.g. PostgreSQL, MySQL).

For example, in this API call:

```
foo.get_list(bar__exact="' OR 1=1")
```

Django will escape the input accordingly, resulting in a statement like this:

```
SELECT * FROM foos WHERE bar = '\' OR 1=1'
```

Completely harmless.

This applies to the entire Django database API, with a couple of exceptions:

- The `where` argument to the `extra()` method (see Appendix XXX). That parameter accepts raw SQL by design.
- Queries done "by hand" using the lower-level database API.

In each of these cases, it's easy to keep yourself protected. In each case, avoid string interpolation in favor of passing in "bind parameters". That is, the example we started this section with should be written:

```
from django.db import connection

def user_contacts(request):
    user = request.GET['username']
    sql = "SELECT * FROM user_contacts WHERE username = %s;"
    cursor = connection.cursor()
    cursor.execute(sql, [user])
    # ... do something with the results
```

The low-level `execute` method takes a SQL string with `%s` placeholders, and automatically escapes and inserts parameters from the list passed as the second argument. You should *always* construct custom SQL this way.

Unfortunately, you can't use bind parameters everywhere in SQL; they're not allowed as identifiers (i.e. table or column names). Thus, if you need to, say, dynamically construct a list of tables from a `POST` variable, you'll need to escape that name in your code. Django provides a function, `django.db.backend.quote_name`, which will escape the identifier according to the current database's quoting scheme.

Cross-site Scripting (XSS)

Probably the most common web vulnerability, **cross-site scripting**, or **XSS**, is found in web applications that fail to properly escape user-submitted content before rendering it into HTML. This allows an attacker to maliciously insert arbitrary HTML, usually in the form of `<script>` tags.

Attackers often use XSS attacks to steal cookie and session info, or to trick users into giving private information to the wrong person (a.k.a **phishing**).

This type of attack can take a number of different forms, and has almost infinite permutations, so we'll just look at a typical example. Let's look at a extremely simple "hello world" view:

```
def say_hello(request):
    name = request.GET.get('name', 'world')
    return render_to_response("hello.html", {"name" : name})
```

This view simply reads a name from a `GET` parameter and passes that name to the `hello.html` template. We might write a template for this view like:

```
<h1>Hello, {{ name }}!</h1>
```

So if we accessed `http://example.com/hello/name=Jacob`, the rendered page would contain:

```
<h1>Hello, Jacob!</h1>
```

But wait — what happens if we access `http://example.com/hello/name=<i>Jacob</i>`? Then we'd get:

```
<h1>Hello, <i>Jacob</i>!</h1>
```

Of course, an attacker wouldn't use something as benign as `<i>` tags; he could include a whole set of HTML that hijacked your page with arbitrary content. This type of attack has been used to trick users into entering data into what looks like their bank's website, but in fact is an XSS-hijacked form that submits your back account information to an attacker.

This gets worse if you storing this data in the database and later display it it on your site.

For example, at one point MySpace was found to be vulnerable to an XSS attack of this nature. A user inserted javascript into his profile that automatically added him as your friend when you visited his profile page. Within a few days he had millions of friends.

Now, this may sound relatively benign, but keep in mind that this attacker managed to get *his* code — not MySpace's — running on *your* computer. This violoates the assumed trust that all the code on MySpace is actually written by MySpace.

MySpace was extremely lucky that this malicious code didn't automatically delete viewer's accounts, change their passwords, flood the site with spam, or any of the other nightmare scenarios this vulnerability unleashes.

The solution

The solution is simple: *always* escape *any* content that might have come from a user. If we simply rewrite our template as:

```
<h1>Hello, {{ name|escape }}!</h1>
```

then we're no longer vulnerable. You should *always* use the `escape` tag (or an analogue) when displaying user-submitted content on your site.

Why doesn't Django just do this for you?

Modifying Django to automatically escape all variables displayed in templates is a frequent topic of discussion on the Django developer mailing list.

So far, Django's templates have avoided this behavior because it subtley and invisibly changes what should be relatively streightforward behavior (displaying variables). It's a tricky issue and a difficult trade-off to evaluate. Adding hidden implicit behavior is against Django's core ideals (and Python's, for that matter), but security is equally important.

All this to say, then, that there's a fair chance that Django will grow some form of auto-escaping (or nearly-auto-escaping) behavior in the future. It's always a good idea to check the official Django documentation; it'll always be more up-to-date than this book (especially the dead-tree version).

Even if Django does add this feature, however, you should *still* be in the habit of thinking "where does this data come from?" at all times. No automatic solution will ever protect your site from XSS attacks 100% of the time.

