



Introduction to Django

This book is about Django, a Web-development framework that saves you time and makes Web development a joy. Using Django, you can build and maintain high-quality Web applications with minimal fuss.

At its best, Web development is an exciting, creative act; at its worst, it can be a repetitive, frustrating nuisance. Django lets you focus on the fun stuff—the crux of your Web application—while easing the pain of the repetitive bits. In doing so, it provides high-level abstractions of common Web development patterns, shortcuts for frequent programming tasks, and clear conventions for how to solve problems. At the same time, Django tries to stay out of your way, letting you work outside the scope of the framework as needed.

The goal of this book is to make you a Django expert. The focus is twofold. First, we explain, in depth, what Django does and how to build Web applications with it. Second, we discuss higher-level concepts where appropriate, answering the question “How can I apply these tools effectively in my own projects?” By reading this book, you’ll learn the skills needed to develop powerful Web sites quickly, with code that is clean and easy to maintain.

What Is a Web Framework?

Django is a prominent member of a new generation of *Web frameworks*—but what does that term mean, precisely?

To answer that question, let’s consider the design of a Web application written in Python *without* a framework. Throughout this book, we’ll take this approach of showing you basic ways to get work done *without* shortcuts, in the hope that you’ll recognize why shortcuts are so helpful. (It’s also valuable to know how to get things done without shortcuts because shortcuts aren’t always available. And most importantly, knowing *why* things work the way they do makes you a better Web developer.)

One of the simplest, most direct ways to build a Python Web app from scratch is to use the Common Gateway Interface (CGI) standard, which was a popular technique circa 1998. Here’s a high-level explanation of how it works: create a Python script that outputs HTML, then save the script to a Web server with a `.cgi` extension and visit the page in your Web browser. That’s it.

Here’s a sample Python CGI script that displays the ten most recently published books from a database. Don’t worry about syntax details; just get a feel for the basic things it’s doing.

```
#!/usr/bin/env python

import MySQLdb

print "Content-Type: text/html\n"
print "<html><head><title>Books</title></head>"
print "<body>"
print "<h1>Books</h1>"
print "<ul>"

connection = MySQLdb.connect(user='me', passwd='letmein', db='my_db')
cursor = connection.cursor()
cursor.execute("SELECT name FROM books ORDER BY pub_date DESC LIMIT 10")

for row in cursor.fetchall():
    print "<li>%s</li>" % row[0]

print "</ul>"
print "</body></html>"

connection.close()
```

First, to fulfill the requirements of CGI, this code prints a “Content-Type” line, followed by a blank line. It prints some introductory HTML, connects to a database, and runs a query to retrieve the names of the latest ten books. Looping over those books, it generates an HTML list of the titles. Finally, it prints the closing HTML and closes the database connection.

With a one-off page like this one, the write-it-from-scratch approach isn’t necessarily bad. For one thing, this code is simple to comprehend—even a novice developer can read these 16 lines of Python and understand everything it does, from start to finish. There’s nothing else to learn, no other code to read. It’s also simple to deploy: just save this code in a file that ends with `.cgi`, upload that file to a Web server, and visit that page with a browser.

Despite its simplicity, this approach has a number of problems and annoyances. Ask yourself these questions:

- What happens when multiple parts of your application need to connect to the database? Surely that database-connecting code shouldn’t need to be duplicated in each individual CGI script. The pragmatic thing to do would be to refactor it into a shared function.
- Should a developer *really* have to worry about printing the “Content-Type” line and remembering to close the database connection? This sort of boilerplate reduces programmer productivity and introduces opportunities for mistakes. These setup- and teardown-related tasks would best be handled by some common infrastructure.

- What happens when this code is reused in multiple environments, each with a separate database and password? At this point, some environment-specific configuration becomes essential.
- What happens when a Web designer who has no experience coding Python wishes to redesign the page? One wrong character could crash the entire application. Ideally, the logic of the page—the retrieval of book titles from the database—would be separate from the HTML display of the page so that a designer could edit the latter without affecting the former.

These problems are precisely what a Web framework intends to solve. A Web framework provides a programming infrastructure for your applications so that you can focus on writing clean, maintainable code without having to reinvent the wheel. In a nutshell, that's what Django does.

The MVC Design Pattern

Let's dive in with a quick example that demonstrates the difference between the previous approach and a Web framework's approach. Here's how you might write the previous CGI code using Django. The first thing to note is that we split it over three Python files (`models.py`, `views.py`, `urls.py`) and an HTML template (`latest_books.html`):

```
# models.py (the database tables)
```

```
from django.db import models
```

```
class Book(models.Model):
    name = models.CharField(max_length=50)
    pub_date = models.DateField()
```

```
# views.py (the business logic)
```

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

```
def latest_books(request):
    book_list = Book.objects.order_by('-pub_date')[:10]
    return render_to_response('latest_books.html', {'book_list': book_list})
```

```
# urls.py (the URL configuration)
```

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

```
urlpatterns = patterns('',
    (r'^latest/$', views.latest_books),
)
```

```
# latest_books.html (the template)

<html><head><title>Books</title></head>
<body>
<h1>Books</h1>
<ul>
{% for book in book_list %}
<li>{{ book.name }}</li>
{% endfor %}
</ul>
</body></html>
```

Again, don't worry about the particulars of syntax; just get a feel for the overall design. The main thing to note here is the *separation of concerns*:

- The `models.py` file contains a description of the database table, represented by a Python class. This class is called a *model*. Using it, you can create, retrieve, update, and delete records in your database using simple Python code rather than writing repetitive SQL statements.
- The `views.py` file contains the business logic for the page. The `latest_books()` function is called a *view*.
- The `urls.py` file specifies which view is called for a given URL pattern. In this case, the URL `/latest/` will be handled by the `latest_books()` function. In other words, if your domain is `example.com`, any visit to the URL `http://example.com/latest/` will call the `latest_books()` function.
- The `latest_books.html` file is an HTML template that describes the design of the page. It uses a template language with basic logic statements—for example, `{% for book in book_list %}`.

Taken together, these pieces loosely follow a pattern called Model-View-Controller (MVC). Simply put, MVC is way of developing software so that the code for defining and accessing data (the model) is separate from request-routing logic (the controller), which in turn is separate from the user interface (the view). (We'll discuss MVC in more depth in Chapter 5.)

A key advantage of such an approach is that components are *loosely coupled*. Each distinct piece of a Django-powered Web application has a single key purpose and can be changed independently without affecting the other pieces. For example, a developer can change the URL for a given part of the application without affecting the underlying implementation. A designer can change a page's HTML without having to touch the Python code that renders it. A database administrator can rename a database table and specify the change in a single place rather than having to search and replace through a dozen files.

In this book, each component of MVC gets its own chapter. Chapter 3 covers views, Chapter 4 covers templates, and Chapter 5 covers models.

Django's History

Before we dive into more code, we should take a moment to explain Django's history. We noted earlier that we'll be showing you how to do things *without* shortcuts so that you more fully understand the shortcuts. Similarly, it's useful to understand *why* Django was created, because knowledge of the history will put into context why Django works the way it does.

If you've been building Web applications for a while, you're probably familiar with the problems in the CGI example we presented earlier. The classic Web developer's path goes something like this:

1. Write a Web application from scratch.
2. Write another Web application from scratch.
3. Realize the application from step 1 shares much in common with the application from step 2.
4. Refactor the code so that application 1 shares code with application 2.
5. Repeat steps 2–4 several times.
6. Realize you've invented a framework.

This is precisely how Django itself was created!

Django grew organically from real-world applications written by a Web-development team in Lawrence, Kansas, USA. It was born in the fall of 2003, when the Web programmers at the *Lawrence Journal-World* newspaper, Adrian Holovaty and Simon Willison, began using Python to build applications.

The World Online team, responsible for the production and maintenance of several local news sites, thrived in a development environment dictated by journalism deadlines. For the sites—including LJWorld.com, Lawrence.com, and KUsports.com—journalists (and management) demanded that features be added and entire applications be built on an intensely fast schedule, often with only days' or hours' notice. Thus, Simon and Adrian developed a time-saving Web-development framework out of necessity—it was the only way they could build maintainable applications under the extreme deadlines.

In summer 2005, after having developed this framework to a point where it was efficiently powering most of World Online's sites, the team, which now included Jacob Kaplan-Moss, decided to release the framework as open source software. They released it in July 2005 and named it Django, after the jazz guitarist Django Reinhardt.

Now, several years later, Django is a well-established open source project with tens of thousands of users and contributors spread across the planet. Two of the original World Online developers (the "Benevolent Dictators for Life," Adrian and Jacob) still provide central guidance for the framework's growth, but it's much more of a collaborative team effort.

This history is relevant because it helps explain two key things. The first is Django's "sweet spot." Because Django was born in a news environment, it offers several features (such as its admin site, covered in Chapter 6) that are particularly well suited for "content" sites—sites like Amazon.com, Craigslist, and The Washington Post that offer dynamic, database-driven information. Don't let that turn you off, though—although Django is particularly good for developing those sorts of sites, that doesn't preclude it from being an effective tool for building any sort of dynamic Web site. (There's a difference between being *particularly effective* at something and being *ineffective* at other things.)

The second matter to note is how Django’s origins have shaped the culture of its open source community. Because Django was extracted from real-world code rather than being an academic exercise or a commercial product, it is acutely focused on solving Web-development problems that Django’s developers themselves have faced—and continue to face. As a result, Django itself is actively improved on an almost daily basis. The framework’s maintainers have a vested interest in making sure Django saves developers time, produces applications that are easy to maintain, and performs well under load. If nothing else, the developers are motivated by their own selfish desires to save themselves time and enjoy their jobs. (To put it bluntly, they eat their own dog food.)

How to Read This Book

In writing this book, we tried to strike a balance between readability and reference, with a bias toward readability. Our goal with this book, as stated earlier, is to make you a Django expert, and we believe the best way to teach is through prose and plenty of examples, rather than providing an exhaustive but bland catalog of Django features. (As the saying goes, you can’t expect to teach somebody how to speak a language merely by teaching them the alphabet.)

With that in mind, we recommend that you read Chapters 1 through 12 in order. They form the foundation of how to use Django; once you’ve read them, you’ll be able to build and deploy Django-powered Web sites. Specifically, Chapters 1 through 7 are the “core curriculum,” Chapters 8 through 11 cover more-advanced Django usage, and Chapter 12 covers deployment. The remaining chapters, 13 through 20, focus on specific Django features and can be read in any order.

The appendixes are for reference. They, along with the free documentation at <http://www.djangoproject.com/>, are probably what you’ll flip back to occasionally to recall syntax or find quick synopses of what certain parts of Django do.

Required Programming Knowledge

Readers of this book should understand the basics of procedural and object-oriented programming: control structures (e.g., `if`, `while`, `for`), data structures (lists, hashes/dictionaries), variables, classes, and objects.

Experience in Web development is, as you may expect, very helpful, but it’s not required to understand this book. Throughout the book, we try to promote best practices in Web development for readers who lack this experience.

Required Python Knowledge

At its core, Django is simply a collection of libraries written in the Python programming language. To develop a site using Django, you write Python code that uses these libraries. Learning Django, then, is a matter of learning how to program in Python and understanding how the Django libraries work.

If you have experience programming in Python, you should have no trouble diving in. By and large, the Django code doesn't perform a lot of "magic" (i.e., programming trickery whose implementation is difficult to explain or understand). For you, learning Django will be a matter of learning Django's conventions and APIs.

If you don't have experience programming in Python, you're in for a treat. It's easy to learn and a joy to use! Although this book doesn't include a full Python tutorial, it highlights Python features and functionality where appropriate, particularly when code doesn't immediately make sense. Still, we recommend you read the official Python tutorial, available online at <http://docs.python.org/tut/>. We also recommend Mark Pilgrim's free book *Dive Into Python* (Apress, 2004), available at <http://www.diveintopython.org/> and published in print by Apress.

Required Django Version

This book covers Django 1.1.

Django's developers maintain backward compatibility within "major version" numbers. This commitment means that, if you write an application for Django 1.1, it will still work for 1.2, 1.3, 1.9, and any other version number that starts with "1." Once Django hits 2.0, though, your applications might need to be rewritten—but version 2.0 is a long way away. As a point of reference, it took more than three years to release version 1.0. (This is very similar to the compatibility policy that applies to the Python language itself: code that was written for Python 2.0 works with Python 2.6, but not necessarily with Python 3.0.) Given that this book covers Django 1.1, it should serve you well for some time.

Getting Help

One of the greatest benefits of Django is its kind and helpful user community. For help with any aspect of Django—from installation to application design to database design to deployment—feel free to ask questions online.

- The Django users mailing list is where thousands of Django users hang out to ask and answer questions. Sign up for free at <http://www.djangoproject.com/r/django-users>.
- The Django IRC channel is where Django users hang out to chat and help each other in real time. Join the fun by logging on to #django on the Freenode IRC network.

What's Next?

In the next chapter, we'll get started with Django, covering installation and initial setup.



Views and URLconfs

In the previous chapter, we explained how to set up a Django project and run the Django development server. In this chapter, you'll learn the basics of creating dynamic Web pages with Django.

Your First Django-Powered Page: Hello World

As a first goal, let's create a Web page that outputs that famous example message: "Hello world."

If you were publishing a simple "Hello world" Web page without a Web framework, you'd simply type "Hello world" into a text file, call it `hello.html`, and upload it to a directory on a Web server somewhere. Notice that you specified two key pieces of information about that Web page: its contents (the string "Hello world") and its URL (`http://www.example.com/hello.html`, or maybe `http://www.example.com/files/hello.html` if you put it in a subdirectory).

With Django, you specify those same two things, but in a different way. The contents of the page are produced by a *view function*, and the URL is specified in a *URLconf*. First, let's write the "Hello world" view function.

Your First View

Within the `mysite` directory that `django-admin.py startproject` made in the last chapter, create an empty file called `views.py`. This Python module will contain the views for this chapter. Note that there's nothing special about the name `views.py`—Django doesn't care what the file is called, as you'll see in a bit—but it's a good idea to call it `views.py` as a convention for the benefit of other developers reading your code.

A "Hello world" view is simple. Here's the entire function, plus import statements, which you should type into the `views.py` file:

```
from django.http import HttpResponse

def hello(request):
    return HttpResponse("Hello world")
```


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

- First, you import the class `HttpResponse`, which lives in the `django.http` module. You need to import this class because it's used later in the code.
- Next, you define a function called `hello`—the view function.
- Each view function takes at least one parameter, called `request` by convention. This is an object that contains information about the current Web request that has triggered this view, and it's an instance of the class `django.http.HttpRequest`. In this example, you don't do anything with `request`, but it must be the first parameter of the view nonetheless.
- Note that the name of the view function doesn't matter; it doesn't have to be named in a certain way for Django to recognize it. We called it `hello` because that name clearly indicates the gist of the view, but it could just as well be named `hello_wonderful_beautiful_world`, or something equally revolting. The next section, “Your First URLconf,” will shed light on how Django finds this function.
- The function is a simple one-liner: it merely returns an `HttpResponse` object that has been instantiated with the text `"Hello world"`.

The main lesson is this: a view is just a Python function that takes an `HttpRequest` as its first parameter and returns an instance of `HttpResponse`. In order for a Python function to be a Django view, it must do these two things. (There are exceptions, but we'll get to them later.)

Your First URLconf

If at this point you run `python manage.py runserver` again, you'll still see the “Welcome to Django” message, with no trace of the “Hello world” view anywhere. That's because the `mysite` project doesn't yet know about the `hello` view; you need to tell Django explicitly that you're activating this view at a particular URL. (Continuing the previous analogy of publishing static HTML files, at this point you've created the HTML file but haven't uploaded it to a directory on the server yet.) To hook a view function to a particular URL with Django, use a `URLconf`.

A *URLconf* is like a table of contents for a Django-powered Web site. Basically, it's a mapping between URLs and the view functions that should be called for those URLs. It's how you tell Django, “For this URL, call this code, and for that URL, call that code.” For example, “When somebody visits the URL `/foo/`, call the view function `foo_view()`, which lives in the Python module `views.py`.”

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

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

# Uncomment the next two lines to enable the admin:
# from django.contrib import admin
# admin.autodiscover()
```

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

    # Uncomment the admin/doc line below and add 'django.contrib.admindocs'
    # to INSTALLED_APPS to enable admin documentation:
    # (r'^admin/doc/', include('django.contrib.admindocs.urls')),

    # Uncomment the next line to enable the admin:
    # (r'^admin/', include(admin.site.urls)),
)
```

This default URLconf includes some commonly used Django features commented out, so activating those features is as easy as uncommenting the appropriate lines. If you ignore the commented-out code, here's the essence of a URLconf:

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

urlpatterns = patterns('',
)
```

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

- The first line imports all objects from the `django.conf.urls.defaults` module, which is Django's URLconf infrastructure. This includes a function called `patterns`.
- The second line calls the function `patterns` and saves the result into a variable called `urlpatterns`. The `patterns` function gets passed only a single argument: the empty string. (The string can be used to supply a common prefix for view functions, which we'll cover in Chapter 8.)

The main thing to note is the variable `urlpatterns`, which Django expects to find in the URLconf module. This variable defines the mapping between URLs and the code that handles those URLs. By default, the URLconf is empty—the Django application is a blank slate.

Note That's how Django knew to show you the “Welcome to Django” page in the last chapter. If your URLconf is empty, Django assumes that you just started a new project, so it displays that message.

To add a URL and view to the URLconf, just add a Python tuple mapping a URL pattern to the view function. Here's how to hook in the hello view:

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

urlpatterns = patterns('',
    ('^hello/$', hello),
)
```

Note We removed the commented-out code for brevity. You can choose to leave those lines in if you want.

Two changes were made:

- First, the `hello` view was imported from its module: `mysite/views.py`, which translates into `mysite.views` in Python import syntax. (This assumes that `mysite/views.py` is on the Python path; see the sidebar for details.)
- Next, the line `('^hello/$', hello)` was added to `urlpatterns`. This line is referred to as a *URL pattern*. It's a Python tuple in which the first element is a pattern-matching string (a regular expression; more on this in a bit) and the second element is the view function to use for that pattern.

In a nutshell, Django was told that any request to the URL `/hello/` should be handled by the `hello` view function.

PYTHON PATH

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

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

If you're interested in seeing the value of your Python path, start the Python interactive interpreter and type this:

```
>>> import sys
>>> print sys.path
```

You usually don't have to worry about setting your Python path—Python and Django take care of things for you automatically behind the scenes. (Setting the Python path is one of the things that the `manage.py` script does.)

It's worth discussing the syntax of this URL pattern because it might not be immediately obvious. Although you want to match the URL `/hello/`, the pattern looks a bit different from that. Here's why:

- Django removes the slash from the front of every incoming URL before it checks the URLpatterns. This means that the URLpattern doesn't include the leading slash in `/hello/`. (At first, this requirement might seem counterintuitive, but it simplifies things—such as the inclusion of URLconfs within other URLconfs, which we'll cover in Chapter 8.)
- The pattern includes a caret (^) and a dollar sign (\$). These regular expression characters have a special meaning: the caret means “require that the pattern matches the start of the string,” and the dollar sign means “require that the pattern matches the end of the string.”
- This concept is best explained by an example. If you had used the pattern `'^hello/'` (without a dollar sign at the end), *any* URL starting with `/hello/` would match (for example, `/hello/foo` and `/hello/bar`, not just `/hello/`). Similarly, if you leave off the initial caret character (for example, `'hello/$'`), Django would match *any* URL that ends with `hello/`, such as `/foo/bar/hello/`. If you simply use `hello/` without a caret *or* a dollar sign, any URL containing `hello/` would match (for example, `/foo/hello/bar`). Thus, you use both the caret and dollar sign to ensure that only the URL `/hello/` matches—nothing more, nothing less.
- Most URLpatterns start with carets and end with dollar signs, but it's nice to have the flexibility to perform more sophisticated matches.
- You might be wondering what happens if someone requests the URL `/hello` (that is, *without* a trailing slash). Because the URLpattern requires a trailing slash, that URL would *not* match. However, by default, any request to a URL that *doesn't* match a URLpattern and *doesn't* end with a slash will be redirected to the same URL with a trailing slash. (This is regulated by the `APPEND_SLASH` Django setting, which is covered in Appendix D.)
- If you're the type of person who likes all URLs to end with slashes (which is the preference of Django's developers), all you need to do is add a trailing slash to each URLpattern and leave `APPEND_SLASH` set to `True`. If you prefer your URLs *not* to have trailing slashes, or if you want to decide it on a per-URL basis, set `APPEND_SLASH` to `False` and put trailing slashes in your URLpatterns as you see fit.

The other thing to note about this URLconf is that the `hello` view function was passed as an object without calling the function. This is a key feature of Python (and other dynamic languages): functions are first-class objects, which means that you can pass them around just like any other variables. Cool stuff, eh?

To test the changes to the URLconf, start the Django development server, as you did in Chapter 2, by running the command `python manage.py runserver`. (If you left it running, that's fine, too. The development server automatically detects changes to your Python code and reloads as necessary, so you don't have to restart the server between changes.) The server is running at the address `http://127.0.0.1:8000/`, so open up a Web browser and go to `http://127.0.0.1:8000/hello/`. You should see the text “Hello world”—the output of your Django view.

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

REGULAR EXPRESSIONS

You can use a *regular expression* (*regex*) as a compact way of specifying patterns in text. While Django URLconfs allow arbitrary regexes for powerful URL matching, you'll probably only use a few regex symbols in practice. Here's a selection of common symbols.

Symbol	Matches
.	Any single character
\d	Any single digit
[A-Z]	Any character between A and Z (uppercase)
[a-z]	Any character between a and z (lowercase)
[A-Za-z]	Any character between a and z (case-insensitive)
+	One or more of the previous expression (for example, \d+ matches one or more digits)
[^/]+	One or more characters until (and not including) a forward slash
?	Zero or one of the previous expression (for example, \d? matches zero or one digits)
*	Zero or more of the previous expression (for example, \d* matches zero, one or more than one digit)
{1,3}	Between one and three (inclusive) of the previous expression (for example, \d{1,3} matches one, two, or three digits)

For more on regular expressions, see <http://www.djangoproject.com/r/python/re-module/>.

A Quick Note About 404 Errors

At this point, the URLconf defines only a single URLpattern: the one that handles requests to the URL `/hello/`. What happens when you request a different URL?

To find out, try running the Django development server and visiting a page such as `http://127.0.0.1:8000/goodbye/`, `http://127.0.0.1:8000/hello/subdirectory/`, or even `http://127.0.0.1:8000/` (the site “root”). You should see a “Page not found” message (see Figure 3-1). Django displays this message because you requested a URL that’s not defined in your URLconf.

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

Naturally, this is sensitive information intended only for you, the Web developer. If this were a production site deployed live on the Internet, you wouldn’t want to expose that information to the public. For that reason, this “Page not found” page is displayed only if your Django project is in *debug mode*. We’ll explain how to deactivate debug mode later. For now, just know that every Django project is in debug mode when you first create it, and if the project is not in debug mode, Django outputs a different 404 response.

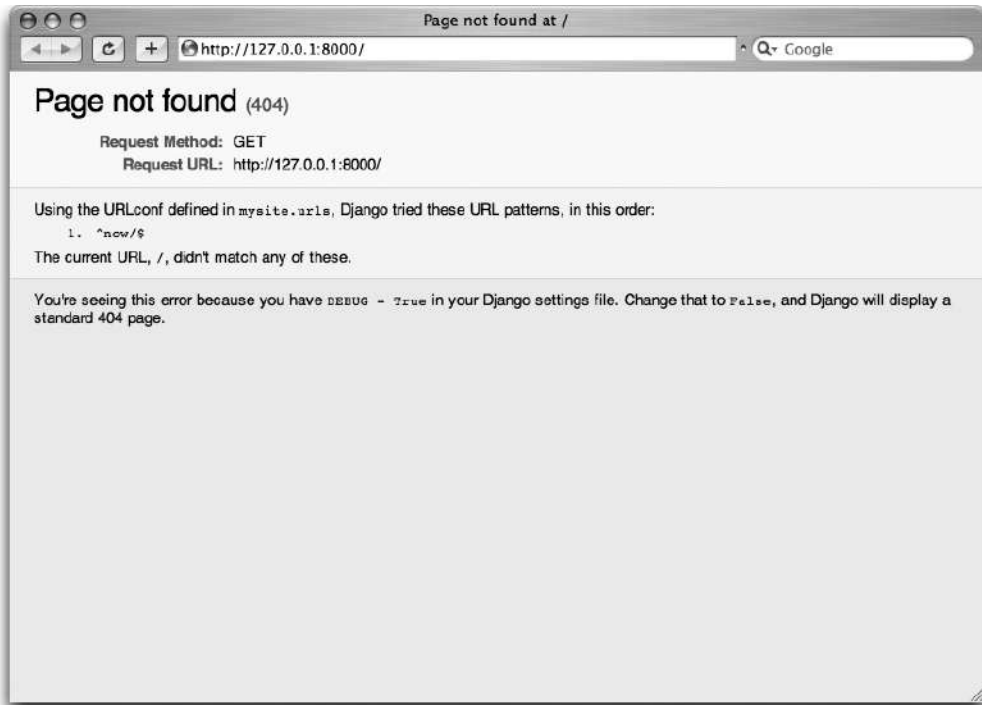


Figure 3-1. *Django's 404 page*

A Quick Note About the Site Root

As explained in the last section, you'll see a 404 error message if you view the site root: `http://127.0.0.1:8000/`. Django doesn't magically add anything to the site root; that URL is not special-cased in any way. It's up to you to assign it to a URLpattern, just like every other entry in your URLconf.

The URLpattern to match the site root is a bit counterintuitive, though, so it's worth mentioning. When you're ready to implement a view for the site root, use the URLpattern `'^$',` which matches an empty string. Here's an example:

```
from mysite.views import hello, my_homepage_view

urlpatterns = patterns('',
    ('^$', my_homepage_view),
    # ...
)
```

How Django Processes a Request

Before continuing to the second view function, let's pause to learn a little more about how Django works. Specifically, when you view your “Hello world” message by visiting `http://127.0.0.1:8000/hello/` in your Web browser, what does Django do behind the scenes?

It all starts with the *settings file*. When you run `python manage.py runserver`, the script looks for a file called `settings.py` in the same directory as `manage.py`. This file contains all sorts of configuration for this particular Django project, all in uppercase: `TEMPLATE_DIRS`, `DATABASE_NAME`, and so on. The most important setting is called `ROOT_URLCONF`. `ROOT_URLCONF` tells Django which Python module should be used as the URLconf for this Web site.

Remember when `django-admin.py startproject` created the files `settings.py` and `urls.py`? The autogenerated `settings.py` contains a `ROOT_URLCONF` setting that points to the autogenerated `urls.py`. Open the `settings.py` file and see for yourself; it should look like this:

```
ROOT_URLCONF = 'mysite.urls'
```

This corresponds to the file `mysite/urls.py`. When a request comes in for a particular URL—say, a request for `/hello/`—Django loads the URLconf pointed to by the `ROOT_URLCONF` setting. Then it checks each of the URLpatterns in that URLconf, in order, comparing the requested URL with the patterns one at a time, until it finds one that matches. When it finds one that matches, it calls the view function associated with that pattern, passing it an `HttpRequest` object as the first parameter. (We'll cover the specifics of `HttpRequest` later.)

As you saw in the first view example, a view function must return an `HttpResponse`. Once it does this, Django does the rest, converting the Python object to a proper Web response with the appropriate HTTP headers and body (the content of the Web page).

In summary, here are the steps:

1. A request comes in to `/hello/`.
2. Django determines the root URLconf by looking at the `ROOT_URLCONF` setting.
3. Django looks at all the URLpatterns in the URLconf for the first one that matches `/hello/`.
4. If it finds a match, it calls the associated view function.
5. The view function returns an `HttpResponse`.
6. Django converts the `HttpResponse` to the proper HTTP response, which results in a Web page.

You now know the basics of how to make Django-powered pages. It's quite simple, really: just write view functions and map them to URLs via URLconfs.

Your Second View: Dynamic Content

The “Hello world” view was instructive for demonstrating the basics of how Django works, but it wasn't an example of a *dynamic* Web page because the contents of the page are always the same. Every time you view `/hello/`, you'll see the same thing; it might as well be a static HTML file.

For the second view, let's create something more dynamic: a Web page that displays the current date and time. This is a nice and simple next step because it doesn't involve a database or any user input; just the output of the server's internal clock. It's only marginally more exciting than "Hello world," but it will demonstrate a few new concepts.

This view needs to do two things: calculate the current date and time, and return an `HttpResponse` containing that value. If you have experience with Python, you know that Python includes a `datetime` module for calculating dates. Here's how to use it:

```
>>> import datetime
>>> now = datetime.datetime.now()
>>> now
datetime.datetime(2008, 12, 13, 14, 9, 39, 2731)
>>> print now
2008-12-13 14:09:39.002731
```

That's simple enough, and it has nothing to do with Django. It's just Python code. (We want to emphasize that you should be aware of what code is "just Python" vs. code that is Django-specific. As you learn Django, we want you to be able to apply your knowledge to other Python projects that don't necessarily use Django.)

To make a Django view that displays the current date and time, you just need to hook this `datetime.datetime.now()` statement into a view and return an `HttpResponse`. Here's how it looks:

```
from django.http import HttpResponse
import datetime

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

As with the hello view function, this should live in `views.py`. Note that we hid the hello function from this example for brevity, but for the sake of completeness, here's what the entire `views.py` looks like:

```
from django.http import HttpResponse
import datetime

def hello(request):
    return HttpResponse("Hello world")

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

Note From now on, we won't display previous code in code examples except when necessary. You should be able to tell from context which parts of an example are new and which are old.

Let's step through the changes made to `views.py` to accommodate the `current_datetime` view:

- An `import datetime` was added to the top of the module, so you can calculate dates.
- The new `current_datetime` function calculates the current date and time as a `datetime.datetime` object and then stores it as the local variable `now`.
- The second line of code within the view constructs an HTML response using Python's "format-string" capability. The `%s` within the string is a placeholder, and the percent sign after the string means "Replace the `%s` in the preceding string with the value of the variable `now`." The `now` variable is technically a `datetime.datetime` object, not a string, but the `%s` format character converts it to its string representation, which is something like this: "2008-12-13 14:09:39.002731". It will result in an HTML string such as "`<html><body>It is now 2008-12-13 14:09:39.002731.</body></html>`".
- Yes, the HTML is invalid, but we're trying to keep the example simple and short.
- Finally, the view returns an `HttpResponse` object that contains the generated response—just as we did in `hello`.

After adding that to `views.py`, add the URLpattern to `urls.py` to tell Django which URL should handle this view. Something like `/time/` would make sense:

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

urlpatterns = patterns('',
    ('^hello/$', hello),
    ('^time/$', current_datetime),
)
```

Two changes were made. First, we imported the `current_datetime` function at the top. Second, and more importantly, we added a URLpattern mapping the URL `/time/` to that new view. Getting the hang of this?

With the view written and URLconf updated, fire up the runserver and visit `http://127.0.0.1:8000/time/` in your browser. You should see the current date and time.

DJANGO'S TIME ZONE

Depending on your computer, the date and time might be a few hours off. That's because Django is time zone-aware and defaults to the America/Chicago time zone. (It has to default to *something*, and that's the time zone where the original developers live.) If you live elsewhere, you'll want to change it in `settings.py`. See the comment in that file for a link to an up-to-date list of worldwide time zone options.

URLconfs and Loose Coupling

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

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

For example, consider the `current_datetime` view. If you wanted to change the URL for the application—say, to move it from `/time/` to `/current-time/`—you could make a quick change to the URLconf without having to worry about the view itself. Similarly, if you wanted to change the view function—altering its logic somehow—you could do that without affecting the URL to which the function is bound.

Furthermore, if you wanted to expose the current-date functionality at *several* URLs, you could easily take care of that by editing the URLconf, without having to touch the view code. In this example, the `current_datetime` is available at two URLs. It's a contrived example, but this technique can come in handy:

```
urlpatterns = patterns('',
    ('^hello/$', hello),
    ('^time/$', current_datetime),
    ('^another-time-page/$', current_datetime),
)
```

URLconfs and views are loose coupling in action. We'll continue to point out examples of this important philosophy throughout this book.

Your Third View: Dynamic URLs

In the `current_datetime` view, the contents of the page—the current date/time—were dynamic, but the URL (`/time/`) was static. In most dynamic Web applications, though, a URL contains parameters that influence the output of the page. For example, an online bookstore might give each book its own URL (for example, `/books/243/` and `/books/81196/`).

Let's create a third view that displays the current date and time offset by a certain number of hours. The goal is to craft a site so that the page `/time/plus/1/` displays the date/time one hour into the future, the page `/time/plus/2/` displays the date/time two hours into the future, the page `/time/plus/3/` displays the date/time three hours into the future, and so on.

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

```
urlpatterns = patterns('',
    ('^time/$', current_datetime),
    ('^time/plus/1/$', one_hour_ahead),
    ('^time/plus/2/$', two_hours_ahead),
    ('^time/plus/3/$', three_hours_ahead),
    ('^time/plus/4/$', four_hours_ahead),
)
```

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

A WORD ABOUT PRETTY URLS

If you're experienced in another Web development platform, such as PHP or Java, you might want to use a query string parameter—something like `/time/plus?hours=3`, in which the hours would be designated by the `hours` parameter in the URL's query string (the part after the `?`).

You *can* do that with Django (and we'll tell you how in Chapter 8 but one of Django's core philosophies is that URLs should be beautiful. The URL `/time/plus/3/` is far cleaner, simpler, more readable, easier to recite to somebody aloud, and just plain prettier than its query string counterpart. Pretty URLs are a characteristic of a quality Web application.

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

How then do you design the application to handle arbitrary hour offsets? The key is to use *wildcard URL patterns*. As mentioned previously, a URL pattern is a regular expression; hence, you can use the regular expression pattern `\d+` to match one or more digits:

```
urlpatterns = patterns('',
    # ...
    (r'^time/plus/\d+/$', hours_ahead),
    # ...
)
```

(We're using the `# ...` to imply there might be other URL patterns that we trimmed from this example.)

This new URL pattern will match any URL such as `/time/plus/2/`, `/time/plus/25/`, or even `/time/plus/1000000000000/`. Come to think of it, let's limit it so that the maximum allowed offset is 99 hours. That means we want to allow either one- or two-digit numbers—and in regular expression syntax, that translates into `\d{1,2}`:

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

Note When building Web applications, it's always important to consider the most outlandish data input possible and decide whether the application should support that input. We've curtailed the outlandishness here by limiting the offset to 99 hours.

One important detail introduced here is that `r` character in front of the regular expression string. This character tells Python that the string is a “raw string”—its contents should not interpret backslashes. In normal Python strings, backslashes are used for escaping special characters—such as in the string `'\n'`, which is a one-character string containing a newline. When you add the `r` to make it a raw string, Python does not apply its backslash escaping, so `r'\n'` is a two-character string containing a literal backslash and a lowercase `n`. There’s a natural collision between Python’s use of backslashes and the backslashes that are found in regular expressions, so it’s strongly suggested that you use raw strings any time you’re defining a regular expression in Python. From now on, all the URLpatterns in this book will be raw strings.

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

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

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

The final URLconf, including the previous two views, looks like this:

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

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

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

CODING ORDER

In this example, the URLpattern was written first and the view was written second, but in the previous examples, the view was written first and then the URLpattern was written. Which technique is better? Well, every developer is different.

If you’re a big-picture type of person, it might make the most sense to you to write all the URLpatterns for your application at the same time, at the start of your project, and then code up the views. This has the advantage of giving you a clear to-do list, and it essentially defines the parameter requirements for the view functions you’ll need to write.

If you’re more of a bottom-up developer, you might prefer to write the views first and then anchor them to URLs afterward. That’s OK, too.

In the end, it comes down to which technique fits your brain the best. Both approaches are valid.

`hours_ahead` is very similar to the `current_datetime` view written earlier with one key difference: it takes an extra argument the number of hours of offset. Here's the view code:

```
from django.http import Http404, HttpResponse
import datetime

def hours_ahead(request, offset):
    try:
        offset = int(offset)
    except ValueError:
        raise Http404()
    dt = datetime.datetime.now() + datetime.timedelta(hours=offset)
    html = "<html><body>In %s hour(s), it will be %s.</body></html>" % (offset, dt)
    return HttpResponse(html)
```

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

- The view function, `hours_ahead`, takes *two* parameters: `request` and `offset`.
- `request` is an `HttpRequest` object, just as in `hello` and `current_datetime`. We'll say it again: each view *always* takes an `HttpRequest` object as its first parameter.
- `offset` is the string captured by the parentheses in the URLpattern. For example, if the requested URL were `/time/plus/3/`, `offset` would be the string `'3'`. If the requested URL were `/time/plus/21/`, `offset` would be the string `'21'`. Note that captured values will always be *strings*, not integers, even if the string is composed of only digits such as `'21'`.

Note Technically, captured values will always be *Unicode objects*, not plain Python byte strings, but don't worry about this distinction at the moment.

- We decided to call the variable `offset`, but you can call it whatever you want as long as it's a valid Python identifier. The variable name doesn't matter; all that matters is that it's the second argument to the function, after `request`. (It's also possible to use keyword, instead of positional, arguments in a URLconf. We cover that in Chapter 8.)
- The first thing you do within the function is call `int()` on `offset`. This converts the string value to an integer.
- Note that Python will raise a `ValueError` exception if you call `int()` on a value that cannot be converted to an integer, such as the string `'foo'`. In this example, if you encounter the `ValueError`, you raise the exception `django.http.Http404`, which, as you can imagine, results in a 404 "Page not found" error.

- Astute readers will wonder how we could ever reach the `ValueError` case, given that the regular expression in the `URLpattern`—`(\d{1,2})`—captures only digits, and therefore `offset` will only ever be a string composed of digits. The answer is that we won't because the `URLpattern` provides a modest but useful level of input validation, *but* we still check for the `ValueError` in case this view function ever gets called in some other way. It's good practice to implement view functions such that they don't make any assumptions about their parameters. Loose coupling, remember?
- In the next line of the function, we calculate the current date/time and add the appropriate number of hours. You've already seen `datetime.datetime.now()` from the `current_datetime` view; the new concept here is that you can perform date/time arithmetic by creating a `datetime.timedelta` object and adding to a `datetime.datetime` object. The result is stored in the variable `dt`.
- This line also shows why we called `int()` on `offset`—the `datetime.timedelta` function requires the `hours` parameter to be an integer.
- Next, the HTML output of this view function is constructed, just as with `current_datetime`. A small difference in this line from the previous line is that it uses Python's format-string capability with *two* values, not just one. Hence, there are two `%s` symbols in the string and a tuple of values to insert: `(offset, dt)`.
- Finally, an `HttpResponse` of the HTML is returned. By now, this is old hat.

With that view function and `URLconf` written, start the Django development server (if it's not already running), and visit `http://127.0.0.1:8000/time/plus/3/` to verify it works. Then try `http://127.0.0.1:8000/time/plus/5/`. Then `http://127.0.0.1:8000/time/plus/24/`. Finally, visit `http://127.0.0.1:8000/time/plus/100/` to verify that the pattern in the `URLconf` accepts only one- or two-digit numbers; Django should display a "Page not found" error in this case, just as you saw in the section "A Quick Note About 404 Errors" earlier. The URL `http://127.0.0.1:8000/time/plus/` (with *no* hour designation) should also throw a 404.

Django's Pretty Error Pages

Take a moment to admire the fine Web application you've made so far and now you'll break it! Let's deliberately introduce a Python error into the `views.py` file by commenting out the `offset = int(offset)` lines in the `hours_ahead` view:

```
def hours_ahead(request, offset):
    # try:
    #     offset = int(offset)
    # except ValueError:
    #     raise Http404()
    dt = datetime.datetime.now() + datetime.timedelta(hours=offset)
    html = "<html><body>In %s hour(s), it will be %s.</body></html>" % (offset, dt)
    return HttpResponse(html)
```

Load up the development server and navigate to `/time/plus/3/`. You'll see an error page with a significant amount of information, including a `TypeError` message displayed at the very top: "unsupported type for timedelta hours component: unicode".

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

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

Here are some things to notice:

- At the top of the page, you get the key information about the exception: the type of exception, any parameters to the exception (the "unsupported type" message in this case), the file in which the exception was raised, and the offending line number.
- Under the key exception information, the page displays the full Python traceback for this exception. This is similar to the standard traceback you get in Python's command-line interpreter, except it's more interactive. For each level ("frame") in the stack, Django displays the name of the file, the function/method name, the line number, and the source code of that line.
- Click the line of source code (in dark gray), and you'll see several lines from before and after the erroneous line, to give you context.
- Click "Local vars" under any frame in the stack to view a table of all local variables and their values, in that frame, at the exact point in the code at which the exception was raised. This debugging information can be a great help.
- Note the "Switch to copy-and-paste view" text under the "Traceback" header. Click those words, and the traceback will switch to an alternate version that can be easily copied and pasted. Use this when you want to share your exception traceback with others to get technical support—such as the kind folks in the Django IRC chat room or on the Django users' mailing list.
- Underneath, the "Share this traceback on a public Web site" button will do this work for you in just one click. Click it to post the traceback to <http://www.dpaste.com/>, where you'll get a distinct URL that you can share with other people.
- Next, the "Request information" section includes a wealth of information about the incoming Web request that spawned the error: GET and POST information, cookie values, and meta-information, such as Common Gateway Interface (CGI) headers. Appendix G has a complete reference of all the information that a request object contains.
- Below the "Request information" section, the "Settings" section lists all the settings for this particular Django installation. (We already mentioned `ROOT_URLCONF` and we'll show you various Django settings throughout the book. All the available settings are covered in detail in Appendix D.)

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

Are you the type of programmer who likes to debug with the help of carefully placed print statements? You can use the Django error page to do so—just without the print statements. At any point in your view, temporarily insert an `assert False` to trigger the error page. Then you can view the local variables and state of the program. Here's an example using the `hours_ahead` view:

```
def hours_ahead(request, offset):
    try:
        offset = int(offset)
    except ValueError:
        raise Http404()
    dt = datetime.datetime.now() + datetime.timedelta(hours=offset)
    assert False
    html = "<html><body>In %s hour(s), it will be %s.</body></html>" % (offset, dt)
    return HttpResponse(html)
```

Finally, it's obvious that much of this information is sensitive—it exposes the innards of your Python code and Django configuration—and it would be foolish to show this information on the public Internet. A malicious person could use it to attempt to reverse-engineer your Web application and do nasty things. For that reason, the Django error page is displayed only when your Django project is in debug mode. We'll explain how to deactivate debug mode in Chapter 12. For now, just know that every Django project is in debug mode automatically when you start it. (Sound familiar? The “Page not found” errors, described earlier in this chapter, work the same way.)

What's Next?

So far, we've been writing the view functions with HTML hard-coded directly in the Python code. We've done that to keep things simple while we demonstrated core concepts, but in the real world, this is nearly always a bad idea.

Django ships with a simple yet powerful template engine that allows you to separate the design of the page from the underlying code. You'll dive into Django's template engine in the next chapter.



Templates

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

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

Although this technique was convenient for the purpose of explaining how views work, it's not a good idea to hard-code HTML directly in your views. Here's why:

- Any change to the design of the page requires a change to the Python code. The design of a site tends to change far more frequently than the underlying Python code, so it would be convenient if the design could change without needing to modify the Python code.
- Writing Python code and designing HTML are two different disciplines, and most professional Web-development environments split these responsibilities between separate people (or even separate departments). Designers and HTML/CSS coders shouldn't be required to edit Python code to get their job done.
- It's most efficient if programmers can work on Python code and designers can work on templates at the same time, rather than one person waiting for the other to finish editing a single file that contains both Python and HTML.

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

Template-System Basics

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

Let's start with a simple example template. This Django template describes an HTML page that thanks a person for placing an order with a company. Think of it as a form letter:

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

<body>

<h1>Ordering notice</h1>

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

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

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

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

{% if ordered_warranty %}
    <p>Your warranty information will be included in the packaging.</p>
{% else %}
    <p>You didn't order a warranty, so you're on your own when
    the products inevitably stop working.</p>
{% endif %}

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

</body>
</html>
```

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

- Any text surrounded by a pair of braces (e.g., `{{ person_name }}`) is a *variable*. This means “insert the value of the variable with the given name.” How do we specify the values of the variables? We'll get to that in a moment.
- Any text that's surrounded by curly braces and percent signs (e.g., `{% if ordered_warranty %}`) is a *template tag*. The definition of a tag is quite broad: a tag just tells the template system to “do something.”

