MODULE-4

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 developed to ease that pain. They take certain common idioms and patterns found in view development and abstract them so that you can quickly write common views of data without having to write too much code. In fact, nearly every view example in the preceding chapters could be rewritten with the help of generic views.

Chapter 8 touched briefly on how you'd go about making a view "generic." To review, we can recognize certain common tasks, like displaying a list of objects, and write code that displays a list of *any* object. Then the model in question can be passed as an extra argument to the URLconf.

Django ships with 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. The `event_list` and `entry_list` views from Chapter 8 are examples of list views. A single event page is an example of what we call a "detail" view.
- 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 would be a typical newspaper's archives.
- 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 a simple URLconf you could use to present a static "about" page:

```
from django.conf.urls.defaults import *
from django.views.generic.simple import direct_to_template

urlpatterns = patterns('',
    ('^about/$', direct_to_template, {
        'template': 'about.html'
    })
)
```

Though this might seem a bit "magical" at first glance—look, a view with no code!— it's actually exactly the same as the examples in Chapter 8. The `direct_to_template` view simply grabs information from the extra-parameters dictionary and uses that information when rendering the view.

Because this generic view—and all the others—is a regular view function like any other, we can reuse it inside our own views. As an example, let's extend our "about" example to map URLs of the form `/about/<whatever>/` to statically rendered `about/<whatever>.html`. We'll do this by first modifying the URLconf to point to a view function:

```
from django.conf.urls.defaults import *
from django.views.generic.simple import direct_to_template
from mysite.books.views import about_pages

urlpatterns = patterns('',
    ('^about/$', direct_to_template, {
        'template': 'about.html'
    }),
    ('^about/(w+)/$', about_pages),
)
```

Next, we'll write the `about_pages` view:

```
from django.http import Http404
from django.template import TemplateDoesNotExist
from django.views.generic.simple import direct_to_template

def about_pages(request, page):
    try:
        return direct_to_template(request, template="about/%s.html" % page)
    except TemplateDoesNotExist:
        raise Http404()
```

Here we're treating `direct_to_template` like any other function. Since it returns an `HttpResponse`, we can simply return it as is. The only slightly tricky business here is dealing with missing templates. We don't want a nonexistent template to cause a server error, so we catch `TemplateDoesNotExist` exceptions and return 404 errors instead.

IS THERE A SECURITY VULNERABILITY HERE?

Sharp-eyed readers may have noticed a possible security hole: we're constructing the template name using interpolated content from the browser (`template="about/%s.html" % page`). At first glance, this looks like a classic *directory traversal* vulnerability (discussed in detail in Chapter 19). But is it really?

Not exactly. Yes, a maliciously crafted value of `page` could cause directory traversal, but although `page` *is* taken from the request URL, not every value will be accepted. They key is in the URLconf: we're using the regular expression `\w+` to match the `page` part of the URL, and `\w` accepts only letters and numbers. Thus, any malicious characters (dots and slashes, here) will be rejected by the URL resolver before they reach the view itself.

Generic Views of Objects

The `direct_to_template` view certainly is useful, but Django's generic views really shine when it comes to presenting views on your database content. Because it's such a common task, Django comes with a handful of built-in generic views that make generating list and detail views of objects incredibly easy.

Let's take a look at one of these generic views: the "object list" view. We'll be using this `Publisher` object from Chapter 5:

```
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 Meta:
        ordering = ["-name"]
```

To build a list page of all books, we'd use a URLconf along these lines:

```
from django.conf.urls.defaults import *
from django.views.generic import list_detail
from mysite.books.models import Publisher

publisher_info = {
    "queryset" : Publisher.objects.all(),
}

urlpatterns = patterns('',
    (r'^publishers/$', list_detail.object_list, publisher_info)
)
```

That's all the Python code we need to write. We still need to write a template, however. We could explicitly tell the `object_list` view which template to use by including a `template_name` key in the extra arguments dictionary, but in the absence of an explicit template Django will infer one from the object's name. In this case, the inferred template will be `"books/publisher_list.html"`—the "books" part comes from the name of the app that defines the model, while the "publisher" bit is just the lowercased version of the model's name.