Cross-site Request Forgery (CSRF)

CSRF happens when a malicious Web site tricks a user into unknowingly loading a URL from a site at which they're already authenticated — hence, taking advantage of their authenticated status.

Django has built-in tools to protect from this kind of attack; both the attack itself and those tools are covered in great detail in Chapter 15.

Session forging/hijacking

This isn't a specific attack, but rather a general class of attacks on a user's session data. It can take a number of different forms:

- A **man-in-the-middle** attack, where an attacker snoops on session data as it travels over the wire (or wireless) network.
- **Session forging**, where an attacker uses a fake session ID (perhaps obtained through a man-in-the-middle attack) to pretend to be another user.

  An example of this first two would be an attacker in a coffee shop using the wireless network to capture a session cookie; he could then use that cookie to impersonate the original user.

- A **cookie forging** attack, where an attacker overrides the supposedly read-only data stored in a cookie. Chapter 12 explains in details how cookies work, and one of the salient points is that it's trivial for browsers and malicious users to change cookies without your knowledge.

  There's a long history of web sites that have stored a cookie like `IsLoggedIn=1` or even `LoggedInAsUser=jacob`; it's almost too easy to exploit these types of attackers.

  On a more subtle level, though, it's never a good idea to trust anything stored in a cookie; you never know who's been poking at them.

- **Session fixation**, where an attacker tricks a user into setting or reseting their session ID.

  For example, PHP allows session identifiers to be passed in the URL (i.e. `http://example.com/?PHPSESSID=fa90197ca25f6ab40bb1374c510d7a32`). An attacker who tricks a user into clicking on a link with a hardcoded session ID will cause the user to pick up that session.

  This has been used in phishing attacks to trick users into entering personal information into the an account which the attacker owns; he can later log into that account and retrieve the data.

- **Session poisoning**, where an attacker injects potentially dangerous data into a user's session — usually through a web form that the user submits to set session data.

  A canonical example is a site that stores a simple user preference (like a page's background color) in a cookie. An attacker could trick a user into clicking on a link to submit a "color" that actually contains an XSS attack; if that color isn't escaped (see above) the user could again inject malicious code into the user's environment.

The solution

There are a number of general principles that can protect from these attacks:

- Never allow session information to be contained in the URL.

  Django's session framework (see Chapter 12) simply doesn't allow sessions to be contained in the URL.

- Don't store data in cookies directly; instead, store a session ID that maps to session data stored on the backend.

  If you use Django's built-in session framework (i.e. `request.session`), this is handled automatically for you. The only cookie that the session framework uses is a single session ID; all the session data is stored in the database.

- Remember to escape session data if you display it in the template. See the XSS section above, and remember that it applies to any user-created content. You should treat session information as user-created.

- Prevent attackers from spoofing session IDs whenever possible.

  Although it's nearly impossible to detect someone who's hijacked a session ID, Django does have built-in protection against a brute-force session attack. Session IDs are stored as hashes (instead of sequential numbers) which prevents a brute-force attack, and a user will always get a new session ID if they try a non-existent one which prevent session fixation.

Notice that none of those principles and tools prevent man-in-the-middle attacks. These types of attacks are nearly impossible to detect. If your site allows logged-in users to see any sort of sensitive data, you should *always* serve that site over HTTPS. Additionally, if you've got an SSL-enabled site, you should set the `SESSION_COOKIE_SECURE` setting to `True`; this will make Django only send session cookies over HTTPS.

E-mail header injection

SQL injection's less-well-known sibling, **e-mail header injection** hijacks email-sending web forms and uses them to send spam. Any form that constructs email headers from web form data is a target for this kind of attack.

Let's look at the canonical contact form found on many sites. Usually this emails a hard-coded email address, and so at first glance doesn't appear vulnerable to spam abuse.

However, most of these forms also allow the user to type in his own subject for the email (along with a from address, body, and sometimes a few other fields). This subject field is used to construct the "subject" header of the email message.

If that header is unescaped when building the email message, an attacker could use something like `"hello\ncc:spamvictim@example.com"` (where `"\n`" is a newline character). That would make the constructed email headers turn into:

```
To: hardcoded@example.com
Subject: hello
cc: spamvictim@example.com
```

Like SQL injection, if we trust the subject line given by the user, we'll allow him to construct a malicious set of headers, and they can use our contact form to send spam.

The solution

We can prevent this attack in the same way we prevent SQL injection: always escape or validate user-submitted content.

Django's built-in mail functions (in `django.core.mail`) simply do not allow newlines in any fields used to construct headers (the from and to addresses and the subject). If you try to use `django.core.mail.send_mail` with a subject that contains newlines, Django will raise a `BadHeaderError` exception.