This example template contains a for tag (`{% for item in item_list %}`) and an if tag (`{% if ordered_warranty %}`).

A for tag works very much like a for statement in Python, letting you loop over each item in a sequence. An if tag, as you may expect, acts as a logical “if” statement. In this particular case, the tag checks whether the value of the `ordered_warranty` variable evaluates to `True`. If it does, the template system will display everything between the `{% if ordered_warranty %}` and `{% else %}`. If not, the template system will display everything between `{% else %}` and `{% endif %}`. Note that the `{% else %}` is optional.

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

Each Django template has access to several built-in tags and filters, many of which are discussed in the sections that follow. Appendix F contains the full list of tags and filters, and it's a good idea to familiarize yourself with that list so you know what's possible. It's also possible to create your own filters and tags; we'll cover that in Chapter 9.

Using the Template System

Let's dive into Django's template system so you can see how it works—but we're *not* yet going to integrate it with the views that we created in the previous chapter. Our goal here is to show you how the system works independent of the rest of Django. (Usually you'll use the template system within a Django view, but we want to make it clear that the template system is just a Python library that you can use *anywhere*, not just in Django views.)

Here is the most basic way you can use Django's template system in Python code:

1. Create a `Template` object by providing the raw template code as a string.
2. Call the `render()` method of the `Template` object with a given set of variables (the *context*). This returns a fully rendered template as a string, with all of the variables and template tags evaluated according to the context.

In code, it looks like this:

```
>>> from django import template
>>> t = template.Template('My name is {{ name }}.')
>>> c = template.Context({'name': 'Adrian'})
>>> print t.render(c)
My name is Adrian.
>>> c = template.Context({'name': 'Fred'})
>>> print t.render(c)
My name is Fred.
```

The following sections describe these steps in much more detail.

Creating Template Objects

The easiest way to create a `Template` object is to instantiate it directly. The `Template` class lives in the `django.template` module, and the constructor takes one argument, the raw template code. Let's dip into the Python interactive interpreter to see how this works in code.

From the `mysite` project directory created by `django-admin.py startproject` (as covered in Chapter 2), type `python manage.py shell` to start the interactive interpreter.

A SPECIAL PYTHON PROMPT

If you've used Python before, you may be wondering why we're running `python manage.py shell` instead of just `python`. Both commands will start the interactive interpreter, but the `manage.py shell` command has one key difference from `python`: before starting the interpreter, it tells Django which settings file to use. Many parts of Django—including the template system—rely on your settings, and you won't be able to use them unless the framework knows which settings to use.

If you're curious, here's how it works behind the scenes. Django looks for an environment variable called `DJANGO_SETTINGS_MODULE`, which should be set to the import path of your `settings.py`. For example, `DJANGO_SETTINGS_MODULE` might be set to `'mysite.settings'`, assuming `mysite` is on your Python path.

When you run `python manage.py shell`, the command takes care of setting `DJANGO_SETTINGS_MODULE` for you. We're encouraging you to use `python manage.py shell` in these examples to minimize the amount of tweaking and configuring you have to do.

As you become more familiar with Django, you'll likely stop using `manage.py shell` and will set `DJANGO_SETTINGS_MODULE` manually in your `.bash_profile` or other shell-environment configuration file.

Let's go through some template-system basics:

```
>>> from django.template import Template
>>> t = Template('My name is {{ name }}.')
>>> print t
```

If you're following along interactively, you'll see something like this:

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

That 0xb7d5f24c will be different every time, but it isn't relevant; it's a Python thing (the Python “identity” of the `Template` object, if you must know).

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

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

The term *block tag* here refers to `{% notatag %}`. *Block tag* and *template tag* are synonymous.

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

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

Rendering a Template

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

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

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

We should point out here that the return value of `t.render(c)` is a Unicode object—not a normal Python string. You can tell this by the `u` in front of the string. Django uses Unicode objects instead of normal strings throughout the framework. If you understand the repercussions of that, be thankful for the sophisticated things Django does behind the scenes to make it work. If you don't understand the repercussions of that, don't worry for now; just know that Django's Unicode support makes it relatively painless for your applications to support a wide variety of character sets beyond the basic “A–Z” of the English language.

DICTIONARIES AND CONTEXTS

A Python dictionary is a mapping between known keys and variable values. A Context is similar to a dictionary, but a Context provides additional functionality, as covered in Chapter 9.

Variable names must begin with a letter (A–Z or a–z) and may contain additional letters, digits, underscores, and dots. (Dots are a special case we’ll discuss in the section “Context Variable Lookup.”) Variable names are case-sensitive.

Here’s an example of template compilation and rendering, using a template similar to the example at the beginning of this chapter:

```
>>> from django.template import Template, Context
>>> raw_template = """<p>Dear {{ person_name }},</p>
...
... <p>Thanks for placing an order from {{ company }}. It's scheduled to
... ship on {{ ship_date|date:"F j, Y" }}.</p>
...
... {% if ordered_warranty %}
... <p>Your warranty information will be included in the packaging.</p>
... {% else %}
... <p>You didn't order a warranty, so you're on your own when
... the products inevitably stop working.</p>
... {% endif %}
...
... <p>Sincerely,<br />{{ company }}</p>"""
>>> t = Template(raw_template)
>>> import datetime
>>> c = Context({'person_name': 'John Smith',
...            'company': 'Outdoor Equipment',
...            'ship_date': datetime.date(2009, 4, 2),
...            'ordered_warranty': False})
>>> t.render(c)
u"<p>Dear John Smith,</p>\n\n<p>Thanks for placing an order from Outdoor
Equipment. It's scheduled to\nship on April 2, 2009.</p>\n\n\n<p>You
didn't order a warranty, so you're on your own when\nthe products
inevitably stop working.</p>\n\n\n\n<p>Sincerely,<br />Outdoor Equipment
</p>"
```

Let’s step through this code one statement at a time:

1. First we import the classes `Template` and `Context`, which both live in the module `django.template`.
2. We save the raw text of our template into the variable `raw_template`. Note that we use triple quotation marks to designate the string, because it wraps over multiple lines; in contrast, strings within single quotation marks cannot be wrapped over multiple lines.

3. Next we create a template object, `t`, by passing `raw_template` to the `Template` class constructor.
4. We import the `datetime` module from Python’s standard library because we’ll need it in the following statement.
5. We create a `Context` object, `c`. The `Context` constructor takes a Python dictionary, which maps variable names to values. Here, for example, we specify that the `person_name` is 'John Smith', `company` is 'Outdoor Equipment', and so forth.
6. Finally, we call the `render()` method on our template object, passing it the context. This returns the rendered template—that is, it replaces template variables with the actual values of the variables, and it executes any template tags.

Note that the “You didn’t order a warranty” paragraph was displayed because the `ordered_warranty` variable evaluated to `False`. Also note the date, April 2, 2009, which is displayed according to the format string `'F j, Y'`. (We’ll explain format strings for the date filter in Appendix E.)

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

```
print t.render(c).
```

Those are the fundamentals of using the Django template system: just write a template string, create a `Template` object, create a `Context`, and call the `render()` method.

Multiple Contexts, Same Template

Once you have a `Template` object, you can render multiple contexts through it. Consider this example:

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

Whenever you’re using the same template source to render multiple contexts like this, it’s more efficient to create the `Template` object *once*, and then call `render()` on it multiple times:

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

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

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

Context Variable Lookup

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

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

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

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

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

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


This example uses a custom class, demonstrating that variable dots also allow attribute access on arbitrary objects:

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

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

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

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

Finally, dots are also used to access list indices, as in this example:

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

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

PYTHON LISTS

A reminder: Python lists have 0-based indices. The first item is at index 0, the second is at index 1, and so on.

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

- Dictionary lookup (e.g., `foo["bar"]`)
- Attribute lookup (e.g., `foo.bar`)
- Method call (e.g., `foo.bar()`)
- List-index lookup (e.g., `foo[2]`)

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

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

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

Method-Call Behavior

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

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

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

>>> class SilentAssertionError(AssertionError):
...     silent_variable_failure = True
>>> class PersonClass4:
...     def first_name(self):
...         raise SilentAssertionError
>>> p = PersonClass4()
>>> t.render(Context({"person": p}))
u'My name is .'
```

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

Say, for instance, you have a `BankAccount` object that has a `delete()` method. If a template includes something like `{{ account.delete }}`, where `account` is a `BankAccount` object, the object would be deleted when the template is rendered!

To prevent this, set the function attribute `alters_data` on the method:

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

- The template system won't execute any method marked in this way. Continuing the current example, if a template includes `{{ account.delete }}` and the `delete()` method has `alters_data=True`, then the `delete()` method will not be executed when the template is rendered. Instead it will fail silently.

How Invalid Variables Are Handled

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

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

The system fails silently rather than raising an exception because it's intended to be resilient to human error. In this case, all of the lookups failed because variable names have the wrong case or name. In the real world, it's unacceptable for a Web site to become inaccessible due to a small template syntax error.

Playing with Context Objects

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

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

Basic Template Tags and Filters

As we've mentioned already, the template system ships with built-in tags and filters. The sections that follow provide a rundown of the most common tags and filters.

Tags

The following sections outline the common Django tags.

if/else

The `{% if %}` tag evaluates a variable, and if that variable is `True` (i.e., it exists, is not empty, and is not a `False` Boolean value), the system will display everything between `{% if %}` and `{% endif %}`, as in this example:

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

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

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

PYTHON "TRUTHINESS"

In Python and in the Django template system, these objects evaluate to `False` in a Boolean context:

- An empty list (`[]`).
- An empty tuple (`()`).
- An empty dictionary (`{}`).
- An empty string (`' '`).
- Zero (`0`).
- The special object `None`.
- The object `False` (obviously).
- Custom objects that define their own Boolean context behavior. (This is advanced Python usage.)

Everything else evaluates to `True`.

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

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

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

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

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

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

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

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

The use of parentheses for controlling order of operations is not supported. If you find yourself needing parentheses, consider performing logic outside the template and passing the result of that as a dedicated template variable. Or just use nested `{% if %}` tags, like this:

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

Multiple uses of the same logical operator are fine, but you can't combine different operators. For example, this is valid:

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

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

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

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

for

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

For example, you could use the following to display a list of athletes given a variable `athlete_list`:

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

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

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

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

```
{% for athlete in athlete_list %}
  <h1>{{ athlete.name }}</h1>
  <ul>
    {% for sport in athlete.sports_played %}
      <li>{{ sport }}</li>
    {% endfor %}
  </ul>
{% endfor %}
```

A common pattern is to check the size of the list before looping over it, and outputting some special text if the list is empty:

```
{% if athlete_list %}
  {% for athlete in athlete_list %}
    <p>{{ athlete.name }}</p>
  {% endfor %}
{% else %}
  <p>There are no athletes. Only computer programmers.</p>
{% endif %}
```

Because this pattern is so common, the for tag supports an optional {% empty %} clause that lets you define what to output if the list is empty. This example is equivalent to the previous one:

```
{% for athlete in athlete_list %}
  <p>{{ athlete.name }}</p>
{% empty %}
  <p>There are no athletes. Only computer programmers.</p>
{% endfor %}
```

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

Within each {% for %} loop, you get access to a template variable called `forloop`. This variable has a few attributes that give you information about the progress of the loop:

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

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

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

- `forloop.revcounter` is always set to an integer representing the number of remaining items in the loop. The first time through the loop, `forloop.revcounter` will be set to the total number of items in the sequence you're traversing. The last time through the loop, `forloop.revcounter` will be set to 1.
- `forloop.revcounter0` is like `forloop.revcounter`, except it's zero-indexed. The first time through the loop, `forloop.revcounter0` will be set to the number of elements in the sequence, minus 1. The last time through the loop, it will be set to 0.
- `forloop.first` is a Boolean value set to `True` if this is the first time through the loop. This is convenient for special-casing:

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

- `forloop.last` is a Boolean value set to `True` if this is the last time through the loop. A common use for this is to put pipe characters between a list of links:

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

The preceding template code might output something like this:

```
Link1 | Link2 | Link3 | Link4
```

Another common use for this is to put a comma between words in a list:

```
Favorite places:
{% for p in places %}{{ p }}{% if not forloop.last %}, {% endif %}{% endfor %}
```

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

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

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

CONTEXT AND THE FORLOOP VARIABLE

Inside the `{% for %}` block, the existing variables are moved out of the way to avoid overwriting the magic `forloop` variable. Django exposes this moved context in `forloop.parentloop`. You generally don't need to worry about this, but if you supply a template variable named `forloop` (though we advise against it, because it could confuse fellow template authors), it will be named `forloop.parentloop` while inside the `{% for %}` block.

ifequal/ifnotequal

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Comments

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

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

The comment will not be output when the template is rendered.

Comments using this syntax cannot span multiple lines. This limitation improves template parsing performance. In the following template, the rendered output will look exactly the same as the template (i.e., the comment tag will not be parsed as a comment):

```
This is a {# this is not
a comment #}
test.
```

If you want to use multiline comments, use the `{% comment %}` template tag, like this:

```
{% comment %}
This is a
multiline comment.
{% endcomment %}
```

Filters

As explained earlier in this chapter, template filters are simple ways of altering the value of variables before they're displayed. Filters use a pipe character, like this:

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

This displays the value of the `{{ name }}` variable after being filtered through the `lower` filter, which converts text to lowercase.

Filters can be *chained*—that is, they can be used in tandem such that the output of one filter is applied to the next. Here's an example that converts the first element in a list to uppercase:

```
{{ my_list|first|upper }}
```

Some filters take arguments. A filter argument comes after a colon and is always in double quotes. Here's an example:

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

This displays the first 30 words of the `bio` variable.

The following are a few of the most important filters. Appendix F covers the rest.

- `addslashes`: Adds a backslash before any backslash, single quote, or double quote. This is useful if the produced text is included in a JavaScript string.
- `date`: Formats a date or datetime object according to a format string given in the parameter, as in this example:

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

Format strings are defined in Appendix F.

- `length`: Returns the length of the value. For a list, this returns the number of elements. For a string, this returns the number of characters. (Python experts, note that this works on any Python object that knows how to determine its own length—that is, any object that has a `__len__()` method.)

Philosophies and Limitations

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

More than any other component of Web applications, template syntax is highly subjective, and programmers' opinions vary wildly. The fact that Python alone has dozens, if not hundreds, of open source template-language implementations supports this point. Each was likely created because its developer deemed all existing template languages inadequate. (In fact, it is said to be a rite of passage for a Python developer to write his or her own template language! If you haven't done this yet, consider it. It's a fun exercise.)

With that in mind, you might be interested to know that Django doesn't require you to use its template language. Because Django is intended to be a full-stack Web framework that provides all the pieces necessary for Web developers to be productive, many times it's more convenient to use Django's template system than other Python template libraries, but it's not a strict requirement in any sense. As you'll see in the upcoming section "Using Templates in Views," it's very easy to use another template language with Django.

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

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

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

- *Syntax should be decoupled from HTML/XML.* Although Django’s template system is used primarily to produce HTML, it’s intended to be just as usable for non-HTML formats, such as plain text. Some other template languages are XML based, placing all template logic within XML tags or attributes, but Django deliberately avoids this limitation. Requiring valid XML for writing templates introduces a world of human mistakes and hard-to-understand error messages, and using an XML engine to parse templates incurs an unacceptable level of overhead in template processing.
- *Designers are assumed to be comfortable with HTML code.* The template system isn’t designed so that templates necessarily are displayed nicely in WYSIWYG editors such as Dreamweaver. That is too severe a limitation and wouldn’t allow the syntax to be as friendly as it is. Django expects template authors to be comfortable editing HTML directly.
- *Designers are assumed not to be Python programmers.* The template-system authors recognize that Web-page templates are most often written by *designers*, not *programmers*, and therefore should not assume Python knowledge.

However, the system also intends to accommodate small teams in which the templates *are* created by Python programmers. It offers a way to extend the system’s syntax by writing raw Python code. (More on this in Chapter 9.)

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

Using Templates in Views

You’ve learned the basics of using the template system; now let’s use this knowledge to create a view. Recall the `current_datetime` view in `mysite.views`, which we started in the previous chapter. Here’s what it looks like:

```
from django.http import HttpResponse
import datetime

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

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

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

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

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

You might first consider saving your template somewhere on your filesystem and using Python's built-in file-opening functionality to read the contents of the template. Here's what that might look like, assuming the template was saved as the file `/home/djangouser/templates/mytemplate.html`:

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

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

This approach, however, is inelegant for these reasons:

- It doesn't handle the case of a missing file, as noted in the code. If the file `mytemplate.html` doesn't exist or isn't readable, the `open()` call will raise an `IOError` exception.
- It hard-codes your template location. If you were to use this technique for every view function, you'd be duplicating the template locations—not to mention that it involves a lot of typing!
- It includes a lot of boring boilerplate code. You've got better things to do than to write calls to `open()`, `fp.read()`, and `fp.close()` each time you load a template.

To solve these issues, we'll use *template loading* and *template inheritance*.

Template Loading

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

To use this template-loading API, first you'll need to tell the framework where you store your templates. The place to do this is in your *settings file*—the `settings.py` file that we mentioned in the last chapter, when we introduced the `ROOT_URLCONF` setting.

If you're following along, open `settings.py` and find the `TEMPLATE_DIRS` setting. By default, it's an empty tuple, likely containing some autogenerated comments:

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

This setting tells Django's template-loading mechanism where to look for templates. Pick a directory where you'd like to store your templates and add it to `TEMPLATE_DIRS`, like so:

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

There are a few things to note:

- You can specify any directory you want, as long as the directory and templates within that directory are readable by the user account under which your Web server runs. If you can't think of an appropriate place to put your templates, we recommend creating a templates directory within your project (i.e., within the `mysite` directory you created in Chapter 2).
- If your `TEMPLATE_DIRS` contains only one directory, don't forget the comma at the end of the directory string!

Bad:

```
# Missing comma!  
TEMPLATE_DIRS = (  
    '/home/django/mysite/templates'  
)
```

Good:

```
# Comma correctly in place.  
TEMPLATE_DIRS = (  
    '/home/django/mysite/templates',  
)
```

Python requires commas within single-element tuples to disambiguate the tuple from a parenthetical expression. This is a common newbie gotcha.

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

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

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

```
import os.path

TEMPLATE_DIRS = (
    os.path.join(os.path.dirname(__file__), 'templates').replace('\\', '/'),
)
```

This example uses the “magic” Python variable `__file__`, which is automatically set to the file name of the Python module in which the code lives. It gets the name of the directory that contains `settings.py` (`os.path.dirname`), joins that with `templates` in a cross-platform way (`os.path.join`), then ensures that everything uses forward slashes instead of backslashes (in the case of Windows).

While we're on the topic of dynamic Python code in settings files, we should point out that it's very important to avoid Python errors in your settings file. If you introduce a syntax error or a runtime error, your Django-powered site will likely crash.

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

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

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

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

Our template in this example is `current_datetime.html`, but there's nothing special about that `.html` extension. You can give your templates whatever extension makes sense for your application, or you can leave off extensions entirely.

To determine the location of the template on your filesystem, `get_template()` combines your template directories from `TEMPLATE_DIRS` with the template name that you pass to `get_template()`. For example, if your `TEMPLATE_DIRS` is set to `['/home/django/mysite/templates']`, the `get_template()` call would look for the template `/home/django/mysite/templates/current_datetime.html`.

If `get_template()` cannot find the template with the given name, it raises a `TemplateDoesNotExist` exception. To see what that looks like, fire up the Django development server by running `python manage.py runserver` within your Django project's directory. Then point your browser at the page that activates the `current_datetime` view (e.g., `http://127.0.0.1:8000/time/`). Assuming `DEBUG` is set to `True` and you haven't yet created a `current_datetime.html` template, you should see a Django error page highlighting the `TemplateDoesNotExist` error, as shown in Figure 4-1.

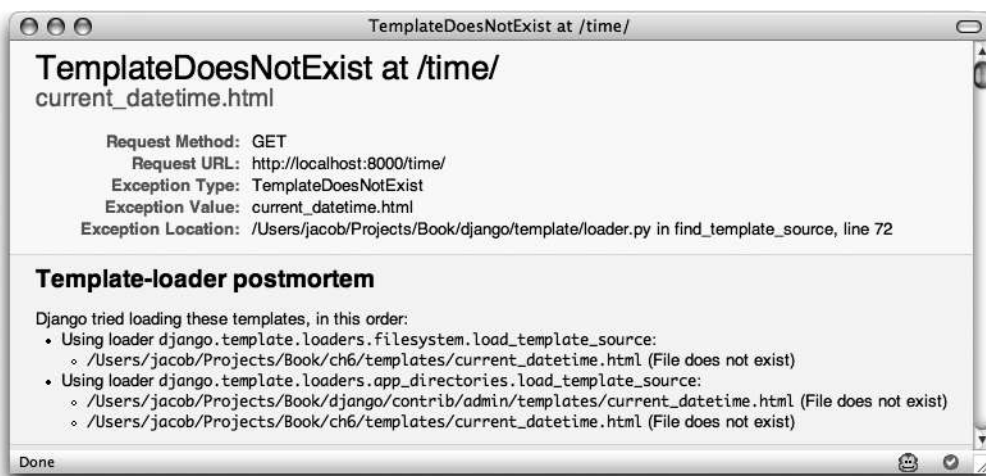


Figure 4-1. The error page shown when a template cannot be found

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

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

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

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

render_to_response()

We've shown you how to load a template, fill a Context, and return an `HttpResponse` object with the result of the rendered template. We've optimized it to use `get_template()` instead of hard-coding templates and template paths. But it still requires a fair amount of typing to do those things. Because these steps are such a common idiom, Django provides a shortcut that lets you load a template, render it, and return an `HttpResponse`—all in one line of code.

This shortcut is a function called `render_to_response()`, which lives in the module `django.shortcuts`. Most of the time you'll be using `render_to_response()` rather than loading templates and creating Context and `HttpResponse` objects manually—unless your employer judges your work by total lines of code written.

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

```
from django.shortcuts import render_to_response
import datetime

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

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

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

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

The locals() Trick

Consider our latest incarnation of `current_datetime`:

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

Many times, as in this example, you'll find yourself calculating some values, storing them in variables (e.g., `now` in the preceding code), and sending those variables to the template. Particularly lazy programmers should note that it's slightly redundant to have to give names for temporary variables *and* give names for the template variables. It also means extra typing.

So if you're one of those lazy programmers and you like keeping code particularly concise, you can take advantage of a built-in Python function called `locals()`. It returns a dictionary mapping all local variable names to their values, where *local* means all variables that have been defined within the current scope. Thus, the preceding view could be rewritten like so:

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

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

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

Subdirectories in `get_template()`

It can get unwieldy to store all of your templates in a single directory. You might like to store templates in subdirectories of your template directory, and that's fine. In fact, we recommend doing so; some more-advanced Django features (such as the generic views system, which we cover in Chapter 11) expect this template layout as a default convention.

Storing templates in subdirectories of your template directory is easy. In your calls to `get_template()`, just include the subdirectory name and a slash before the template name, like so:

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

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

```
return render_to_response('dateapp/current_datetime.html', {'current_date': now})
```

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

Note Windows users, be sure to use forward slashes rather than backslashes. `get_template()` assumes a Unix-style file-name designation.

The include Template Tag

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

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

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

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

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

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

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

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

Included templates are evaluated with the context of the template that's including them. For example, consider these two templates:

```
# mypage.html

<html>
<body>
{% include "includes/nav.html" %}
<h1>{{ title }}</h1>
</body>
</html>

# includes/nav.html

<div id="nav">
    You are in: {{ current_section }}
</div>
```

If you render `mypage.html` with a context containing `current_section`, then the variable will be available in the included template, as you would expect.

If, in an `{% include %}` tag, a template with the given name isn't found, Django will do one of two things:

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

Template Inheritance

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

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

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

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

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
<html lang="en">
<head>
    <title>The current time</title>
</head>
<body>
    <h1>My helpful timestamp site</h1>
    <p>It is now {{ current_date }}.</p>

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

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

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

```

<body>
  <h1>My helpful timestamp site</h1>
  <p>In {{ hour_offset }} hour(s), it will be {{ next_time }}.</p>

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

```

Clearly, we've just duplicated a lot of HTML. Imagine if we had a more typical site, including a navigation bar, a few style sheets, perhaps some JavaScript—we'd end up putting all sorts of redundant HTML into each template.

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

```

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

```

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

```

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

```

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

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

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

```

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

```

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

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

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

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

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

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

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

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

{% block title %}Future time{% endblock %}

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

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

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

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

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

Inheritance doesn't affect the template context. In other words, any template in the inheritance tree will have access to every one of your template variables from the context.

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

1. Create a `base.html` template that holds the main look and feel of your site. This is the stuff that rarely, if ever, changes.
2. Create a `base_SECTION.html` template for each “section” of your site (e.g., `base_photos.html` and `base_forum.html`). These templates extend `base.html` and include section-specific styles/design.
3. Create individual templates for each type of page, such as a forum page or a photo gallery. These templates extend the appropriate section template.

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

Here are some guidelines for working with template inheritance:

- If you use `{% extends %}` in a template, it must be the first template tag in that template. Otherwise, template inheritance won’t work.
- Generally, the more `{% block %}` tags in your base templates, the better. Remember, child templates don’t have to define all parent blocks, so you can fill in reasonable defaults in a number of blocks and then define only the ones you need in the child templates. It’s better to have more hooks than fewer hooks.
- If you find yourself duplicating code in a number of templates, it probably means you should move that code to a `{% block %}` in a parent template.
- If you need to get the content of the block from the parent template, use `{{ block.super }}`, which is a “magic” variable providing the rendered text of the parent template. This is useful if you want to add to the contents of a parent block instead of completely overriding it.
- You may not define multiple `{% block %}` tags with the same name in the same template. This limitation exists because a block tag works in both directions. That is, a block tag doesn’t just provide a hole to fill; it also defines the content that fills the hole in the *parent*. If there were two similarly named `{% block %}` tags in a template, that template’s parent wouldn’t know which one of the blocks’ content to use.
- The template name you pass to `{% extends %}` is loaded using the same method that `get_template()` uses. That is, the template name is appended to your `TEMPLATE_DIRS` setting.
- In most cases, the argument to `{% extends %}` will be a string, but it can be a variable if you don’t know the name of the parent template until runtime. This lets you do some cool, dynamic stuff.

What's Next?

You now have the basics of Django's template system under your belt. What's next?

Many modern Web sites are *database-driven*: the content of the Web site is stored in a relational database. This allows a clean separation of data and logic (in the same way views and templates allow the separation of logic and display).

The next chapter covers the tools Django gives you to interact with a database.



Models

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

In modern Web applications, the arbitrary logic often involves interacting with a database. Behind the scenes, a *database-driven Web site* connects to a database server, retrieves some data out of it, and displays that data on a Web page. The site might also provide ways for site visitors to populate the database on their own.

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

Django is well suited for making database-driven Web sites because it comes with easy yet powerful tools for performing database queries using Python. This chapter explains that functionality: Django's database layer.

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

The “Dumb” Way to Do Database Queries in Views

Just as Chapter 3 detailed a “dumb” way to produce output within a view (by hard-coding the text directly within the view), there's a “dumb” way to retrieve data from a database in a view. It's simple: just use any existing Python library to execute an SQL query and do something with the results.

In this example view, we use the MySQLdb library (available via <http://www.djangoproject.com/r/python-mysql/>) to connect to a MySQL database, retrieve some records, and feed them to a template for display as a Web page:

```
from django.shortcuts import render_to_response
import MySQLdb

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

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

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

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

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

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

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

The MTV (or MVC) Development Pattern

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

As we mentioned in previous chapters, Django is designed to encourage loose coupling and strict separation between pieces of an application. If you follow this philosophy, it's easy to make changes to one particular piece of the application without affecting the other pieces.

In view functions, for instance, we discussed the importance of separating the business logic from the presentation logic by using a template system. With the database layer, we're applying that same philosophy to data access logic.

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

WHY THE ACRONYM?

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

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

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

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

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

If you're familiar with other MVC Web-development frameworks, such as Ruby on Rails, you may consider Django views to be the “controllers” and Django templates to be the “views.” This is an unfortunate confusion brought about by differing interpretations of MVC.

In Django’s interpretation of MVC, the “view” describes the data that gets presented to the user; it’s not necessarily just *how* the data looks, but *which* data is presented. In contrast, Ruby on Rails and similar frameworks suggest that the controller’s job includes deciding which data gets presented to the user, whereas the view is strictly *how* the data looks, not *which* data is presented.

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

Configuring the Database

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

We’ll assume you’ve set up a database server, activated it, and created a database within it (e.g., using a `CREATE DATABASE` statement). If you’re using SQLite, no such setup is required because SQLite uses standalone files on the filesystem to store its data.

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

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

Here’s a rundown of each setting.

- `DATABASE_ENGINE` tells Django which database engine to use. If you’re using a database with Django, `DATABASE_ENGINE` must be set to one of the strings shown in Table 5-1.

Table 5-1. *Database Engine Settings*

Setting	Database	Required Adapter
postgresql	PostgreSQL	psycopg version 1.x, http://www.djangoproject.com/r/python-psycopg/1/ .
postgresql_psycopg2	PostgreSQL	psycopg version 2.x, http://www.djangoproject.com/r/python-psycopg/ .
mysql	MySQL	MySQLdb, http://www.djangoproject.com/r/python-mysqldb/ .
sqlite3	SQLite	No adapter needed if using Python 2.5+. Otherwise, pysqlite, http://www.djangoproject.com/r/python-sqlite/ .
oracle	Oracle	cx_Oracle, http://www.djangoproject.com/r/python-oracle/ .

- Note that for whichever database back-end you use, you'll need to download and install the appropriate database adapter. Each one is available for free on the Web; just follow the links in the "Required Adapter" column in Table 5-1. If you're on Linux, your distribution's package-management system might offer convenient packages. (Look for packages called `python-postgresql` or `python-psycopg2`.) For example:

```
DATABASE_ENGINE = 'postgresql_psycopg2'
```

- `DATABASE_NAME` tells Django the name of your database. For example:

```
DATABASE_NAME = 'mydb'
```

If you're using SQLite, specify the full filesystem path to the database file on your filesystem. For example:

```
DATABASE_NAME = '/home/django/mydata.db'
```

As for where to put that SQLite database, we're using the `/home/django` directory in this example, but you should pick a directory that works best for you.

- `DATABASE_USER` tells Django which username to use when connecting to your database. If you're using SQLite, leave this blank.
- `DATABASE_PASSWORD` tells Django which password to use when connecting to your database. If you're using SQLite or have an empty password, leave this blank.
- `DATABASE_HOST` tells Django which host to use when connecting to your database. If your database is on the same computer as your Django installation (i.e., `localhost`), leave this blank. If you're using SQLite, leave this blank.

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

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

Once you've entered those settings and saved `settings.py`, it's a good idea to test your configuration. To do this, run `python manage.py shell`, as in the last chapter, from within the `mysite` project directory. (As discussed in the previous chapter, `manage.py shell` is a way to run the Python interpreter with the correct Django settings activated. This is necessary in our case because Django needs to know which settings file to use in order to get your database connection information.)

In the shell, type these commands to test your database configuration:

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

If nothing happens, then your database is configured properly. Otherwise, check the error message for clues about what's wrong. Table 5-2 shows some common errors.

Table 5-2. *Database Configuration Error Messages*

Error Message	Solution
You haven't set the DATABASE_ENGINE setting yet.	Set the DATABASE_ENGINE setting to something other than an empty string. Valid values are shown in Table 5-1.
Environment variable DJANGO_SETTINGS_MODULE is undefined.	Run the command <code>python manage.py shell</code> rather than <code>python</code> .
Error loading ____ module: No module named ____.	You haven't installed the appropriate database-specific adapter (e.g., <code>psycopg</code> or <code>MySQLdb</code>). Adapters are <i>not</i> bundled with Django, so it's your responsibility to download and install them on your own.
____ isn't an available database back-end.	Set your DATABASE_ENGINE setting to one of the valid engine settings described previously. Perhaps you made a typo?
Database ____ does not exist	Change the DATABASE_NAME setting to point to a database that exists, or execute the appropriate <code>CREATE DATABASE</code> statement in order to create it.
Role ____ does not exist	Change the DATABASE_USER setting to point to a user that exists, or create the user in your database.
Could not connect to server	Make sure DATABASE_HOST and DATABASE_PORT are set correctly, and make sure the database server is running.

Your First App

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

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

- A project is an instance of a certain set of Django apps, plus the configuration for those apps.
- Technically, the only requirement of a project is that it supplies a settings file, which defines the database connection information, the list of installed apps, the `TEMPLATE_DIRS`, and so forth.
- An app is a portable set of Django functionality, usually including models and views, that lives together in a single Python package.
- For example, Django comes with a number of apps, such as a commenting system and an automatic admin interface. A key thing to note about these apps is that they're portable and reusable across multiple projects.

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

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

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

Within the `mysite` project directory, type this command to create a `books` app:

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

This command does not produce any output, but it does create a `books` directory within the `mysite` directory. Let's look at the contents of that directory:

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

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

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

Defining Models in Python

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

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

- Introspection requires overhead and is imperfect. In order to provide convenient data-access APIs, Django needs to know the database layout *somehow*, and there are two ways of accomplishing this. The first way is to explicitly describe the data in Python, and the second way is to introspect the database at runtime to determine the data models.
- This second way seems cleaner, because the metadata about your tables lives in only one place, but it introduces a few problems. First, introspecting a database at runtime obviously requires overhead. If the framework had to introspect the database each time it processed a request, or even only when the Web server was initialized, this would incur an unacceptable level of overhead. (While some believe that level of overhead is acceptable, Django’s developers aim to trim as much framework overhead as possible.) Second, some databases, notably older versions of MySQL, do not store sufficient metadata for accurate and complete introspection.
- Writing Python is fun, and keeping everything in Python limits the number of times your brain has to do a “context switch.” It helps productivity if you keep yourself in a single programming environment/mentality for as long as possible. Having to write SQL, then Python, and then SQL again is disruptive.
- Having data models stored as code rather than in your database makes it easier to keep your models under version control. This way, you can easily keep track of changes to your data layouts.
- SQL allows for only a certain level of metadata about a data layout. Most database systems, for example, do not provide a specialized data type for representing e-mail addresses or URLs. Django models do. The advantage of higher-level data types is higher productivity and more reusable code.
- SQL is inconsistent across database platforms. If you’re distributing a Web application, for example, it’s much more pragmatic to distribute a Python module that describes your data layout than separate sets of `CREATE TABLE` statements for MySQL, PostgreSQL, and SQLite.

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

Finally, we should note that Django includes a utility that can generate models by introspecting an existing database. This is useful for quickly getting up and running with legacy data. We’ll cover this in Chapter 18.

Your First Model

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

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

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

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

```
from django.db import models

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

class Author(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=40)
    email = models.EmailField()

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

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

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

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

Indeed, Django can generate that `CREATE TABLE` statement automatically, as we'll show you in a moment.

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

For a full list of field types and model syntax options, see Appendix B.

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

Installing the Model

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

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

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

Temporarily comment out all four of those strings by putting a hash character (`#`) in front of them. (They're included by default as a common-case convenience, but we'll activate and discuss them in subsequent chapters.) While you're at it, comment out the default `MIDDLEWARE_CLASSES` setting, too; the default values in `MIDDLEWARE_CLASSES` depend on some of the apps we just commented out. Then, add `'mysite.books'` to the `INSTALLED_APPS` list, so the setting ends up looking like this:

```
MIDDLEWARE_CLASSES = (
    # 'django.middleware.common.CommonMiddleware',
    # 'django.contrib.sessions.middleware.SessionMiddleware',
    # 'django.contrib.auth.middleware.AuthenticationMiddleware',
)
```

```

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

```

As discussed in the last chapter, when you set `TEMPLATE_DIRS`, be sure to include the trailing comma in `INSTALLED_APPS` because it's a single-element tuple. By the way, this book's authors prefer to put a comma after *every* element of a tuple, regardless of whether the tuple has only a single element. This avoids the issue of forgetting commas, and there's no penalty for using that extra comma.

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

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

```
python manage.py validate
```

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

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

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

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

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

```

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

```

```

CREATE TABLE "books_author" (
    "id" serial NOT NULL PRIMARY KEY,
    "first_name" varchar(30) NOT NULL,
    "last_name" varchar(40) NOT NULL,
    "email" varchar(75) NOT NULL
)
;
CREATE TABLE "books_book" (
    "id" serial NOT NULL PRIMARY KEY,
    "title" varchar(100) NOT NULL,
    "publisher_id" integer NOT NULL REFERENCES "books_publisher" ("id")
        DEFERRABLE INITIALLY DEFERRED,
    "publication_date" date NOT NULL
)
;
CREATE TABLE "books_book_authors" (
    "id" serial NOT NULL PRIMARY KEY,
    "book_id" integer NOT NULL REFERENCES "books_book" ("id")
        DEFERRABLE INITIALLY DEFERRED,
    "author_id" integer NOT NULL REFERENCES "books_author" ("id")
        DEFERRABLE INITIALLY DEFERRED,
    UNIQUE ("book_id", "author_id")
)
;
CREATE INDEX "books_book_publisher_id" ON "books_book" ("publisher_id");
COMMIT;

```

Note the following:

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

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

```
python manage.py syncdb
```

Run that command and you'll see something like this:

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

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

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

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

Basic Data Access

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

```
>>> from books.models import Publisher
>>> p1 = Publisher(name='Apress', address='2855 Telegraph Avenue',
...     city='Berkeley', state_province='CA', country='U.S.A.',
...     website='http://www.apress.com/')
>>> p1.save()
>>> p2 = Publisher(name="O'Reilly", address='10 Fawcett St.',
...     city='Cambridge', state_province='MA', country='U.S.A.',
...     website='http://www.oreilly.com/')
>>> p2.save()
>>> publisher_list = Publisher.objects.all()
>>> publisher_list
[<Publisher: Publisher object>, <Publisher: Publisher object>]
```

These few lines of code accomplish quite a bit. Here are the highlights:

- First, import the `Publisher` model class. This lets you interact with the database table that contains publishers.
- Create a `Publisher` object by instantiating it with values for each field: `name`, `address`, and so on.
- To save the object to the database, call its `save()` method. Behind the scenes, Django executes an SQL `INSERT` statement here.
- To retrieve publishers from the database, use the attribute `Publisher.objects`, which you can think of as a set of all publishers. Fetch a list of *all* `Publisher` objects in the database with the statement `Publisher.objects.all()`. Behind the scenes, Django executes an SQL `SELECT` statement here.

One thing is worth mentioning, in case it wasn't clear from this example. When you create objects using the Django model API, Django doesn't save the objects to the database until you call the `save()` method:

```
p1 = Publisher(...)
# At this point, p1 is not saved to the database yet!
p1.save()
# Now it is.
```

If you want to create an object and save it to the database in a single step, use the `objects.create()` method. This example is equivalent to the preceding example:

```
>>> p1 = Publisher.objects.create(name='Apress',
...     address='2855 Telegraph Avenue',
...     city='Berkeley', state_province='CA', country='U.S.A.',
...     website='http://www.apress.com/')
>>> p2 = Publisher.objects.create(name="O'Reilly",
...     address='10 Fawcett St.', city='Cambridge',
...     state_province='MA', country='U.S.A.',
...     website='http://www.oreilly.com/')
>>> publisher_list = Publisher.objects.all()
>>> publisher_list
```

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

Adding Model String Representations

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

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

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

```
from django.db import models

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

    def __unicode__(self):
        return self.name

class Author(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=40)
    email = models.EmailField()

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

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

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

As you can see, a `__unicode__()` method can do whatever it needs to do in order to return a representation of an object. Here, the `__unicode__()` methods for `Publisher` and `Book` simply return the object’s name and title, respectively, but the `__unicode__()` for `Author` is slightly more complex: it pieces together the `first_name` and `last_name` fields, separated by a space. The only requirement for `__unicode__()` is that it return a Unicode object. If `__unicode__()` doesn’t return a Unicode object—if it returns, say, an integer—Python will raise a `TypeError` with a message such as “coercing to Unicode: need string or buffer, int found”.