This template will be rendered against a context containing a variable called `object_list` that contains all the book objects. A very simple template might look like the following:

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

{% block content %}
    <h2>Publishers</h2>
    <ul>
        {% for publisher in object_list %}
            <li>{{ publisher.name }}</li>
        {% endfor %}
    </ul>
{% endblock %}
```

That's really all there is to it. All the cool features of generic views come from changing the "info" dictionary passed to the generic view. Appendix D documents all the generic views and their options in detail. In the rest of this chapter, we'll consider some of the common ways you might customize and extend generic views.

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 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 handful of patterns dealt with in the sections that follow.

Making "Friendly" Template Contexts

You might have noticed that the sample publisher list template stores all the books in a variable named `object_list`. While this works just fine, it isn't all that "friendly" to template authors: they have to "just know" that they're dealing with books here. A better name for that variable would be `publisher_list`; that variable's content is pretty obvious.

We can change the name of that variable easily with the `template_object_name` argument:

```
publisher_info = {
    "queryset" : Publisher.objects.all(),
    "template_object_name" : "publisher",
}
```

```
urlpatterns = patterns('',
    (r'^publishers/$', list_detail.object_list, publisher_info)
)
```

Providing a useful `template_object_name` is always a good idea. Your co-workers who design templates will thank you.

Adding Extra Context

Often you simply need to present some extra information beyond that provided by the generic view. For example, think of showing a list of all the other publishers on each publisher detail page. The `object_detail` generic view provides the publisher to the context, but it seems there's no way to get a list of *all* publishers in that template.

But there is: all generic views take an extra optional parameter, `extra_context`. This is a dictionary of extra objects that will be added to the template's context. So, to provide the list of all publishers on the detail view, we'd use an info dict like this:

```
publisher_info = {
    "queryset" : Publisher.objects.all(),
    "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.

However, there's actually a subtle bug here—can you spot it? The problem has to do with when the queries in `extra_context` are evaluated. Because this example puts `Publisher.objects.all()` in the URLconf, it will be evaluated only once (when the URLconf is first loaded). Once you add or remove publishers, you'll notice that the generic view doesn't reflect those changes until you reload the Web server (see "Caching and QuerySets" in Appendix C for more information about when `QuerySet`s are cached and evaluated).

■ **Note** This problem doesn't apply to the `queryset` generic view argument. Since Django knows that particular `QuerySet` should *never* be cached, the generic view takes care of clearing the cache when each view is rendered.

The solution is to use a callback in `extra_context` instead of a value. Any callable (i.e., a function) that's passed to `extra_context` will be evaluated when the view is rendered (instead of only once). You could do this with an explicitly defined function:

```
def get_publishers():
    return Publisher.objects.all()

book_info = {
    "queryset" : Publisher.objects.all(),
    "extra_context" : {"publisher_list" : get_publishers}
}
```

or you could use a less obvious but shorter version that relies on the fact that `Publisher.objects.all` is itself a callable:

```
book_info = {
    "queryset" : Book.objects.all(),
    "extra_context" : {"publisher_list" : Publisher.objects.all}
}
```

Notice the lack of parentheses after `Publisher.objects.all`; this references the function without actually calling it (which the generic view will do later).

Viewing Subsets of Objects

Now let's take a closer look at this `queryset` key we've been using all along. Most generic views take one of these `queryset` arguments—it's how the view knows which set of objects to display (see "Selecting Objects" in Chapter 5 for an introduction to `QuerySets`, and see Appendix C for the complete details).

To pick a basic example, we might want to order a list of books by publication date, with the most recent first:

```
book_info = {
    "queryset" : Book.objects.all().order_by("-publication_date"),
}
```

That's a pretty simple example, but it illustrates the idea nicely. Of course, you'll usually want to do more than just reorder objects. If you want to present a list of books by a particular publisher, you can use the same technique:

```
apress_books = {
    "queryset": Book.objects.filter(publisher__name="Apress Publishing"),
    "template_name" : "books/apress_list.html"
}

urlpatterns = patterns('',
    (r'^publishers/$', list_detail.object_list, publisher_info),
    (r'^books/apress/$', list_detail.object_list, apress_books),
)
```

Notice that along with a filtered `queryset`, we're also using a custom template name. If we didn't, the generic view would use the same template as the "vanilla" object list, which might not be what we want.

Also notice that this isn't a very elegant way of doing publisher-specific books. If we want to add another publisher page, we need another handful of lines in the URLconf, and more than a few publishers would get unreasonable. We'll deal with this problem in the next section.

■ **Note** If you get a 404 when requesting `/books/apress/`, check to ensure that you actually have a publisher with the name "Apress Publishing." If you don't have that publisher, you'll get a 404. Generic views have an `allow_empty` parameter that changes this behavior; see Appendix D for details.