If you decide to use other methods of sending email, you'll need to make sure that newlines in headers either cause an error or are stripped. You may want to examine the `SafeMIMEText` class in `django.core.mail` to see how Django does this.

Directory traversal

Directory traversal is another injection-style attack wherein a malicious user tricks filesystem code into reading and/or writing files that the web server shouldn't have access to.

An example might be a view that reads files from the disk without carefully sanitizing the file name:

```
def dump_file(request):
    filename = request.GET["filename"]
    filename = os.path.join(BASE_PATH, filename)
    content = open(filename).read()

    # ...
```

Thought it looks like that view restricts file access to files beneath `BASE_PATH` (by using `os.path.join`), if the attacker passes in a `filename` containing `..` (that's two periods, the UNIX shorthand for "the parent directory"), he can access files "above" `BASE_PATH`. It's only a matter of time before he can discover the correct number of dots to successfully access, say, `../../../../../etc/passwd`.

Anything that reads files without proper escaping is vulnerable to this problem. Views that *write* files are just as vulnerable, but the consequences are doubly dire.

Another permutation of this problem lies in code that dynamically loads modules based on the URL or other request information. A well-publicized example came from the world of Ruby on Rails. Prior to mid-2006, Rails used URLs like `http://example.com/person/poke/1` directly to load modules and call methods. The result was that a carefully-constructed URL could automatically load arbitrary code, including a database reset script!

The solution

If your code ever needs to read or write files based on user input, you need to very carefully sanitize the requested path to ensure that an attacker isn't able to escape from the base directory you're restricting access to.

Note

Needless to say, you should **never** write code that can read from any area of the disk!

A good example of how to do this escaping lies in the Django's built-in static content serving view (in `django.views.static`). Here's the relevant code:

```
import os
import posixpath

# ...

path = posixpath.normpath(urllib.unquote(path))
newpath = ''
for part in path.split('/'):
    if not part:
        # strip empty path components
        continue

    drive, part = os.path.splitdrive(part)
    head, part = os.path.split(part)
    if part in (os.curdir, os.pardir):
        # strip '.' amd '..' in path
        continue

    newpath = os.path.join(newpath, part).replace('\\', '/')
```

Django itself doesn't read files (unless you use the `static.serve` function, but that's protected with the code shown above), so this vulnerability doesn't affect the core code much.

In addition, the use of the URLconf abstraction means that Django will *never* load code you've not explicitly told it to load. There's no way to create a URL that causes Django to load something not mentioned in a URLconf.

Exposed error messages

During development, being able to see tracebacks and errors live in your browser is extremely useful. Django has "pretty" and informative debug messages specifically to make debugging easier.

However, if these errors get displayed once the site goes live, they can sometimes unintentionally reveal aspects of your code or configuration that could aid an attacker.

Furthermore, errors and tracebacks aren't at all useful to end users. Django's philosophy is that site visitors should never see application-related error messages. If your code raises an unhandled exception, a site visitor should not see the full traceback — or *any* hint of code snippets or Python (programmer-oriented) error messages. Instead, the visitor should see a friendly "This page is unavailable" message.

Naturally, of course, developers need to see tracebacks to debug problems in their code. So the framework should hide all error messages from the public, but it should display them to the trusted site developers.

The solution

Django has a simple flag that controls the display of these error messages. If the `DEBUG` setting is set to `True`, error messages will be displayed in the browser. If not, Django will render return a HTTP 500 ("internal server error") message and render an error template that you provide. This error template is called `500.html`, and should live in the root of one of your template directories.

Since developers still need to see errors generated on a live site, any errors handled this way will send an email with the full traceback to any addresses given in the `ADMINS` setting.

Users deploying under Apache and mod_python should also make sure they have `PythonDebug Off` in their Apache conf files; this will ensure that any errors that occur before Django's had a chance to load won't be displayed publicly.

A final word

Hopefully all this talk of security problems isn't too intimidating. It's true that the web can be a wild and wooly world, but with a little bit of foresight you can have an incredibly secure website.

Keep in mind that web security is a constantly changing field; if you're reading the dead-tree version of this book, be sure to check more up-to-date security resources for any new vulnerabilities that have been discovered. In fact, it's always a good idea to spend some time each month or week researching and keeping current on the state of web application security. It's a small investment to make, but the protection you'll get for your site and your users is priceless.

Did we miss anything? Are there other security vulnerabilities you think we should cover in this chapter? Did we get something wrong (`$DEITY` forbid)? Leave a note on this paragraph and let us know!

« previous ◊ table of contents ◊ next »