UNICODE OBJECTS

What are Unicode objects?

You can think of a Unicode object as a Python string that can handle more than a million different types of characters, from accented versions of Latin characters, to non-Latin characters, to curly quotes and obscure symbols.

Normal Python strings are *encoded*, which means they use an encoding such as ASCII, ISO-8859-1, or UTF-8. If you're storing fancy characters (anything beyond the standard 128 ASCII characters such as 0–9 and A–Z) in a normal Python string, you have to keep track of which encoding your string is using, or else the fancy characters might appear messed up when they're displayed or printed. Problems occur when you have data that's stored in one encoding and you try to combine it with data in a different encoding, or when you try to display it in an application that assumes a certain encoding. We've all seen Web pages and e-mail that are littered with “??? ?????” or other characters in odd places; that generally suggests there's an encoding problem.

Unicode objects, however, have no encoding; they use a consistent, universal set of characters called, well, *Unicode*. When you deal with Unicode objects in Python, you can mix and match them safely without having to worry about encoding issues.

Django uses Unicode objects throughout the framework. Model objects are retrieved as Unicode objects, views interact with Unicode data, and templates are rendered as Unicode. You usually won't have to worry about making sure that your encodings are right; things should just work.

Note that this has been a *very* high-level, dumbed-down overview of Unicode objects, and you owe it to yourself to learn more about the topic. A good place to start is <http://www.joelonsoftware.com/articles/Unicode.html>.

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

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

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

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

Inserting and Updating Data

You've already seen this done: to insert a row into your database, first create an instance of your model using keyword arguments, like so:


```
>>> p = Publisher(name='Apress',
...               address='2855 Telegraph Ave.',
...               city='Berkeley',
...               state_province='CA',
...               country='U.S.A.',
...               website='http://www.apress.com/')
```

This act of instantiating a model class does *not* touch the database. The record isn't saved into the database until you call `save()`, like this:

```
>>> p.save()
```

In SQL, this can roughly be translated into the following:

```
INSERT INTO books_publisher
  (name, address, city, state_province, country, website)
VALUES
  ('Apress', '2855 Telegraph Ave.', 'Berkeley', 'CA',
   'U.S.A.', 'http://www.apress.com/');
```

Because the `Publisher` model uses an autoincrementing primary key `id`, the initial call to `save()` does one more thing: it calculates the primary key value for the record and sets it to the `id` attribute on the instance:

```
>>> p.id
52    # this will differ based on your own data
```

Subsequent calls to `save()` will save the record in place, without creating a new record (i.e., performing an SQL `UPDATE` statement instead of an `INSERT`):

```
>>> p.name = 'Apress Publishing'
>>> p.save()
```

The preceding `save()` statement will result in roughly the following SQL:

```
UPDATE books_publisher SET
  name = 'Apress Publishing',
  address = '2855 Telegraph Ave.',
  city = 'Berkeley',
  state_province = 'CA',
  country = 'U.S.A.',
  website = 'http://www.apress.com'
WHERE id = 52;
```

Note that *all* the fields will be updated, not just the ones that have been changed. Depending on your application, this may cause a race condition. See the section “Updating Multiple Objects in One Statement” to find out how to execute this (slightly different) query:

```
UPDATE books_publisher SET
  name = 'Apress Publishing'
WHERE id=52;
```

Selecting Objects

Knowing how to create and update database records is essential, but chances are that the Web applications you'll build will be doing more querying of existing objects than creating new ones. You've already seen a way to retrieve *every* record for a given model:

```
>>> Publisher.objects.all()
[<Publisher: Apress>, <Publisher: O'Reilly>]
```

This roughly translates to this SQL:

```
SELECT id, name, address, city, state_province, country, website
FROM books_publisher;
```

Note Django doesn't use `SELECT *` when looking up data and instead lists all fields explicitly. This is by design: in certain circumstances `SELECT *` can be slower, and (more important) listing fields more closely follows one tenet of the Zen of Python: "Explicit is better than implicit." For more on the Zen of Python, try typing `import this` at a Python prompt.

Let's take a close look at each part of this `Publisher.objects.all()` line:

- First, we have the model we defined, `Publisher`. No surprise here: when you want to look up data, you use the model for that data.
- Next, we have the `objects` attribute, which is called a *manager*. Managers are discussed in detail in Chapter 10. For now, all you need to know is that managers take care of all "table-level" operations on data including, most important, data lookup.
- All models automatically get an objects manager; you'll use it any time you want to look up model instances.
- Finally, we have `all()`. This is a method on the objects manager that returns all the rows in the database. Though this object *looks* like a list, it's actually a *QuerySet*—an object that represents a specific set of rows from the database. Appendix C deals with QuerySets in detail. For the rest of this chapter, we'll just treat them like the lists they emulate.

Any database lookup is going to follow this general pattern—we'll call methods on the manager attached to the model we want to query against.

Filtering Data

Naturally, it's rare to want to select *everything* from a database at once; in most cases, you'll want to deal with a subset of your data. In the Django API, you can filter your data using the `filter()` method:

```
>>> Publisher.objects.filter(name='Apress')
[<Publisher: Apress>]
```

`filter()` takes keyword arguments that get translated into the appropriate SQL WHERE clauses. The preceding example would get translated into something like this:

```
SELECT id, name, address, city, state_province, country, website
FROM books_publisher
WHERE name = 'Apress';
```

You can pass multiple arguments into `filter()` to narrow down things further:

```
>>> Publisher.objects.filter(country="U.S.A.", state_province="CA")
[<Publisher: Apress>]
```

Those multiple arguments get translated into SQL AND clauses. Thus, the example in the code snippet translates into the following:

```
SELECT id, name, address, city, state_province, country, website
FROM books_publisher
WHERE country = 'U.S.A.'
AND state_province = 'CA';
```

Notice that by default the lookups use the SQL `=` operator to do exact match lookups. Other lookup types are available:

```
>>> Publisher.objects.filter(name__contains="press")
[<Publisher: Apress>]
```

That's a *double* underscore there between `name` and `contains`. Like Python itself, Django uses the double underscore to signal that something “magic” is happening—here, the `__contains` part gets translated by Django into an SQL LIKE statement:

```
SELECT id, name, address, city, state_province, country, website
FROM books_publisher
WHERE name LIKE '%press%';
```

Many other types of lookups are available, including `icontains` (case-insensitive LIKE), `startswith` and `endswith`, and `range` (SQL BETWEEN queries). Appendix C describes all of these lookup types in detail.

Retrieving Single Objects

The previous `filter()` examples all returned a `QuerySet`, which you can treat like a list. Sometimes it's more convenient to fetch only a single object instead of a list. That's what the `get()` method is for:

```
>>> Publisher.objects.get(name="Apress")
<Publisher: Apress>
```

Instead of a list (rather, `QuerySet`), only a single object is returned. Because of that, a query resulting in multiple objects will cause an exception:

```
>>> Publisher.objects.get(country="U.S.A.")
Traceback (most recent call last):
...
MultipleObjectsReturned: get() returned more than one Publisher --
it returned 2! Lookup parameters were {'country': 'U.S.A.'}
```

A query that returns no objects also causes an exception:

```
>>> Publisher.objects.get(name="Penguin")
Traceback (most recent call last):
...
DoesNotExist: Publisher matching query does not exist.
```

The `DoesNotExist` exception is an attribute of the model's class: `Publisher.DoesNotExist`. In your applications, you'll want to trap these exceptions, like this:

```
try:
    p = Publisher.objects.get(name='Apress')
except Publisher.DoesNotExist:
    print "Apress isn't in the database yet."
else:
    print "Apress is in the database."
```

Ordering Data

As you play around with the previous examples, you might discover that the objects are being returned in a seemingly random order. You aren't imagining things; so far we haven't told the database how to order its results, so we're simply getting back data in some arbitrary order chosen by the database.

In your Django applications, you'll probably want to order your results according to a certain value—say, alphabetically. To do this, use the `order_by()` method:

```
>>> Publisher.objects.order_by("name")
[<Publisher: Apress>, <Publisher: O'Reilly>]
```

This doesn't look much different from the earlier `all()` example, but the SQL now includes a specific ordering:

```
SELECT id, name, address, city, state_province, country, website
FROM books_publisher
ORDER BY name;
```

You can order by any field you like:

```
>>> Publisher.objects.order_by("address")
[<Publisher: O'Reilly>, <Publisher: Apress>]

>>> Publisher.objects.order_by("state_province")
[<Publisher: Apress>, <Publisher: O'Reilly>]
```

To order by multiple fields (where the second field is used to disambiguate ordering in cases where the first is the same), use multiple arguments:

```
>>> Publisher.objects.order_by("state_province", "address")
[<Publisher: Apress>, <Publisher: O'Reilly>]
```

You can also specify reverse ordering by prefixing the field name with a - (that's a minus character):

```
>>> Publisher.objects.order_by("-name")
[<Publisher: O'Reilly>, <Publisher: Apress>]
```

While this flexibility is useful, using `order_by()` all the time can be quite repetitive. Most of the time you'll have a particular field you usually want to order by. In these cases, Django lets you specify a default ordering in the model:

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

    def __unicode__(self):
        return self.name

    class Meta:
        ordering = ['name']
```

Here, we've introduced a new concept: the class `Meta`, which is a class that's embedded within the `Publisher` class definition (it's indented to be within `class Publisher`). You can use this `Meta` class on any model to specify various model-specific options. A full reference of `Meta` options is available in Appendix B, but for now, we're concerned with the `ordering` option. If you specify this, it tells Django that unless an ordering is given explicitly with `order_by()`, all `Publisher` objects should be ordered by the `name` field whenever they're retrieved with the Django database API.

Chaining Lookups

You've seen how you can filter data, and you've seen how you can order it. You'll often need to do both, of course. In these cases, you simply "chain" the lookups together:

```
>>> Publisher.objects.filter(country="U.S.A.").order_by("-name")
[<Publisher: O'Reilly>, <Publisher: Apress>]
```

As you might expect, this translates to an SQL query with both a `WHERE` and an `ORDER BY`:

```
SELECT id, name, address, city, state_province, country, website
FROM books_publisher
WHERE country = 'U.S.A.'
ORDER BY name DESC;
```

Slicing Data

Another common need is to look up only a fixed number of rows. Imagine that you have thousands of publishers in your database, but you want to display only the first one. You can do this using Python's standard list-slicing syntax:

```
>>> Publisher.objects.order_by('name')[0]
<Publisher: Apress>
```

This translates roughly to:

```
SELECT id, name, address, city, state_province, country, website
FROM books_publisher
ORDER BY name
LIMIT 1;
```

Similarly, you can retrieve a specific subset of data using Python's range-slicing syntax:

```
>>> Publisher.objects.order_by('name')[0:2]
```

This returns two objects, translating roughly to the following:

```
SELECT id, name, address, city, state_province, country, website
FROM books_publisher
ORDER BY name
OFFSET 0 LIMIT 2;
```

Note that negative slicing is *not* supported:

```
>>> Publisher.objects.order_by('name')[-1]
Traceback (most recent call last):
...
AssertionError: Negative indexing is not supported.
```

This is easy to get around, though. Just change the `order_by()` statement like this:

```
>>> Publisher.objects.order_by('-name')[0]
```

Updating Multiple Objects in One Statement

We pointed out in the “Inserting and Updating Data” section that the model `save()` method updates *all* columns in a row. Depending on your application, you might want to update only a subset of columns.

For example, suppose that you want to update the Apress Publisher to change the name from 'Apress' to 'Apress Publishing'. Using `save()`, it would look something like this:

```
>>> p = Publisher.objects.get(name='Apress')
>>> p.name = 'Apress Publishing'
>>> p.save()
```

This roughly translates to the following SQL:

```
SELECT id, name, address, city, state_province, country, website
FROM books_publisher
WHERE name = 'Apress';
```

```
UPDATE books_publisher SET
    name = 'Apress Publishing',
    address = '2855 Telegraph Ave.',
    city = 'Berkeley',
    state_province = 'CA',
    country = 'U.S.A.',
    website = 'http://www.apress.com'
WHERE id = 52;
```

Note This example assumes that Apress has a publisher ID of 52.

You can see in this example that Django's `save()` method sets *all* the column values, not just the name column. If you're in an environment in which other columns of the database might change because of some other process, it's smarter to change *only* the column you need to change. To do this, use the `update()` method on `QuerySet` objects. Here's an example:

```
>>> Publisher.objects.filter(id=52).update(name='Apress Publishing')
```

The SQL translation here is much more efficient and has no chance of race conditions:

```
UPDATE books_publisher
SET name = 'Apress Publishing'
WHERE id = 52;
```

The `update()` method works on any `QuerySet`, which means that you can edit multiple records in bulk. Here's how you might change the country from 'U.S.A.' to USA in each `Publisher` record:

```
>>> Publisher.objects.all().update(country='USA')
2
```

The `update()` method has a return value: an integer that represents how many records changed. In the preceding example, it was 2.

Deleting Objects

To delete an object from your database, simply call the object's `delete()` method:

```
>>> p = Publisher.objects.get(name="O'Reilly")
>>> p.delete()
>>> Publisher.objects.all()
[<Publisher: Apress Publishing>]
```

You can also delete objects in bulk by calling `delete()` on the result of any `QuerySet`. This is similar to the `update()` method shown in the last section:

```
>>> Publisher.objects.filter(country='USA').delete()
>>> Publisher.objects.all().delete()
>>> Publisher.objects.all()
[]
```

Be careful when deleting your data! As a precaution against deleting all the data in a particular table, Django requires you to explicitly use `all()` if you want to delete *everything* in your table.

For example, this doesn't work:

```
>>> Publisher.objects.delete()
Traceback (most recent call last):
  File "<console>", line 1, in <module>
AttributeError: 'Manager' object has no attribute 'delete'
```

But it does work if you add the `all()` method:

```
>>> Publisher.objects.all().delete()
```

If you're just deleting a subset of your data, you don't need to include `all()`. To repeat a previous example:

```
>>> Publisher.objects.filter(country='USA').delete()
```

What's Next?

After reading this chapter, you now have enough knowledge of Django models to be able to write basic database applications. Chapter 10 will provide some information on more advanced usage of Django's database layer.

Once you've defined your models, the next step is to populate your database with data. You might have legacy data, in which case Chapter 18 will give you advice about integrating with legacy databases. You might rely on site users to supply your data, in which case Chapter 7 will teach you how to process user-submitted form data.

But in some cases, you or your team might need to enter data manually, in which case it would be helpful to have a Web-based interface for entering and managing data. The next chapter covers Django's admin interface, which exists precisely for that reason.



The Django Admin Site

For a certain class of Web sites, an *admin interface* is an essential part of the infrastructure. This is a Web-based interface, limited to trusted site administrators, that enables the adding, editing, and deletion of site content. Some common examples are: the interface you use to post to your blog, the back-end site managers use to moderate user-generated comments, the tool your clients use to update the press releases on the Web site you built for them.

There's a problem with admin interfaces, though: it's boring to build them. Web development is fun when you're developing public-facing functionality, but building admin interfaces is always the same. You have to authenticate users, display and handle forms, validate input, and so on. It's boring and it's repetitive.

So what's Django's approach to these boring, repetitive tasks? It does it all for you—in just a couple of lines of code, no less. With Django, building an admin interface is a solved problem.

This chapter is about Django's automatic admin interface. The feature works by reading metadata in your model to provide a powerful and production-ready interface that site administrators can start using immediately. We discuss how to activate, use, and customize this feature.

Note that we recommend reading this chapter even if you don't intend to use the Django admin site, because we introduce a few concepts that apply to all of Django, regardless of admin-site usage.

The `django.contrib` Packages

Django's automatic admin is part of a larger suite of Django functionality called `django.contrib`—the part of the Django codebase that contains various useful add-ons to the core framework. You can think of `django.contrib` as Django's equivalent of the Python standard library—optional, de facto implementations of common patterns. They're bundled with Django so that you don't have to reinvent the wheel in your own applications.

The admin site is the first part of `django.contrib` that we're covering in this book; technically, it's called `django.contrib.admin`. Other available features in `django.contrib` include a user-authentication system (`django.contrib.auth`), support for anonymous sessions (`django.contrib.sessions`), and even a system for user comments (`django.contrib.comments`). You'll get

to know the various `django.contrib` features as you become a Django expert, and we'll spend some more time discussing them in Chapter 16. For now, just know that Django ships with many nice add-ons, and `django.contrib` is generally where they live.

Activating the Admin Interface

The Django admin site is entirely optional, because only certain types of sites need this functionality. That means you'll need to take a few steps to activate it in your project.

First, make a few changes to your settings file:

1. Add `'django.contrib.admin'` to the `INSTALLED_APPS` setting. (The order of `INSTALLED_APPS` doesn't matter, but we like to keep things alphabetical so it's easy for a human to read.)
2. Make sure `INSTALLED_APPS` contains `'django.contrib.auth'`, `'django.contrib.contenttypes'`, and `'django.contrib.sessions'`. The Django admin site requires these three packages. (If you're following along with our ongoing `mysite` project, note that we commented out these three `INSTALLED_APPS` entries in Chapter 5. Uncomment them now.)
3. Make sure `MIDDLEWARE_CLASSES` contains `'django.middleware.common.CommonMiddleware'`, `'django.contrib.sessions.middleware.SessionMiddleware'`, and `'django.contrib.auth.middleware.AuthenticationMiddleware'`. (Again, if you're following along, note that we commented them out in Chapter 5, so uncomment them.)

Second, run `python manage.py syncdb`. This step will install the extra database tables that the admin interface uses. The first time you run `syncdb` with `'django.contrib.auth'` in `INSTALLED_APPS`, you'll be asked about creating a superuser. If you don't do this, you'll need to run `python manage.py createsuperuser` separately to create an admin user account; otherwise you won't be able to log in to the admin site. (Potential gotcha: the `python manage.py createsuperuser` command is available only if `'django.contrib.auth'` is in your `INSTALLED_APPS`.)

Third, add the admin site to your `URLconf` (in `urls.py`, remember). By default, the `urls.py` generated by `django-admin.py startproject` contains commented-out code for the Django admin, and all you have to do is uncomment it. For the record, here are the bits you need to make sure are in there:

```
# Include these import statements...
from django.contrib import admin
admin.autodiscover()

# And include this URLpattern...
urlpatterns = patterns('',
    # ...
    (r'^admin/', include(admin.site.urls)),
    # ...
)
```

With that bit of configuration out of the way, now you can see the Django admin site in action. Just run the development server (`python manage.py runserver`, as in previous chapters) and visit `http://127.0.0.1:8000/admin/` in your Web browser.

Using the Admin Site

The admin site is designed to be used by nontechnical users, and as such it should be pretty self-explanatory. Nevertheless, we'll give you a quick walkthrough of the basic features.

The first thing you'll see is a login screen, as shown in Figure 6-1.

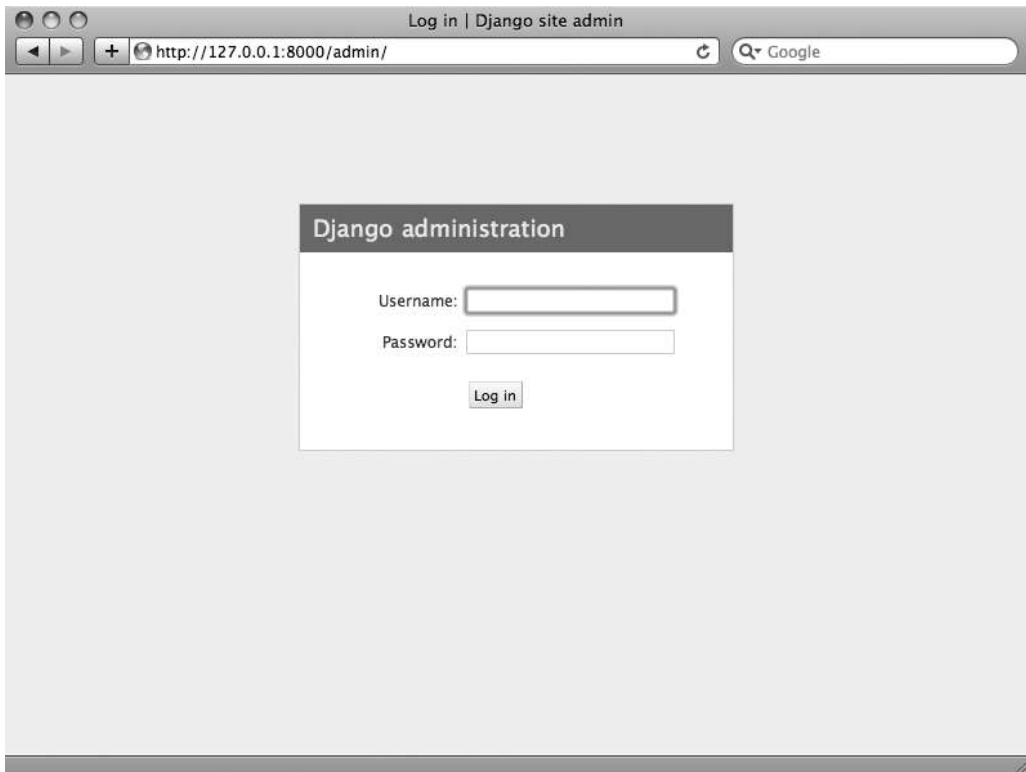


Figure 6-1. *Django's login screen*

Log in with the username and password you set up when you added your superuser. If you're unable to log in, make sure you've actually created a superuser—try running `python manage.py createsuperuser`.

Once you're logged in, the first thing you'll see will be the admin home page (Figure 6-2). This page lists all the available types of data that can be edited on the admin site. At this point, because we haven't activated any of our own models yet, the list is sparse: it includes only Groups and Users, which are the two default admin-editable models.

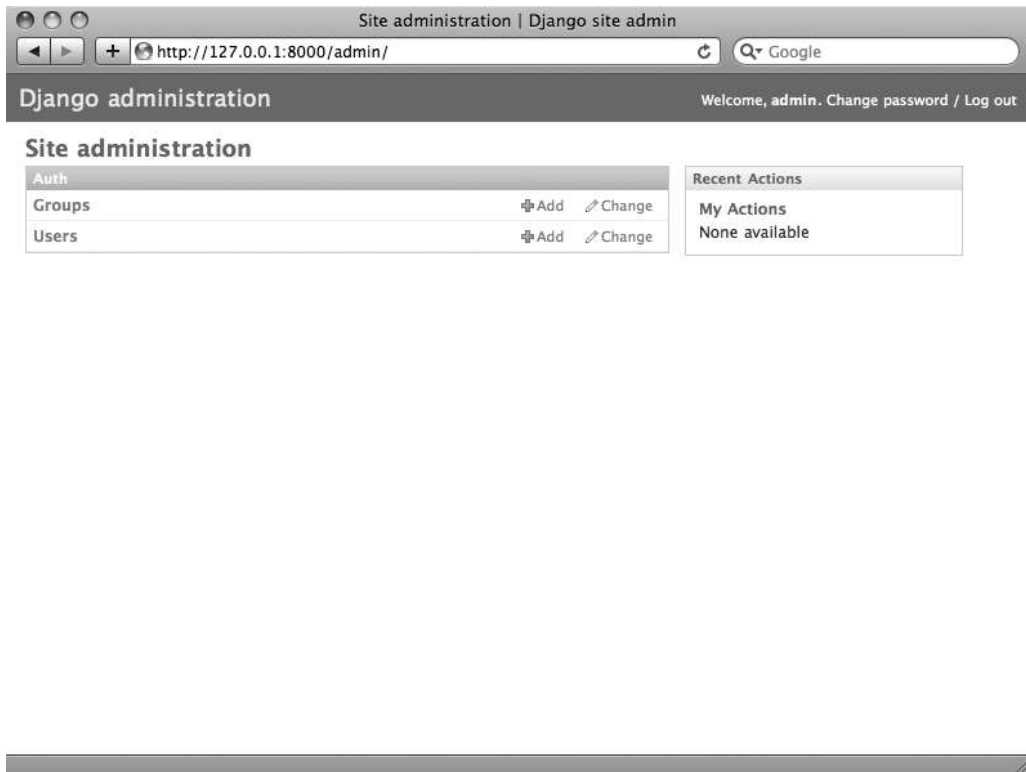


Figure 6-2. *The Django admin home page*

Each type of data in the Django admin site has a *change list* and an *edit form*. Change lists show you all the available objects in the database, and edit forms let you add, change, or delete particular records in your database.

OTHER LANGUAGES

If your primary language is not English and your Web browser is configured to prefer a language other than English, you can make a quick change to see whether the Django admin site has been translated into your language. Just add `'django.middleware.locale.LocaleMiddleware'` to your `MIDDLEWARE_CLASSES` setting, making sure it appears *after* `'django.contrib.sessions.middleware.SessionMiddleware'`.

When you've done that, reload the admin index page. If a translation for your language is available, then the various parts of the interface—from the Change Password and Log Out links at the top of the page to the Groups and Users links in the middle—will appear in your language instead of English. Django ships with translations for dozens of languages.

For much more on Django's internationalization features, see Chapter 19.

Click the Change link in the Users row to load the change-list page for users (Figure 6-3).

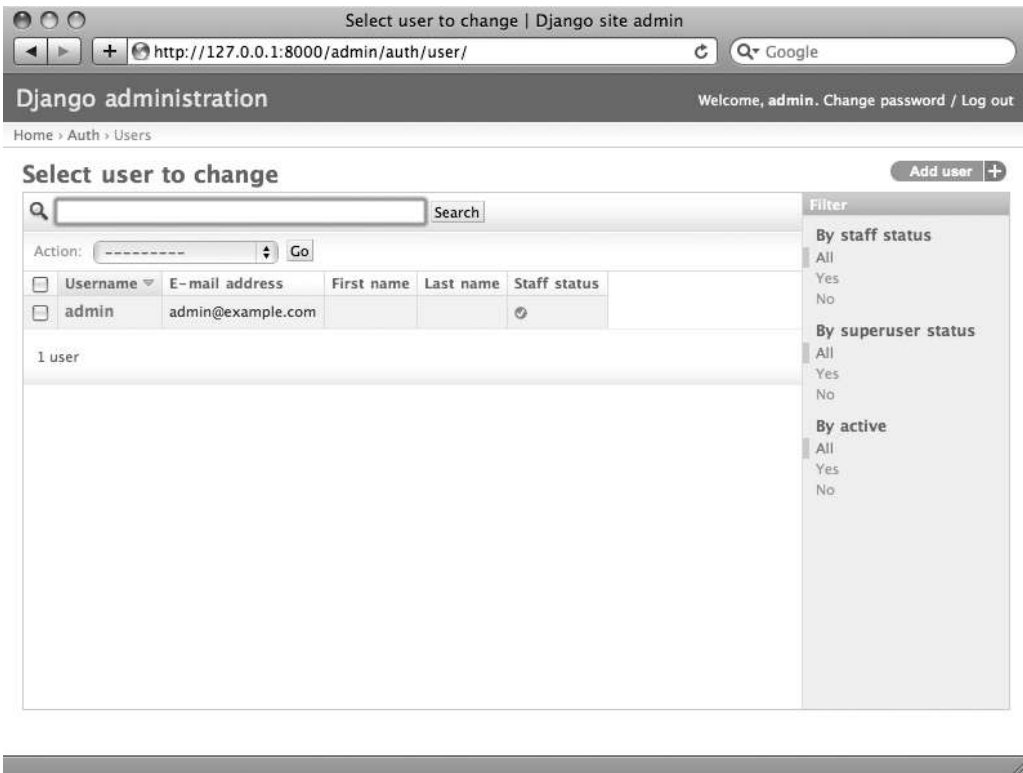


Figure 6-3. *The user change-list page*

This page displays all users in the database; you can think of it as a prettied-up Web version of a `SELECT * FROM auth_user;` SQL query. If you're following along with our ongoing example, you'll see only one user here, assuming you've added only one, but once you have more users, you'll probably find the filtering, sorting, and searching options useful. Filtering options are at the right, sorting is available by clicking a column header, and the search box at the top lets you search by username.

Click the username of the user you created, and you'll see the edit form for that user (Figure 6-4).

The screenshot shows the Django administration interface for editing a user. The browser address bar shows the URL `http://127.0.0.1:8000/admin/auth/user/1/`. The page title is "Change user | Django site admin". The Django administration header includes a welcome message for "admin" and links for "Change password" and "Log out". The breadcrumb trail is "Home > Auth > Users > admin".

The main form is titled "Change user" and includes a "History" link and a "View on site" link. The form is divided into several sections:

- Username:** A text input field containing "admin". Below it, a note states: "Required. 30 characters or fewer. Alphanumeric characters only (letters, digits and underscores)."

Username:	admin
-----------	-------
- Password:** A text input field containing a SHA-256 hash: `sha1578e575ba08922431ebd1bcc5a34e`. Below it, a note states: "Use '[algo]\${salt}\${hexdigest}' or use the change password form."

Password:	sha1578e575ba08922431ebd1bcc5a34e
-----------	-----------------------------------
- Personal info:** A section with three text input fields:
 - First name: (empty)
 - Last name: (empty)
 - E-mail address: `admin@example.com`
- Permissions:** A section with three checkboxes, all of which are checked:
 - ☒ Staff status: Designates whether the user can log into this admin site.
 - ☒ Active: Designates whether this user should be treated as active. Unselect this instead of deleting accounts.
 - ☒ Superuser status

Figure 6-4. *The user edit form*

This page lets you change the attributes of the user, like the first/last names and various permissions. (Note that to change a user's password, you should click Change Password Form under the password field rather than editing the hashed code.) Another thing to note here is that fields of different types get different widgets—for example, date/time fields have calendar controls, Boolean fields have check boxes, and character fields have simple text input fields.

You can delete a record by clicking the Delete button at the bottom left of the record's edit form. That'll take you to a confirmation page, which, in some cases, will display any dependent objects that will be deleted, too. (For example, if you delete a publisher, any book with that publisher will be deleted, as well!)

You can add a record by clicking Add in the appropriate column of the admin home page. This will give you an empty version of the edit page, ready for you to fill out.

You'll notice that the admin interface handles input validation for you. Try leaving a required field blank or putting an invalid date into a date field, and you'll see those errors when you try to save, as shown in Figure 6-5.

When you edit an existing object, you'll notice a History link in the upper-right corner of the window. Every change made through the admin interface is logged, and you can examine this log by clicking the History link (see Figure 6-6).

The screenshot shows the 'Change user' form in the Django admin interface. The browser address bar indicates the URL is `http://127.0.0.1:8000/admin/auth/user/1/`. The page title is 'Change user | Django site admin'. The Django administration header shows 'Welcome, admin. Change password / Log out' and a breadcrumb trail 'Home > Auth > Users > admin'. The form title is 'Change user' with links for 'History' and 'View on site'. A message at the top says 'Please correct the error below.' Below this, a red error box states 'This field is required.' The 'Username' field is empty, with a note: 'Required. 30 characters or fewer. Alphanumeric characters only (letters, digits and underscores).' The 'Password' field contains a SHA-1 hash: `sha1$78e57$ba08922431ebd1bcc5a34f`, with a note: 'Use [algo]\${salt}\${hexdigest} or use the change password form.' The 'Personal info' section has fields for 'First name', 'Last name', and 'E-mail address' (which is filled with `admin@example.com`). The 'Permissions' section has two checked checkboxes: 'Staff status' (with a note: 'Designates whether the user can log into this admin site.') and 'Active'.

Figure 6-5. An edit form displaying errors

The screenshot shows the 'Change history' page for the user 'admin' in the Django admin interface. The browser address bar indicates the URL is `http://127.0.0.1:8000/admin/auth/user/1/history/`. The page title is 'Change history: admin | Django site admin'. The Django administration header shows 'Welcome, admin. Change password / Log out' and a breadcrumb trail 'Home > Auth > Users > admin > History'. The form title is 'Change history: admin'. Below the title is a table showing the history of changes.

Date/time	User	Action
June 9, 2009, 11:23 a.m.	admin	Changed email.

Figure 6-6. An object history page

Adding Your Models to the Admin Site

There's one crucial part we haven't done yet. Let's add our own models to the admin site so we can add, change, and delete objects in our custom database tables using this nice interface. We'll continue the books example from Chapter 5, where we defined three models: `Publisher`, `Author`, and `Book`.

Within the books directory (`mysite/books`), create a file called `admin.py`, and type in the following lines of code:

```
from django.contrib import admin
from mysite.books.models import Publisher, Author, Book

admin.site.register(Publisher)
admin.site.register(Author)
admin.site.register(Book)
```

This code tells the Django admin site to offer an interface for each of these models.

Once you've done that, go to your admin home page in your Web browser (<http://127.0.0.1:8000/admin/>). You should see a Books section with links for Authors, Books, and Publishers. (You might have to stop and start the runserver for the changes to take effect.)

You now have a fully functional admin interface for each of those three models. That was easy!

Take some time to add and change records, to populate your database with some data. If you followed Chapter 5's examples of creating `Publisher` objects (and you didn't delete them), you'll already see those records on the publisher change-list page.

One feature worth mentioning here is the admin site's handling of foreign keys and many-to-many relationships, both of which appear in the `Book` model. As a reminder, here's what the `Book` model looks like:

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

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

On the Django admin site's Add Book page (<http://127.0.0.1:8000/admin/books/book/add/>), the publisher (a `ForeignKey`) is represented by a select box, and the authors field (a `ManyToManyField`) is represented by a multiple-select box. Both fields sit next to a green plus sign that lets you add related records of that type. For example, if you click the green plus sign next to the Publisher field, you'll get a pop-up window that lets you add a publisher. After you successfully create the publisher in the pop-up, the Add Book form will be updated with the newly created publisher. Slick.

How the Admin Site Works

Behind the scenes, how does the admin site work? It's pretty straightforward.

When Django loads your `URLconf` from `urls.py` at server startup, it executes the `admin.autodiscover()` statement that we added as part of activating the admin. This function iterates over your `INSTALLED_APPS` setting and looks for a file called `admin.py` in each installed app. If an `admin.py` exists in a given app, it executes the code in that file.

In the `admin.py` in our `books` app, each call to `admin.site.register()` simply registers the given model with the admin. The admin site will display an edit/change interface for only models that have been explicitly registered.

The app `django.contrib.auth` includes its own `admin.py`, which is why `Users` and `Groups` showed up automatically in the admin. Other `django.contrib` apps, such as `django.contrib.redirects`, also add themselves to the admin, as do many third-party Django applications you might download from the Web.

Beyond that, the Django admin site is just a Django application, with its own models, templates, views, and `URLpatterns`. You add it to your application by hooking it into your `URLconf`, just as you hook in your own views. You can inspect its templates, views, and `URLpatterns` by poking around in `django/contrib/admin` in your copy of the Django codebase—but don't be tempted to change anything directly in there, as there are plenty of hooks for you to customize the way the admin site works. (If you do decide to poke around the Django admin application, keep in mind it does some rather complicated things in reading metadata about models, so it would probably take a good amount of time to read and understand the code.)

Making Fields Optional

After you play around with the admin site for a while, you'll probably notice a limitation—the edit forms require every field to be filled out, whereas in many cases you'd want certain fields to be optional. Let's say, for example, that we want our `Author` model's `email` field to be optional—that is, a blank string should be allowed. In the real world, you might not have an e-mail address on file for every author.

To specify that the `email` field is optional, edit the `Book` model (which, as you'll recall from Chapter 5, lives in `mysite/books/models.py`). Simply add `blank=True` to the `email` field, like so:

```
class Author(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=40)
    email = models.EmailField(blank=True)
```

This tells Django that a blank value is indeed allowed for authors' e-mail addresses. By default, all fields have `blank=False`, which means blank values are not allowed.

There's something interesting happening here. Until now, with the exception of the `__unicode__()` method, our models have served as definitions of our database tables—Pythonic expressions of `SQL CREATE TABLE` statements, essentially. In adding `blank=True`, we have begun expanding our model beyond a simple definition of what the database table looks like. Now our model class is starting to become a richer collection of knowledge about what `Author` objects are and what they can do. Not only is the `email` field represented by a `VARCHAR` column in the database, it's also an optional field in contexts such as the Django admin site.

Once you’ve added that `blank=True`, reload the author edit form (<http://127.0.0.1:8000/admin/books/author/add/>), and you’ll notice the field’s label—Email—is no longer bolded. This signifies it’s not a required field. You can now add authors without needing to provide e-mail addresses; you won’t get the loud red “This field is required” message anymore if the field is submitted empty.

Making Date and Numeric Fields Optional

A common gotcha related to `blank=True` has to do with date and numeric fields, but it requires a fair amount of background explanation.

SQL has its own way of specifying blank values—a special value called `NULL`. `NULL` could mean “unknown,” or “invalid,” or some other application-specific meaning. In SQL, a value of `NULL` is different from an empty string, just as the special Python object `None` is different from an empty Python string (`""`). This means it’s possible for a particular character field (e.g., a `VARCHAR` column) to contain both `NULL` values and empty string values.

This can cause unwanted ambiguity and confusion: “Why does this record have a `NULL` but this other one has an empty string? Is there a difference, or was the data just entered inconsistently?” And “How do I get all the records that have a blank value—should I look for both `NULL` records and empty strings, or do I select only the ones with empty strings?”

To help avoid such ambiguity, Django’s automatically generated `CREATE TABLE` statements (which were covered in Chapter 5) add an explicit `NOT NULL` to each column definition. For example, here’s the generated statement for our `Author` model, from Chapter 5:

```
CREATE TABLE "books_author" (
    "id" serial NOT NULL PRIMARY KEY,
    "first_name" varchar(30) NOT NULL,
    "last_name" varchar(40) NOT NULL,
    "email" varchar(75) NOT NULL
);
```

In most cases, this default behavior is optimal for your application and will save you from data-inconsistency headaches. And it works nicely with the rest of Django, such as the Django admin site, which inserts an empty string (*not* a `NULL` value) when you leave a character field blank.

But there’s an exception with database column types that do not accept empty strings as valid values—such as dates, times, and numbers. If you try to insert an empty string into a date or an integer column, you’ll likely get a database error, depending on which database you’re using. (PostgreSQL, which is strict, will raise an exception here; MySQL might accept it or might not, depending on the version you’re using, the time of day, and the phase of the moon.) In this case, `NULL` is the only way to specify an empty value. In Django models, you can specify that `NULL` is allowed by adding `null=True` to a field.

In short, if you want to allow blank values in a date field (e.g., `DateField`, `TimeField`, `DateTimeField`) or numeric field (e.g., `IntegerField`, `DecimalField`, `FloatField`), you’ll need to use both `null=True` *and* `blank=True`.

For the sake of example, let's change our Book model to allow a blank `publication_date`. Here's the revised code:

```
class Book(models.Model):
    title = models.CharField(max_length=100)
    authors = models.ManyToManyField(Author)
    publisher = models.ForeignKey(Publisher)
    publication_date = models.DateField(blank=True, null=True)
```

Adding `null=True` is more complicated than adding `blank=True`, because `null=True` changes the semantics of the database—that is, it changes the `CREATE TABLE` statement to remove the `NOT NULL` from the `publication_date` field. To complete this change, we'll need to update the database.

For a number of reasons, Django does not attempt to automate changes to database schemas, so it's your own responsibility to execute the appropriate `ALTER TABLE` statement whenever you make such a change to a model. Recall that you can use `manage.py dbshell` to enter your database server's shell. Here's how to remove the `NOT NULL` in this particular case:

```
ALTER TABLE books_book ALTER COLUMN publication_date DROP NOT NULL;
```

(Note that this SQL syntax is specific to PostgreSQL.) We'll cover schema changes in more depth in Chapter 10.

Bringing this back to the admin site, now the Add Book edit form should allow for empty publication-date values.

Customizing Field Labels

On the admin site's edit forms, each field's label is generated from its model field name. The algorithm is simple: Django just replaces underscores with spaces and capitalizes the first character, so, for example, the Book model's `publication_date` field has the label `Publication Date`.

However, field names don't always lend themselves to nice admin field labels, so in some cases you might want to customize a label. You can do this by specifying `verbose_name` in the appropriate model field.

For example, here's how we can change the label of the `Author.email` field to “e-mail,” with a hyphen:

```
class Author(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=40)
    email = models.EmailField(blank=True, verbose_name='e-mail')
```

Make that change and reload the server, and you should see the field's new label on the author edit form.

Note that you shouldn't capitalize the first letter of a `verbose_name` unless it should *always* be capitalized (e.g., “USA state”). Django will automatically capitalize it when it needs to, and it will use the exact `verbose_name` value in places that don't require capitalization.

Finally, note that you can pass the `verbose_name` as a positional argument, for a slightly more compact syntax. This example is equivalent to the previous one:

```
class Author(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=40)
    email = models.EmailField('e-mail', blank=True)
```

This won't work with `ManyToManyField` or `ForeignKey` fields, though, because they require the first argument to be a model class. In those cases, specifying `verbose_name` explicitly is the way to go.

Custom ModelAdmin Classes

The changes we've made so far—`blank=True`, `null=True`, and `verbose_name`—are really model-level changes, not admin-level changes. That is, these changes are fundamentally a part of the model and just so happen to be used by the admin site; there's nothing admin-specific about them.

Beyond these, the Django admin site offers a wealth of options that let you customize how the admin site works for a particular model. Such options live in `ModelAdmin` classes, which are classes that contain configuration for a specific model in a specific admin site instance.

Customizing Change Lists

Let's dive into admin customization by specifying the fields that are displayed on the change list for our `Author` model. By default, the change list displays the result of `__unicode__()` for each object. In Chapter 5 we defined the `__unicode__()` method for `Author` objects to display the first name and last name together:

```
class Author(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=40)
    email = models.EmailField(blank=True, verbose_name='e-mail')

    def __unicode__(self):
        return u'%s %s' % (self.first_name, self.last_name)
```

As a result, the change list for `Author` objects displays each author's first name and last name together, as you can see in Figure 6-7.

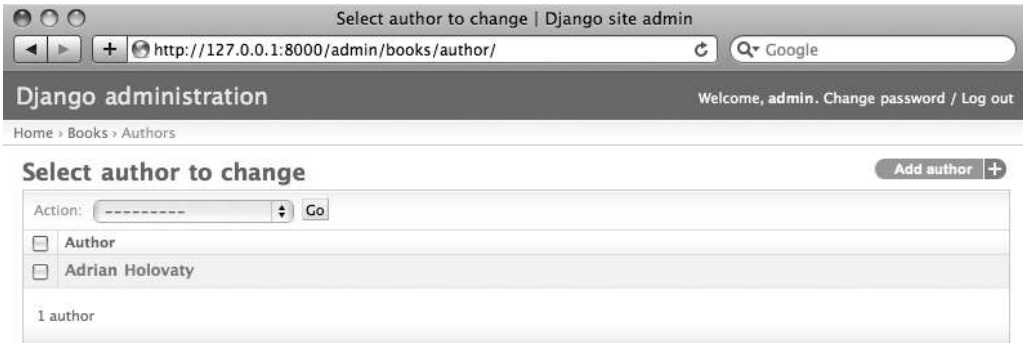


Figure 6-7. *The author change-list page*

We can improve on this default behavior by adding a few other fields to the change-list display. It'd be handy, for example, to see each author's e-mail address in this list, and it'd be nice to be able to sort by first and last name.

To make this happen, we'll define a `ModelAdmin` class for the `Author` model. This class is the key to customizing the admin, and one of the most basic things it lets you do is specify the list of fields to display on change-list pages. Edit `admin.py` to make these changes:

```
from django.contrib import admin
from mysite.books.models import Publisher, Author, Book

class AuthorAdmin(admin.ModelAdmin):
    list_display = ('first_name', 'last_name', 'email')

admin.site.register(Publisher)
admin.site.register(Author, AuthorAdmin)
admin.site.register(Book)
```

Here's what we've done:

- We created the class `AuthorAdmin`. This class, which subclasses `django.contrib.admin.ModelAdmin`, holds custom configuration for a specific admin model. We've specified only one customization—`list_display`, which is set to a tuple of field names to display on the change-list page. These field names must exist in the model, of course.
- We altered the `admin.site.register()` call to add `AuthorAdmin` after `Author`. You can read this as “Register the `Author` model with the `AuthorAdmin` options.”

The `admin.site.register()` function takes a `ModelAdmin` subclass as an optional second argument. If you don't specify a second argument (as is the case for `Publisher` and `Book`), Django will use the default admin options for that model.

With that tweak made, reload the author change-list page, and you'll see it's now displaying three columns—the first name, last name, and e-mail address. In addition, each of those columns is sortable by clicking on the column header. (See Figure 6-8.)

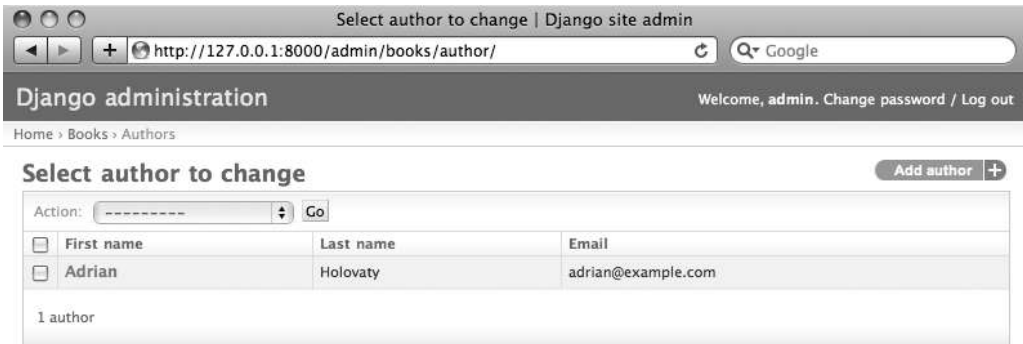


Figure 6-8. The author change-list page after `list_display`

Next let's add a simple search bar. Add `search_fields` to `AuthorAdmin`, like so:

```
class AuthorAdmin(admin.ModelAdmin):
    list_display = ('first_name', 'last_name', 'email')
    search_fields = ('first_name', 'last_name')
```

Reload the page in your browser, and you should see a search bar at the top. (See Figure 6-9.) We've just told the admin change-list page to include a search bar that searches against the `first_name` and `last_name` fields. As a user might expect, this is case insensitive and searches both fields, so searching for the string "bar" would find both an author with the first name Barney and an author with the last name Hobarson.

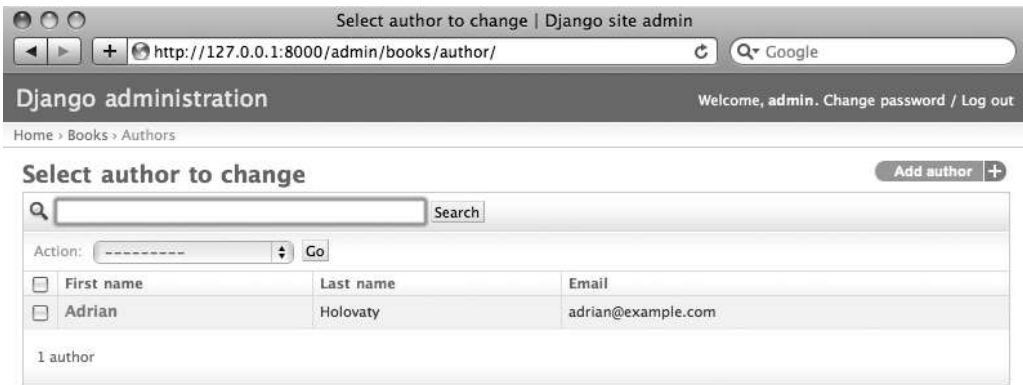


Figure 6-9. *The author change-list page after `search_fields`*

Next let's add some date filters to our `Book` model's change-list page:

```
from django.contrib import admin
from mysite.books.models import Publisher, Author, Book

class AuthorAdmin(admin.ModelAdmin):
    list_display = ('first_name', 'last_name', 'email')
    search_fields = ('first_name', 'last_name')
```

```

class BookAdmin(admin.ModelAdmin):
    list_display = ('title', 'publisher', 'publication_date')
    list_filter = ('publication_date',)

admin.site.register(Publisher)
admin.site.register(Author, AuthorAdmin)
admin.site.register(Book, BookAdmin)

```

Here, because we're dealing with a different set of options, we created a separate `ModelAdmin` class—`BookAdmin`. First we defined a `list_display` just to make the change list look a bit nicer. Then we used `list_filter`, which is set to a tuple of fields to use to create filters along the right side of the change-list page. For date fields, Django provides shortcuts to filter the list to “Today,” “Past 7 days,” “This month,” and “This year”—shortcuts that Django's developers have found hit the common cases for filtering by date. Figure 6-10 shows what that looks like.

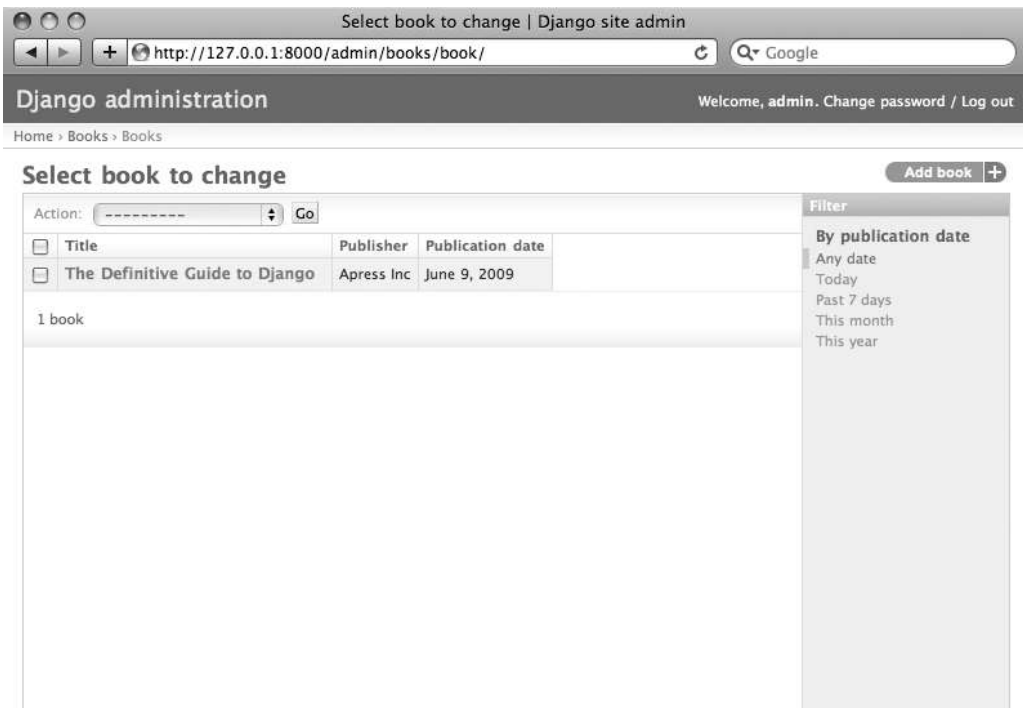


Figure 6-10. *The book change-list page after `list_filter`*

`list_filter` also works on fields of other types, not just `DateField`. (Try it with `BooleanField` and `ForeignKey` fields, for example.) The filters show up as long as there are at least two values to choose from.

Another way to offer date filters is to use the `date_hierarchy` admin option, like this:

```
class BookAdmin(admin.ModelAdmin):
    list_display = ('title', 'publisher', 'publication_date')
    list_filter = ('publication_date',)
    date_hierarchy = 'publication_date'
```

With this in place, the change-list page gets a date drill-down navigation bar at the top of the list, as shown in Figure 6-11. It starts with a list of available years, then drills down into months and individual days.

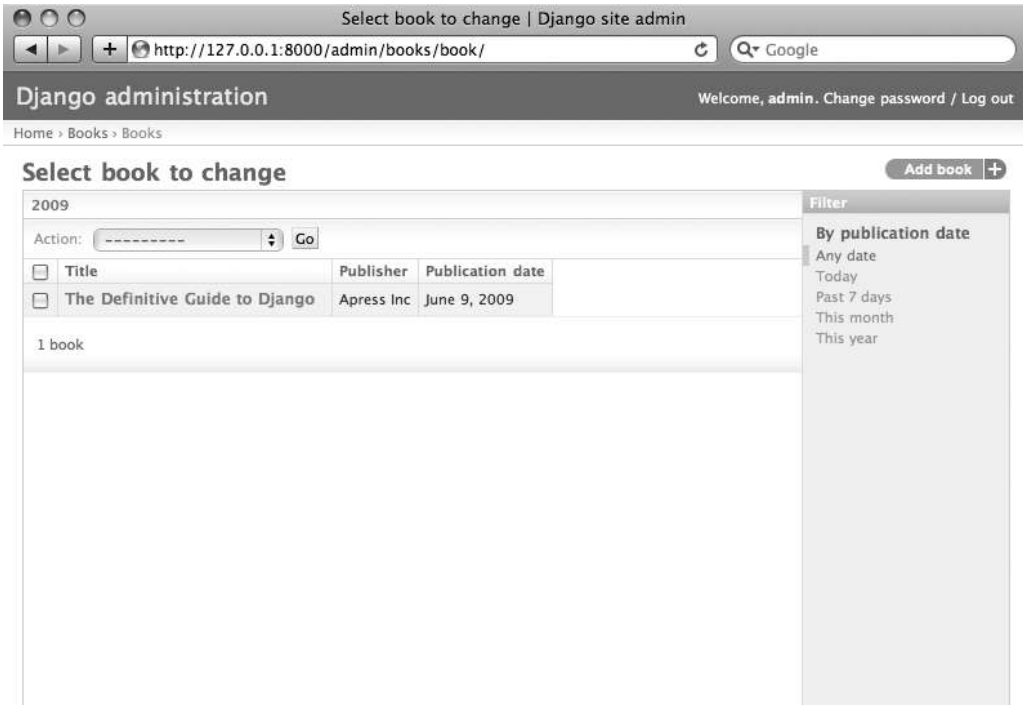


Figure 6-11. The book change-list page after `date_hierarchy`

Note that `date_hierarchy` takes a *string*, not a tuple, because only one date field can be used to make the hierarchy.

Finally, let's change the default ordering so that books on the change-list page are always ordered descending by their publication date. By default, the change list orders objects according to their model's ordering within class `Meta` (which we covered in Chapter 5)—but if you haven't specified this ordering value, then the ordering is undefined.

```
class BookAdmin(admin.ModelAdmin):
    list_display = ('title', 'publisher', 'publication_date')
    list_filter = ('publication_date',)
    date_hierarchy = 'publication_date'
    ordering = ('-publication_date',)
```

This admin ordering option works exactly as the ordering in a model's class `Meta`, except that it uses only the first field name in the list. Just pass a list or tuple of field names, and add a minus sign to a field to use descending sort order.

Reload the book change list to see this in action. Note that the Publication Date header now includes a small arrow that indicates which way the records are sorted. (See Figure 6-12.)

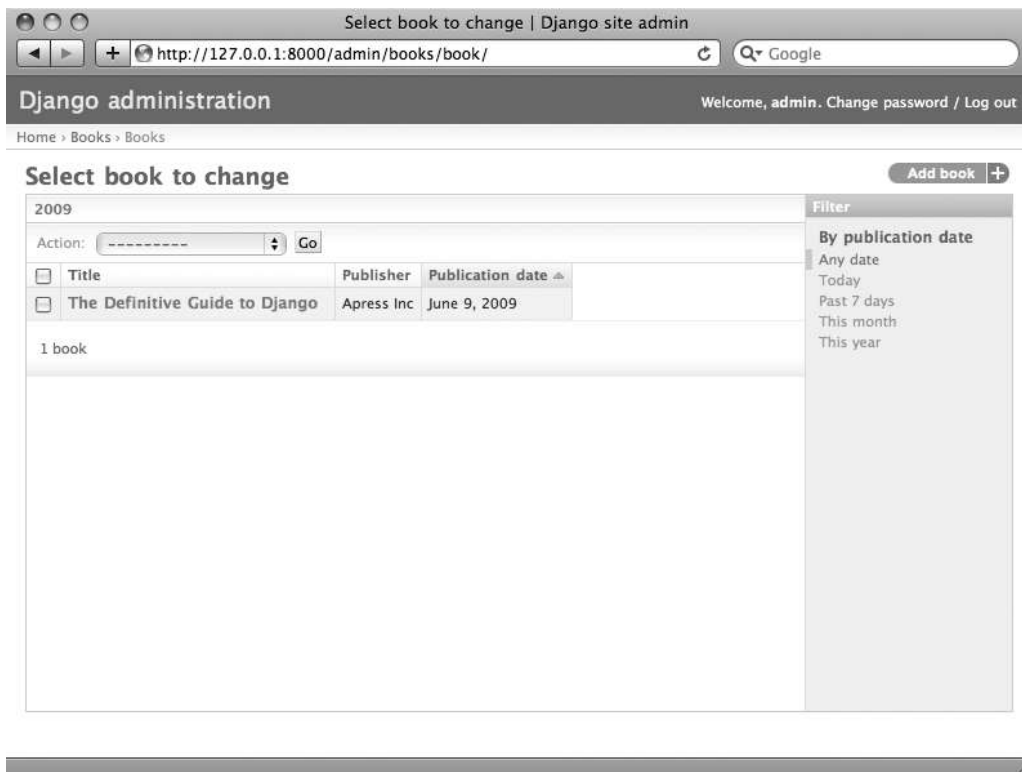


Figure 6-12. The book change-list page after ordering

We've covered the main change-list options here. Using these options, you can make a very powerful, production-ready, data-editing interface with only a few lines of code.

Customizing Edit Forms

Just as the change list can be customized, edit forms can be customized in many ways.

First, let's customize the way fields are ordered. By default, the order of fields in an edit form corresponds to the order in which they're defined in the model. We can change that using the `fields` option in our `ModelAdmin` subclass:

```
class BookAdmin(admin.ModelAdmin):
    list_display = ('title', 'publisher', 'publication_date')
    list_filter = ('publication_date',)
    date_hierarchy = 'publication_date'
    ordering = ('-publication_date',)
    fields = ('title', 'authors', 'publisher', 'publication_date')
```

After this change, the edit form for books will use the given ordering for fields. It's slightly more natural to have the authors after the book title. Of course, the field order should depend on your data-entry workflow. Every form is different.

Another useful thing the `fields` option lets you do is to *exclude* certain fields from being edited. Just leave out the field(s) you want to exclude. You might use this if your admin users are trusted to edit only a certain segment of your data, or if parts of your fields are changed by some outside, automated process. For example, in our book database, we could prevent the `publication_date` field from being editable:

```
class BookAdmin(admin.ModelAdmin):
    list_display = ('title', 'publisher', 'publication_date')
    list_filter = ('publication_date',)
    date_hierarchy = 'publication_date'
    ordering = ('-publication_date',)
    fields = ('title', 'authors', 'publisher')
```

As a result, the edit form for books doesn't offer a way to specify the publication date. This could be useful if, say, you're an editor who prefers that his authors not push back publication dates. (This is purely a hypothetical example, of course.)

When a user employs this incomplete form to add a new book, Django will simply set the `publication_date` to `None`—so make sure that field has `null=True`.

Another commonly used edit-form customization has to do with many-to-many fields. As we've seen on the edit form for books, the admin site represents each `ManyToManyField` as a multiple-select box, which is the most logical HTML input widget to utilize—but multiple-select boxes can be difficult to use. If you want to select multiple items, you have to hold down the Control key, or Command on a Mac. The admin site helpfully inserts a bit of text that explains this, but, still, it gets unwieldy when your field contains hundreds of options.

The admin site's solution is `filter_horizontal`. Let's add that to `BookAdmin` and see what it does.

```
class BookAdmin(admin.ModelAdmin):
    list_display = ('title', 'publisher', 'publication_date')
    list_filter = ('publication_date',)
    date_hierarchy = 'publication_date'
    ordering = ('-publication_date',)
    filter_horizontal = ('authors',)
```

(If you're following along, note that we've also removed the `fields` option to restore all the fields in the edit form.)

Reload the edit form for books, and you'll see that the Authors section now uses a fancy JavaScript filter interface that lets you search through the options dynamically and move specific authors from Available Authors to the Chosen Authors box, and vice versa.

The screenshot shows the Django administration interface for editing a book. The browser address bar shows `http://127.0.0.1:8000/admin/books/book/1/`. The page title is "Change book | Django site admin". The Django administration header shows "Welcome, admin. Change password / Log out". The breadcrumb trail is "Home > Books > Books > The Definitive Guide to Django".

The "Change book" form contains the following fields:

- Title:** The Definitive Guide to Django
- Publisher:** Apress Inc
- Publication date:** 2009-06-09 Today
- Authors:**
 - Available authors:** A search box and a list of authors.
 - Chosen authors:** A list of authors selected for the book. Adrian Holovaty is currently selected.

At the bottom of the form, there are buttons: "Delete", "Save and add another", "Save and continue editing", and "Save".

Figure 6-13. *The book edit form after adding `filter_horizontal`*

We'd highly recommend using `filter_horizontal` for any `ManyToManyField` that has more than ten items. It's far easier to use than a simple multiple-select widget. Also, note you can use `filter_horizontal` for multiple fields—just specify each name in the tuple.

ModelAdmin classes also support a `filter_vertical` option. This works exactly as `filter_horizontal`, but the resulting JavaScript interface stacks the two boxes vertically instead of horizontally. It's a matter of personal taste.

`filter_horizontal` and `filter_vertical` work on only `ManyToManyField` fields, not `ForeignKey` fields. By default, the admin site uses simple `<select>` boxes for `ForeignKey` fields, but, as for `ManyToManyField`, sometimes you don't want to incur the overhead of having to select all the related objects to display in the drop-down. For example, if our book database grows to include thousands of publishers, the Add Book form could take a while to load, because it would have to load every publisher for display in the `<select>` box.

You can fix this with an option called `raw_id_fields`. Set this to a tuple of `ForeignKey` field names, and those fields will be displayed in the admin with a simple text-input box (<input type="text">) instead of a <select>. See Figure 6-14.

```
class BookAdmin(admin.ModelAdmin):
    list_display = ('title', 'publisher', 'publication_date')
    list_filter = ('publication_date',)
    date_hierarchy = 'publication_date'
    ordering = ('-publication_date',)
    filter_horizontal = ('authors',)
    raw_id_fields = ('publisher',)
```

The screenshot shows the Django administration interface for editing a book. The browser address bar shows `http://127.0.0.1:8000/admin/books/book/1/`. The page title is "Change book | Django site admin". The Django administration header shows "Welcome, admin. Change password / Log out". The breadcrumb trail is "Home > Books > Books > The Definitive Guide to Django". The main heading is "Change book" with a "History" button. The form fields are: "Title:" with the value "The Definitive Guide to Django"; "Publisher:" with a text input containing "1" and a magnifying glass icon, with "Apress Inc" displayed below; "Publication date:" with a date input showing "2009-06-09" and a "Today" link; and "Authors:" which is a multi-select interface. The "Available authors" list is empty, and the "Chosen authors" list contains "Adrian Holovaty". There are "Choose all" and "Clear all" buttons at the bottom of the author selection area. At the very bottom of the form are four buttons: "Delete", "Save and add another", "Save and continue editing", and "Save".

Figure 6-14. *The book edit form after adding `raw_id_fields`*

What do you enter in this input box? The database ID of the publisher. Given that humans don't normally memorize database IDs, there's a magnifying-glass icon that you can click to pull up a pop-up window from which you can select the publisher.

Users, Groups, and Permissions

Because you’re logged in as a superuser, you have access to create, edit, and delete any object. Naturally, different environments require different permission systems—not everybody can or should be a superuser. Django’s admin site uses a permissions system that you can use to give specific users access to only the portions of the interface that they need.

These user accounts are meant to be generic enough to be used outside of the admin interface, but we’ll just treat them as admin user accounts for now. In Chapter 14 we’ll cover how to integrate user accounts with the rest of your site (i.e., not just the admin site).

You can edit users and permissions through the admin interface just like any other object. We saw this earlier in this chapter, when we played around with the User and Group sections of the admin. User objects have the standard username, password, e-mail, and real-name fields you might expect, along with a set of fields that define what the user is allowed to do in the admin interface. First, there’s a set of three Boolean flags:

- The “active” flag controls whether the user is active at all. If this flag is off and the user tries to log in, he won’t be allowed in, even with a valid password.
- The “staff” flag controls whether the user is allowed to log in to the admin interface (i.e., whether that user is considered a “staff member” in your organization). Since this same user system can be used to control access to public (i.e., nonadmin) sites—see Chapter 14—this flag differentiates between public users and administrators.
- The “superuser” flag gives the user full access to add, create, and delete any item in the admin interface. If a user has this flag set, then all regular permissions (or lack thereof) are ignored for that user.

“Normal” admin users—that is, active, nonsuperuser staff members—are granted admin access through assigned permissions. Each object editable through the admin interface (e.g., books, authors, publishers) has three permissions: *create*, *edit*, and *delete*. Assigning permissions to a user grants the user the associated level of access.

When you create a user, that user has no permissions; it’s up to you to assign specific ones. For example, you can give a user permission to add and change publishers but not to delete them. Note that these permissions are defined per model, not per object—so they let you say, “John can make changes to any book,” but they don’t let you say, “John can make changes to any book published by Apress.” Per-object permissions are a bit more complicated and are outside the scope of this book (but are covered in the Django documentation).

Note Access to edit users and permissions is also controlled by this permissions system. If you give someone permission to edit users, she will be able to edit her own permissions, which might not be what you want! Giving a user permission to edit other users is essentially turning a user into a superuser.

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

When and Why to Use the Admin Interface— And When Not To

After having worked through this chapter, you should have a good idea of how to use Django’s admin site. But we want to make a point of covering *when* and *why* you might want to use it—and when *not* to use it.

Django’s admin site especially shines when nontechnical users need to be able to enter data; that’s the purpose behind the feature, after all. At the newspaper where Django was first developed, creation of a typical online feature—say, a special report on water quality in the municipal supply—would go something like this:

1. The reporter responsible for the project meets with one of the developers and describes the available data.
2. The developer designs Django models to fit this data and then opens up the admin site to the reporter.
3. The reporter inspects the admin site to point out any missing or extraneous fields—better now than later. The developer changes the models iteratively.
4. When the models are agreed upon, the reporter begins entering data using the admin site. At the same time, the programmer can focus on developing the publicly accessible views/templates (the fun part!).

In other words, the *raison d’être* of Django’s admin interface is to facilitate the simultaneous work of content producers and programmers.

However, beyond these obvious data-entry tasks, the admin site is useful in a few other cases:

- *Inspecting data models:* Once you’ve defined a few models, it can be quite useful to call them up in the admin interface and enter some dummy data. In some cases, this might reveal data-modeling mistakes or other problems with your models.
- *Managing acquired data:* For applications that rely on data coming from external sources (e.g., users or Web crawlers), the admin site gives you an easy way to inspect or edit this data. You can think of it as a less powerful but more convenient version of your database’s command-line utility.
- *Quick and dirty data-management apps:* You can use the admin site to build a very lightweight data-management app—say, to keep track of expenses. If you’re just building something for your own needs, not for public consumption, the admin site can take you a long way. In this sense, you can think of it as a beefed-up, relational version of a spreadsheet.

One final point we want to make clear is that the admin site is not an end-all-be-all. Over the years, we’ve seen it hacked and chopped up to serve a variety of functions it wasn’t intended to serve. It’s not intended to be a *public* interface to data, nor is it intended to allow for sophisticated sorting and searching of your data. As we said early in this chapter, it’s for trusted site administrators. Keeping this sweet spot in mind is the key to effective admin-site usage.

What's Next?

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



Forms

HTML forms are the backbone of interactive Web sites, from the simplicity of Google's single search box to ubiquitous blog comment-submission forms to complex custom data-entry interfaces. This chapter covers how you can use Django to access user-submitted form data, validate it, and do something with it. Along the way, we'll cover `HttpRequest` and `Form` objects.

Getting Data from the Request Object

We introduced `HttpRequest` objects in Chapter 3 when we first covered view functions, but we didn't have much to say about them at the time. Recall that each view function takes an `HttpRequest` object as its first parameter, as in our `hello()` view:

```
from django.http import HttpResponse

def hello(request):
    return HttpResponse("Hello world")
```

`HttpRequest` objects, such as the variable `request` here, have a number of interesting attributes and methods that you should familiarize yourself with so that you know what's possible. You can use these attributes to get information about the current request (i.e., the user/Web browser that's loading the current page on your Django-powered site) at the time the view function is executed.

Information About the URL

`HttpRequest` objects contain several pieces of information about the currently requested URL, as Table 7-1 shows.

Table 7-1. *HttpRequest Attributes and Methods*

Attribute/Method	Description	Example
<code>request.path</code>	The full path, not including the domain but including the leading slash	<code>"/hello/"</code>
<code>request.get_host()</code>	The host (i.e., the “domain,” in common parlance)	<code>"127.0.0.1:8000"</code> or <code>"www.example.com"</code>
<code>request.get_full_path()</code>	The path, plus a query string (if available)	<code>"/hello/?print=true"</code>
<code>request.is_secure()</code>	True if the request was made via HTTPS; otherwise, False	True or False

Always use the attributes/methods outlined in Table 7-1 instead of hard-coding URLs in your views. This makes for more flexible code that can be reused in other places. Here’s a simplistic example:

```
# BAD!
def current_url_view_bad(request):
    return HttpResponse("Welcome to the page at /current/")

# GOOD
def current_url_view_good(request):
    return HttpResponse("Welcome to the page at %s" % request.path)
```

Other Information About the Request

`request.META` is a Python dictionary containing all available HTTP headers for the given request—including the user’s IP address and user agent (generally the name and version of the Web browser). Note that the full list of available headers depends on which headers the user sent and which headers your Web server sets. The following are some commonly available keys in this dictionary:

- `HTTP_REFERER`: The referring URL, if any. (Note the misspelling of `REFERER`.)
- `HTTP_USER_AGENT`: The user-agent string (if any) of the user’s browser. This looks something like the following:

```
"Mozilla 5.0 (X11; U; Linux i686) Gecko/20080829 Firefox/2.0.0.17"
```

- `REMOTE_ADDR`: The IP address of the client—for instance, `"12.345.67.89"`. (If the request has passed through any proxies, then this might be a comma-separated list of IP addresses, such as `"12.345.67.89,23.456.78.90"`.)

Note that because `request.META` is just a basic Python dictionary, you'll get a `KeyError` exception if you try to access a key that doesn't exist. (Because HTTP headers are *external* data—that is, they're submitted by your users' browsers—they shouldn't be trusted, and you should always design your application to fail gracefully if a particular header is empty or doesn't exist.) You should either use a `try/except` clause or the `get()` method to handle the case of undefined keys, as in this example:

```
# BAD!
def ua_display_bad(request):
    ua = request.META['HTTP_USER_AGENT'] # Might raise KeyError!
    return HttpResponse("Your browser is %s" % ua)

# GOOD (VERSION 1)
def ua_display_good1(request):
    try:
        ua = request.META['HTTP_USER_AGENT']
    except KeyError:
        ua = 'unknown'
    return HttpResponse("Your browser is %s" % ua)

# GOOD (VERSION 2)
def ua_display_good2(request):
    ua = request.META.get('HTTP_USER_AGENT', 'unknown')
    return HttpResponse("Your browser is %s" % ua)
```

We encourage you to write a small view that displays all of the `request.META` data so you can get to know what's available. Here's what that view might look like:

```
def display_meta(request):
    values = request.META.items()
    values.sort()
    html = []
    for k, v in values:
        html.append('<tr><td>%s</td><td>%s</td></tr>' % (k, v))
    return HttpResponse('<table>%s</table>' % '\n'.join(html))
```

As an exercise, see whether you can convert this view to use Django's template system instead of hard-coding the HTML. Also try adding `request.path` and the other `HttpRequest` methods from the previous section.

Information About Submitted Data

Beyond basic metadata about the request, `HttpRequest` objects have two attributes that contain user-submitted information: `request.GET` and `request.POST`. Both of these are dictionary-like objects that give you access to GET and POST data.

POST data generally is submitted from an HTML `<form>`, while GET data can come from a `<form>` or the query string in the page's URL.

DICTIONARY-LIKE OBJECTS

When we say `request.GET` and `request.POST` are dictionary-like objects, we mean they behave like standard Python dictionaries but aren't technically dictionaries under the hood. For example, `request.GET` and `request.POST` both have `get()`, `keys()`, and `values()` methods, and you can iterate over the keys by doing `for key in request.GET`.

So why do we refer to these as “dictionary-like” objects as opposed to normal dictionaries? Because both `request.GET` and `request.POST` have additional methods that normal dictionaries don't have.

You might have encountered the similar term “file-like objects”—Python objects that have a few basic methods, such as `read()`, that let them act as stand-ins for “real” file objects.

A Simple Form-Handling Example

Continuing this book's ongoing example of books, authors, and publishers, let's create a simple view that lets users search our book database by title.

Generally, there are two parts to developing a form: the HTML user interface and the back-end view code that processes the submitted data. The first part is easy; let's just set up a view that displays a search form:

```
from django.shortcuts import render_to_response

def search_form(request):
    return render_to_response('search_form.html')
```

As you learned in Chapter 3, this view can live anywhere on your Python path. For this example, put it in `books/views.py`.

The accompanying template, `search_form.html`, could look like this:

```
<html>
<head>
    <title>Search</title>
</head>
<body>
    <form action="/search/" method="get">
        <input type="text" name="q">
        <input type="submit" value="Search">
    </form>
</body>
</html>
```

The URL pattern in `urls.py` could look like this:

```
from mysite.books import views

urlpatterns = patterns('',
    # ...
    (r'^search-form/$', views.search_form),
    # ...
)
```

Note that we're importing the views module directly, instead of something like `from mysite.views import search_form`, because the former is less verbose. We'll cover this importing approach in more detail in Chapter 8.

Now, if you run the runserver and visit `http://127.0.0.1:8000/search-form/`, you'll see the search interface. Simple enough.

Try submitting the form, though, and you'll get a Django 404 error. The form points to the URL `/search/`, which hasn't yet been implemented. Let's fix that with a second view function:

```
# urls.py

urlpatterns = patterns('',
    # ...
    (r'^search-form/$', views.search_form),
    (r'^search/$', views.search),
    # ...
)

# views.py

def search(request):
    if 'q' in request.GET:
        message = 'You searched for: %r' % request.GET['q']
    else:
        message = 'You submitted an empty form.'
    return HttpResponse(message)
```

For the moment, this merely displays the user's search term so we can make sure the data is being submitted to Django properly and so you can get a feel for how the search term flows through the system. In short, here's what happens:

1. The HTML `<form>` defines a variable `q`. When it's submitted, the value of `q` is sent via GET (method="get") to the URL `/search/`.
2. The Django view that handles the URL `/search/` (`search()`) has access to the `q` value in `request.GET`.

Note that we explicitly check that `'q'` exists in `request.GET`. As we pointed out in the `request.META` discussion earlier in this chapter, you shouldn't trust anything submitted by users or even assume that they've submitted anything in the first place. If we didn't add this check, any submission of an empty form would raise `KeyError` in the view:

```
# BAD!

def bad_search(request):
    # The following line will raise KeyError if 'q' hasn't
    # been submitted!
    message = 'You searched for: %r' % request.GET['q']
    return HttpResponse(message)
```

QUERY-STRING PARAMETERS

Because GET data is passed in the query string (e.g., `/search/?q=django`), you can use `request.GET` to access query-string variables. In Chapter 3's introduction of Django's URLconf system, we compared Django's pretty URLs to more traditional PHP/Java URLs such as `/time/plus?hours=3` and said we'd show you how to use the latter in Chapter 7. Now you know how to access query-string parameters in your views (like `hours=3` in this example)—use `request.GET`.

POST data works the same way as GET data—just use `request.POST` instead of `request.GET`. What's the difference between GET and POST? Use GET when the act of submitting the form is just a request to “get” data. Use POST whenever the act of submitting the form will have some side effect—*changing* data or sending an e-mail, or something else that's beyond simple *display* of data. In our book-search example, we're using GET because the query doesn't change any data on our server. (See <http://www.w3.org/2001/tag/doc/whenToUseGet.html> if you want to learn more about GET and POST.)

Now that we've verified `request.GET` is being passed in properly, let's hook the user's search query into our book database (again, in `views.py`):

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

def search(request):
    if 'q' in request.GET and request.GET['q']:
        q = request.GET['q']
        books = Book.objects.filter(title__icontains=q)
        return render_to_response('search_results.html',
                                  {'books': books, 'query': q})
    else:
        return HttpResponse('Please submit a search term.')
```

Some notes on what we did here:

- In addition to checking that 'q' exists in `request.GET`, we made sure that `request.GET['q']` is a nonempty value before passing it to the database query.
- We used `Book.objects.filter(title__icontains=q)` to query our book table for all books whose title includes the given submission. The `icontains` is a lookup type (as explained in Chapter 5 and Appendix B), and the statement can be roughly translated as “Get the books whose title contains q, without being case-sensitive.”

This is a very simple way to do a book search. We wouldn't recommend using a simple `icontains` query on a large production database, as it can be slow. (In the real world, you'd want to use a custom search system of some sort. Search the Web for *open-source full-text search* to get an idea of the possibilities.)

- We passed `books`, a list of `Book` objects, to the template. The template code for `search_results.html` might include something like this:

```
<p>You searched for: <strong>{{ query }}</strong></p>

{% if books %}
    <p>Found {{ books|length }} book{{ books|pluralize }}.</p>
    <ul>
        {% for book in books %}
            <li>{{ book.title }}</li>
        {% endfor %}
    </ul>
{% else %}
    <p>No books matched your search criteria.</p>
{% endif %}
```

Note the usage of the `pluralize` template filter, which outputs an “s” if appropriate, based on the number of books found.

Improving Our Simple Form-Handling Example

As in previous chapters, we’ve shown you the simplest thing that could possibly work. Now we’ll point out some problems and show you how to improve it.

First, our `search()` view’s handling of an empty query is poor—we’re just displaying a “Please submit a search term.” message, requiring the user to hit the browser’s Back button. This is horrid and unprofessional, and if you ever actually implement something like this in the wild, your Django privileges will be revoked.

It would be much better to redisplay the form, with an error above it, so that the user can try again immediately. The easiest way to do that would be to render the template again, like this:

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

def search_form(request):
    return render_to_response('search_form.html')

def search(request):
    if 'q' in request.GET and request.GET['q']:
        q = request.GET['q']
        books = Book.objects.filter(title__icontains=q)
        return render_to_response('search_results.html',
            {'books': books, 'query': q})
    else:
        return render_to_response('search_form.html', {'error': True})
```

(Note that we’ve included `search_form()` here so you can see both views in one place.)

Here we’ve improved `search()` to render the `search_form.html` template again if the query is empty. And because we need to display an error message in that template, we pass a template variable. Now we can edit `search_form.html` to check for the error variable:

```

<html>
<head>
    <title>Search</title>
</head>
<body>
    {% if error %}
        <p style="color: red;">Please submit a search term.</p>
    {% endif %}
    <form action="/search/" method="get">
        <input type="text" name="q">
        <input type="submit" value="Search">
    </form>
</body>
</html>

```

We can still use this template from our original view, `search_form()`, because `search_form()` doesn't pass error to the template—so the error message won't show up in that case.

With this change in place, it's a better application but it now begs the question: is a dedicated `search_form()` view really necessary? As it stands, a request to the URL `/search/` (without any GET parameters) will display the empty form (but with an error). We can remove the `search_form()` view, along with its associated URL pattern, as long as we change `search()` to hide the error message when somebody visits `/search/` with no GET parameters:

```

def search(request):
    error = False
    if 'q' in request.GET:
        q = request.GET['q']
        if not q:
            error = True
        else:
            books = Book.objects.filter(title__icontains=q)
            return render_to_response('search_results.html',
                                     {'books': books, 'query': q})
    return render_to_response('search_form.html',
                              {'error': error})

```

In this updated view, if a user visits `/search/` with no GET parameters, he'll see the search form with no error message. If a user submits the form with an empty value for `'q'`, he'll see the search form *with* an error message. And, finally, if a user submits the form with a non-empty value for `'q'`, he'll see the search results.

We can make one final improvement to this application, to remove a bit of redundancy. Now that we've rolled the two views and URLs into one and `/search/` handles both search-form display and result display, the HTML `<form>` in `search_form.html` doesn't have to hard-code a URL. Instead of this

```
<form action="/search/" method="get">
```

it can be changed to this:

```
<form action="" method="get">
```


The `action=""` means “Submit the form to the same URL as the current page.” With this change in place, you won’t have to remember to change the `action` if you ever hook the `search()` view to another URL.

Simple Validation

Our search example is still reasonably simple, particularly in terms of its data validation; we're merely checking to make sure the search query isn't empty. Many HTML forms include a level of validation that's more complex than making sure the value is nonempty. We've all seen the following error messages on Web sites:

- “Please enter a valid e-mail address. ‘foo’ is not an e-mail address.”
- “Please enter a valid five-digit U.S. ZIP code. ‘123’ is not a ZIP code.”
- “Please enter a valid date in the format YYYY-MM-DD.”
- “Please enter a password that is at least 8 characters long and contains at least one number.”

A NOTE ON JAVASCRIPT VALIDATION

JavaScript validation is beyond the scope of this book, but you can use JavaScript to validate data on the client side, directly in the browser. Be warned, however: even if you do this, you *must* validate data on the server side. Some people have JavaScript turned off, and some malicious users might submit raw, unvalidated data directly to your form handler to see whether they can cause mischief.

There's nothing you can do about this, other than to *always* validate user-submitted data server-side (i.e., in your Django views). You should think of JavaScript validation as a bonus usability feature, not as your only means of validating.

Let's tweak our `search()` view so it validates that the search term is less than or equal to 20 characters long. (For this example, let's say anything longer than that might make the query too slow.) How might we do that? The simplest thing would be to embed the logic directly in the view, like this:

[illegible]

```
return render_to_response('search_form.html',
    {'error': error})
```

Now if you try submitting a search query greater than 20 characters long, you'll get an error message. But that error message in `search_form.html` currently says, "Please submit a search term."—so we'll have to change it to be accurate for both cases (an empty search term or a search term that's too long).