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. Earlier we hard-coded the publisher's name in the URLconf, but what if we wanted to write a view that displayed all the books by some arbitrary publisher? We can "wrap" the `object_list` generic view to avoid writing a lot of code by hand. As usual, we'll start by writing a URLconf:

```
urlpatterns = patterns('',
    (r'publishers/$', list_detail.object_list, publisher_info),
    (r'books/(w+)/$', books_by_publisher),
)
```

Next, we'll write the `books_by_publisher` view itself:

```
from django.http import Http404
from django.views.generic import list_detail
from mysite.books.models import Book, Publisher

def books_by_publisher(request, name):

    # Look up the publisher (and raise a 404 if it can't be found).
    try:
        publisher = Publisher.objects.get(name__iexact=name)
    except Publisher.DoesNotExist:
        raise Http404

    # Use the object_list view for the heavy lifting.
    return list_detail.object_list(
        request,
        queryset = Book.objects.filter(publisher=publisher),
        template_name = "books/books_by_publisher.html",
        template_object_name = "books",
        extra_context = {"publisher" : publisher}
    )
```

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 the next section) handing things off to the generic view.

Notice that in the preceding example we passed the current publisher being displayed in the `extra_context`. This is usually a good idea in wrappers of this nature; it lets the template know which "parent" object is currently being browsed.

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 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 add an author detail bit in the URLconf to point to a custom view:

```
from mysite.books.views import author_detail

urlpatterns = patterns('',
    #...
    (r'^authors/(?P<author_id>d+)/$', author_detail),
)
```

Then we'd write our wrapper function:

```
import datetime
from mysite.books.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,
    )
```

This code won't actually work unless you add a `last_accessed` field to your `Author` model and create a `books/author_detail.html` template.

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(
        request,
        queryset = Author.objects.all(),
        mimetype = "text/plain",
        template_name = "books/author_list.txt"
    )
    response["Content-Disposition"] = "attachment; filename=authors.txt"
    return response
```

This works because the generic views return simple `HttpResponse` objects that 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.

CHAPTER 11

Generating Non-HTML Content

Usually when we talk about developing Web sites, we're talking about producing HTML. Of course, there's a lot more to the Web than HTML; we use the Web to distribute data in all sorts of formats: RSS, PDFs, images, and so forth.

So far we've focused 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 (an XML format originally developed by Google that gives hints to search engines)

We'll examine each of those tools a little later on, but first we'll cover the basic principles.

The Basics: Views and MIME Types

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
- Return an `HttpResponse` instance

The key to returning non-HTML content from a view lies in the `HttpResponse` class, specifically the `mimetype` constructor argument. By tweaking the MIME type, we can indicate to the browser that we've returned a response of a different format.

For 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 use an `HttpResponse` instance in 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 a comma (CSV stands for comma-separated values). For example, here's some data on "unruly" airline passengers in CSV format:

```
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
```

The preceding listing contains real numbers; they come courtesy of the US Federal Aviation Administration. See `http://www.faa.gov/data_statistics/passengers_cargo/unruly_passengers/`.

Though CSV looks simple, 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.

Because the `csv` module operates on filelike objects, it's a snap to use an `HttpResponse` instead:

```
import csv
from django.http import HttpResponse
```

```
# Number of unruly passengers each year 1995 - 2005. In a real application
# this would likely come from a database or some other back-end data store.
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 special mention:

- The response is given the `text/csv` MIME type (instead of the default `text/html`). This tells browsers that the document is a CSV 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 file name is arbitrary; call it whatever you want. It will be used by browsers in the Save As dialog.
- Hooking into the CSV-generation API is easy: just pass `response` as the first argument to `csv.writer`. The `csv.writer` function expects a filelike 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 will do the right thing.

This is the general pattern you'll use any time you need to return non-HTML content: create an `HttpResponse` response object (with a special MIME type), pass it to something expecting a file, and then return the response.

Let's look at a few more examples.

Generating PDFs

Portable Document Format (PDF) 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 someone needs to give a document to someone else to print.

You can easily generate PDFs with Python and Django thanks to the 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.

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 (naturally available only as a PDF file) at `http://www.reportlab.org/rsrc/userguide.pdf` has additional installation instructions.

If you're using a modern Linux distribution, you might want to check your package management utility before installing ReportLab. Most package repositories have added ReportLab. For example, if you're using the (excellent) Ubuntu distribution, a simple `apt-get 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

Like CSV, generating PDFs dynamically with Django is easy because the ReportLab API acts on filelike 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
```

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 off this information, browsers will probably interpret the response 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 filelike 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 you'll end up with a corrupted PDF file).

Complex PDFs

If you're creating a complex PDF document (or any large data blob), consider using the `cStringIO` library as a temporary holding place for your PDF file. The `cStringIO` library provides a filelike object interface that is written in C for maximum efficiency.

Here's the previous "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'

    pdfbuffer = StringIO()

    # Create the PDF object, using the StringIO object as its "file."
    p = canvas.Canvas(pdfbuffer)

    # 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(pdfbuffer.getvalue())
    return response
```

Other Possibilities

There's a whole host 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:

- *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 images*: The Python Imaging Library (PIL; `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/`) 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/`), can be 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.

■ **Note** 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 get information on 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.

The high-level feed-generating framework is a view that's hooked to `/feeds/` by convention. Django uses the remainder of the URL (everything after `/feeds/`) to determine which feed to return.

To create a feed, you'll write a `Feed` class and point to it in your URLconf (see Chapters 3 and 8 for more about URLconfs).

Initialization

To activate syndication feeds on your Django site, add this URLconf:

```
(r'^feeds/(?P<url>.*)/$',
 'django.contrib.syndication.views.feed',
 {'feed_dict': feeds}
),
```

This line 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 (a 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 preceding 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'll need to define the `Feed` classes themselves.

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 Feed

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 are as follows:

- 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 previous `LatestEntries` example, 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 More Complex Feed

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 would be silly to create a separate `Feed` class for each police beat; that would violate the Don't Repeat Yourself (DRY) principle and would couple data to programming logic. Instead, the syndication framework lets you make generic feeds that return items based on information in the feed's URL.

On chicagocrime.org, the police-beat feeds are accessible via URLs like this:

- `http://www.chicagocrime.org/rss/beats/0613/`: Returns recent crimes for beat 0613
- `http://www.chicagocrime.org/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, 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's `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:

   a. It tries to call a method, passing the `obj` argument, where `obj` is the object returned by `get_object()`.

   b. Failing that, it tries to call a method with no arguments.

   c. 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()`, and then finally an `items` class attribute (which should be a list).

Full documentation of all the methods and attributes of the `Feed` classes is always available from the official Django documentation (`http://www.djangoproject.com/documentation/syndication/`).

Specifying the Type of Feed

By default, the syndication framework produces 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 shown in Table 11-1.

Table 11-1. *Feed Types*

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 (i.e., media resources associated with feed items such as MP3 podcast feeds), use the `item_enclosure_url`, `item_enclosure_length`, and `item_enclosure_mime_type` hooks:

```
from myproject.models import Song

class MyFeedWithEnclosures(Feed):
    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, that 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 here's 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

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 example, here's a piece of the sitemap for Django's Web site (`http://www.djangoproject.com/sitemap.xml`):

```
<?xml version="1.0" encoding="UTF-8"?>
<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">
  <url>
    <loc>http://www.djangoproject.com/documentation/</loc>
    <changefreq>weekly</changefreq>
    <priority>0.5</priority>
  </url>
  <url>
    <loc>http://www.djangoproject.com/documentation/0_90/</loc>
    <changefreq>never</changefreq>
    <priority>0.1</priority>
  </url>
  ...
</urlset>
```

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 application, 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 need to change this only if you've changed that setting.
3. Make sure you've installed the sites framework (see Chapter 14).

The sitemap application doesn't install any database tables. The only reason it needs to go into `INSTALLED_APPS` is so 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 line 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 (as described shortly).

`Sitemap` classes must subclass `django.contrib.sitemaps.Sitemap`. They can live anywhere in your code tree.

For example, 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
```

Declaring a `Sitemap` should look very similar to declaring a `Feed`; that's by design. Like `Feed` classes, `Sitemap` members can be either methods or attributes. See the steps in the earlier "A Complex Example" section 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 the absolute URL for a given object. Here, "absolute URL" means a URL that doesn't include the protocol or domain. Here are some 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 specification) are as follows:
  - `'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 `http://sitemaps.org` documentation for more about how `priority` works.

Shortcuts

The sitemap framework provides a couple convenience classes for common cases. These are described in the sections that follow.

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 14 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 earlier):

```
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 as follows:

- 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 previous example:

```
(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 of this writing, only Google responds to sitemap pings. However, it's quite likely that Yahoo and/or MSN 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 lookup on 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()`.