```
<html>
<head>
    <title>Search</title>
</head>
<body>
    {% if error %}
        <p style="color: red;">
            Please submit a search term
            20 characters or shorter.
        </p>
    {% endif %}
    <form action="/search/" method="get">
        <input type="text" name="q">
        <input type="submit" value="Search">
    </form>
</body>
</html>
```

There's something ugly about this. Our one-size-fits-all error message is potentially confusing. Why should the error message for an empty form submission mention anything about a 20-character limit? Error messages should be specific and clear.

The problem is that we're using a simple Boolean value for error, whereas we should be using a *list* of error-message strings. Here's how we might fix that:

```
def search(request):
    errors = []
    if 'q' in request.GET:
        q = request.GET['q']
        if not q:
            errors.append('Enter a search term.')
        elif len(q) > 20:
            errors.append('Please enter at most 20 characters.')
        else:
            books = Book.objects.filter(title__icontains=q)
            return render_to_response('search_results.html',
                {'books': books, 'query': q})
    return render_to_response('search_form.html',
        {'errors': errors})
```

Then we need to make a small tweak to the `search_form.html` template to reflect that it's now passed an errors list instead of an error Boolean value:

```

<html>
<head>
    <title>Search</title>
</head>
<body>
    {% if errors %}
        <ul>
            {% for error in errors %}
                <li>{{ error }}</li>
            {% endfor %}
        </ul>
    {% endif %}
    <form action="/search/" method="get">
        <input type="text" name="q">
        <input type="submit" value="Search">
    </form>
</body>
</html>

```

Making a Contact Form

Although we iterated over the book-search-form example several times and improved it nicely, it's still fundamentally simple: just a single field, 'q'. Because it's so simple, we didn't even use Django's form library to deal with it. But more complex forms call for more complex treatment—and now we'll develop something more complex: a site contact form that lets site users submit a bit of feedback, along with an optional e-mail return address. After the form is submitted and the data is validated, we'll automatically send the message via e-mail to the site staff.

We'll start with our template, `contact_form.html`.

```

<html>
<head>
    <title>Contact us</title>
</head>
<body>
    <h1>Contact us</h1>

    {% if errors %}
        <ul>
            {% for error in errors %}
                <li>{{ error }}</li>
            {% endfor %}
        </ul>
    {% endif %}

```

```

<form action="/contact/" method="post">
    <p>Subject: <input type="text" name="subject"></p>
    <p>Your e-mail (optional): <input type="text" name="e-mail"></p>
    <p>Message: <textarea name="message" rows="10" cols="50"></textarea></p>
    <input type="submit" value="Submit">
</form>
</body>
</html>

```

We've defined three fields: the subject, e-mail address, and message. The second is optional, but the other two fields are required. Note we're using `method="post"` here instead of `method="get"` because this form submission has a side effect—it sends an e-mail. Also, we copied the error-displaying code from our previous template `search_form.html`.

If we continue down the road established by our `search()` view from the previous section, a naive version of our `contact()` view might look like this:

```

from django.core.mail import send_mail
from django.http import HttpResponseRedirect
from django.shortcuts import render_to_response

def contact(request):
    errors = []
    if request.method == 'POST':
        if not request.POST.get('subject', ''):
            errors.append('Enter a subject.')
        if not request.POST.get('message', ''):
            errors.append('Enter a message.')
        if request.POST.get('e-mail') and '@' not in request.POST['e-mail']:
            errors.append('Enter a valid e-mail address.')
        if not errors:
            send_mail(
                request.POST['subject'],
                request.POST['message'],
                request.POST.get('e-mail', 'noreply@example.com'),
                ['siteowner@example.com'],
            )
            return HttpResponseRedirect('/contact/thanks/')
    return render_to_response('contact_form.html',
        {'errors': errors})

```

Note You may be wondering whether to put this view in the `books/views.py` file. It doesn't have anything to do with the books application, so should it live elsewhere? It's up to you; Django doesn't care, as long as you're able to point to the view from your `URLconf`. Our personal preference would be to create a separate directory, `contact`, at the same level in the directory tree as `books`. This would contain an empty `__init__.py` and `views.py`.

Several new things are happening here:

- We're checking that `request.method` is `'POST'`. This will be true only in the case of a form submission; it won't be true if somebody is merely viewing the contact form. (In the latter case, `request.method` will be set to `'GET'` because in normal Web browsing, browsers use GET, not POST.) This makes it a nice way to isolate the “form display” case from the “form processing” case.
- Instead of `request.GET`, we're using `request.POST` to access the submitted form data. This is necessary because the HTML `<form>` in `contact_form.html` uses `method="post"`. If this view is accessed via POST, then `request.GET` will be empty.
- We have *two* required fields, `subject` and `message`, so we have to validate both. Note that we're using `request.POST.get()` and providing a blank string as the default value; this is a nice, short way of handling both the cases of missing keys and missing data.
- Although the e-mail field is not required, we still validate it if it is indeed submitted. Our validation algorithm here is fragile—we're just checking that the string contains an `@` character. In the real world, you'd want more robust validation (and Django provides it, which we'll show you in the “Your First Form Class” section later in this chapter).
- We're using the function `django.core.mail.send_mail` to send an e-mail. This function has four required arguments: the e-mail subject, the e-mail body, the “from” address, and a list of recipient addresses. `send_mail` is a convenient wrapper around Django's `E-mailMessage` class, which provides advanced features such as attachments, multipart e-mails, and full control over e-mail headers.

Note that in order to send e-mail using `send_mail()`, your server must be configured to send mail, and Django must be told about your outbound e-mail server. See <http://docs.djangoproject.com/en/dev/topics/e-mail/> for the specifics.

- After the e-mail is sent, we redirect to a “success” page by returning an `HttpResponseRedirect` object. We'll leave the implementation of that “success” page up to you (it's a simple `view/URLconf/template`), but we'll explain why we initiate a redirect instead of, for example, simply calling `render_to_response()` with a template right there.

The reason: if a user hits Refresh on a page that was loaded via POST, that request will be repeated. This can often lead to undesired behavior, such as a duplicate record being added to the database—or, in our example, the e-mail being sent twice. If the user is redirected to another page after the POST, then there's no chance of repeating the request.

You should *always* issue a redirect for successful POST requests. It's a Web-development best practice.

This view works, but those validation functions are kind of crufty. Imagine processing a form with a dozen fields; would you really want to have to write all of those `if` statements?

Another problem is *form redisplay*. In the case of validation errors, it's best practice to redisplay the form with the previously submitted data already filled in so the user can see what he did wrong (and doesn't have to re-enter data in fields that were submitted correctly). We *could* manually pass the POST data back to the template, but we'd have to edit each HTML field to insert the proper value in the proper place:

```
# views.py

def contact(request):
    errors = []
    if request.method == 'POST':
        if not request.POST.get('subject', ''):
            errors.append('Enter a subject.')
        if not request.POST.get('message', ''):
            errors.append('Enter a message.')
        if request.POST.get('e-mail') and '@' not in request.POST['e-mail']:
            errors.append('Enter a valid e-mail address.')
        if not errors:
            send_mail(
                request.POST['subject'],
                request.POST['message'],
                request.POST.get('e-mail', 'noreply@example.com'),
                ['siteowner@example.com'],
            )
            return HttpResponseRedirect('/contact/thanks/')
    return render_to_response('contact_form.html', {
        'errors': errors,
        'subject': request.POST.get('subject', ''),
        'message': request.POST.get('message', ''),
        'e-mail': request.POST.get('e-mail', ''),
    })

# contact_form.html

<html>
<head>
    <title>Contact us</title>
</head>
<body>
    <h1>Contact us</h1>

    {% if errors %}
        <ul>
            {% for error in errors %}
                <li>{{ error }}</li>
            {% endfor %}
        </ul>
    {% endif %}

    <form action="/contact/" method="post">
        <p>Subject: <input type="text" name="subject" value="{{ subject }}"></p>
        <p>Your e-mail (optional):
            <input type="text" name="e-mail" value="{{ e-mail }}">

```

```

    </p>
    <p>Message:
        <textarea name="message" rows="10" cols="50">
            **{{ message }}**
        </textarea>
    </p>
    <input type="submit" value="Submit">
</form>
</body>
</html>

```

This is a lot of cruft, and it introduces a lot of opportunities for human error. We hope you’re starting to see the opportunity for some higher-level library that handles form- and validation-related tasks.

Your First Form Class

Django comes with a form library, called `django.forms`, that handles many of the issues we’ve been exploring in this chapter—from HTML form display to validation. Let’s dive in and rework our contact-form application using the Django forms framework.

DJANGO’S “NEWFORMS” LIBRARY

Throughout the Django community, you might see chatter about something called `django.newforms`. When people speak of `django.newforms`, they’re talking about what is now `django.forms`—the library covered in this chapter.

When Django was first released to the public, it had a complicated, confusing forms system, `django.forms`. It was completely rewritten, and the new version was called `django.newforms` so that people could still use the old system. When Django 1.0 was released, the old `django.forms` went away, and `django.newforms` became `django.forms`.

The primary way to use the forms framework is to define a `Form` class for each HTML `<form>` you’re dealing with. In our case, we only have one `<form>`, so we’ll have one `Form` class. This class can live anywhere you want—including directly in your `views.py` file—but community convention is to keep `Form` classes in a separate file called `forms.py`. Create this file in the same directory as your `views.py`, and enter the following:

```

from django import forms

class ContactForm(forms.Form):
    subject = forms.CharField()
    e-mail = forms.EmailField(required=False)
    message = forms.CharField()

```

This is pretty intuitive, and it's similar to Django's model syntax. Each field in the form is represented by a type of Field class—CharField and EmailField are the only types of fields used here—as attributes of a Form class. Each field is required by default, so to make e-mail optional, we specify `required=False`.

Let's hop into the Python interactive interpreter and see what this class can do. The first thing it can do is display itself as HTML:

```
>>> from contact.forms import ContactForm
>>> f = ContactForm()
>>> print f
<tr><th><label for="id_subject">Subject:</label></th><td>
<input type="text" name="subject" id="id_subject" /></td></tr>
<tr><th><label for="id_e-mail">E-mail:</label></th><td>
<input type="text" name="e-mail" id="id_e-mail" /></td></tr>
<tr><th><label for="id_message">Message:</label></th><td>
<input type="text" name="message" id="id_message" /></td></tr>
```

Django adds a label to each field, along with `<label>` tags for accessibility. The idea is to make the default behavior as optimal as possible.

This default output is in the format of an HTML `<table>`, but there are a few other built-in outputs:

```
>>> print f.as_ul()
<li><label for="id_subject">Subject:</label>
<input type="text" name="subject" id="id_subject" /></li>
<li><label for="id_e-mail">E-mail:</label>
<input type="text" name="e-mail" id="id_e-mail" /></li>
<li><label for="id_message">Message:</label>
<input type="text" name="message" id="id_message" /></li>
>>> print f.as_p()
<p><label for="id_subject">Subject:</label>
<input type="text" name="subject" id="id_subject" /></p>
<p><label for="id_e-mail">E-mail:</label>
<input type="text" name="e-mail" id="id_e-mail" /></p>
<p><label for="id_message">Message:</label>
<input type="text" name="message" id="id_message" /></p>
```

Note that the opening and closing `<table>`, ``, and `<form>` tags aren't included in the output, so you can add any additional rows and customization if necessary.

These methods are just shortcuts for the common case of “display the entire form.” You can also display the HTML for a particular field:

```
>>> print f['subject']
<input type="text" name="subject" id="id_subject" />
>>> print f['message']
<input type="text" name="message" id="id_message" />
```

The second thing Form objects can do is validate data. To do this, create a new Form object and pass it a dictionary of data that maps field names to data:


```
>>> f = ContactForm({'subject': 'Hello', 'e-mail': 'adrian@example.com',
... 'message': 'Nice site!'})
```

Once you’ve associated data with a Form instance, you’ve created a *bound* Form:

```
>>> f.is_bound
True
```

Call the `is_valid()` method on any bound Form to find out whether its data is valid. We’ve passed a valid value for each field, so the Form in its entirety is valid:

```
>>> f.is_valid()
True
```

If we don’t pass the e-mail field, it’s still valid, because we’ve specified `required=False` for that field:

```
>>> f = ContactForm({'subject': 'Hello', 'message': 'Nice site!'})
>>> f.is_valid()
True
```

But if we leave off either subject or message, the Form is no longer valid:

```
>>> f = ContactForm({'subject': 'Hello'})
>>> f.is_valid()
False
>>> f = ContactForm({'subject': 'Hello', 'message': ''})
>>> f.is_valid()
False
```

You can drill down to get field-specific error messages:

```
>>> f = ContactForm({'subject': 'Hello', 'message': ''})
>>> f['message'].errors
[u'This field is required.']
>>> f['subject'].errors
[]
>>> f['e-mail'].errors
[]
```

Each bound Form instance has an `errors` attribute that gives you a dictionary mapping field names to error-message lists:

```
>>> f = ContactForm({'subject': 'Hello', 'message': ''})
>>> f.errors
{'message': [u'This field is required.']}
```

Finally, for Form instances whose data has been found to be valid, a `cleaned_data` attribute is available. This is a dictionary of the submitted data, “cleaned up.” Django’s forms framework not only validates data, but cleans it up by converting values to the appropriate Python types, as shown here:

```
>>> f = ContactForm({'subject': 'Hello', 'e-mail': 'adrian@example.com',
... 'message': 'Nice site!'})
>>> f.is_valid()
True
>>> f.cleaned_data
{'message': u'Nice site!', 'e-mail': u'adrian@example.com', 'subject': u'Hello'}
```

Our contact form deals only with strings, which are “cleaned” into Unicode objects—but if we were to use an `IntegerField` or a `DateField`, the forms framework would ensure that `cleaned_data` used proper Python integers or `datetime.date` objects for the given fields.

Tying Form Objects into Views

Now that you have some basic knowledge about Form classes, you might see how we can use this infrastructure to replace some of the cruft in our `contact()` view. Here’s how we can rewrite `contact()` to use the forms framework:

```
# views.py

from django.shortcuts import render_to_response
from mysite.contact.forms import ContactForm

def contact(request):
    if request.method == 'POST':
        form = ContactForm(request.POST)
        if form.is_valid():
            cd = form.cleaned_data
            send_mail(
                cd['subject'],
                cd['message'],
                cd.get('e-mail', 'noreply@example.com'),
                ['siteowner@example.com'],
            )
            return HttpResponseRedirect('/contact/thanks/')
    else:
        form = ContactForm()
        return render_to_response('contact_form.html', {'form': form})

# contact_form.html

<html>
<head>
    <title>Contact us</title>
</head>
<body>
    <h1>Contact us</h1>
```

```

{% if form.errors %}
    <p style="color: red;">
        Please correct the error{{ form.errors|pluralize }} below.
    </p>
{% endif %}

<form action="" method="post">
    <table>
        {{ form.as_table }}
    </table>
    <input type="submit" value="Submit">
</form>
</body>
</html>

```

Look at how much cruft we've been able to remove! Django's forms framework handles the HTML display, the validation, data cleanup, and form redisplay-with-errors.

Try running this locally. Load the form, submit it with none of the fields filled out, submit it with an invalid e-mail address, then finally submit it with valid data. (Of course, depending on your mail-server configuration, you might get an error when `send_mail()` is called, but that's another issue.)

Changing How Fields Are Rendered

Probably the first thing you'll notice when you render this form locally is that the message field is displayed as an `<input type="text">`, and it ought to be a `<textarea>`. We can fix that by setting the field's *widget*:

```

from django import forms

class ContactForm(forms.Form):
    subject = forms.CharField()
    e-mail = forms.EmailField(required=False)
    message = forms.CharField(widget=forms.Textarea)

```

The forms framework separates out the presentation logic for each field into a set of widgets. Each field type has a default widget, but you can easily override the default or provide a custom widget of your own.

Think of the Field classes as representing *validation logic*, while widgets represent *presentation logic*.

Setting a Maximum Length

One of the most common validation needs is to check that a field is of a certain size. For good measure, we should improve our `ContactForm` to limit the subject to 100 characters. To do that, just supply a `max_length` to the `CharField`, like this:

```
from django import forms

class ContactForm(forms.Form):
    subject = forms.CharField(max_length=100)
    e-mail = forms.EmailField(required=False)
    message = forms.CharField(widget=forms.Textarea)
```

An optional `min_length` argument is also available.

Setting Initial Values

As an improvement to this form, let's add an *initial value* for the subject field: "I love your site!" (A little power of suggestion can't hurt.) To do this, we can use the `initial` argument when we create a Form instance:

```
def contact(request):
    if request.method == 'POST':
        form = ContactForm(request.POST)
        if form.is_valid():
            cd = form.cleaned_data
            send_mail(
                cd['subject'],
                cd['message'],
                cd.get('e-mail', 'noreply@example.com'),
                ['siteowner@example.com'],
            )
            return HttpResponseRedirect('/contact/thanks/')
    else:
        form = ContactForm(
            initial={'subject': 'I love your site!'}
        )
    return render_to_response('contact_form.html', {'form': form})
```

Now the subject field will be displayed prepopulated with that kind statement.

Note that there is a difference between passing *initial* data and passing data that *binds* the form. If you're just passing *initial* data, then the form will be *unbound*, which means it won't have any error messages.

Adding Custom Validation Rules

Imagine we've launched our feedback form, and the e-mails have started tumbling in. There's just one problem: some of the submitted messages are just one or two words, which isn't long enough for us to make sense of. We decide to adopt a new validation policy: four words or more, please.

There are various ways to hook custom validation into a Django form. If our rule is something we will reuse again and again, we can create a custom field type. Most custom validations are one-off affairs, though, and can be tied directly to the Form class.

We want additional validation on the message field, so we add a `clean_message()` method to our Form class:

```

from django import forms

class ContactForm(forms.Form):
    subject = forms.CharField(max_length=100)
    e-mail = forms.EmailField(required=False)
    message = forms.CharField(widget=forms.Textarea)

    def clean_message(self):
        message = self.cleaned_data['message']
        num_words = len(message.split())
        if num_words < 4:
            raise forms.ValidationError("Not enough words!")
        return message

```

Django's form system automatically looks for any method whose name starts with `clean_` and ends with the name of a field. If any such method exists, it's called during validation.

Specifically, the `clean_message()` method will be called *after* the default validation logic for a given field (in this case, the validation logic for a required `CharField`). Because the field data has already been partially processed, we pull it out of `self.cleaned_data`. Also, we don't have to worry about checking that the value exists and is nonempty; the default validator does that.

We naively use a combination of `len()` and `split()` to count the number of words. If the user has entered too few words, we raise a `forms.ValidationError`. The string attached to this exception will be displayed to the user as an item in the error list.

It's important that we explicitly return the cleaned value for the field at the end of the method. This allows us to modify the value (or convert it to a different Python type) within our custom validation method. If we forget the return statement, then `None` will be returned and the original value will be lost.

Specifying Labels

By default, the labels on Django's autogenerated form HTML are created by replacing underscores with spaces and capitalizing the first letter—so the label for the e-mail field is "E-mail". (Sound familiar? It's the same simple algorithm that Django's models use to calculate default `verbose_name` values for fields, which we covered in Chapter 5.)

But, as with Django's models, we can customize the label for a given field. Just use `label`, like so:

```

class ContactForm(forms.Form):
    subject = forms.CharField(max_length=100)
    e-mail = forms.EmailField(required=False, label='Your e-mail address')
    message = forms.CharField(widget=forms.Textarea)

```

Customizing Form Design

Our `contact_form.html` template uses `{{ form.as_table }}` to display the form, but we can display the form in other ways to get more granular control over the display.

The quickest way to customize forms' presentation is with CSS. Error lists, in particular, could do with some visual enhancement, and the autogenerated error lists use `<ul class="errorlist">` precisely so that you can target them with CSS. The following CSS really makes our errors stand out:

```
<style type="text/css">
    ul.errorlist {
        margin: 0;
        padding: 0;
    }
    .errorlist li {
        background-color: red;
        color: white;
        display: block;
        font-size: 10px;
        margin: 0 0 3px;
        padding: 4px 5px;
    }
</style>
```

Although it's convenient to have our form's HTML generated for us, in many cases you'll want to override the default rendering. `{{ form.as_table }}` and friends are useful shortcuts while you develop your application, but everything about the way a form is displayed can be overridden, mostly within the template itself, and you'll likely override the defaults often.

Each field's widget (`<input type="text">`, `<select>`, `<textarea>`, etc.) can be rendered individually by accessing `{{ form.fieldname }}` in the template, and any errors associated with a field are available as `{{ form.fieldname.errors }}`. With this in mind, we can construct a custom template for our contact form with the following template code:

```
<html>
<head>
    <title>Contact us</title>
</head>
<body>
    <h1>Contact us</h1>

    {% if form.errors %}
        <p style="color: red;">
            Please correct the error{{ form.errors|pluralize }} below.
        </p>
    {% endif %}

    <form action="" method="post">
        <div class="field">
            {{ form.subject.errors }}
            <label for="id_subject">Subject:</label>
            {{ form.subject }}
        </div>
```

```

<div class="field">
    {{ form.e-mail.errors }}
    <label for="id_e-mail">Your e-mail address:</label>
    {{ form.e-mail }}
</div>
<div class="field">
    {{ form.message.errors }}
    <label for="id_message">Message:</label>
    {{ form.message }}
</div>
<input type="submit" value="Submit">
</form>
</body>
</html>

```

`{{ form.message.errors }}` displays a `<ul class="errorlist">` if errors are present and a blank string if the field is valid (or the form is unbound). We can also treat `form.message.errors` as a Boolean or even iterate over it as a list. Consider this example:

```

<div class="field{% if form.message.errors %} errors{% endif %}">
    {% if form.message.errors %}
        <ul>
            {% for error in form.message.errors %}
                <li><strong>{{ error }}</strong></li>
            {% endfor %}
        </ul>
    {% endif %}
    <label for="id_message">Message:</label>
    {{ form.message }}
</div>

```

In the case of validation errors, this will add an errors class to the containing `<div>` and display the list of errors in an unordered list.

What's Next?

This chapter concludes the introductory material in this book—the so-called “core curriculum.” The next section of the book, Chapters 8 to 12, goes into more detail about advanced Django usage, including how to deploy a Django application (Chapter 12).

After these first seven chapters, you should know enough to start writing your own Django projects. The rest of the material in this book will help fill in the missing pieces. We'll start in Chapter 8 by doubling back and taking a closer look at views and URLconfs (introduced first in Chapter 3).



Advanced Views and URLconfs

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

URLconf Tricks

There’s nothing “special” about URLconfs—like anything else in Django, they’re just Python code. You can take advantage of this in several ways, as described in the sections that follow.

Streamlining Function Imports

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

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

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

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

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


```

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

urlpatterns = patterns('',
    (r'^hello/$', views.hello),
    (r'^time/$', views.current_datetime),
    (r'^time/plus/(d{1,2})/$', views.hours_ahead),
)

```

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

```

from django.conf.urls.defaults import *

urlpatterns = patterns('',
    (r'^hello/$', 'mysite.views.hello'),
    (r'^time/$', 'mysite.views.current_datetime'),
    (r'^time/plus/(d{1,2})/$', 'mysite.views.hours_ahead'),
)

```

(Note the quotes around the view names. We're using `'mysite.views.current_datetime'`—with quotes—instead of `mysite.views.current_datetime`.)

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

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

```

from django.conf.urls.defaults import *

urlpatterns = patterns('mysite.views',
    (r'^hello/$', 'hello'),
    (r'^time/$', 'current_datetime'),
    (r'^time/plus/(d{1,2})/$', 'hours_ahead'),
)

```

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

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

Advantages of the string approach are as follows:

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

Advantages of the function object approach are as follows:

- It allows for easy “wrapping” of view functions. See the section “Wrapping View Functions” later in this chapter.
- It’s more “Pythonic”—that is, it’s more in line with Python traditions, such as passing functions as objects.

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

Using Multiple View Prefixes

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

Old:

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

urlpatterns = patterns('',
    (r'^hello/$', 'mysite.views.hello'),
    (r'^time/$', 'mysite.views.current_datetime'),
    (r'^time/plus/(d{1,2})/$', 'mysite.views.hours_ahead'),
    (r'^tag/(w+)/$', 'weblog.views.tag'),
)
```

New:

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

urlpatterns = patterns('mysite.views',
    (r'^hello/$', 'hello'),
    (r'^time/$', 'current_datetime'),
    (r'^time/plus/(d{1,2})/$', 'hours_ahead'),
)

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

All the framework cares about is that there’s a module-level variable called `urlpatterns`. This variable can be constructed dynamically, as we do in this example. We should specifically point out that the objects returned by `patterns()` can be added together, which is something you might not have expected.

Special-Casing URLs in Debug Mode

Speaking of constructing `urlpatterns` dynamically, you might want to take advantage of this technique to alter your `URLconf`'s behavior while in Django's debug mode. To do this, just check the value of the `DEBUG` setting at runtime, like so:

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

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

if settings.DEBUG:
    urlpatterns += patterns('',
        (r'^debuginfo/$', views.debug),
    )
```

In this example, the URL `/debuginfo/` will be available only if your `DEBUG` setting is set to `True`.

Using Named Groups

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

KEYWORD ARGUMENTS VS. POSITIONAL ARGUMENTS

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

For example, consider this simple function:

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

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

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

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

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

Finally, you can mix keyword and positional arguments, as long as all positional arguments are listed before keyword arguments. The following statements are equivalent to the previous examples:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Of course, the benefits of named groups come at the cost of brevity; some developers find the named-group syntax ugly and too verbose. Still, another advantage of named groups is readability, especially by those who aren't intimately familiar with regular expressions or your particular Django application. It's easier to see what's happening, at a glance, in a URLconf that uses named groups.

Understanding the Matching/Grouping Algorithm

A caveat with using named groups in a URLconf is that a single URLconf pattern cannot contain both named and non-named groups. If you do this, Django won't throw any errors, but you'll probably find that your URLs aren't matching as you expect. Specifically, here's the algorithm the URLconf parser follows, with respect to named groups vs. non-named groups in a regular expression:

- If there are any named arguments, it will use those, ignoring non-named arguments.
- Otherwise, it will pass all non-named arguments as positional arguments.
- In both cases, it will pass any extra options as keyword arguments. See the next section for more information.

Passing Extra Options to View Functions

Sometimes you'll find yourself writing view functions that are quite similar, with only a few small differences. For example, say you have two views whose contents are identical except for the templates they use:

```
# urls.py

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

urlpatterns = patterns('',
    (r'^foo/$', views.foo_view),
    (r'^bar/$', views.bar_view),
)

# views.py

from django.shortcuts import render_to_response
from mysite.models import MyModel
```

```
def foo_view(request):
    m_list = MyModel.objects.filter(is_new=True)
    return render_to_response('template1.html', {'m_list': m_list})

def bar_view(request):
    m_list = MyModel.objects.filter(is_new=True)
    return render_to_response('template2.html', {'m_list': m_list})
```

We're repeating ourselves in this code, and that's inelegant. At first, you may think to remove the redundancy by using the same view for both URLs, putting parentheses around the URL to capture it, and checking the URL within the view to determine the template, like so:

```
# urls.py

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

urlpatterns = patterns('',
    (r'^(foo)/$', views.foo_bar_view),
    (r'^(bar)/$', views.foo_bar_view),
)

# views.py

from django.shortcuts import render_to_response
from mysite.models import MyModel

def foo_bar_view(request, url):
    m_list = MyModel.objects.filter(is_new=True)
    if url == 'foo':
        template_name = 'template1.html'
    elif url == 'bar':
        template_name = 'template2.html'
    return render_to_response(template_name, {'m_list': m_list})
```

The problem with that solution, though, is that it couples your URLs to your code. If you decide to rename /foo/ to /fooe/, you'll have to remember to change the view code.

The elegant solution involves an optional URLconf parameter. Each pattern in a URLconf may include a third item: a dictionary of keyword arguments to pass to the view function.

With this in mind, we can rewrite our ongoing example like this:

```
# urls.py

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

urlpatterns = patterns('',
    (r'^foo/$', views.foo_bar_view, {'template_name': 'template1.html'}),
    (r'^bar/$', views.foo_bar_view, {'template_name': 'template2.html'}),
)
```

```
# views.py

from django.shortcuts import render_to_response
from mysite.models import MyModel

def foobar_view(request, template_name):
    m_list = MyModel.objects.filter(is_new=True)
    return render_to_response(template_name, {'m_list': m_list})
```

As you can see, the URLconf in this example specifies `template_name` in the URLconf. The view function treats it as just another parameter.

This extra URLconf options technique is a nice way of sending additional information to your view functions with minimal fuss. As such, it's used by a couple of Django's bundled applications, most notably its generic views system, which we cover in Chapter 11.

The following sections contain a couple of ideas on how you can use the extra URLconf options technique in your own projects.

Faking Captured URLconf Values

Say you have a set of views that match a pattern, along with another URL that doesn't fit the pattern but whose view logic is the same. In this case, you can “fake” the capturing of URL values by using extra URLconf options to handle that extra URL with the same view.

For example, you might have an application that displays some data for a particular day, with URLs such as these:

```
/mydata/jan/01/
/mydata/jan/02/
/mydata/jan/03/
# ...
/mydata/dec/30/
/mydata/dec/31/
```

This is simple enough to deal with—you can capture those in a URLconf like this (using named group syntax):

```
urlpatterns = patterns('',
    (r'^mydata/(?P<month>\w{3})/(?P<day>\d\d)/$', views.my_view),
)
```

And the view function signature would look like this:

```
def my_view(request, month, day):
    # ....
```

This approach is straightforward—it's nothing you haven't seen before. The trick comes in when you want to add another URL that uses `my_view` but whose URL doesn't include a month and/or day.

For example, you might want to add another URL, `/mydata/birthday/`, which would be equivalent to `/mydata/jan/06/`. You can take advantage of extra URLconf options like so:

```
urlpatterns = patterns('',
    (r'^mydata/birthday/$', views.my_view, {'month': 'jan', 'day': '06'}),
    (r'^mydata/(?P<month>w{3})/(?P<day>d\d)/$', views.my_view),
)
```

The cool thing here is that you don't have to change your view function at all. The view function only cares that it *gets* month and day parameters—it doesn't matter whether they come from the URL capturing itself or extra parameters.

Making a View Generic

It's good programming practice to “factor out” commonalities in code. For example, with these two Python functions:

```
def say_hello(person_name):
    print 'Hello, %s' % person_name

def say_goodbye(person_name):
    print 'Goodbye, %s' % person_name
```

we can factor out the greeting to make it a parameter:

```
def greet(person_name, greeting):
    print '%s, %s' % (greeting, person_name)
```

You can apply this same philosophy to your Django views by using extra URLconf parameters.

With this in mind, you can start making higher-level abstractions of your views. Instead of thinking to yourself, “This view displays a list of Event objects,” and “That view displays a list of BlogEntry objects,” realize they're both specific cases of “A view that displays a list of objects, where the type of object is variable.”

Take this code, for example:

```
# urls.py

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

urlpatterns = patterns('',
    (r'^events/$', views.event_list),
    (r'^blog/entries/$', views.entry_list),
)

# views.py

from django.shortcuts import render_to_response
from mysite.models import Event, BlogEntry

def event_list(request):
    obj_list = Event.objects.all()
    return render_to_response('mysite/event_list.html', {'event_list': obj_list})
```



```
def entry_list(request):
    obj_list = BlogEntry.objects.all()
    return render_to_response('mysite/blogentry_list.html',
                              {'entry_list': obj_list})
```

The two views do essentially the same thing: they display a list of objects. So let's factor out the type of object they're displaying:

```
# urls.py

from django.conf.urls.defaults import *
from mysite import models, views

urlpatterns = patterns('',
    (r'^events/$', views.object_list, {'model': models.Event}),
    (r'^blog/entries/$', views.object_list, {'model': models.BlogEntry}),
)

# views.py

from django.shortcuts import render_to_response

def object_list(request, model):
    obj_list = model.objects.all()
    template_name = 'mysite/%s_list.html' % model.__name__.lower()
    return render_to_response(template_name, {'object_list': obj_list})
```

With those small changes, we suddenly have a reusable, model-agnostic view! From now on, anytime we need a view that lists a set of objects, we can simply reuse this `object_list` view rather than writing view code. Here are a couple of notes about what we did:

- We passed the model classes directly, as the `model` parameter. The dictionary of extra URLconf options can pass any type of Python object—not just strings.
- The `model.objects.all()` line is an example of *duck typing*: “If it walks like a duck and talks like a duck, we can treat it like a duck.” Note the code doesn't know what type of object model is; the only requirement is that `model` have an `objects` attribute, which in turn has an `all()` method.
- We used `model.__name__.lower()` in determining the template name. Every Python class has a `__name__` attribute that returns the class name. This feature is useful at times like this, when we don't know the type of class until runtime. For example, the `BlogEntry` class's `__name__` is the string `'BlogEntry'`.
- In a slight difference between this example and the previous example, we passed the generic variable name `object_list` to the template. We could easily change this variable name to be `blogentry_list` or `event_list`, but we've left that as an exercise for the reader.

Because database-driven Web sites have several common patterns, Django comes with a set of “generic views” that use this exact technique to save you time. We cover Django’s built-in generic views in Chapter 11.

Giving a View Configuration Options

If you’re distributing a Django application, chances are that your users will want some degree of configuration. In this case, it’s a good idea to add hooks to your views for any configuration options you think people may want to change. You can use extra URLconf parameters for this purpose.

A common bit of an application to make configurable is the template name:

```
def my_view(request, template_name):
    var = do_something()
    return render_to_response(template_name, {'var': var})
```

Understanding Precedence of Captured Values vs. Extra Options

When there’s a conflict, extra URLconf parameters get precedence over captured parameters. In other words, if your URLconf captures a named-group variable and an extra URLconf parameter includes a variable with the same name, the extra URLconf parameter value will be used.

For example, consider this URLconf:

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

urlpatterns = patterns('',
    (r'^mydata/(?P<id>\d+)/$', views.my_view, {'id': 3}),
)
```

Here, both the regular expression and the extra dictionary include an `id`. The hard-coded `id` gets precedence. That means any request (e.g., `/mydata/2/` or `/mydata/432432/`) will be treated as if `id` is set to 3, regardless of the value captured in the URL.

Astute readers will note that in this case, it’s a waste of time and typing to capture the `id` in the regular expression, because its value will always be overridden by the dictionary’s value. That’s correct; we bring this up only to help you avoid making the mistake.

Using Default View Arguments

Another convenient trick is to specify default parameters for a view’s arguments. This tells the view which value to use for a parameter by default if none is specified.

Here’s an example:

```
# urls.py

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

```
urlpatterns = patterns('',
    (r'^blog/$', views.page),
    (r'^blog/page(?P<num>\d+)/$', views.page),
)

# views.py

def page(request, num='1'):
    # Output the appropriate page of blog entries, according to num.
    # ...
```

Here, both URLpatterns point to the same view—`views.page`—but the first pattern doesn't capture anything from the URL. If the first pattern matches, the `page()` function will use its default argument for `num`, `'1'`. If the second pattern matches, `page()` will use whatever `num` value was captured by the regular expression.

Note We've been careful to set the default argument's value to the *string* `'1'`, not the integer `1`. That's for consistency because any captured value for `num` will always be a string.

It's common to use this technique in conjunction with configuration options, as explained earlier. This example makes a slight improvement to the example in the “Giving a View Configuration Options” section by providing a default value for `template_name`:

```
def my_view(request, template_name='mysite/my_view.html'):
    var = do_something()
    return render_to_response(template_name, {'var': var})
```

Special-Casing Views

Sometimes you'll have a pattern in your URLconf that handles a large set of URLs, but you'll need to special-case one of them. In this case, take advantage of the linear way a URLconf is processed and put the special case first.

For example, you can think of the “add an object” pages in Django's admin site as represented by a URLpattern like this:

```
urlpatterns = patterns('',
    # ...
    (r'^([^\s]+)/([^\s]+)/add/$', views.add_stage),
    # ...
)
```

This matches URLs such as `/myblog/entries/add/` and `/auth/groups/add/`. However, the “add” page for a user object (`/auth/user/add/`) is a special case—it doesn't display all of the form fields, it displays two password fields, and so forth. We *could* solve this problem by special-casing in the view, like so:

```
def add_stage(request, app_label, model_name):
    if app_label == 'auth' and model_name == 'user':
        # do special-case code
    else:
        # do normal code
```

but that's inelegant for a reason we've touched on multiple times in this chapter: it puts URL logic in the view. As a more elegant solution, we can take advantage of the fact that URLconfs are processed in order from top to bottom:

```
urlpatterns = patterns('',
    # ...
    ('^auth/user/add/$', views.user_add_stage),
    ('^([^\s]+)/([^\s]+)/add/$', views.add_stage),
    # ...
)
```

With this in place, a request to `/auth/user/add/` will be handled by the `user_add_stage` view. Although that URL matches the second pattern, it matches the top one first. (This is short-circuit logic.)

Capturing Text in URLs

Each captured argument is sent to the view as a plain Python Unicode string, regardless of what sort of match the regular expression makes. For example, in this URLconf line, the year argument to `views.year_archive()` will be a string, not an integer, even though `\d{4}` will only match integer strings:

```
(r'^articles/(?P<year>\d{4})/$', views.year_archive),
```

This is important to keep in mind when you're writing view code. Many built-in Python functions are fussy (and rightfully so) about accepting only objects of a certain type. A common error is to attempt to create a `datetime.date` object with string values instead of integer values:

```
>>> import datetime
>>> datetime.date('1993', '7', '9')
Traceback (most recent call last):
...
TypeError: an integer is required
>>> datetime.date(1993, 7, 9)
datetime.date(1993, 7, 9)
```

Translated to a URLconf and view, the error looks like this:

```
# urls.py

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

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

```
# views.py
```

```
import datetime
```

```
def day_archive(request, year, month, day):
    # The following statement raises a TypeError!
    date = datetime.date(year, month, day)
```

Instead, `day_archive()` can be written correctly like this:

```
def day_archive(request, year, month, day):
    date = datetime.date(int(year), int(month), int(day))
```

Note that `int()` itself raises a `ValueError` when you pass it a string that is not composed solely of digits, but we're avoiding that error in this case because the regular expression in our `URLconf` has ensured that only strings containing digits are passed to the view function.

Determining What the URLconf Searches Against

When a request comes in, Django tries to match the `URLconf` patterns against the requested URL, as a Python string. This does not include GET or POST parameters, or the domain name. It also does not include the leading slash, because every URL has a leading slash.

For example, in a request to `http://www.example.com/myapp/`, Django will try to match `myapp/`. In a request to `http://www.example.com/myapp/?page=3`, Django will try to match `myapp/`.

The request method (e.g., POST, GET) is *not* taken into account when traversing the `URLconf`. In other words, all request methods will be routed to the same function for the same URL. It's the responsibility of a view function to perform branching based on the request method.

Higher-Level Abstractions of View Functions

And speaking of branching based on the request method, let's take a look at how we might build a nice way of doing that. Consider this `URLconf/view` layout:

```
# urls.py
```

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

```
urlpatterns = patterns('',
    # ...
    (r'^somepage/$', views.some_page),
    # ...
)
```

```
# views.py
```

```

from django.http import Http404, HttpResponseRedirect
from django.shortcuts import render_to_response

def some_page(request):
    if request.method == 'POST':
        do_something_for_post()
        return HttpResponseRedirect('/someurl/')
    elif request.method == 'GET':
        do_something_for_get()
        return render_to_response('page.html')
    else:
        raise Http404()

```

In this example, the `some_page()` view's handling of POST vs. GET requests is quite different. The only thing they have in common is a shared URL: `/somepage/`. As such, it's kind of inelegant to deal with both POST and GET in the same view function. It would be nice if we could have two separate view functions—one handling GET requests and the other handling POST—and ensuring that each one was called only when appropriate.

We can do that by writing a view function that delegates to other views, either before or after executing some custom logic. Here's an example of how this technique could help simplify our `some_page()` view:

```

# views.py

from django.http import Http404, HttpResponseRedirect
from django.shortcuts import render_to_response

def method_splitter(request, GET=None, POST=None):
    if request.method == 'GET' and GET is not None:
        return GET(request)
    elif request.method == 'POST' and POST is not None:
        return POST(request)
    raise Http404

def some_page_get(request):
    assert request.method == 'GET'
    do_something_for_get()
    return render_to_response('page.html')

def some_page_post(request):
    assert request.method == 'POST'
    do_something_for_post()
    return HttpResponseRedirect('/someurl/')

# urls.py

```

```

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

```

```
urlpatterns = patterns('',
    # ...
    (r'^somepage/$', views.method_splitter,
     {'GET': views.some_page_get, 'POST': views.some_page_post}),
    # ...
)
```

Let's go through what this does:

- We wrote a new view, `method_splitter()`, that delegates to other views based on `request.method`. It looks for two keyword arguments, GET and POST, which should be *view functions*. If `request.method` is 'GET', it calls the GET view. If `request.method` is 'POST', it calls the POST view. If `request.method` is something else (HEAD, and so on), or if GET or POST were not supplied to the function, it raises an `Http404`.
- In the URLconf, we point `/somepage/` at `method_splitter()` and pass it extra arguments—the view functions to use for GET and POST, respectively.
- Finally, we split the `some_page()` view into two view functions: `some_page_get()` and `some_page_post()`. This is much nicer than shoving all that logic into a single view.

Note These view functions technically no longer have to check `request.method` because `method_splitter()` does that. (By the time `some_page_post()` is called, for example, we can be confident that `request.method` is 'POST'.) Still, just to be safe, and also to serve as documentation, we stuck in an `assert`, ensuring that `request.method` is what we expect it to be.

Now we have a nice generic view function that encapsulates the logic of delegating a view by `request.method`. Nothing about `method_splitter()` is tied to our specific application, of course, so we can reuse it in other projects.

But there's one way to improve on `method_splitter()`. As it's written, it assumes that the GET and POST views take no arguments other than `request`. What if we wanted to use `method_splitter()` with views that, for example, capture text from URLs or take optional keyword arguments?