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.

In this chapter you'll learn how to handle this lack of state. We'll start at the lowest level (cookies), and work up to the high-level tools for handling sessions, users, and registration.

Cookies

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.

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 the following:

```
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 in this chapter. However, we'll pause and look at how to read and write cookies at a low level. This should help you understand how the rest of the tools discussed in the chapter actually work, and it will 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 an `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"])

        return response

    else:
        return HttpResponse("You didn't give a favorite color.")
```

You can also pass a number of optional arguments to `response.set_cookie()` that control aspects of the cookie, as shown in Table 12-1.

Table 12-1. *Cookie Options*

Parameter	Default	Description
`max_age`	`None`	Age (in seconds) that the cookie should last. If this parameter is `None`, the cookie will last only until the browser is closed.
`expires`	`None`	The date/time when the cookie should expire. It needs to be in the format `"Wdy, DD-Mth-YY HH:MM:SS GMT"`. If given, this parameter 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 parameter 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 parameter 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 this parameter is set to `None`, a cookie will only be readable by the domain that set it.
`secure`	`False`	If set to `True`, this parameter instructs the browser to return this cookie only 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:

- Storage of cookies is essentially voluntary; browsers don't guarantee anything. In fact, all browsers enable users to control the 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.

  Despite their nearly universal use, cookies are still the definition of unreliability. This means that developers should check that a user actually accepts cookies before relying on them.

More important, you should *never* store important data in cookies. The Web is filled with horror stories of developers who have stored unrecoverable information in browser cookies, only to have that data purged by the browser for one reason or another.

- Cookies (especially those not sent over HTTPS) are not 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 19 discusses attacks of this nature in depth, as well as ways to prevent them.

- 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 (`http://wwwsearch.sourceforge.net/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 examples of those "pain points" in Web development. Of course, Django's goal is to be an effective painkiller, so it 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.

Let's look at how to enable sessions and use them in views.

Enabling Sessions

Sessions are implemented via a piece of middleware (see Chapter 15) and a Django model. To enable sessions, you'll need to follow these steps:

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` have 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`. Doing so will save you only a 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 uses only a small number of underscore-prefixed session variables, but unless you know what they all are (and you are willing to keep up with any changes in Django itself), staying away from underscore prefixes will keep Django from interfering with your application.
- Don't replace `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's a simple (but not particularly secure) way of preventing a user from posting more than one comment:

```
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):
try:

    m = Member.objects.get(username__exact=request.POST['username'])
    if m.password == request.POST['password']:
        request.session['member_id'] = m.id
        return HttpResponse("You're logged in.")
except Member.DoesNotExist:
    pass
return HttpResponse("Your username and password didn't match.")
```

And this one logs out a member, according to `login()`:

```
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 shortly handles this task for you in a much more robust and useful manner. These examples are deliberately simplistic so that you can easily see what's going on.

Setting Test Cookies

As mentioned earlier, you can't rely on every browser accepting cookies. So, as a convenience, Django provides an easy way to test whether a 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 friendlier 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 authentication 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`. Each session is identified by a more-or-less random 32-character hash stored in a cookie. 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(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 database if 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 `SESSION_SAVE_EVERY_REQUEST` 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 sent only 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 that the cookie Google sent us contained `expires=Sun, 17-Jan-2038 19:14:07 GMT;`. Cookies can optionally contain an expiration date that 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 his or 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 two weeks, or 1,209,600 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, a few other settings influence how Django's session framework uses cookies, as shown in Table 12-2.

Table 12-2. *Settings That Influence Cookie Behavior*

Setting	Default	Description
`SESSION_COOKIE_DOMAIN`	`None`	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.
`SESSION_COOKIE_NAME`	`"sessionid"`	The name of the cookie to use for sessions. This can be any string.
`SESSION_COOKIE_SECURE`	`False`	Indication of 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.

TECHNICAL DETAILS

For the curious, here are a few technical notes about the inner workings of the session framework:

- The session dictionary accepts any Python object capable of being "pickled." See the documentation for Python's built-in `pickle` module for information about how this works.
- Session data is stored in a database table named `django_session`.
- Session data is fetched upon demand. If you never access `request.session`, Django won't hit that database table.
- Django sends a cookie only 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 (e.g., PHP, JSP) do. This is an intentional design decision. Putting sessions in URLs doesn't just make URLs ugly, but it also 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 more details.

Users and Authentication

We're now halfway to linking browsers directly to real people. Sessions give us a way of persisting data through multiple browser requests; the second part of the equation is using those sessions for user login. Of course, we can't just trust that users are who they say they are, so we 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* (authentication and authorization) system. That name recognizes that dealing with users is often a two-step process. We need to

1. Verify (*authenticate*) that a user is who he or she claims to be (usually by checking a username and password against a database of users).
2. Verify that the user is *authorized* to perform some given operation (usually by checking against a table of permissions).

Following these needs, Django's auth/auth system consists of a number of parts:

- *Users*: People registered with your site
- *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 (detailed in Chapter 6), you've already seen many of these tools, and if you've edited users or groups in the admin tool, you've actually been editing data in the auth system's database tables.

Enabling Authentication Support

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 as described earlier in this chapter. 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 the following section 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 have a `User`—often from `request.user`, but possibly through one of the other methods discussed shortly—you have a number of fields and methods available on that object. `AnonymousUser` objects emulate *some* of this interface, but not all of it, so you should always check `user.is_authenticated()` before assuming you're dealing with a bona fide user object. Tables 12-3 and 12-4 list the fields and methods, respectively, on `User` objects.

Table 12-3. *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. Email address.
`password`	Required. A hash of, and metadata about, the password (Django doesn't store the raw password). See the "Passwords" section for more about this value.
`is_staff`	Boolean. Designates whether this user can access the admin site.

Field	Description
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. This is set to the current date/time by default.
date_joined	A datetime designating when the account was created. This is set to the current date/time by default when the account is created.

Table 12-4. *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 it doesn't check if the user is active. It only indicates that the user has successfully 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 he or 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(app_label)	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 email 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 "Profiles" section 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 user's 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
>>> user = auth.authenticate(username='john', password='secret')
>>> if user is not None:
...     print "Correct!"
... else:
...     print "Oops, that's wrong!"
```

authenticate() only verifies a user's credentials. To log in a user, 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, 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.

In practice, you usually will not need to write your own login/logout functions; the authentication 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 them 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.

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 this:

```
{% 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|escape }}" />
  <form action='.' method='post'>

{% endblock %}
```

If the user successfully logs in, he or she will 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 will be automatically added to the context as a variable called `next` that you can insert into that hidden field.

The logout 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 logout.

Limiting Access to Logged-in Users

Of course, the reason we're going through all this trouble is so 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.

Limiting Access to Users Who Pass a Test

Limiting access based on certain permissions or some other test, or providing a different location for the login 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` (more about how permissions works follows):

```
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 called `user_passes_test`. It takes arguments and generates a specialized decorator for your particular situation:

```
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 follows:

```
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 interface. Chapter 6 discusses how to use Django's admin interface 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, and we discuss these in the sections that follow.

Creating Users

Create users with 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 (`create_user()` doesn't actually call `save()` itself). 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 A "SALTED HASH" SOME KIND OF DRUG?

No, a *salted hash* has nothing to do with marijuana; it's actually a common way to securely store passwords. A *hash* is a one-way cryptographic 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 takes some time, but less than you might think—computers are incredibly fast.

Worse, there are publicly available *rainbow tables*, or 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 to breaking passwords. Since salts differ from password to password, they 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 convenience.

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(data)
            return HttpResponseRedirect("/books/")
    else:
        data, errors = {}, {}

    return render_to_response("registration/register.html", {
        'form' : forms.FormWrapper(form, data, errors)
    })
```

This form 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 or 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 the default. Again, see Chapter 10 for more information.

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 a template-friendly proxy to a couple of permission methods described shortly.

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 for some given application, 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 are a few other bits of the authentication framework that we've only dealt with in passing. We'll take a closer look at them in the following sections.

Permissions

Permissions are a simple way to "mark" users and groups as being able to perform some action. They are usually used by the Django admin site, but you can easily use them 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, change, 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 have a `polls` application 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 applications.

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.

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.

Groups are also 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 those users access to a members-only portion of your site, or send them members-only email messages.

Like users, the easiest way to manage groups is through the admin interface. 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 timestamps.

Messages are used by the Django admin interface 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 application. The API is simple:

- To create a new message, use `user.message_set.create(message='message_text')`.
- To retrieve/delete messages, use `user.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 or 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 it 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 application 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 could raise a `SiteProfileNotAvailable` exception if `AUTH_PROFILE_MODULE` isn't defined, or it could 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).