To do that, we can use a nice Python feature: variable arguments with asterisks. We'll show the example first and then explain it:

```
def method_splitter(request, *args, **kwargs):
    get_view = kwargs.pop('GET', None)
    post_view = kwargs.pop('POST', None)
    if request.method == 'GET' and get_view is not None:
        return get_view(request, *args, **kwargs)
    elif request.method == 'POST' and post_view is not None:
        return post_view(request, *args, **kwargs)
    raise Http404
```

Here, we refactored `method_splitter()` to remove the GET and POST keyword arguments in favor of `*args` and `**kwargs` (note the asterisks). This is a Python feature that allows a function to accept a dynamic arbitrary number of arguments whose names aren't known until runtime. If you put a single asterisk in front of a parameter in a function definition, any *positional* arguments to that function will be rolled up into a single tuple. If you put two asterisks in front of a parameter in a function definition, any *keyword* arguments to that function will be rolled up into a single dictionary.

For example, note this function:

```
def foo(*args, **kwargs):
    print "Positional arguments are:"
    print args
    print "Keyword arguments are:"
    print kwargs
```

Here's how it would work:

```
>>> foo(1, 2, 3)
Positional arguments are:
(1, 2, 3)
Keyword arguments are:
{}
>>> foo(1, 2, name='Adrian', framework='Django')
Positional arguments are:
(1, 2)
Keyword arguments are:
{'framework': 'Django', 'name': 'Adrian'}
```

Bringing this back to `method_splitter()`, you can see we're using `*args` and `**kwargs` to accept *any* arguments to the function and pass them along to the appropriate view. But before we do that, we make two calls to `kwargs.pop()` to get the GET and POST arguments, if they're available. (We're using `pop()` with a default value of `None` to avoid `KeyError` if one or the other isn't defined.)

Wrapping View Functions

Our final view trick takes advantage of an advanced Python technique. Suppose that you find yourself repeating a bunch of code throughout various views, as in this example:

```
def my_view1(request):
    if not request.user.is_authenticated():
        return HttpResponseRedirect('/accounts/login/')
    # ...
    return render_to_response('template1.html')

def my_view2(request):
    if not request.user.is_authenticated():
        return HttpResponseRedirect('/accounts/login/')
    # ...
    return render_to_response('template2.html')
```



```
def my_view3(request):
    if not request.user.is_authenticated():
        return HttpResponseRedirect('/accounts/login/')
    # ...
    return render_to_response('template3.html')
```

Here, each view starts by checking that `request.user` is authenticated—that is, the current user has successfully logged into the site—and redirects to `/accounts/login/` if not.

Note We haven't yet covered `request.user`—Chapter 14 does—but `request.user` represents the current user, either logged-in or anonymous.

It would be nice if we could remove that bit of repetitive code from each of these views and just mark them as requiring authentication. We can do that by making a view wrapper. Take a moment to study this:

```
def requires_login(view):
    def new_view(request, *args, **kwargs):
        if not request.user.is_authenticated():
            return HttpResponseRedirect('/accounts/login/')
        return view(request, *args, **kwargs)
    return new_view
```

This function, `requires_login`, takes a view function (`view`) and returns a new view function (`new_view`). The new function, `new_view`, is defined *within* `requires_login` and handles the logic of checking `request.user.is_authenticated()` and delegating to the original view (`view`).

Now, we can remove the `if not request.user.is_authenticated()` checks from our views and simply wrap them with `requires_login` in our URLconf:

```
from django.conf.urls.defaults import *
from mysite.views import requires_login, my_view1, my_view2, my_view3

urlpatterns = patterns('',
    (r'^view1/$', requires_login(my_view1)),
    (r'^view2/$', requires_login(my_view2)),
    (r'^view3/$', requires_login(my_view3)),
)
```

This has the same effect as before, but with less code redundancy. Now we've created a nice generic function—`requires_login()` that we can wrap around any view in order to make it require a login.

Including Other URLconfs

If you intend your code to be used on multiple Django-based sites, you should consider arranging your URLconfs in such a way that allows for “including.”

At any point, your URLconf can “include” other URLconf modules. This essentially “roots” a set of URLs below other ones. For example, this URLconf includes other URLconfs:

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

urlpatterns = patterns('',
    (r'^weblog/', include('mysite.blog.urls')),
    (r'^photos/', include('mysite.photos.urls')),
    (r'^about/$', 'mysite.views.about'),
)
```

You saw this before in Chapter 6, when we introduced the Django admin site. The admin site has its own URLconf that you merely include() within yours.

There’s an important gotcha here: the regular expressions in this example that point to an include() do *not* have a \$ (end-of-string match character) but *do* include a trailing slash. Whenever Django encounters include(), it chops off whatever part of the URL matched up to that point and sends the remaining string to the included URLconf for further processing.

Continuing this example, here’s the URLconf `mysite.blog.urls`:

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

urlpatterns = patterns('',
    (r'^(\d\d\d\d)/$', 'mysite.blog.views.year_detail'),
    (r'^(\d\d\d\d)/(\d\d)/$', 'mysite.blog.views.month_detail'),
)
```

With these two URLconfs, here’s how a few sample requests would be handled:

- `/weblog/2007/`: In the first URLconf, the pattern `r'^weblog/'` matches. Because it is an include(), Django strips all the matching text, which is `'weblog/'` in this case. The remaining part of the URL is `2007/`, which matches the first line in the `mysite.blog.urls` URLconf.
- `/weblog//2007/` (with two slashes): In the first URLconf, the pattern `r'^weblog/'` matches. Because it is an include(), Django strips all the matching text, which is `'weblog/'` in this case. The remaining part of the URL is `/2007/` (with a leading slash), which does not match any of the lines in the `mysite.blog.urls` URLconf.
- `/about/`: This matches the view `mysite.views.about` in the first URLconf, demonstrating that you can mix include() patterns with non-include() patterns.

How Captured Parameters Work with include()

An included URLconf receives any captured parameters from parent URLconfs, for example:

```
# root urls.py

from django.conf.urls.defaults import *

urlpatterns = patterns('',
    (r'^(?P<username>\w+)/blog/', include('foo.urls.blog')),
)
```

```
# foo/urls/blog.py

from django.conf.urls.defaults import *

urlpatterns = patterns('',
    (r'^$', 'foo.views.blog_index'),
    (r'^archive/$', 'foo.views.blog_archive'),
)
```

In this example, the captured username variable is passed to the included URLconf and, hence, to *every* view function within that URLconf.

Note that the captured parameters will *always* be passed to *every* line in the included URLconf, regardless of whether the line's view actually accepts those parameters as valid. For this reason, this technique is useful only if you're certain that every view in the included URLconf accepts the parameters you're passing.

How Extra URLconf Options Work with include()

Similarly, you can pass extra URLconf options to `include()`, just as you can pass extra URLconf options to a normal view—as a dictionary. When you do this, *each* line in the included URLconf will be passed the extra options.

For example, the following two URLconf sets are functionally identical.

Set one:

```
# urls.py

from django.conf.urls.defaults import *

urlpatterns = patterns('',
    (r'^blog/', include('inner'), {'blogid': 3}),
)
```

```
# inner.py

from django.conf.urls.defaults import *

urlpatterns = patterns('',
    (r'^archive/$', 'mysite.views.archive'),
    (r'^about/$', 'mysite.views.about'),
    (r'^rss/$', 'mysite.views.rss'),
)
```

Set two:

```
# urls.py

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

```
urlpatterns = patterns('',
    (r'^blog/', include('inner')),
)

# inner.py

from django.conf.urls.defaults import *

urlpatterns = patterns('',
    (r'^archive/$', 'mysite.views.archive', {'blogid': 3}),
    (r'^about/$', 'mysite.views.about', {'blogid': 3}),
    (r'^rss/$', 'mysite.views.rss', {'blogid': 3}),
)
```

As is the case with captured parameters (explained in the previous section), extra options will *always* be passed to *every* line in the included URLconf, regardless of whether the line's view actually accepts those options as valid. For this reason, this technique is useful only if you're certain that every view in the included URLconf accepts the extra options you're passing.

What's Next?

This chapter provided many advanced tips and tricks for views and URLconfs. In Chapter 9, we'll give this advanced treatment to Django's template system.



Advanced Templates

Although most of your interactions with Django’s template language will be in the role of template author, you may want to customize and extend the template engine—either to make it do something it doesn’t already do, or to make your job easier in some other way.

This chapter delves deep into the guts of Django’s template system. It covers what you need to know if you plan to extend the system or if you’re just curious about how it works. It also covers the autoescaping feature, a security measure you’ll no doubt notice over time as you continue to use Django.

If you’re looking to use the Django template system as part of another application (i.e., without the rest of the framework), make sure to read the “Configuring the Template System in Standalone Mode” section later in the chapter.

Template Language Review

First, let’s quickly review a number of terms introduced in Chapter 4:

- A *template* is a text document, or a normal Python string, that is marked up using the Django template language. A template can contain template tags and variables.
- A *template tag* is a symbol within a template that does something. This definition is deliberately vague. For example, a template tag can produce content, serve as a control structure (an if statement or a for loop), grab content from a database, or enable access to other template tags.

Template tags are surrounded by `{%` and `%}`:

```
{% if is_logged_in %}  
    Thanks for logging in!  
{% else %}  
    Please log in.  
{% endif %}
```

- A *variable* is a symbol within a template that outputs a value.

Variable tags are surrounded by `{{` and `}}`:

My first name is `{{ first_name }}`. My last name is `{{ last_name }}`.

- A *context* is a name-value mapping (similar to a Python dictionary) that is passed to a template.
- A template *renders* a context by replacing the variable “holes” with values from the context and executing all template tags.

For more details about the basics of these terms, refer back to Chapter 4.

The rest of this chapter discusses ways of extending the template engine. First, though, let’s take a quick look at a few internals left out of Chapter 4 for simplicity.

RequestContext and Context Processors

When rendering a template, you need a context. Usually this is an instance of `django.template.Context`, but Django also comes with a special subclass, `django.template.RequestContext`, that acts slightly differently. `RequestContext` adds a bunch of variables to your template context by default—things like the `HttpRequest` object or information about the currently logged-in user.

Use `RequestContext` when you don’t want to have to specify the same set of variables in a series of templates. For example, consider these two views:

```
from django.template import loader, Context

def view_1(request):
    # ...
    t = loader.get_template('template1.html')
    c = Context({
        'app': 'My app',
        'user': request.user,
        'ip_address': request.META['REMOTE_ADDR'],
        'message': 'I am view 1.'
    })
    return t.render(c)

def view_2(request):
    # ...
    t = loader.get_template('template2.html')
    c = Context({
        'app': 'My app',
        'user': request.user,
        'ip_address': request.META['REMOTE_ADDR'],
        'message': 'I am the second view.'
    })
    return t.render(c)
```

(Note that we’re deliberately *not* using the `render_to_response()` shortcut in these examples—we’re manually loading the templates, constructing the context objects, and rendering the templates. We’re “spelling out” all of the steps for the purpose of clarity.)

Each view passes the same three variables—`app`, `user`, and `ip_address`—to its template. Wouldn’t it be nice if we could remove that redundancy?

`RequestContext` and *context processors* were created to solve this problem. Context processors let you specify a number of variables that get set in each context automatically—without you having to specify the variables in each `render_to_response()` call. The catch is that you have to use `RequestContext` instead of `Context` when you render a template.

The most low-level way of using context processors is to create some processors and pass them to `RequestContext`. Here’s how the preceding example could be written with context processors:

```
from django.template import loader, RequestContext

def custom_proc(request):
    "A context processor that provides 'app', 'user' and 'ip_address'."
    return {
        'app': 'My app',
        'user': request.user,
        'ip_address': request.META['REMOTE_ADDR']
    }

def view_1(request):
    # ...
    t = loader.get_template('template1.html')
    c = RequestContext(request, {'message': 'I am view 1.'},
                       processors=[custom_proc])
    return t.render(c)

def view_2(request):
    # ...
    t = loader.get_template('template2.html')
    c = RequestContext(request, {'message': 'I am the second view.'},
                       processors=[custom_proc])
    return t.render(c)
```

Let’s step through this code:

- First, we define a function `custom_proc`. This is a context processor—it takes an `HttpRequest` object and returns a dictionary of variables to use in the template context. That’s all it does.
- We’ve changed the two view functions to use `RequestContext` instead of `Context`. There are two differences in how the context is constructed. First, `RequestContext` requires the first argument to be an `HttpRequest` object—the one that was passed into the view function in the first place (`request`). Second, `RequestContext` takes an optional `processors` argument, which is a list or tuple of context processor functions to use. Here, we pass in `custom_proc`, the custom processor we defined earlier.

- Each view no longer has to include `app`, `user`, or `ip_address` in its context construction, because those are provided by `custom_proc`.
- Each view *still* has the flexibility to introduce any custom template variables it might need. In this example, the message template variable is set differently in each view.

In Chapter 4, we introduced the `render_to_response()` shortcut, which saves you from having to call `loader.get_template()`, then create a `Context`, then call the `render()` method on the template. In order to demonstrate the lower-level workings of context processors, the previous examples didn't use `render_to_response()`. But it's possible—and preferable—to use context processors with `render_to_response()`. Do this with the `context_instance` argument, like so:

```
from django.shortcuts import render_to_response
from django.template import RequestContext

def custom_proc(request):
    "A context processor that provides 'app', 'user' and 'ip_address'."
    return {
        'app': 'My app',
        'user': request.user,
        'ip_address': request.META['REMOTE_ADDR']
    }

def view_1(request):
    # ...
    return render_to_response('template1.html',
        {'message': 'I am view 1.'},
        context_instance=RequestContext(request, processors=[custom_proc]))

def view_2(request):
    # ...
    return render_to_response('template2.html',
        {'message': 'I am the second view.'},
        context_instance=RequestContext(request, processors=[custom_proc]))
```

Here, we've trimmed down each view's template-rendering code to a single (wrapped) line.

This is an improvement, but, evaluating the conciseness of this code, we have to admit we're now almost overdosing on the *other* end of the spectrum. We've removed redundancy in data (our template variables) at the cost of adding redundancy in code (in the `processors` call). Using context processors doesn't save you much typing if you have to type `processors` all the time.

For that reason, Django provides support for *global* context processors. The `TEMPLATE_CONTEXT_PROCESSORS` setting (in your `settings.py`) designates which context processors should *always* be applied to `RequestContext`. This removes the need to specify `processors` each time you use `RequestContext`.

By default, `TEMPLATE_CONTEXT_PROCESSORS` is set to the following:

```
TEMPLATE_CONTEXT_PROCESSORS = (  
    'django.core.context_processors.auth',  
    'django.core.context_processors.debug',  
    'django.core.context_processors.i18n',  
    'django.core.context_processors.media',  
)
```

This setting is a tuple of callables that use the same interface as the preceding `custom_proc` function—functions that take a request object as their argument and return a dictionary of items to be merged into the context. Note that the values in `TEMPLATE_CONTEXT_PROCESSORS` are specified as *strings*, which means the processors are required to be somewhere on your Python path (so you can refer to them from the setting).

Each processor is applied in order. That is, if one processor adds a variable to the context and a second processor adds a variable with the same name, the second will override the first.

Django provides a number of simple context processors, including the ones that are enabled by default.

django.core.context_processors.auth

If `TEMPLATE_CONTEXT_PROCESSORS` contains this processor, every `RequestContext` will contain these variables:

- `user`: A `django.contrib.auth.models.User` instance representing the current logged-in user (or an `AnonymousUser` instance, if the client isn't logged in).
- `messages`: A list of messages (as strings) for the current logged-in user. Behind the scenes, this variable calls `request.user.get_and_delete_messages()` for every request. That method collects the user's messages and deletes them from the database.
- `perms`: An instance of `django.core.context_processors.PermWrapper`, which represents the permissions the current logged-in user has.

See Chapter 14 for more information on users, permissions, and messages.

django.core.context_processors.debug

This processor pushes debugging information down to the template layer. If `TEMPLATE_CONTEXT_PROCESSORS` contains this processor, every `RequestContext` will contain these variables:

- `debug`: The value of your `DEBUG` setting (either `True` or `False`). You can use this variable in templates to test whether you're in debug mode.
- `sql_queries`: A list of `{ 'sql': ..., 'time': ... }` dictionaries representing every SQL query that has happened so far during the request and how long it took. The list is in the order in which the queries were issued.

Because debugging information is sensitive, this context processor will add variables to the context only if both of the following conditions are true:

- The `DEBUG` setting is `True`.
- The request came from an IP address in the `INTERNAL_IPS` setting.

Astute readers will notice that the debug template variable will never have the value `False` because if `DEBUG` is `False`, the debug template variable won't be populated in the first place.

`django.core.context_processors.i18n`

If this processor is enabled, every `RequestContext` will contain these variables:

- `LANGUAGES`: The value of the `LANGUAGES` setting.
- `LANGUAGE_CODE`: `request.LANGUAGE_CODE` if it exists; otherwise, the value of the `LANGUAGE_CODE` setting.

Appendix D provides more information about these two settings.

`django.core.context_processors.request`

If this processor is enabled, every `RequestContext` will contain a variable `request`, which is the current `HttpRequest` object. Note that this processor is not enabled by default; you have to activate it.

You might want to use this if you find your templates needing to access attributes of the current `HttpRequest` such as the IP address:

```
{{ request.REMOTE_ADDR }}
```

Guidelines for Writing Your Own Context Processors

Here are a few tips for rolling your own:

- Make each context processor responsible for the smallest subset of functionality possible. It's easy to use multiple processors, so you might as well split functionality into logical pieces for future reuse.
- Keep in mind that any context processor in `TEMPLATE_CONTEXT_PROCESSORS` will be available in *every* template powered by that settings file, so try to pick variable names that are unlikely to conflict with variable names your templates might be using independently. Because variable names are case-sensitive, it's not a bad idea to use all uppercase letters for variables that a processor provides.
- It doesn't matter where on the filesystem the processors live, as long as they're on your Python path so you can point to them from the `TEMPLATE_CONTEXT_PROCESSORS` setting. With that said, the convention is to save them in a file called `context_processors.py` within your app or project.

Automatic HTML Escaping

When generating HTML from templates, there's always a risk that a variable will include characters that affect the resulting HTML. For example, consider this template fragment:

```
Hello, {{ name }}.
```

At first, it seems like a harmless way to display a user's name, but consider what would happen if the user entered his name this way:

```
<script>alert('hello')</script>
```

With this name value, the template would be rendered as follows:

```
Hello, <script>alert('hello')</script>
```

This means the browser would display a JavaScript alert box! Similarly, what if the name contained a '`<`' symbol, like this?

```
<b>username
```

It would result in a rendered template like this:

```
Hello, <b>username
```

This, in turn, would result in the remainder of the Web page being bold!

Clearly, user-submitted data shouldn't be trusted blindly and inserted directly into your Web pages because a malicious user could use this kind of hole to do potentially bad things. This type of security exploit is called a cross-site scripting (XSS) attack.

Tip For more on security, see Chapter 20.

To avoid this problem, you have two options:

- You can make sure to run each untrusted variable through the escape filter, which converts potentially harmful HTML characters to unarmful ones. This was the default solution in Django for its first few years, but the problem is that it puts the onus on *you*, the developer/template author, to ensure that you're escaping everything. It's easy to forget to escape data.
- You can take advantage of Django's automatic HTML escaping. The remainder of this section describes how autoescaping works.

By default, in Django every template automatically escapes the output of every variable tag. Specifically, these five characters are escaped:

- < is converted to <
- > is converted to >
- ' (single quote) is converted to '
- " (double quote) is converted to "
- & is converted to &

Again, we stress that this behavior is on by default. If you're using Django's template system, you're protected.

How to Turn It Off

If you don't want data to be autoescaped on a per-site, per-template, or per-variable level, you can turn it off in several ways.

Why would you want to turn it off? Because sometimes template variables contain data that you *intend* to be rendered as raw HTML, in which case you don't want their contents to be escaped. For example, you might store a blob of trusted HTML in your database and want to embed it directly into your template. Or you might be using Django's template system to produce text that is *not* HTML—like an e-mail message, for instance.

For Individual Variables

To disable autoescaping for an individual variable, use the `safe` filter:

```
This will be escaped: {{ data }}
This will not be escaped: {{ data|safe }}
```

Think of *safe* as shorthand for *safe from further escaping or can be safely interpreted as HTML*. In this example, if data contains '``', the output will be the following:

```
This will be escaped: &lt;b&gt;
This will not be escaped: <b>
```

For Template Blocks

To control autoescaping for a template, wrap the template (or just a particular section of the template) in the `autoescape` tag, like so:

```
{% autoescape off %}
    Hello {{ name }}
{% endautoescape %}
```

The autoescape tag takes either on or off as its argument. At times, you might want to force autoescaping when it would otherwise be disabled. Here is an example template:

```
Autoescaping is on by default. Hello {{ name }}
```

```
{% autoescape off %}
    This will not be autoescaped: {{ data }}.
```

```

    Nor this: {{ other_data }}
{% autoescape on %}
    Autoescaping applies again: {{ name }}
{% endautoescape %}
{% endautoescape %}
```

The autoescaping tag passes its effect on to templates that extend the current one as well as templates included via the include tag, just like all block tags. For example:

```
# base.html
```

```
{% autoescape off %}
<h1>{% block title %}{% endblock %}</h1>
{% block content %}
{% endblock %}
{% endautoescape %}
```

```
# child.html
```

```
{% extends "base.html" %}
{% block title %}This & that{% endblock %}
{% block content %}{{ greeting }}{% endblock %}
```

Because autoescaping is turned off in the base template, it will also be turned off in the child template, resulting in the following rendered HTML when the greeting variable contains the string `Hello!`:

```
<h1>This & that</h1>
<b>Hello!</b>
```

Notes

Template authors usually don't need to worry about autoescaping very much. Developers on the Python side (people writing views and custom filters) need to think about the cases in which data shouldn't be escaped, and mark data appropriately, so things work in the template.

If you're creating a template that might be used in situations in which you're not sure whether autoescaping is enabled, add an escape filter to any variable that needs escaping. When autoescaping is on, there's no danger of the escape filter *double-escaping* data—the escape filter does not affect autoescaped variables.

Automatic Escaping of String Literals in Filter Arguments

As mentioned earlier, filter arguments can be strings:

```
{{ data|default:"This is a string literal." }}
```

All string literals are inserted *without* any automatic escaping into the template—they act as if they were all passed through the safe filter. The reasoning behind this is that the template author is in control of what goes into the string literal, so they can make sure the text is correctly escaped when the template is written.

This means you would write the following:

```
{{ data|default:"3 &lt; 2" }}
```

instead of the following:

```
{{ data|default:"3 < 2" }} <-- Bad! Don't do this.
```

This doesn't affect what happens to data coming from the variable itself. The variable's contents are still automatically escaped, if necessary, because they're beyond the control of the template author.

Inside Template Loading

Generally, you'll store templates in files on your filesystem, but you can use custom *template loaders* to load templates from other sources.

Django has two ways to load templates:

- `django.template.loader.get_template(template_name)`: `get_template` returns the compiled template (a `Template` object) for the template with the given name. If the template doesn't exist, a `TemplateDoesNotExist` exception will be raised.
- `django.template.loader.select_template(template_name_list)`: `select_template` is just like `get_template`, except it takes a list of template names. Of the list, it returns the first template that exists. If none of the templates exist, a `TemplateDoesNotExist` exception will be raised.

As covered in Chapter 4, each of these functions by default uses your `TEMPLATE_DIRS` setting to load templates. Internally, however, these functions actually delegate to a template loader for the heavy lifting.

Some of loaders are disabled by default, but you can activate them by editing the `TEMPLATE_LOADERS` setting. `TEMPLATE_LOADERS` should be a tuple of strings, where each string represents a template loader. These template loaders ship with Django:

- `django.template.loaders.filesystem.load_template_source`: This loader loads templates from the filesystem, according to `TEMPLATE_DIRS`. It is enabled by default.
- `django.template.loaders.app_directories.load_template_source`: This loader loads templates from Django applications on the filesystem. For each application in `INSTALLED_APPS`, the loader looks for a `templates` subdirectory. If the directory exists, Django looks for templates there.

This means you can store templates with your individual applications, making it easy to distribute Django applications with default templates. For example, if `INSTALLED_APPS` contains `('myproject.polls', 'myproject.music')`, then `get_template('foo.html')` will look for templates in this order:

- `/path/to/myproject/polls/templates/foo.html`
- `/path/to/myproject/music/templates/foo.html`

Note that the loader performs an optimization when it is first imported: it caches a list of which `INSTALLED_APPS` packages have a `templates` subdirectory.

This loader is enabled by default.

- `django.template.loaders.eggs.load_template_source`: This loader is just like `app_directories`, except it loads templates from Python eggs rather than from the filesystem. This loader is disabled by default; you'll need to enable it if you're using eggs to distribute your application. (Python eggs are a way of compressing Python code into a single file.)

Django uses the template loaders in order according to the `TEMPLATE_LOADERS` setting. It uses each loader until a loader finds a match.

Extending the Template System

Now that you understand a bit more about the internals of the template system, let's look at how to extend the system with custom code.

Most template customization comes in the form of custom template tags and/or filters. Although the Django template language comes with many built-in tags and filters, you'll probably assemble your own libraries of tags and filters that fit your own needs. Fortunately, it's quite easy to define your own functionality.

Creating a Template Library

Whether you're writing custom tags or filters, the first thing to do is to create a *template library*—a small bit of infrastructure Django can hook into.

Creating a template library is a two-step process:

1. First, decide which Django application should house the template library. If you've created an app via `manage.py startapp`, you can put it in there, or you can create another app solely for the template library. We recommend the latter because your filters might be useful to you in future projects.

Whichever route you take, make sure to add the app to your `INSTALLED_APPS` setting. We'll explain this shortly.

2. Second, create a `templatetags` directory in the appropriate Django application's package. It should be on the same level as `models.py`, `views.py`, and so forth. For example:

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

Create two empty files in the `templatetags` directory: an `__init__.py` file (to indicate to Python that this is a package containing Python code) and a file that will contain your custom tag/filter definitions. The name of the latter file is what you'll use to load the tags later. For example, if your custom tags/filters are in a file called `poll_extras.py`, you'd write the following in a template:

```
{% load poll_extras %}
```

The `{% load %}` tag looks at your `INSTALLED_APPS` setting and only allows the loading of template libraries within installed Django applications. This is a security feature; it allows you to host Python code for many template libraries on a single computer without enabling access to all of them for every Django installation.

If you write a template library that isn't tied to any particular models/views, it's valid and quite normal to have a Django application package that contains only a `templatetags` package. There's no limit on how many modules you put in the `templatetags` package. Just keep in mind that a `{% load %}` statement will load tags/filters for the given Python module name, not the name of the application.

Once you've created that Python module, you'll just have to write a bit of Python code, depending on whether you're writing filters or tags.

To be a valid tag library, the module must contain a module-level variable named `register` that is an instance of `template.Library`. This is the data structure in which all the tags and filters are registered. So, near the top of your module, insert the following:

```
from django import template

register = template.Library()
```

Note For a fine selection of examples, read the source code for Django's default filters and tags. They're in `django/template/defaultfilters.py` and `django/template/defaulttags.py`, respectively. Some applications in `django.contrib` also contain template libraries.

Once you've created this `register` variable, you'll use it to create template filters and tags.

Writing Custom Template Filters

Custom filters are just Python functions that take one or two arguments:

- The value of the variable (input)
- The value of the argument, which can have a default value or be left out altogether

For example, in the filter `{{ var|foo:"bar" }}`, the filter `foo` would be passed the contents of the variable `var` and the argument `"bar"`.

Filter functions should always return something. They shouldn't raise exceptions, and they should fail silently. If there's an error, they should return either the original input or an empty string, whichever makes more sense.

Here's an example filter definition:

```
def cut(value, arg):
    "Removes all values of arg from the given string"
    return value.replace(arg, '')
```

And here's an example of how that filter would be used to cut spaces from a variable's value:

```
{{ somevariable|cut:" " }}
```

Most filters don't take arguments. In this case, just leave the argument out of your function:

```
def lower(value): # Only one argument.
    "Converts a string into all lowercase"
    return value.lower()
```

When you've written your filter definition, you need to register it with your `Library` instance, to make it available to Django's template language:

```
register.filter('cut', cut)
register.filter('lower', lower)
```

The `Library.filter()` method takes two arguments:

- The name of the filter (a string)
- The filter function itself

If you're using Python 2.4 or above, you can use `register.filter()` as a decorator instead:

```
@register.filter(name='cut')
def cut(value, arg):
    return value.replace(arg, '')
```

```
@register.filter
def lower(value):
    return value.lower()
```

If you leave off the name argument, as in the second example, Django will use the function's name as the filter name.

Here, then, is a complete template library example, supplying the cut filter:

```
from django import template

register = template.Library()

@register.filter(name='cut')
def cut(value, arg):
    return value.replace(arg, '')
```

Writing Custom Template Tags

Tags are more complex than filters, because tags can do nearly anything.

Chapter 4 describes how the template system works in a two-step process: compiling and rendering. To define a custom template tag, you need to tell Django how to manage *both* of these steps when it gets to your tag.

When Django compiles a template, it splits the raw template text into *nodes*. Each node is an instance of `django.template.Node` and has a `render()` method. Thus, a compiled template is simply a list of `Node` objects. For example, consider this template:

```
Hello, {{ person.name }}.
```

```
{% ifequal name.birthday today %}
    Happy birthday!
{% else %}
    Be sure to come back on your birthday
    for a splendid surprise message.
{% endifequal %}
```

In compiled template form, this template is represented as this list of nodes:

- Text node: "Hello, "
- Variable node: `person.name`
- Text node: ".\n\n"
- IfEqual node: `name.birthday` and `today`

When you call `render()` on a compiled template, the template calls `render()` on each `Node` in its node list, with the given context. The results are all concatenated together to form the output of the template. Thus, to define a custom template tag, you specify how the raw template tag is converted into a `Node` (the compilation function) and what the node's `render()` method does.

In the sections that follow, we cover all the steps in writing a custom tag.

Writing the Compilation Function

For each template tag the parser encounters, it calls a Python function with the tag contents and the parser object itself. This function is responsible for returning a `Node` instance based on the contents of the tag.

For example, let's write a template tag, `{% current_time %}`, that displays the current date/time, formatted according to a parameter given in the tag, in `strftime` syntax (see <http://www.djangoproject.com/r/python/strftime/>). It's a good idea to decide the tag syntax before anything else. In our case, let's say the tag should be used like this:

```
<p>The time is {% current_time "%Y-%m-%d %I:%M %p" %}.</p>
```

Note Yes, this template tag is redundant—Django’s default `{% now %}` tag does the same task with simpler syntax. This template tag is presented here just for example purposes.

The parser for this function should grab the parameter and create a `Node` object:

```
from django import template

register = template.Library()

def do_current_time(parser, token):
    try:
        # split_contents() knows not to split quoted strings.
        tag_name, format_string = token.split_contents()
    except ValueError:
        msg = '%r tag requires a single argument' % token.split_contents()[0]
        raise template.TemplateSyntaxError(msg)
    return CurrentTimeNode(format_string[1:-1])
```

There’s a lot going here:

- Each template tag compilation function takes two arguments: `parser` and `token`. `parser` is the template parser object. We don’t use it in this example. `token` is the token currently being parsed by the parser.
- `token.contents` is a string of the raw contents of the tag. In our example, it’s `'current_time "%Y-%m-%d %I:%M %p"'`.
- The `token.split_contents()` method separates the arguments on spaces while keeping quoted strings together. Avoid using `token.contents.split()` (which just uses Python’s standard string-splitting semantics). It’s not as robust, as it naively splits on *all* spaces, including those within quoted strings.
- This function is responsible for raising `django.template.TemplateSyntaxError`, with helpful messages, for any syntax error.
- Don’t hard-code the tag’s name in your error messages, because that couples the tag’s name to your function. `token.split_contents()[0]` will *always* be the name of your tag—even when the tag has no arguments.
- The function returns a `CurrentTimeNode` (which we’ll create shortly) containing everything the node needs to know about this tag. In this case, it just passes the argument `"%Y-%m-%d %I:%M %p"`. The leading and trailing quotes from the template tag are removed with `format_string[1:-1]`.
- Template tag compilation functions *must* return a `Node` subclass; any other return value is an error.

Writing the Template Node

The second step in writing custom tags is to define a Node subclass that has a `render()` method. Continuing the preceding example, we need to define `CurrentTimeNode`:

```
import datetime

class CurrentTimeNode(template.Node):
    def __init__(self, format_string):
        self.format_string = str(format_string)

    def render(self, context):
        now = datetime.datetime.now()
        return now.strftime(self.format_string)
```

These two functions (`__init__()` and `render()`) map directly to the two steps in template processing (compilation and rendering). Thus, the initialization function only needs to store the format string for later use, and the `render()` function does the real work.

Like template filters, these rendering functions should fail silently instead of raising errors. The only time that template tags are allowed to raise errors is at compilation time.

Registering the Tag

Finally, you need to register the tag with your module's `Library` instance. Registering custom tags is very similar to registering custom filters (as explained previously). Just instantiate a `template.Library` instance and call its `tag()` method. For example:

```
register.tag('current_time', do_current_time)
```

The `tag()` method takes two arguments:

- The name of the template tag (string).
- The compilation function.

As with filter registration, it is also possible to use `register.tag` as a decorator in Python 2.4 and above:

```
@register.tag(name="current_time")
def do_current_time(parser, token):
    # ...

@register.tag
def shout(parser, token):
    # ...
```

If you leave off the `name` argument, as in the second example, Django will use the function's name as the tag name.

Setting a Variable in the Context

The previous section's example simply returned a value. Often it's useful to set template variables instead of returning values. That way, template authors can just use the variables that your template tags set.

To set a variable in the context, use dictionary assignment on the context object in the `render()` method. Here's an updated version of `CurrentTimeNode` that sets a template variable, `current_time`, instead of returning it:

```
class CurrentTimeNode2(template.Node):
    def __init__(self, format_string):
        self.format_string = str(format_string)

    def render(self, context):
        now = datetime.datetime.now()
        context['current_time'] = now.strftime(self.format_string)
        return ''
```

Note We'll leave the creation of a `do_current_time2` function, plus the registration of that function to a `current_time2` template tag, as exercises for you.

Note that `render()` returns an empty string. `render()` should always return a string, so if all the template tag does is set a variable, `render()` should return an empty string.

Here's how you'd use this new version of the tag:

```
{% current_time2 "%Y-%M-%d %I:%M %p" %}
<p>The time is {{ current_time }}.</p>
```

But there's a problem with `CurrentTimeNode2`: the variable name `current_time` is hard-coded. This means you'll need to make sure your template doesn't use `{{ current_time }}` anywhere else, because `{% current_time2 %}` will blindly overwrite that variable's value.

A cleaner solution is to make the template tag specify the name of the variable to be set, like so:

```
{% get_current_time "%Y-%M-%d %I:%M %p" as my_current_time %}
<p>The current time is {{ my_current_time }}.</p>
```

To do so, you'll need to refactor both the compilation function and the `Node` class, as follows:

```
import re

class CurrentTimeNode3(template.Node):
    def __init__(self, format_string, var_name):
        self.format_string = str(format_string)
        self.var_name = var_name
```

```

def render(self, context):
    now = datetime.datetime.now()
    context[self.var_name] = now.strftime(self.format_string)
    return ''

def do_current_time(parser, token):
    # This version uses a regular expression to parse tag contents.
    try:
        # Splitting by None == splitting by spaces.
        tag_name, arg = token.contents.split(None, 1)
    except ValueError:
        msg = '%r tag requires arguments' % token.contents[0]
        raise template.TemplateSyntaxError(msg)

    m = re.search(r'(.*) as (\w+)', arg)
    if m:
        fmt, var_name = m.groups()
    else:
        msg = '%r tag had invalid arguments' % tag_name
        raise template.TemplateSyntaxError(msg)

    if not (fmt[0] == fmt[-1] and fmt[0] in ('"', "'")):
        msg = "%r tag's argument should be in quotes" % tag_name
        raise template.TemplateSyntaxError(msg)

    return CurrentTimeNode3(fmt[1:-1], var_name)

```

Now `do_current_time()` passes the format string and the variable name to `CurrentTimeNode3`.

Parsing Until Another Template Tag

Template tags can work as blocks containing other tags (such as `{% if %}`, `{% for %}`, etc.). To create a template tag like this, use `parser.parse()` in your compilation function.

Here's how the standard `{% comment %}` tag is implemented:

```

def do_comment(parser, token):
   odelist = parser.parse(('endcomment',))
    parser.delete_first_token()
    return CommentNode()

class CommentNode(template.Node):
    def render(self, context):
        return ''

```

`parser.parse()` takes a tuple of names of template tags to parse until. It returns an instance of `django.template.NodeList`, which is a list of all `Node` objects that the parser encountered *before* it encountered any of the tags named in the tuple.

So in the preceding example, `odelist` is a list of all nodes between `{% comment %}` and `{% endcomment %}`, not counting `{% comment %}` and `{% endcomment %}` themselves.

After `parser.parse()` is called, the parser hasn't yet “consumed” the `{% endcomment %}` tag, so the code needs to explicitly call `parser.delete_first_token()` to prevent that tag from being processed twice.

Then `CommentNode.render()` simply returns an empty string. Anything between `{% comment %}` and `{% endcomment %}` is ignored.

Parsing Until Another Template Tag and Saving Contents

In the previous example, `do_comment()` discarded everything between `{% comment %}` and `{% endcomment %}`. It's also possible to do something with the code between template tags instead.

For example, here's a custom template tag, `{% upper %}`, that capitalizes everything between itself and `{% endupper %}`:

```
{% upper %}
    This will appear in uppercase, {{ user_name }}.
{% endupper %}
```

As in the previous example, we'll use `parser.parse()`. This time, we pass the resulting `odelist` to `Node`:

```
def do_upper(parser, token):
    oodelist = parser.parse(('endupper',))
    parser.delete_first_token()
    return UpperNode(oodelist)

class UpperNode(template.Node):
    def __init__(self, oodelist):
        self.oodelist = oodelist

    def render(self, context):
        output = self.oodelist.render(context)
        return output.upper()
```

The only new concept here is `self.oodelist.render(context)` in `UpperNode.render()`. This simply calls `render()` on each `Node` in the node list.

For more examples of complex rendering, see the source code for `{% if %}`, `{% for %}`, `{% ifequal %}`, and `{% ifchanged %}`. They live in `django/template/defaulttags.py`.

Shortcut for Simple Tags

Many template tags take a single argument—a string or a template variable reference—and return a string after doing some processing based solely on the input argument and some external information. For example, the `current_time` tag we wrote earlier is of this variety. We give it a format string, and it returns the time as a string.

To ease the creation of these types of tags, Django provides a helper function, `simple_tag`. This function, which is a method of `django.template.Library`, takes a function that accepts one argument, wraps it in a `render` function and the other necessary bits mentioned previously, and registers it with the template system.

Our earlier `current_time` function could thus be written like this:

```
def current_time(format_string):
    try:
        return datetime.datetime.now().strftime(str(format_string))
    except UnicodeEncodeError:
        return ''

register.simple_tag(current_time)
```

In Python 2.4, the decorator syntax also works:

```
@register.simple_tag
def current_time(token):
    # ...
```

Notice a couple of things about the `simple_tag` helper function:

- Only the (single) argument is passed into our function.
- Checking for the required number of arguments has already been done by the time our function is called, so we don't need to do that.
- The quotes around the argument (if any) have already been stripped away, so we receive a plain Unicode string.

Inclusion Tags

Another common template tag is the type that displays some data by rendering *another* template. For example, Django's admin interface uses custom template tags to display the buttons along the bottom of the “add/change” form pages. Those buttons always look the same, but the link targets change depending on the object being edited. They're a perfect case for using a small template that is filled with details from the current object.

These sorts of tags are called *inclusion tags*. Writing inclusion tags is probably best demonstrated by example. Let's write a tag that produces a list of books for a given Author object. We'll use the tag like this:

```
{% books_for_author author %}
```

The result will be something like this:

```
<ul>
  <li>The Cat In The Hat</li>
  <li>Hop On Pop</li>
  <li>Green Eggs And Ham</li>
</ul>
```

First, we define the function that takes the argument and produces a dictionary of data for the result. Notice that we need to return only a dictionary, not anything more complex. This will be used as the context for the template fragment:


```
def books_for_author(author):
    books = Book.objects.filter(authors__id=author.id)
    return {'books': books}
```

Next, we create the template used to render the tag's output. Following our example, the template is very simple:

```
<ul>
{% for book in books %}
    <li>{{ book.title }}</li>
{% endfor %}
</ul>
```

Finally, we create and register the inclusion tag by calling the `inclusion_tag()` method on a `Library` object.

Following our example, if the preceding template is in a file called `book_snippet.html`, we register the tag like this:

```
register.inclusion_tag('book_snippet.html')(books_for_author)
```

Python 2.4 decorator syntax works as well, so we could have written this instead:

```
@register.inclusion_tag('book_snippet.html')
def books_for_author(author):
    # ...
```

Sometimes, your inclusion tags need access to values from the parent template's context. To solve this, Django provides a `takes_context` option for inclusion tags. If you specify `takes_context` in creating an inclusion tag, the tag will have no required arguments, and the underlying Python function will have one argument: the template context as of when the tag was called.

For example, say you're writing an inclusion tag that will always be used in a context that contains `home_link` and `home_title` variables that point back to the main page. Here's what the Python function would look like:

```
@register.inclusion_tag('link.html', takes_context=True)
def jump_link(context):
    return {
        'link': context['home_link'],
        'title': context['home_title'],
    }
```

Note The first parameter to the function *must* be called `context`.

The template `link.html` might contain the following:

Jump directly to {{ title }}.

Then, anytime you want to use that custom tag, load its library and call it without any arguments, like so:

```
{% jump_link %}
```

Writing Custom Template Loaders

Django’s built-in template loaders (described in the “Inside Template Loading” section) will usually cover all your template-loading needs, but it’s pretty easy to write your own if you need special loading logic. For example, you could load templates from a database, or directly from a Subversion repository using Subversion’s Python bindings, or (as shown shortly) from a ZIP archive.

A template loader—that is, each entry in the `TEMPLATE_LOADERS` setting—is expected to be a callable object with this interface:

```
load_template_source(template_name, template_dirs=None)
```

The `template_name` argument is the name of the template to load (as passed to `loader.get_template()` or `loader.select_template()`), and `template_dirs` is an optional list of directories to search instead of `TEMPLATE_DIRS`.

If a loader is able to successfully load a template, it should return a tuple: `(template_source, template_path)`. Here, `template_source` is the template string that will be compiled by the template engine, and `template_path` is the path the template was loaded from. That path might be shown to the user for debugging purposes, so it should quickly identify where the template was loaded from.

If the loader is unable to load a template, it should raise `django.template.TemplateDoesNotExist`.

Each loader function should also have an `is_usable` function attribute. This is a Boolean that informs the template engine whether this loader is available in the current Python installation. For example, the eggs loader (which is capable of loading templates from Python eggs) sets `is_usable` to `False` if the `pkg_resources` module isn’t installed, because `pkg_resources` is necessary to read data from eggs.

An example should help clarify all of this. Here’s a template loader function that can load templates from a ZIP file. It uses a custom setting, `TEMPLATE_ZIP_FILES`, as a search path instead of `TEMPLATE_DIRS`, and it expects each item on that path to be a ZIP file containing templates:

```
from django.conf import settings
from django.template import TemplateDoesNotExist
import zipfile

def load_template_source(template_name, template_dirs=None):
    "Template loader that loads templates from a ZIP file."

    template_zipfiles = getattr(settings, "TEMPLATE_ZIP_FILES", [])
```

```
# Try each ZIP file in TEMPLATE_ZIP_FILES.
for fname in template_zipfiles:
    try:
        z = zipfile.ZipFile(fname)
        source = z.read(template_name)
    except (IOError, KeyError):
        continue
    z.close()
    # We found a template, so return the source.
    template_path = "%s:%s" % (fname, template_name)
    return (source, template_path)

# If we reach here, the template couldn't be loaded
raise TemplateDoesNotExist(template_name)

# This loader is always usable (since zipfile is included with Python)
load_template_source.is_usable = True
```

The only step left if we want to use this loader is to add it to the `TEMPLATE_LOADERS` setting. If we put this code in a package called `mysite.zip_loader`, then we add `mysite.zip_loader.load_template_source` to `TEMPLATE_LOADERS`.

Configuring the Template System in Standalone Mode

Note This section is only of interest to people trying to use the template system as an output component in another application. If you are using the template system as part of a Django application, the information presented here doesn't apply to you.

Normally, Django loads all the configuration information it needs from its own default configuration file, combined with the settings in the module given in the `DJANGO_SETTINGS_MODULE` environment variable. (This was explained in “A Special Python Prompt” in Chapter 4.) But if you're using the template system independent of the rest of Django, the environment variable approach isn't very convenient, because you probably want to configure the template system in line with the rest of your application rather than dealing with settings files and pointing to them via environment variables.

To solve this problem, you need to use the manual configuration option described fully in Appendix D. In a nutshell, you need to import the appropriate pieces of the template system and then, *before* you call any of the template functions, call `django.conf.settings.configure()` with any settings you wish to specify.

You might want to consider setting at least `TEMPLATE_DIRS` (if you are going to use template loaders), `DEFAULT_CHARSET` (although the default of `utf-8` is probably fine), and `TEMPLATE_DEBUG`. All available settings are described in Appendix D, and any setting starting with `TEMPLATE_` is of obvious interest.

What's Next?

Continuing this section's theme of advanced topics, the next chapter covers the advanced usage of Django models.



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 or 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.

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. (See “Passing Extra Options to View Functions” in Chapter 8 for an overview of this technique.)

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('',
    (r'^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('',
    (r'^about/$', direct_to_template, {
        'template': 'about.html'
    }),
    (r'^about/(\w+)/$', 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_page(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 20). 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. The 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 (such as dots and slashes) 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(max_length=30)
    address = models.CharField(max_length=50)
    city = models.CharField(max_length=60)
    state_province = models.CharField(max_length=30)
    country = models.CharField(max_length=50)
    website = models.URLField()

    def __unicode__(self):
        return self.name

class Meta:
    ordering = ['name']
```

To build a list page of all publishers, 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 can explicitly tell the `object_list` view which template to use by including a `template_name` key in the extra-arguments dictionary:

```
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(),
    'template_name': 'publisher_list_page.html',
}

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

In the absence of `template_name`, though, the `object_list` generic view 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 publisher 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 %}
```


(Note that this assumes the existence of a `base.html` template, as we provided in an example in Chapter 4.)

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 C documents all the generic views and all their options in detail; the rest of this chapter will 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, one of the most common questions 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 the handful of patterns dealt with in the following sections.

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:

```
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(),
    'template_name': 'publisher_list_page.html',
    'template_object_name': 'publisher',
}

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

In the template, the generic view will append `_list` to the `template_object_name` to create the variable name representing the list of items.

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

Adding Extra Context

Sometimes you might need to present information beyond that provided in 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 dictionary like this:

```
publisher_info = {
    'queryset': Publisher.objects.all(),
    'template_object_name': 'publisher',
    'extra_context': {'book_list': Book.objects.all()}
}
```

This would populate a `{{ book_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 `Book.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 B for more information about when `QuerySet` objects 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_books():
    return Book.objects.all()

publisher_info = {
    'queryset': Publisher.objects.all(),
    'template_object_name': 'publisher',
    'extra_context': {'book_list': get_books}
}
```

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

```
publisher_info = {
    'queryset': Publisher.objects.all(),
    'template_object_name': 'publisher',
    'extra_context': {'book_list': Book.objects.all}
}
```

Notice the lack of parentheses after `Book.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 `QuerySet` objects, and see Appendix B for the complete details).

Suppose, for example, that you want to order a list of books by publication date, with the most recent first:

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

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

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'd 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.

Complex Filtering with Wrapper Functions

Another common need is to filter 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? The solution is to “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.shortcuts import get_object_or_404
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).
    publisher = get_object_or_404(Publisher, name__iexact=name)

    # 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 = 'book',
        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.

Note 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 django.shortcuts import get_object_or_404
from django.views.generic import list_detail
from mysite.books.models import Author

def author_detail(request, author_id):
    # Delegate to the generic view and get an HttpResponse.
    response = list_detail.object_detail(
        request,
        queryset = Author.objects.all(),
        object_id = author_id,
    )

    # Record the last accessed date. We do this *after* the call
    # to object_detail(), not before it, so that this won't be called
    # unless the Author actually exists. (If the author doesn't exist,
    # object_detail() will raise Http404, and we won't reach this point.)
    now = datetime.datetime.now()
    Author.objects.filter(id=author_id).update(last_accessed=now)

    return response
```

Note This code won't 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.

What's Next?

In this chapter we looked at only a couple of the generic views Django ships with, but the general ideas presented here should apply pretty closely to any generic view. Appendix C covers all the available views in detail, and it's recommended reading if you want to get the most out of this powerful feature.

This concludes the section of this book devoted to “advanced usage.” In the next chapter we cover deployment of Django applications.



Deploying Django

This chapter covers the last essential step of building a Django application: deploying it to a production server.

If you've been following along with our ongoing examples, you probably used the `runserver`, which makes things very easy (you don't have to worry about Web server setup). But `runserver` is intended only for development on your local machine, not for exposure on the public Web. To deploy your Django application, you'll need to hook it into an industrial-strength Web server such as Apache. In this chapter, we'll show you how to do that, but first we'll give you a checklist of things to do in your codebase before you go live.

Preparing Your Codebase for Production

Fortunately, the `runserver` approximates a “real” Web server closely enough that not very many changes need to be made to a Django application in order to make it production-ready. But there are a few *essential things* you should do before you turn the switch.

Turning Off Debug Mode

When we created a project in Chapter 2, the command `django-admin.py startproject` created a `settings.py` file with `DEBUG` set to `True`. Many internal parts of Django check this setting and change their behavior if `DEBUG` mode is on. For example, if `DEBUG` is set to `True`, then:

- All database queries will be saved in memory as the object `django.db.connection.queries`. As you can imagine, this eats up memory!
- Any 404 error will be rendered by Django's special 404 error page (covered in Chapter 3) instead of returning a proper 404 response. This page contains potentially sensitive information and should *not* be exposed to the public Internet.
- Any uncaught exception in your Django application—from basic Python syntax errors to database errors to template syntax errors—will be rendered by the Django pretty error page that you've likely come to know and love. This page contains even *more* sensitive information than the 404 page and should *never* be exposed to the public.

In short, setting `DEBUG` to `True` tells Django to assume that only trusted developers are using your site. The Internet is full of untrustworthy hooligans, and the first thing you should do when you're preparing your application for deployment is set `DEBUG` to `False`.

Turning Off Template Debug Mode

Similarly, you should set `TEMPLATE_DEBUG` to `False` in production. If `True`, this setting tells Django's template system to save some extra information about every template for use on the pretty error pages.

Implementing a 404 Template

If `DEBUG` is `True`, Django displays the useful 404 error page. But if `DEBUG` is `False`, it does something different: it renders a template called `404.html` in your root template directory. So, when you're ready to deploy, you'll need to create this template and put a useful "Page not found" message in it.

Here's a sample `404.html` you can use as a starting point. It assumes that you're using template inheritance and have defined a `base.html` with blocks called `title` and `content`:

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

{% block title %}Page not found{% endblock %}

{% block content %}
<h1>Page not found</h1>

<p>Sorry, but the requested page could not be found.</p>
{% endblock %}
```

To test that your `404.html` is working, just change `DEBUG` to `False` and visit a nonexistent URL. (This works on the runserver just as well as it works on a production server.)

Implementing a 500 Template

Similarly, if `DEBUG` is `False`, then Django no longer displays its useful error/traceback pages in case of an unhandled Python exception. Instead, it looks for a template called `500.html` and renders it. Like `404.html`, this template should live in your root template directory.

There's one slightly tricky thing about `500.html`. You can never be sure *why* this template is being rendered, so it shouldn't do anything that requires a database connection or relies on any potentially broken part of your infrastructure. (For example, it should not use custom template tags.) If it uses template inheritance, then the parent template(s) shouldn't rely on potentially broken infrastructure, either. Therefore, the best approach is to avoid template inheritance and use something very simple. Here's an example `500.html` as a starting point:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01//EN"
    "http://www.w3.org/TR/html4/strict.dtd">
<html lang="en">
<head>
    <title>Page unavailable</title>
</head>
```



```

<body>
  <h1>Page unavailable</h1>

  <p>Sorry, but the requested page is unavailable due to a
  server hiccup.</p>

  <p>Our engineers have been notified, so check back later.</p>
</body>
</html>

```

Setting Up Error Alerts

When your Django-powered site is running and an exception is raised, you'll want to know about it, so you can fix it. By default, Django is configured to send an e-mail to the site developers whenever your code raises an unhandled exception—but you need to do two things to set it up.

First, change your `ADMINS` setting to include your e-mail address, along with the e-mail addresses of any other people who need to be notified. This setting takes `(name, email)` tuples, like this:

```

ADMINS = (
    ('John Lennon', 'jlennon@example.com'),
    ('Paul McCartney', 'pmacca@example.com'),
)

```

Second, make sure that your server is configured to send e-mail. Setting up postfix, sendmail, or any other mail server is outside the scope of this book, but on the Django side of things, you'll want to make sure that your `EMAIL_HOST` setting is set to the proper hostname for your mail server. It's set to `'localhost'` by default, which works out of the box for most shared-hosting environments. You might also need to set `EMAIL_HOST_USER`, `EMAIL_HOST_PASSWORD`, `EMAIL_PORT`, or `EMAIL_USE_TLS`, depending on the complexity of your arrangement.

Also, you can set `EMAIL_SUBJECT_PREFIX` to control the prefix Django uses in front of its error e-mail. It is set to `'[Django]'` by default.

Setting Up Broken Link Alerts

If you have the `CommonMiddleware` installed (e.g., if your `MIDDLEWARE_CLASSES` setting includes `'django.middleware.common.CommonMiddleware'`, which it does by default), you have the option of receiving an e-mail any time somebody visits a page on your Django-powered site that raises 404 with a non-empty referrer—that is, every broken link. If you want to activate this feature, set `SEND_BROKEN_LINK_EMAILS` to `True` (it's `False` by default) and set your `MANAGERS` setting to a person or people who will receive this broken-link e-mail. `MANAGERS` uses the same syntax as `ADMINS`. For example:

```

MANAGERS = (
    ('George Harrison', 'gharrison@example.com'),
    ('Ringo Starr', 'ringo@example.com'),
)

```

Note that error e-mail can get annoying; they're not for everybody.

Using Different Settings for Production

So far in this book, we’ve dealt with only a single settings file: the `settings.py` generated by `django-admin.py startproject`. But as you get ready to deploy, you’ll likely find yourself needing multiple settings files to keep your development environment isolated from your production environment. (For example, you probably won’t want to change `DEBUG` from `False` to `True` whenever you want to test code changes on your local machine.) Django makes this very easy by allowing you to use multiple settings files.

If you want to organize your settings files into “production” and “development” settings, you can accomplish it in three ways:

- Set up two full-blown, independent settings files.
- Set up a “base” settings file (say, for development) and a second (say, production) settings file that merely imports from the first one and defines whatever overrides it needs to define.
- Use only a single settings file that has Python logic to change the settings based on context.

We’ll take these one at a time.

First, the most basic approach is to define two separate settings files. If you’re following along, you’ve already got `settings.py`. Now, just make a copy of it called `settings_production.py`. (We made this name up; you can call it whatever you want.) In this new file, change `DEBUG`, and so on.

The second approach is similar, but cuts down on redundancy. Instead of having two settings files whose contents are mostly similar, you can treat one as the “base” file and create another file that imports from it. For example:

```
# settings.py

DEBUG = True
TEMPLATE_DEBUG = DEBUG

DATABASE_ENGINE = 'postgresql_psycopg2'
DATABASE_NAME = 'devdb'
DATABASE_USER = ''
DATABASE_PASSWORD = ''
DATABASE_PORT = ''

# ...

# settings_production.py

from settings import *

DEBUG = TEMPLATE_DEBUG = False
DATABASE_NAME = 'production'
DATABASE_USER = 'app'
DATABASE_PASSWORD = 'letmein'
```

Here, `settings_production.py` imports everything from `settings.py` and just redefines the settings that are particular to production. In this case, `DEBUG` is set to `False`, but we also set different database access parameters for the production setting. (The latter goes to show that you can redefine *any* setting, not just the basic ones such as `DEBUG`.)

Finally, the most concise way of accomplishing two settings environments is to use a single settings file that branches based on the environment. One way to do this is to check the current hostname. For example:

```
# settings.py

import socket

if socket.gethostname() == 'my-laptop':
    DEBUG = TEMPLATE_DEBUG = True
else:
    DEBUG = TEMPLATE_DEBUG = False

# ...
```

Here, we import the `socket` module from Python's standard library and use it to check the current system's hostname. We can check the hostname to determine whether the code is being run on the production server.

A core lesson here is that settings files are *just Python code*. They can import from other files, they can execute arbitrary logic, and so on. Just make sure that if you go down this road, the Python code in your settings files is bulletproof. If it raises any exceptions, Django will likely crash badly.

RENAMING SETTINGS.PY

Feel free to rename your `settings.py` to `settings_dev.py`, or `settings/dev.py`, or `foobar.py`—Django doesn't care, as long as you tell it what settings file you're using.

But if you *do* rename the `settings.py` file that is generated by `django-admin.py startproject`, you'll find that `manage.py` will give you an error message saying that it can't find the `settings`. That's because it tries to import a module called `settings`. You can fix this either by editing `manage.py` to change `settings` to the name of your module, or by using `django-admin.py` instead of `manage.py`. In the latter case, you'll need to set the `DJANGO_SETTINGS_MODULE` environment variable to the Python path to your settings file (e.g., `'mysite.settings'`).

DJANGO_SETTINGS_MODULE

With those code changes out of the way, the next part of this chapter will focus on deployment instructions for specific environments, such as Apache. The instructions are different for each environment, but one thing remains the same: in each case, you have to tell the Web server your `DJANGO_SETTINGS_MODULE`. This is the entry point into your Django application. The `DJANGO_SETTINGS_MODULE` points to your settings file, which points to your `ROOT_URLCONF`, which points to your views, and so on.

DJANGO_SETTINGS_MODULE is the Python path to your settings file. For example, assuming that the `mysite` directory is on your Python path, the `DJANGO_SETTINGS_MODULE` for our ongoing example is `'mysite.settings'`.

Using Django with Apache and `mod_python`

Apache with `mod_python` historically has been the suggested setup for using Django on a production server.

`mod_python` (http://www.djangoproject.com/r/mod_python/) is an Apache plug-in that embeds Python within Apache and loads Python code into memory when the server starts. Code stays in memory throughout the life of an Apache process, which leads to significant performance gains over other server arrangements.

Django requires Apache 2.x and `mod_python` 3.x.

Note Configuring Apache is *well* beyond the scope of this book, so we'll simply mention details as needed. Luckily, many great resources are available if you need to learn more about Apache. A few of them we like are the following.

- The free online Apache documentation, available via <http://www.djangoproject.com/r/apache/docs/>
- *Pro Apache, Third Edition* by Peter Wainwright (Apress, 2004), available via <http://www.djangoproject.com/r/books/pro-apache/>
- *Apache: The Definitive Guide, Third Edition* by Ben Laurie and Peter Laurie (O'Reilly, 2002), available via <http://www.djangoproject.com/r/books/pro-apache/>

Basic Configuration

To configure Django with `mod_python`, first make sure you have Apache installed with the `mod_python` module activated. This usually means having a `LoadModule` directive in your Apache configuration file. It will look something like this:

```
LoadModule python_module /usr/lib/apache2/modules/mod_python.so
```

Then, edit your Apache configuration file and add a `<Location>` directive that ties a specific URL path to a specific Django installation. For example:

```
<Location "/">
    SetHandler python-program
    PythonHandler django.core.handlers.modpython
    SetEnv DJANGO_SETTINGS_MODULE mysite.settings
    PythonDebug Off
</Location>
```

Make sure to replace `mysite.settings` with the appropriate `DJANGO_SETTINGS_MODULE` for your site.

This tells Apache, “Use `mod_python` for any URL at or under ‘/’, using the Django `mod_python` handler.” It passes the value of `DJANGO_SETTINGS_MODULE` so `mod_python` knows which settings to use.

Note that we’re using the `<Location>` directive, not the `<Directory>` directive. The latter is used for pointing at places on your filesystem, whereas `<Location>` points at places in the URL structure of a Web site. `<Directory>` would be meaningless here.

Apache likely runs as a different user than your normal login and may have a different path and `sys.path`. You may need to tell `mod_python` how to find your project and Django itself.

```
PythonPath ["'/path/to/project', '/path/to/django'] + sys.path"
```

You can also add directives such as `PythonAutoReload Off` for performance. See the `mod_python` documentation for a full list of options.

Note that you should set `PythonDebug Off` on a production server. If you leave `PythonDebug On`, your users will see ugly (and revealing) Python tracebacks if something goes wrong within `mod_python`.

Restart Apache, and any request to your site (or virtual host if you’ve put this directive inside a `<VirtualHost>` block) will be served by Django.

Running Multiple Django Installations on the Same Apache Instance

It’s entirely possible to run multiple Django installations on the same Apache instance. You might want to do this if you’re an independent Web developer with multiple clients but only a single server.

To accomplish this, just use `VirtualHost` like so:

```
NameVirtualHost *
```

```
<VirtualHost *>
    ServerName www.example.com
    # ...
    SetEnv DJANGO_SETTINGS_MODULE mysite.settings
</VirtualHost>
```

```
<VirtualHost *>
    ServerName www2.example.com
    # ...
    SetEnv DJANGO_SETTINGS_MODULE mysite.other_settings
</VirtualHost>
```

If you need to put two Django installations within the same `VirtualHost`, you’ll need to take a special precaution to ensure `mod_python`’s code cache doesn’t mess things up. Use the `PythonInterpreter` directive to give different `<Location>` directives separate interpreters:

```

<VirtualHost *>
    ServerName www.example.com
    # ...
    <Location "/something">
        SetEnv DJANGO_SETTINGS_MODULE mysite.settings
        PythonInterpreter mysite
    </Location>

    <Location "/otherthing">
        SetEnv DJANGO_SETTINGS_MODULE mysite.other_settings
        PythonInterpreter mysite_other
    </Location>
</VirtualHost>

```

The values of `PythonInterpreter` don't really matter, as long as they're different between the two `Location` blocks.

Running a Development Server with `mod_python`

Because `mod_python` caches loaded Python code, when deploying Django sites on `mod_python` you'll need to restart Apache each time you make changes to your code. This can be a hassle, so here's a quick trick to avoid it: just add `MaxRequestsPerChild 1` to your config file to force Apache to reload everything for each request. But don't do that on a production server, or we'll revoke your Django privileges.

If you're the type of programmer who debugs using scattered print statements (we are), note that print statements have no effect in `mod_python`; they don't appear in the Apache log, as you might expect. If you have the need to print debugging information in a `mod_python` setup, you'll probably want to use Python's standard logging package. More information is available at <http://docs.python.org/lib/module-logging.html>.

Serving Django and Media Files from the Same Apache Instance

Django should not be used to serve media files itself; leave that job to whichever Web server you choose. We recommend using a separate Web server (i.e., one that's not also running Django) for serving media. For more information, see the "Scaling" section.

If, however, you have no option but to serve media files on the same Apache `VirtualHost` as Django, here's how you can turn off `mod_python` for a particular part of the site:

```

<Location "/media/">
    SetHandler None
</Location>

```

Change `Location` to the root URL of your media files.

You can also use `<LocationMatch>` to match a regular expression. For example, this sets up Django at the site root but explicitly disables Django for the `media` subdirectory and any URL that ends with `.jpg`, `.gif`, or `.png`:

```

<Location "/">
    SetHandler python-program
    PythonHandler django.core.handlers.modpython
    SetEnv DJANGO_SETTINGS_MODULE mysite.settings
</Location>

<Location "/media/">
    SetHandler None
</Location>

<LocationMatch "\.(jpg|gif|png)$">
    SetHandler None
</LocationMatch>

```

In all of these cases, you'll need to set the `DocumentRoot` directive so Apache knows where to find your static files.

Error Handling

When you use Apache/mod_python, errors will be caught by Django—in other words, they won't propagate to the Apache level and won't appear in the Apache `error_log`.

The exception to this is if something is really messed up in your Django setup. In that case, you'll see an ominous “Internal Server Error” page in your browser and the full Python traceback in your Apache `error_log` file. The `error_log` traceback is spread over multiple lines. (Yes, this is ugly and rather hard to read, but it's how mod_python does things.)

Handling a Segmentation Fault

Sometimes, Apache segfaults when you install Django. When this happens, it's almost *always* one of two causes mostly unrelated to Django itself:

- It may be that your Python code is importing the `pyexpat` module (used for XML parsing), which may conflict with the version embedded in Apache. For full information, see “Expat Causing Apache Crash” at <http://www.djangoproject.com/r/articles/expat-apache-crash/>.
- It may be because you're running mod_python and mod_php in the same Apache instance, with MySQL as your database back-end. In some cases, this causes a known mod_python issue due to version conflicts in PHP and the Python MySQL back-end. There's full information in a mod_python FAQ entry, accessible via <http://www.djangoproject.com/r/articles/php-modpython-faq/>.

If you continue to have problems setting up mod_python, a good thing to do is get a bare-bones mod_python site working, without the Django framework. This is an easy way to isolate mod_python-specific problems. The article “Getting mod_python Working” details this procedure: <http://www.djangoproject.com/r/articles/getting-modpython-working/>.

The next step should be to edit your test code and add an import of any Django-specific code you're using—your views, your models, your URLconf, your RSS configuration, and so forth. Put these imports in your test handler function and access your test URL in a browser. If this causes a crash, you've confirmed it's the importing of Django code that causes the problem. Gradually reduce the set of imports until it stops crashing, so as to find the specific module that causes the problem. Drop down further into modules and look into their imports as necessary. For more help, system tools like `ldconfig` on Linux, `otool` on Mac OS, and `ListDLLs` (from SysInternals) on Windows can help you identify shared dependencies and possible version conflicts.

An Alternative: `mod_wsgi`

As an alternative to `mod_python`, you might consider using `mod_wsgi` (<http://code.google.com/p/modwsgi/>), which has been developed more recently than `mod_python` and is getting some traction in the Django community. A full overview is outside the scope of this book, but see the official Django documentation for more information.

Using Django with FastCGI

Although Django under Apache and `mod_python` is the most robust deployment setup, many people use shared hosting, on which FastCGI is the only available deployment option.

Additionally, in some situations, FastCGI allows better security and possibly better performance than `mod_python`. For small sites, FastCGI can also be more lightweight than Apache.

FastCGI Overview

FastCGI is an efficient way of letting an external application serve pages to a Web server. The Web server delegates the incoming Web requests (via a socket) to FastCGI, which executes the code and passes the response back to the Web server, which, in turn, passes it back to the client's Web browser.

Like `mod_python`, FastCGI allows code to stay in memory, allowing requests to be served with no startup time. Unlike `mod_python`, a FastCGI process doesn't run inside the Web server process, but in a separate, persistent process.

WHY RUN CODE IN A SEPARATE PROCESS?

The traditional `mod_*` arrangements in Apache embed various scripting languages (most notably PHP, Python/`mod_python`, and Perl/`mod_perl`) inside the process space of your Web server. Although this lowers startup time (because code doesn't have to be read off disk for every request), it comes at the cost of memory use.

Each Apache process gets a copy of the Apache engine, complete with all the features of Apache that Django simply doesn't take advantage of. FastCGI processes, on the other hand, only have the memory overhead of Python and Django.

Due to the nature of FastCGI, it's also possible to have processes that run under a different user account than the Web server process. That's a nice security benefit on shared systems, because it means you can secure your code from other users.

Before you can start using FastCGI with Django, you'll need to install `flup`, a Python library for dealing with FastCGI. Some users have reported stalled pages with older `flup` versions, so you may want to use the latest SVN version. Get `flup` at <http://www.djangoproject.com/r/flup/>.

Running Your FastCGI Server

FastCGI operates on a client/server model, and in most cases you'll be starting the FastCGI server process on your own. Your Web server (be it Apache, `lighttpd`, or otherwise) contacts your Django-FastCGI process only when the server needs a dynamic page to be loaded. Because the daemon is already running with the code in memory, it's able to serve the response very quickly.

Note If you're on a shared hosting system, you'll probably be forced to use Web server-managed FastCGI processes. If you're in this situation, you should read the section titled "Running Django on a Shared-Hosting Provider with Apache," later in this chapter.

A Web server can connect to a FastCGI server in one of two ways: it can use either a Unix domain socket (a *named pipe* on Win32 systems) or a TCP socket. What you choose is a matter of preference; a TCP socket is usually easier due to permissions issues.

To start your server, first change into the directory of your project (wherever your `manage.py` is), and then run `manage.py` with the `runfcgi` command:

```
./manage.py runfcgi [options]
```

If you specify `help` as the only option after `runfcgi`, a list of all the available options will display.


You'll need to specify either a socket or both host and port. Then, when you set up your Web server, you'll just need to point it at the socket or host/port you specified when starting the FastCGI server.

A few examples should help explain this:

- Running a threaded server on a TCP port:

```
./manage.py runfcgi method=threaded host=127.0.0.1 port=3033
```

- Running a preforked server on a Unix domain socket:

```
./manage.py runfcgi method=prefork   
socket=/home/user/mysite.sock pidfile=django.pid
```

- Running without daemonizing (backgrounding) the process (good for debugging):

```
./manage.py runfcgi daemonize=false socket=/tmp/mysite.sock
```

Stopping the FastCGI Daemon

If you have the process running in the foreground, it's easy enough to stop it: simply press Ctrl+C to stop and quit the FastCGI server. However, when you're dealing with background processes, you'll need to resort to the Unix `kill` command.

If you specify the `pidfile` option to your `manage.py runfcgi`, you can kill the running FastCGI daemon like this:

```
kill 'cat $PIDFILE'
```

where `$PIDFILE` is the pidfile you specified.

To easily restart your FastCGI daemon on Unix, you can use this small shell script:

```
#!/bin/bash

# Replace these three settings.
PROJDIR="/home/user/myproject"
PIDFILE="$PROJDIR/mysite.pid"
SOCKET="$PROJDIR/mysite.sock"

cd $PROJDIR
if [ -f $PIDFILE ]; then
    kill 'cat-$PIDFILE'
    rm -f-$PIDFILE
fi

exec /usr/bin/env - \
    PYTHONPATH="../python:.." \
    ./manage.py runfcgi socket=$SOCKET pidfile=$PIDFILE
```

Using Django with Apache and FastCGI

To use Django with Apache and FastCGI, you'll need Apache installed and configured, with `mod_fastcgi` installed and enabled. Consult the Apache and `mod_fastcgi` documentation for instructions: http://www.djangoproject.com/r/mod_fastcgi/.

Once you've completed the setup, point Apache at your Django FastCGI instance by editing the `httpd.conf` (Apache configuration) file. You'll need to do two things:

- Use the `FastCGIExternalServer` directive to specify the location of your FastCGI server.
- Use `mod_rewrite` to point URLs at FastCGI as appropriate.

Specifying the Location of the FastCGI Server

The `FastCGIExternalServer` directive tells Apache how to find your FastCGI server. As the `FastCGIExternalServer` docs (http://www.djangoproject.com/r/mod_fastcgi/FastCGIExternalServer/) explain, you can specify either a socket or a host. Here are examples of both:

```
# Connect to FastCGI via a socket/named pipe:
FastCGIExternalServer /home/user/public_html/mysite.fcgi ➡
-socket /home/user/mysite.sock

# Connect to FastCGI via a TCP host/port:
FastCGIExternalServer /home/user/public_html/mysite.fcgi -host 127.0.0.1:3033
```

In either case, the directory `/home/user/public_html/` should exist, though the file `/home/user/public_html/mysite.fcgi` doesn't actually have to exist. It's just a URL used by the Web server internally—a hook for signifying which requests at a URL should be handled by FastCGI. (More on this in the next section.)

Using `mod_rewrite` to Point URLs at FastCGI

The second step is telling Apache to use FastCGI for URLs that match a certain pattern. To do this, use the `mod_rewrite` module and rewrite URLs to `mysite.fcgi` (or whatever you specified in the `FastCGIExternalServer` directive, as explained in the previous section).

In this example, we tell Apache to use FastCGI to handle any request that doesn't represent a file on the filesystem and doesn't start with `/media/`. This is probably the most common case, if you're using Django's admin site:

```
<VirtualHost 12.34.56.78>
  ServerName example.com
  DocumentRoot /home/user/public_html
  Alias /media /home/user/python/django/contrib/admin/media
  RewriteEngine On
  RewriteRule ^/(media.*)$ /$1 [QSA,L]
  RewriteCond %{REQUEST_FILENAME} !-f
  RewriteRule ^/(.*)$ /mysite.fcgi/$1 [QSA,L]
</VirtualHost>
```

FastCGI and `lighttpd`

`lighttpd` (<http://www.djangoproject.com/r/lighttpd/>) is a lightweight Web server commonly used for serving static files. It supports FastCGI natively and thus is also an ideal choice for serving both static and dynamic pages, if your site doesn't have any Apache-specific needs.

Make sure `mod_fastcgi` is in your modules list, somewhere after `mod_rewrite` and `mod_access`, but not after `mod_accesslog`. You'll probably want `mod_alias` as well, for serving admin media.

Add the following to your `lighttpd` config file:

```
server.document-root = "/home/user/public_html"
fastcgi.server = (
    "/mysite.fcgi" => (
        "main" => (
            # Use host / port instead of socket for TCP fastcgi
            # "host" => "127.0.0.1",
            # "port" => 3033,
            "socket" => "/home/user/mysite.sock",
            "check-local" => "disable",
        )
    ),
)
alias.url = (
    "/media/" => "/home/user/django/contrib/admin/media/",
)

url.rewrite-once = (
    "^(/media.*)$" => "$1",
    "^/favicon\.ico$" => "/media/favicon.ico",
    "^(/.*)$" => "/mysite.fcgi$1",
)
```

Running Multiple Django Sites on One `lighttpd` Instance

`lighttpd` lets you use “conditional configuration” to allow configuration to be customized per host. To specify multiple FastCGI sites, just add a conditional block around your FastCGI config for each site:

```
# If the hostname is 'www.example1.com'...
$HTTP["host"] == "www.example1.com" {
    server.document-root = "/foo/site1"
    fastcgi.server = (
        ...
    )
    ...
}

# If the hostname is 'www.example2.com'...
$HTTP["host"] == "www.example2.com" {
    server.document-root = "/foo/site2"
    fastcgi.server = (
        ...
    )
    ...
}
```

You can also run multiple Django installations on the same site simply by specifying multiple entries in the `fastcgi.server` directive. Add one FastCGI host for each.

Running Django on a Shared-Hosting Provider with Apache

Many shared-hosting providers don't allow you to run your own server daemons or edit the `httpd.conf` file. In these cases, it's still possible to run Django using Web server-spawned processes.

Note If you're using Web server-spawned processes, as explained in this section, there's no need for you to start the FastCGI server on your own. Apache will spawn a number of processes, scaling as it needs to.

In your Web root directory, add this to a file named `.htaccess`:

```
AddHandler fastcgi-script .fcgi
RewriteEngine On
RewriteCond %{REQUEST_FILENAME} !-f
RewriteRule ^(.*)$ mysite.fcgi/$1 [QSA,L]
```

Then, create a small script that tells Apache how to spawn your FastCGI program. Create a file, `mysite.fcgi`, and place it in your Web directory, and be sure to make it executable:

```
#!/usr/bin/python
import sys, os

# Add a custom Python path.
sys.path.insert(0, "/home/user/python")

# Switch to the directory of your project. (Optional.)
# os.chdir("/home/user/myproject")

# Set the DJANGO_SETTINGS_MODULE environment variable.
os.environ['DJANGO_SETTINGS_MODULE'] = "myproject.settings"

from django.core.servers.fastcgi import runfastcgi
runfastcgi(method="threaded", daemonize="false")
```

Restarting the Spawned Server

If you change any Python code on your site, you'll need to tell FastCGI the code has changed. But there's no need to restart Apache in this case. Rather, just reupload `mysite.fcgi`—or edit the file—so that the timestamp on the file changes. When Apache sees the file has been updated, it will restart your Django application for you.

If you have access to a command shell on a Unix system, you can accomplish this easily by using the `touch` command:

```
touch mysite.fcgi
```

Scaling

Now that you know how to get Django running on a single server, let's look at how you can scale out a Django installation. This section walks through how a site might scale from a single server to a large-scale cluster that could serve millions of hits an hour.

It's important to note, however, that nearly every large site is large in different ways, so scaling is anything but a one-size-fits-all operation. The following coverage should suffice to show the general principle, and whenever possible we'll try to point out where different choices could be made.

First off, we'll make a pretty big assumption and exclusively talk about scaling under Apache and `mod_python`. Though we know of a number of successful medium- to large-scale FastCGI deployments, we're much more familiar with Apache.

Running on a Single Server

Most sites start out running on a single server, with an architecture that looks something like Figure 12-1.

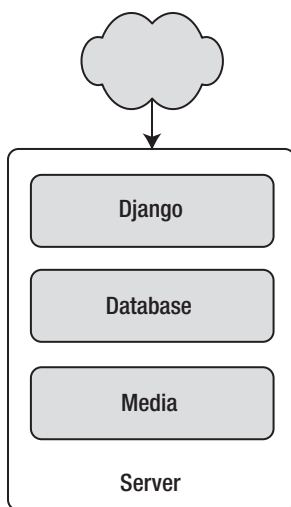


Figure 12-1. A single-server Django setup

This works just fine for small- to medium-sized sites, and it's relatively cheap—you can put together a single-server site designed for Django for well under \$3,000.

However, as traffic increases you'll quickly run into *resource contention* between the different pieces of software. Database servers and Web servers *love* to have the entire server to themselves, so when run on the same server they often end up “fighting” over the same resources (RAM, CPU) that they'd prefer to monopolize.

This is solved easily by moving the database server to a second machine, as explained in the following section.

Separating Out the Database Server

As far as Django is concerned, the process of separating out the database server is extremely easy: you'll simply need to change the `DATABASE_HOST` setting to the IP or DNS name of your database server. It's probably a good idea to use the IP if at all possible, as relying on DNS for the connection between your Web server and database server isn't recommended.

With a separate database server, our architecture now looks like Figure 12-2.

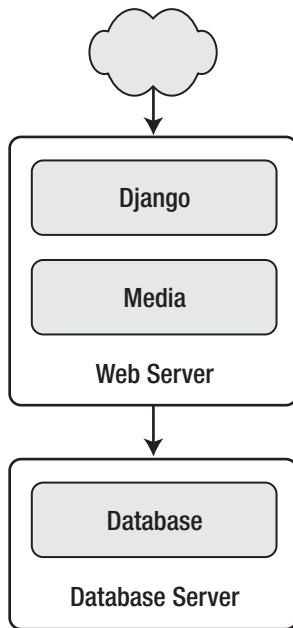


Figure 12-2. *Moving the database onto a dedicated server*

Here we're starting to move into what's usually called *n-tier* architecture. Don't be scared by the buzzword—it just refers to the fact that different "tiers" of the Web stack get separated out onto different physical machines.

At this point, if you anticipate ever needing to grow beyond a single database server, it's probably a good idea to start thinking about connection pooling and/or database replication. Unfortunately, there's not nearly enough space to do those topics justice in this book, so you'll need to consult your database's documentation and/or community for more information.

Running a Separate Media Server

We still have a big problem left over from the single-server setup: the serving of media from the same box that handles dynamic content.

Those two activities perform best under different circumstances, and by smashing them together on the same box you end up with neither performing particularly well. So the next step is to separate out the media—that is, anything *not* generated by a Django view—onto a dedicated server (see Figure 12-3).

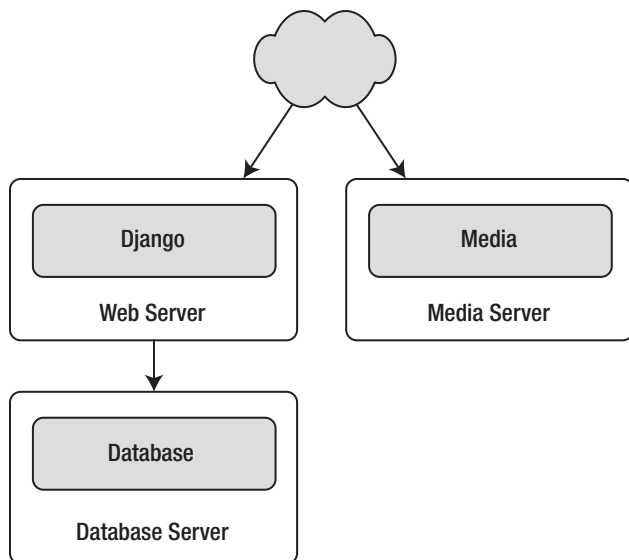


Figure 12-3. *Separating out the media server*

Ideally, this media server should run a stripped-down Web server optimized for static media delivery. `lighttpd` and `tux` (<http://www.djangoproject.com/r/tux/>) are both excellent choices here, but a heavily stripped down Apache could work, too.

For sites heavy in static content (photos, videos, etc.), moving to a separate media server is doubly important and should likely be the *first* step in scaling up.

This step can be slightly tricky, however. If your application involves file uploads, Django needs to be able to write uploaded media to the media server. If media lives on another server, you'll need to arrange a way for that write to happen across the network.

Implementing Load Balancing and Redundancy

At this point, we've broken things down as much as possible. This three-server setup should handle a very large amount of traffic—we served around 10 million hits a day from an architecture of this sort—so if you grow further, you'll need to start adding redundancy.

This is a good thing, actually. One glance at Figure 12-3 shows you that if even a single one of your three servers fails, you'll bring down your entire site. So as you add redundant servers, not only do you increase capacity, but you also increase reliability.

For the sake of this example, let's assume that the Web server hits capacity first. It's relatively easy to get multiple copies of a Django site running on different hardware—just copy all the code onto multiple machines, and start Apache on all of them.

However, you'll need another piece of software to distribute traffic over your multiple servers: a *load balancer*. You can buy expensive and proprietary hardware load balancers, but there are a few high-quality open source software load balancers out there.

Apache's `mod_proxy` is one option, but we've found `Perlbal` (<http://www.djangoproject.com/r/perlbal/>) to be fantastic. It's a load balancer and reverse proxy written by the same folks who wrote `Memcached` (see Chapter 15).

Note If you're using FastCGI, you can accomplish this same distribution/load-balancing step by separating your front-end Web servers and back-end FastCGI processes onto different machines. The front-end server essentially becomes the load balancer, and the back-end FastCGI processes replace the Apache/`mod_python`/Django servers.

With the Web servers now clustered, our evolving architecture starts to look more complex, as shown in Figure 12-4.

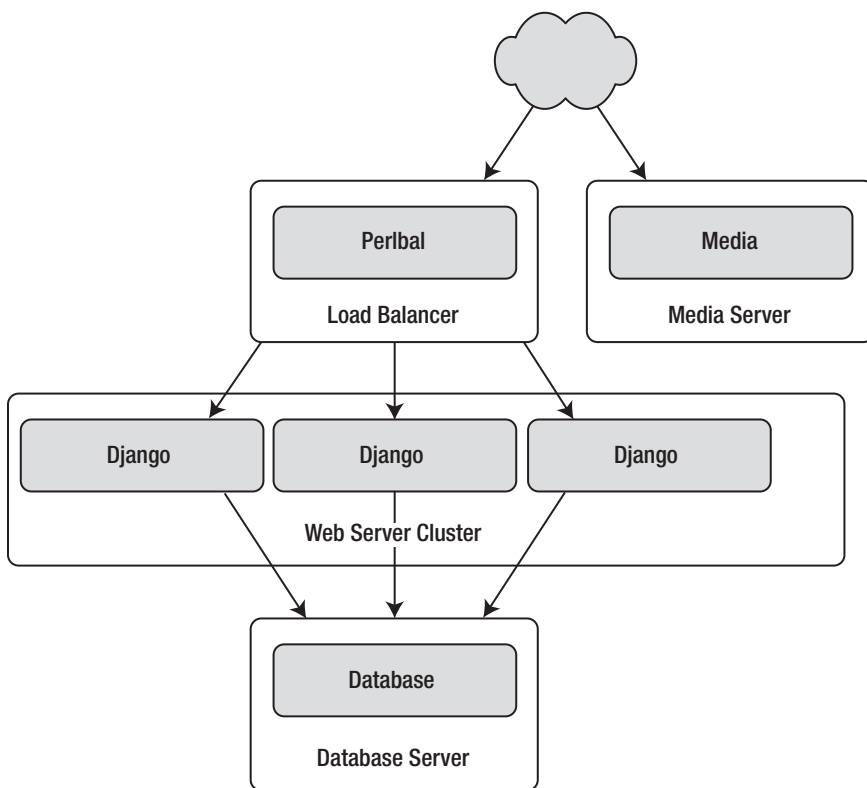


Figure 12-4. A load-balanced, redundant server setup

Notice that in the diagram the Web servers are referred to as a “cluster” to indicate that the number of servers is basically variable. Once you have a load balancer out front, you can easily add and remove back-end Web servers without a second of downtime.

Going Big

At this point, the next few steps are pretty much derivatives of the last one:

- As you need more database performance, you might want to add replicated database servers. MySQL includes built-in replication; PostgreSQL users should look into Slony (<http://www.djangoproject.com/r/slony/>) and pgpool (<http://www.djangoproject.com/r/pgpool/>) for replication and connection pooling, respectively.
- If the single load balancer isn't enough, you can add more load balancer machines out front and distribute among them using round-robin DNS.
- If a single media server doesn't suffice, you can add more media servers and distribute the load with your load-balancing cluster.
- If you need more cache storage, you can add dedicated cache servers.
- At any stage, if a cluster isn't performing well, you can add more servers to the cluster.

After a few of these iterations, a large-scale architecture might look like Figure 12-5.

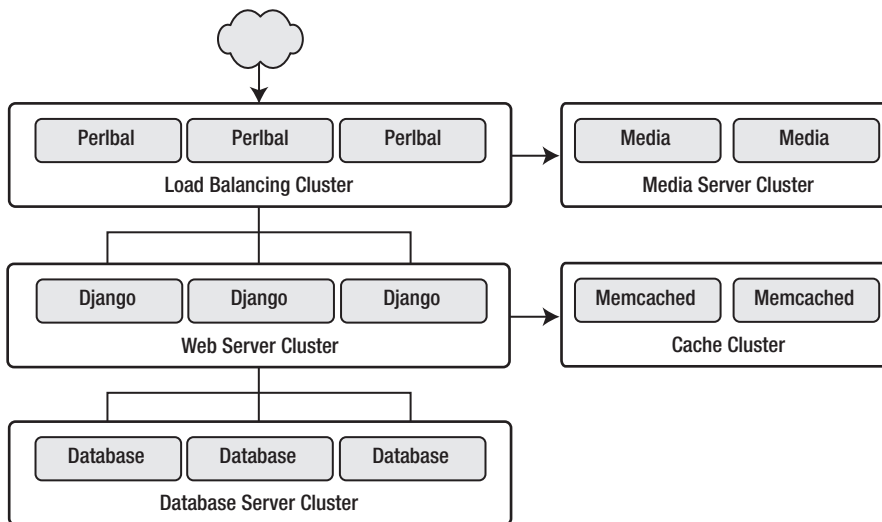


Figure 12-5. *An example large-scale Django setup*

Though we've shown only two or three servers at each level, there's no fundamental limit to how many you can add.

Performance Tuning

If you have a huge amount of money, you can just keep throwing hardware at scaling problems. For the rest of us, though, performance tuning is a must.

Note Incidentally, if anyone with monstrous gobs of cash is actually reading this book, please consider a substantial donation to the Django Foundation. We accept uncut diamonds and gold ingots, too.

Unfortunately, performance tuning is much more of an art than a science, and it is even more difficult to write about than scaling. If you're serious about deploying a large-scale Django application, you should spend a great deal of time learning how to tune each piece of your stack.

The following sections, though, present a few Django-specific tuning tips we've discovered over the years.

There's No Such Thing As Too Much RAM

Even the really expensive RAM is relatively affordable these days. Buy as much RAM as you can possibly afford, and then buy a little bit more.

Faster processors won't improve performance all that much; most Web servers spend up to 90% of their time waiting on disk I/O. As soon as you start swapping, performance will just die. Faster disks might help slightly, but they're much more expensive than RAM, such that it doesn't really matter.

If you have multiple servers, the first place to put your RAM is in the database server. If you can afford it, get enough RAM to get fit your entire database into memory. This shouldn't be too hard; we've developed a site with more than half a million newspaper articles, and it took under 2GB of space.

Next, max out the RAM on your Web server. The ideal situation is one where neither server swaps—ever. If you get to that point, you should be able to withstand most normal traffic.

Turn Off Keep-Alive

Keep-Alive is a feature of HTTP that allows multiple HTTP requests to be served over a single TCP connection, avoiding the TCP setup/teardown overhead.

This looks good at first glance, but it can kill the performance of a Django site. If you're properly serving media from a separate server, each user browsing your site will only request a page from your Django server every ten seconds or so. This leaves HTTP servers waiting around for the next keep-alive request, and an idle HTTP server just consumes RAM that an active one should be using.

Use Memcached

Although Django supports a number of different cache back-ends, none of them even come *close* to being as fast as Memcached. If you have a high-traffic site, don't even bother with the other back-ends—go straight to Memcached.

Use Memcached Often

Of course, selecting Memcached does you no good if you don't actually use it. Chapter 15 is your best friend here: learn how to use Django's cache framework, and use it everywhere possible. Aggressive, preemptive caching is usually the only thing that will keep a site up under major traffic.

Join the Conversation

Each piece of the Django stack—from Linux to Apache to PostgreSQL or MySQL—has an awesome community behind it. If you really want to get that last 1% out of your servers, join the open source communities behind your software and ask for help. Most free-software community members will be happy to help.

And also be sure to join the Django community. Your humble authors are only two members of an incredibly active, growing group of Django developers. Our community has a huge amount of collective experience to offer.

What's Next?

The remaining chapters focus on other Django features that you might or might not need, depending on your application. Feel free to read them in any order you choose.

1

jQuery and Ajax Integration in Django

We will be working with the leading Python web framework, Django, on the server side, and jQuery-powered Ajax on the client side. During the course of this book, we will cover the basic technologies and then see them come together in an employee intranet photo directory that shares some Web 2.0 strengths.

There is more than one good JavaScript library; we will be working with jQuery, which has reached acceptance as a standard lightweight JavaScript library. It might be suggested that Pythonistas may find much to like in jQuery: jQuery, like Python, was carefully designed to enable the developer to get powerful results easily.

In this chapter, we will:

- Discuss Ajax as not a single technology but a technique which is overlaid on other technologies
- Cover the basic technologies used in Ajax JavaScript
- Cover "Hello, world!" in a Django kickstart
- Introduce the Django templating engine
- Cover how to serve up static content in Django

Overall, what we will be doing is laying a solid foundation and introducing the working pieces of Django Ajax to be explored in this book.

Ajax and the XMLHttpRequest object

Ajax is not a technology like JavaScript or CSS, but is more like an overlaid function. So, what exactly is that?

Human speech: An overlaid function

Human speech is an **overlaid function**. What is meant by this is reflected in the answer to a question: "What part of the human body has the basic job of speech?" The tongue, for one answer, is used in speech, but it also tastes food and helps us swallow. The lungs and diaphragm, for another answer, perform the essential task of breathing. The brain cannot be overlooked, but it also does a great many other jobs. All of these parts of the body do something more essential than speech and, for that matter, all of these can be found among animals that cannot talk. Speech is something that is *overlaid* over organs that are there in the first place because of something other than speech.

Something similar to this is true for **Ajax**, which is not a technology in itself, but something **overlaid** on top of other technologies. Ajax, some people say, stands for **Asynchronous JavaScript and XML**, but that was a retroactive expansion. JavaScript was introduced almost a decade before people began seriously talking about Ajax. Not only is it technically possible to use Ajax without JavaScript (one can substitute VBScript at the expense of browser compatibility), but there are quite a few substantial reasons to use **JavaScript Object Notation (JSON)** in lieu of heavy-on-the-wire **eXtensible Markup Language (XML)**. Performing the *overlaid function* of Ajax with JSON replacing XML is just as eligible to be considered full-fledged Ajax as a solution incorporating XML.

Ajax: Another overlaid function

What exactly is this overlaid function?

Ajax is a way of using client-side technologies to talk with a server and perform partial page updates. Updates may be to all or part of the page, or simply to data handled behind the scenes. It is an alternative to the older paradigm of having a whole page replaced by a new page loaded when someone clicks on a link or submits a form. Partial page updates, in Ajax, are associated with **Web 2.0**, while whole page updates are associated with **Web 1.0**; it is important to note that "Web 2.0" and "Ajax" are not interchangeable. Web 2.0 includes more decentralized control and contributions besides Ajax, and for some objectives it may make perfect sense to develop an e-commerce site that uses Ajax but does not open the door to the same kind of community contributions as Web 2.0.

Some of the key features common in Web 2.0 include:

- Partial page updates with JavaScript communicating with a server and rendering to a page
- An emphasis on user-centered design
- Enabling community participation to update the website
- Enabling information sharing as core to what this communication allows

The concept of "partial page updates" may not sound very big, but part of its significance may be seen in an unintended effect. The original expectation of partial page updates was that it would enable web applications that were more responsive. The expectation was that if submitting a form would only change a small area of a page, using Ajax to just load the change would be faster than reloading the entire page for every minor change. That much was true, but once programmers began exploring, what they used Ajax for was not simply minor page updates, but making client-side applications that took on challenges more like those one would expect a desktop program to do, and the more interesting Ajax applications usually became slower. Again, this was not because you could not fetch part of the page and update it faster, but because programmers were trying to do things on the client side that simply were not possible under the older way of doing things, and were pushing the envelope on the concept of a web application and what web applications can do.

The technologies Ajax is overlaid on

Now let us look at some of the technologies where Ajax may be said to be *overlaid*.

JavaScript

JavaScript deserves pride of place, and while it is possible to use VBScript for Internet Explorer as much more than a proof of concept, for now if you are doing Ajax, it will almost certainly be Ajax running JavaScript as its engine. Your application will have JavaScript working with XMLHttpRequest, JavaScript working with HTML, XHTML, or HTML5; JavaScript working with the DOM, JavaScript working with CSS, JavaScript working with XML or JSON, and perhaps JavaScript working with other things.

While addressing a group of Django developers or Pythonistas, it would seem appropriate to open with, "I share your enthusiasm." On the other hand, while addressing a group of JavaScript programmers, in a few ways it is more appropriate to say, "I feel your pain." JavaScript is a language that has been discovered as a gem, but its warts were enough for it to be largely unappreciated for a long time. "Ajax is the gateway drug to JavaScript," as it has been said – however, JavaScript needs a gateway drug before people get hooked on it. JavaScript is an excellent language and a terrible language rolled into one.

Before discussing some of the strengths of JavaScript – and the language does have some truly deep strengths – I would like to say "I feel your pain" and discuss two quite distinct types of pain in the JavaScript language.

The first source of pain is some of the language decisions in JavaScript:

- The Wikipedia article says it was designed to resemble Java but be easier for non-programmers, a decision reminiscent of SQL and COBOL.
- The Java programmer who finds the C-family idiom of `for(i = 0; i < 100; ++i)` available will be astonished to find that the functions are clobbering each other's assignments to `i` until they are explicitly declared local to the function by declaring the variables with `var`. There is more pain where that came from.

The following two functions will not perform the naively expected mathematical calculation correctly; the assignments to `i` and the result will clobber each other:

```
function outer()
{
    result = 0;
    for(i = 0; i < 100; ++i)
    {
        result += inner(i);
    }
    return result
}

function inner(limit)
{
    result = 0;
    for(i = 0; i < limit; ++i)
    {
        result += i;
    }
    return result;
}
```


The second source of pain is quite different. It is a pain of inconsistent implementation: the pain of, "Write once, debug everywhere." Strictly speaking, this is not JavaScript's fault; browsers are inconsistent. And it need not be a pain in the server-side use of JavaScript or other non-browser uses. However, it comes along for the ride for people who wish to use JavaScript to do Ajax. Cross-browser testing is a foundational practice in web development of any stripe; a good web page with semantic markup and good CSS styling that is developed on Firefox will usually look sane on Internet Explorer (or vice versa), even if not quite pixel-perfect. But program directly for the JavaScript implementation on one version of a browser, and you stand rather sharp odds of your application not working at all on another browser. The most important object by far for Ajax is the `XMLHttpRequest` and not only is it not the case that you may have to do different things to get an `XMLHttpRequest` in different browsers or sometimes different (common) versions of the same browser, and, even when you have code that will get an `XMLHttpRequest` object, the objects you have can be incompatible so that code that works on one will show strange bugs for another. Just because you have done the work of getting an `XMLHttpRequest` object in all of the major browsers, it doesn't mean you're home free.

Before discussing some of the strengths of the JavaScript language itself, it would be worth pointing out that a good library significantly reduces the second source of pain. Almost any sane library will provide a single, consistent way to get `XMLHttpRequest` functionality, and consistent behavior for the access it provides. In other words, one of the services provided by a good JavaScript library is a much more uniform behavior, so that you are programming for only one model, or as close as it can manage, and not, for instance, pasting conditional boilerplate code to do simple things that are handled differently by different browser versions, often rendering surprisingly different interpretations of JavaScript. We will be using the jQuery library in this book as a standard, well-designed, lightweight library. Many of the things we will see done well as we explore jQuery are also done well in other libraries.

We previously said that JavaScript is an excellent language and a terrible language rolled into one; what is to be said in favor of JavaScript? The list of faults is hardly all that is wrong with JavaScript, and saying that libraries can dull the pain is not itself a great compliment. But in fact, something much stronger can be said for JavaScript: *If you can figure out why Python is a good language, you can figure out why JavaScript is a good language.*

I remember, when I was chasing pointer errors in what became 60,000 lines of C, teasing a fellow student for using Perl instead of a real language. It was clear in my mind that there were interpreted scripting languages, such as the bash scripting that I used for minor convenience scripts, and then there were *real* languages, which were compiled to machine code. I was sure that a real language was identified with being compiled, among other things, and that power in a language was the sort of thing C traded in. (I wonder why he didn't ask me if he wasn't a real programmer because he didn't spend half his time chasing pointer errors.) Within the past year or so I've been asked if "Python is a real programming language or is just used for scripting," and something similar to the attitude shift I needed to appreciate Perl and Python is needed to properly appreciate JavaScript.

The name "JavaScript" is unfortunate; like calling Python "Assembler Kit", it's a way to ask people not to see its real strengths. (Someone looking for tools for working on an assembler would be rather disgusted to buy an "Assembler Kit" and find Python inside. People looking for Java's strengths in JavaScript will almost certainly be disappointed.)

JavaScript code may look like Java in an editor, but the resemblance is a façade; besides Mocha, which had been renamed LiveScript, being renamed to JavaScript just when Netscape was announcing Java support in web browsers, it has been described as being descended from NewtonScript, Self, Smalltalk, and Lisp, as well as being influenced by Scheme, Perl, Python, C, and Java. What's under the Java façade is pretty interesting. And, in the sense of the simplifying "façade" design pattern, JavaScript was marketed in a way almost guaranteed not to communicate its strengths to programmers. It was marketed as something that nontechnical people could add snippets of, in order to achieve minor, and usually annoying, effects on their web pages. It may not have been a toy language, but it sure was dressed up like one.

Python may not have functions clobbering each other's variables (at least not unless they are explicitly declared global), but Python and JavaScript are both multiparadigm languages that support object-oriented programming, and their versions of "object-oriented" have a lot in common, particularly as compared to (for instance) Java. In Java, an object's class defines its methods and the type of its fields, and this much is set in stone. In Python, an object's class defines what an object starts off as, but methods and fields can be attached and detached at will. In JavaScript, classes as such do not exist (unless simulated by a library such as Prototype), but an object can inherit from another object, making a prototype and by implication a prototype chain, and like Python it is dynamic in that fields can be attached and detached at will. In Java, the `instanceof` keyword is important, as are class casts, associated with strong, static typing; Python doesn't have casts, and its `isinstance()` function is seen by some as a mistake, hence the blog posting "`isinstance()` considered harmful" at <http://www.canonical.org/~kragen/isinstance/>.

The concern is that Python, like JavaScript, is a duck-typing language: *If it looks like a duck, and it quacks like a duck, it's a duck!* In a duck-typing language, if you write a program that polls weather data, and there's a `ForecastFromScreenscraper` object that is several years old and screenscrapes an HTML page, you should be able to write a `ForecastFromRSS` object that gets the same information much more cleanly from an RSS feed. You should be able to use it as a drop-in replacement as long as you have the interface right. That is different from Java; at least if it were a `ForecastFromScreenscraper` object, code would break immediately if you handed it a `ForecastFromRSS` object. Now, in fairness to Java, the "best practices" Java way to do it would probably separate out an `IForecast` interface, which would be implemented by both `ForecastFromScreenscraper` and later `ForecastFromRSS`, and Java has ways of allowing drop-in replacements *if* they have been explicitly foreseen and planned for. However, in duck-typed languages, the reality goes beyond the fact that if the people in charge designed things carefully and used an interface for a particular role played by an object, you can make a drop-in replacement. In a duck-typed language, you can make a drop-in replacement for things that the original developers never imagined you would want to replace.

JavaScript's reputation is changing. More and more people are recognizing that there's more to the language than design flaws. More and more people are looking past the fact that JavaScript is packaged like Java, like packaging a hammer to give the impression that it is basically like a wrench. More and more people are looking past the silly "toy language" Halloween costume that JavaScript was stuffed into as a kid.

One of the ways good programmers grow is by learning new languages, and JavaScript is not just the gateway to mainstream Ajax; it is an interesting language in itself. With that much stated, we will be making a carefully chosen, selective use of JavaScript, and not make a language lover's exploration of the JavaScript language, overall. Much of our work will be with the jQuery library; if you have just programmed a little "bare JavaScript", discovering jQuery is a bit like discovering Python, in terms of a tool that cuts like a hot knife through butter. It takes learning, but it yields power and interesting results soon as well as having some room to grow.

XMLHttpRequest

The `XMLHttpRequest` object is the reason why the kind of games that can be implemented with Ajax technologies do not stop at clones of Tetris and other games that do not know or care if they are attached to a network. They include massive multiplayer online role-playing games where the network is the computer. Without having something like `XMLHttpRequest`, "Ajax chess" would probably mean a game of chess against a chess engine running in your browser's JavaScript engine; with `XMLHttpRequest`, "Ajax chess" is more likely man-to-man chess against another human player connected via the network. The `XMLHttpRequest` object is the object that lets Gmail, Google Maps, Bing Maps, Facebook, and many less famous Ajax applications deliver on Sun's promise: the network *is* the computer.

There are differences and some incompatibilities between different versions of `XMLHttpRequest`, and efforts are underway to advance "level-2-compliant" `XMLHttpRequest` implementations, featuring everything that is expected of an `XMLHttpRequest` object today and providing further functionality in addition, somewhat in the spirit of level 2 or level 3 CSS compliance. We will not be looking at level 2 efforts, but we will look at the baseline of what is expected as standard in most `XMLHttpRequest` objects.

The basic way that an `XMLHttpRequest` object is used is that the object is created or reused (the preferred practice usually being to reuse rather than create and discard a large number), a callback event handler is specified, the connection is opened, the data is sent, and then when the network operation completes, the callback handler retrieves the response from `XMLHttpRequest` and takes an appropriate action.

A bare-bones `XMLHttpRequest` object can be expected to have the following methods and properties.

Methods

A bare-bones `XMLHttpRequest` object can be expected to have the following methods:

1. `XMLHttpRequest.abort()`
This cancels any active request.
2. `XMLHttpRequest.getAllResponseHeaders()`
This returns all HTTP response headers sent with the response.
3. `XMLHttpRequest.getResponseHeader(headerName)`
This returns the requested header if available, or a browser-dependent false value if the header is not defined.

4. `XMLHttpRequest.open(method, URL),`
`XMLHttpRequest.open(method, URL, asynchronous),`
`XMLHttpRequest.open(method, URL, asynchronous, username),`
`XMLHttpRequest.open(method, URL, asynchronous, username,`
`password)`

The method is GET, POST, HEAD, or one of the other less frequently used methods defined for HTTP.

The URL is the relative or absolute URL to fetch. As a security measure for JavaScript running in browsers on trusted internal networks, a **same origin policy** is in effect, prohibiting direct access to servers other than one the web page came from. Note that this is less restrictive than it sounds, as it is entirely permissible for the server to act as a proxy for any server it has access to: for developers willing to undertake the necessary chores, other sites on the public internet are "virtually accessible".

The asynchronous variable defaults to `true`, meaning that the method call should return quickly in most cases, instead of waiting for the network operation to complete. Normally this default value should be preserved. Among other problems, setting it to `false` can lock up the visitor's browser.

The last two arguments are the username and password as optionally specified in HTTP. If they are not specified, they default to any username and password defined for the web page.

5. `XMLHttpRequest.send(content)`

Content can be a string or a reference to a document.

Properties

A bare-bones `XMLHttpRequest` object can be expected to have the following properties:

1. `XMLHttpRequest.onreadystatechange,`
`XMLHttpRequest.readyState`

In addition to the provided methods, the reference to one other method is supplied by the developer as a property, `XMLHttpRequest.onreadystatechange`, which is called without argument each time the ready state of `XMLHttpRequest` changes. An `XMLHttpRequest` object can have five ready states:

- Uninitialized, meaning that `open()` has not been called.
- Open, meaning that `open()` has been called but `send()` has not.
- Sent, meaning that `send()` has been called, and headers and status are available, but the response is not yet available.

- Receiving, meaning that the response is being downloaded and `responseText` has the portion that is presently available.
 - Loaded, meaning that the network operation has completed. If it has completed successfully (that is, the HTTP status stored in `XMLHttpRequest.status` is 200), this is when the web page would be updated based on the response.
2. `XMLHttpRequest.responseText`, `XMLHttpRequest.responseXML`
- The text of the response. It is important to note that while the name "XMLHttpRequest" is now very well established, and it was originally envisioned as a tool to get *XML*, the job done today is quite often to get *text* that may or may not happen to be XML. While there have been problems encountered with using `XMLHttpRequest` to fetch raw binary data, the `XMLHttpRequest` object is commonly used to fetch not only XML but XHTML, HTML, plain text, and JSON, among others. If it were being named today, it would make excellent sense to name it "*TextHttpRequest*." Once the request reaches a ready state of 4 ("loaded"), the `responseText` field will contain the text that was served up, whether the specific text format is XML or anything else. In addition, if the format does turn out to be XML, the `responseXML` field will hold a parsed XML document.
3. `XMLHttpRequest.status`, `XMLHttpRequest.statusText`
- The `status` field contains the HTTP code, such as 200 for OK; the `statusText` field has a short text description, like OK. The callback event handler should ordinarily check `XMLHttpRequest.readyState` and wait before acting on server-provided data until the `readyState` is 4. In addition, because there could be a server error or a network error, the callback will check whether the status is 200 or something else: a code like 4xx or 5xx in particular needs to be treated as an error. If the server-response has been transmitted successfully, the `readyState` will be 4 and the `status` will be 200.

This is the basic work that needs to be done for the `XMLHttpRequest` side of Ajax. Other frameworks may simplify this and do much of the cross-browser debugging work for you; we will see in the next chapter how jQuery simplifies this work. But this kind of task is something you will need to have done with any library, and it's worth knowing what's behind the simplified interfaces that jQuery and other libraries provide.

HTML/XHTML

HTML and XHTML make up the bedrock markup language for the web. JavaScript and CSS were introduced in relation to HTML; perhaps some people are now saying that JavaScript is a very interesting language independent of web browsers and using standalone interpreters such as SpiderMonkey and Rhino. However, HTML was on the scene first and other players on the web exist in relation to HTML's story. Even when re-implemented as XHTML, to do HTML's job while potentially making much more sense to parsers, a very early web page, the beginning of the source at <http://www.w3.org/History/19921103-hypertext/hypertext/WWW/TheProject.html>, is still quite intelligible:

```
<HEADER>
<TITLE>The World Wide Web project</TITLE>
<NEXTID N="55">
</HEADER>
<BODY>
<H1>World Wide Web</H1>The WorldWideWeb (W3)
is a wide-area
<A NAME=0 HREF="WhatIs.html">
hypermedia</A> information retrieval
initiative aiming to give universal
access to a large universe of documents.<P>
Everything there is online about
W3 is linked directly or indirectly
to this document, including an
<A NAME=24 HREF="Summary.html">executive
summary</A> of the project,
<A NAME=29 HREF="Administration/Mailing/Overview.html">
Mailing lists</A> ,
<A NAME=30 HREF="Policy.html">Policy</A> , November's
<A NAME=34 HREF="News/9211.html">W3 news</A> ,
<A NAME=41 HREF="FAQ/List.html">Frequently Asked Questions
</A> .
<DL>
...
```

At the time of this writing, HTML 5 is taking shape but is not "out in the wild", and so there are no reports of how the shoe feels after the public has worn it for a while. Code in this book, where possible, will be written in XHTML 1.0 Strict. Depending on your situation, this may or may not be the right decision for you; if you are working with an existing project, the right HTML/XHTML is often the one that maintains consistency within the project.

XML

eXtensible Markup Language (XML) is tied to an attempt to clean up early HTML. At least in earliest forms, HTML was a black sheep among specific markup languages derived from the generalized and quite heavy **Standard Generalized Markup Language (SGML)**. Forgiving web browsers meant, in part, that early web hobbyists could write terrible markup and it would still display well in a browser. The amount of terrible markup on the web was not just an issue for purists; it meant that making a parser that could make sense of early "Wild West" web pages in general was a nearly impossible task. XML is vastly simplified from SGML, but it provides a generic space where an HTML variant, XHTML, could pick up the work done by HTML but not present parsers with unpredictable tag soup. XHTML could be described as HTML brought back into the fold, still good for doing web development, but without making machine interpretation such a hopeless cause. Where early HTML was developed with browsers that were meant to be forgiving, XML requested draconian error handling, and validated XML or XHTML documents are documents that can be parsed in a sensible way.

XML works for exchanging information, and it works where many of its predecessors had failed: it provides interoperability between different systems after a long history of failed attempts at automating B2B communication and failed attempts at automated conversion between text data formats. Notwithstanding this, it is a heavy and verbose solution, with a bureaucratic ambiance, compared in particular to a lean, mean JSON. XML-based approaches to data storage and communication are increasingly critiqued in discussions on the web. If you have a reasonable choice between XML and JSON, we suggest that you seriously consider JSON.

JSON

JavaScript Object Notation (JSON) is a brilliantly simple idea. While formats like XML, ReStructuredText, and so on share the assumption that "if you're going to parse this from your language, your language will need to have a parser added," JSON simply takes advantage of how an object would be specified in JavaScript, and clarifies a couple of minor points to make JSON conceptually simpler and cross-browser friendly. JSON is clear, simple, and concise enough that not only is it a format of choice for JavaScript, but it is gaining traction in other languages, and it is being used for communication between languages that need a (simple, added) parser to parse JSON. The other languages can't use `eval()` to simply run JSON, and in JavaScript you should have JSON checked to make sure it does not contain malicious JavaScript you should not `eval()`. However, JSON is turning out to have a much broader impact than the initial "in communicating with JavaScript, just give it code to declare the object being communicated that can simply be evaluated to construct the object."

CSS

Cascading Style Sheets (CSS) may have introduced some new possibilities for presentation, but quite a lot of presentation was already possible beforehand. CSS did not so much add styling capabilities, as it added good engineering *to* styling (good engineering is the essence of "separating presentation from content"), and make the combination of semantic markup *and* attractive appearance a far more attainable goal. It allows parlor tricks such as in-place rebranding of websites: making changes in images and changing one stylesheet is, at least in principle, enough to reskin an extensive website without touching a single character of its HTML/XHTML markup. In Ajax, as for the rest of the web, the preferred practice is to use semantic, structural markup, and then add styles in a stylesheet (not inline) so that a particular element, optionally belonging to the right class or given the right ID, will have the desired appearance. Tables are not deprecated but should be used for semantic presentation of tabular data where it makes sense to use not only a `td` but a `th` as well. What is discouraged is using the side effect that tables can position content that is not, semantically speaking, tabular data.

The DOM

As far as direct human browsing is concerned, HTML and associated technologies are vehicles to deliver a pickled **Document Object Model (DOM)**, and nothing more. In this respect, HTML is a means to an end: the DOM is the "deserialized object," or better, the "live form" of what we really deliver to people. HTML may help provide a complete blueprint, and the "complete blueprint" is a means to the "fully realized building." This is why solving Ajax problems on the level of HTML text are like answering the wrong question, or at least solving a problem on the wrong level. It is like deciding that you want a painting hung on a wall of a building, and then going about getting it by adding the painting to the blueprint and asking construction personnel to implement the specified change. It may be better to hang the painting on the wall directly, as is done in Ajax DOM manipulations.

`document.write()` and `document.getElementById().innerHTML()` still have a place in web development. It is a sensible optimization to want a static, cacheable HTML/XHTML file include that will only be downloaded once in the usual multi-page visit. A JavaScript include with a series of `document.write()` may be the least Shanghaiing you can do to technologies and still achieve that goal. But this is *not* Ajax; it is barely JavaScript, and this is not where we should be getting our bearings. In Ajax, a serious alternative to this kind of solution for altering part of a web page is with the DOM.

As the book progresses, we will explore Ajax development that works with the DOM.

iframes and other Ajax variations

Ajax includes several variations; **Comet** for instance, is a variation on standard Ajax in which either an XMLHttpRequest object's connection to a server is kept open and streaming indefinitely, or a new connection is opened whenever an old one is closed, creating an Ajax environment in which the server as well as the client can push material. This is used, for instance, in some instant messaging implementations. One much more essential Ajax variation has to do with loading documents into seamlessly integrated iframes instead of making DOM manipulations to a single, frame-free web page.

If you click around on the page for a Gmail account, you will see partial page refreshes that look consistent with Ajax DOM manipulations: what happens when you click on **Compose Mail**, or a filter, or a message subject, looks very much like an Ajax update where the Gmail web application talks with the server if it needs to, and then updates the DOM in accordance with your clicks. However, there is one important difference between Gmail's behavior and a similar Ajax clone that updates the DOM for one frameless web page: what happens when you click the browser "Back" button. Normally, if you click on a link, you trigger an Ajax event but not a whole page refresh, and Ajax optionally communicates with a server and updates some part of the DOM. This does not register in the browser's history, and hitting the **Back** button would not simply reset the last Ajax partial page update. If you made an Ajax clone of Gmail that used DOM manipulations instead of seamlessly integrated iframes, there would be one important difference in using the clone: hitting **Back** would do far more than reverse the last DOM manipulation. *It would take you back to the login or load screen.* In Gmail, the browser's **Back** button works with surgical accuracy, and the reason it can do something much better than take you back to the login screen is that Gmail is carefully implemented with iframes, and every change that the **Back** button can undo is implemented by a fresh page load in one of the seamlessly integrated iframes. That creates browsing history.

For that matter, a proof of concept has been created for an Ajax application that does not use client-side scripting or programming, instead using, on the client side, a system of frames/iframes, targets, links, form submissions, and meta refresh tags in order to perform partial page updates. Whether this variant technique lends itself to creating graceful alternatives to standard Ajax implementations, or is only a curiosity merely lending itself to proofs of concept, it is in principle possible to make an Ajax application that loses nothing if a visitor's browser has turned off scripting completely.

Comet and iframes are two of many possible variations on the basic Ajax technique; what qualifies as Ajax is more a matter of Python- or JavaScript-style duck-typing than Java-style static typing. "Asynchronous JavaScript and XML" describes a reference example more than a strict definition, and it is not appropriate to say "if

you replace XML with JSON then, by definition, it isn't really Ajax." This is a case of, "the proof of the pudding is in the eating," not what technologies or even techniques are in the kitchen.

JavaScript/Ajax Libraries

This book advocates taking advantage of libraries, and as a limitation of scope focuses on jQuery. If you only learn one library, or if you are starting with just one library, jQuery is a good choice, and it is widely used. It is powerful, but it is also a much easier environment to get started in than some other libraries; in that way, it is somewhat like Python. However, it is best not to ask, "Which one library is best?" but "Which library or libraries are the right tools for this job?", and it is common real-world practice to use more than one library, possibly several.

JavaScript libraries offer several advantages. They can reduce chores and boilerplate code, significantly lessening the pain of JavaScript, and provide a more uniform interface. They can also provide (for instance) ready-made widgets; we will be working with a jQuery slider later on in this book. And on a broad scale, they can let the JavaScript you write be higher-level and a little more Pythonic.

Server-side technologies

Many of the "usual suspects" in client-side technologies have been mentioned. The list of client-side technologies is generally constrained by what is available in common web browsers; the list of available server-side technologies is only constrained by what will work on the server, and any general-purpose programming language *can* do the job. The question on the server is not "What is available?" but "Which option would you choose?" Python and Django make an excellent choice of server-side technology, and we will work with them in this book.

A look at Django

Django's developers call it "the web framework for perfectionists with deadlines," and it is one of the most popular Python web frameworks, perhaps the most popular. In contrast to the MVC pattern, which separates concerns into Model, View, and Controller, it could be described as an MTV pattern, which separates concerns into Model, Template, and View. The **Model** is a class that ties into an ORM where instances correspond to rows in the table but act and feel like Python objects. The **Template** is a system designed to be easy for non-Python developers (though easy for Pythonistas too), and limits the extent to which HTML needs to be sprinkled throughout the Python source. The **View** is a function that renders, in most cases, from a template. Let's look at a kickstart example of Django in action.

Django templating kickstart

Let us briefly go through how to install Django, create a sample project, and create and use a basic template that can serve as a basis for further tinkering.

Django installation instructions are at <http://docs.djangoproject.com/en/dev/intro/install/>; for Ubuntu, for instance, you will want to run `sudo apt-get install python-django`.

Once you have Django installed, create a project named `sample`:

```
django-admin.py startproject sample
```

Go into the `sample` directory, and create the directory `templates`. Enter the `templates` directory.

Create a template file named `index.html` containing the following template:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en-US" lang="en-US">
  <head>
    <title>{% block title %}Hello, world!{% endblock title %}
  </title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
  </head>
  <body>
    {% block body %}
      <h1>{% block heading %}Hello, world!
    {% endblock heading %}</h1>
      {% block content %}<p>Greetings from the Django templating engine!</p>{% endblock content %}
    {% endblock body %}
  </body>
</html>
```

Go up one level to the `sample` directory and edit the `urls.py` file so that the first line after `urlpatterns = patterns('', is:`

```
(r'^$', 'sample.views.home'),
```

Then create the `views.py` file containing the following:

```
#!/usr/bin/python/

from django.shortcuts import render_to_response

def home(request):
    return render_to_response(u'index.html')
```

Edit the `settings.py` file, and add:

```
os.path.join(os.path.dirname(__file__), "templates"),
```

right after:

```
TEMPLATE_DIRS = (
```

Then, from the command line, run:

```
python manage.py runserver
```

This makes the server accessible to your computer only by entering the URL `http://localhost:8080/` in your web browser.

If you are in a protected environment behind a firewall, appropriate NATting, or the like, you can make the development server available to the network by running:

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

There is one point of clarification we would like to make clear. Django is packaged with a minimal, single-threaded web server that is intended to be just enough to start exploring Django in a development environment. Django's creators are attempting to make a good, competitive web framework and not a good, competitive web server, and *the development server has never undergone a security audit*. The explicit advice from Django's creators is: when deploying, use a good, serious web server; they also provide instructions for doing this.

A more complete glimpse at Django templating

Before further exploring technical details, it would be worth taking a look at the opinions and philosophy behind the Django templating language, because an understandable approach of, "Oh, it's a general purpose programming language used for templating," is a recipe for needless frustration and pain. The Django developers themselves acknowledge that their opinions in the templating language are one just opinion in an area where different people have different opinions, and you are welcome to disagree with them if you want. If you don't like the templating system that Django comes with, Django is designed to let you use another. But it is worth understanding what exactly the philosophy is behind the templating language; even if this is not the only philosophy one could use, it is carefully thought out.

The Django templating language is intended to foster the separation of presentation and logic. In its design decisions, both large and small, Django's templating engine is optimized primarily for designers to use for designing, rather than programmers to use for programming, and its limitations are almost as carefully chosen as the features it provides. Unlike ASP, JSP, and PHP, it is not a programming language interspersed with HTML. It provides enough power for presentation, and is intended *not* to provide enough power to do serious programming work where it doesn't belong (in the Django opinion), and is simple enough that some non-programmers can pick it up in a day. For a programmer, the difficulty of learning the templating basics is comparable to the difficulty of simple HTML or SQL: it is simple, and a good bit easier to learn than wrapping your arms around a regular programming language. Some programmers like it immediately, but there are some who started by asking, "Why doesn't the templating language just let you mix Python and HTML?" and after playing with it, found themselves saying, "This isn't what I would have come up with myself, but I really, really like it."

Additional benefits include it being *fast* (most of the work is done by a single regular expression call, and the founders talk about disabling caching because it wasn't as fast as the template rendering), *secure* (it is designed so that it can be used by untrusted designers without allowing a malicious designer to execute arbitrary code), and *versatile* enough to generate whatever text format you want: plain text, HTML, XML, XHTML, JSON, JavaScript, CSV, ReStructuredText, and so on. We will be using it to generate web pages and JSON, but Django's templating language is a general-purpose text templating solution.

Following the Django site's lead, let us use a template intended as an example of how one might begin a base template for a site, then start to walk through its contents, and then look at some of how it could be used and parts overridden to create a specific document.

This renders as follows, if we strip out blank lines:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en-US"
lang="en-US">
  <head>
    <title></title>
    <link rel="icon" href="/static/favicon.ico"
      type="x-icon" />
    <link rel="shortcut icon" href="/static/favicon.ico"
      type="x-icon" />
    <meta http-equiv="Content-Type" content="text/html;
      charset=UTF-8" />
    <meta http-equiv="Content-Language" value="en-US" />
```

```

    <link rel="stylesheet" type="text/css"
        href="/static/css/style.css" />
</head>
<body>
    <div id="sidebar">
    </div>
    <div id="content">
        <div id="header">
            <h1></h1>
        </div>
    </div>
    <div id="footer">
    </div>
</body>
<script language="JavaScript" type="text/javascript"
    src="/static/js/jquery.js"></script>
</html>

```

Let us unwrap what is going on here; there is more to the template than how it renders to this page, but let us start with that much.

The `{% block dtd %}` style tags begin, or the case of `{% endblock dtd %}` end, a semantic block of text that can be left untouched or can be replaced. In the case of this one template, the effect is to strip them out like comments, but they will yield benefits later on, much like semantic HTML markup with CSS yields benefits later on.

Django templating reflects a choice to go with hooks rather than includes because hooks provide the more versatile solution. The "beginner's mistake" version of a header to include might be something like the following:

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.
w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en-US"
lang="en-US">
    <head>
        <title>Welcome to my site!</title>
        <link rel="icon" href="/favicon.ico" type="x-icon" />
        <link rel="shortcut icon" href="/favicon.ico"
            type="x-icon" />
        <meta http-equiv="Content-Type" content="text/html;
            charset=UTF-8" />
        <meta http-equiv="Content-Language" value="en-US" />
        <link rel="stylesheet" type="text/css"
            href="/css/style.css" />
    </head>
    <body>

```

This solution could be continued by adding a sidebar, but there is a minor problem, or at least it seems minor at first: there are bits of this header that are not generic. For a serious site, having every page titled, "Welcome to my site!" would be an embarrassment. The language is declared to be "en-US", meaning U.S. English, which is wonderful if the entire site is in U.S. English, but if it expands to include more than U.S. English content, hardcoding "en-US" will be a problem. If the only concern is to accurately label British English, then the more expansive "en" could be substituted in, but hardcoding "en-US" and "en" are equally unhelpful if the site expands to feature a section in Russian. This header does not include other meta tags that might be desirable, such as "description", which is ideally written for a specific page and not done as site-wide boilerplate. Including the header verbatim solves a problem, but it doesn't provide a very flexible solution.

The previous example builds in hooks. It does specify:

```
{% block html_tag %}<html xmlns="http://www.w3.org/1999/xhtml" xml:
lang="en-US" lang="en-US">{% endblock html_tag %}
```

If not overridden, this will render as:

```
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en-US"
lang="en-US">
```

However, in a template that extends this, by having `{% extends "base.html" %}` as its opening tag, if the base is loaded as `base.html`, then:

```
{% block html_tag %}<html xmlns="http://www.w3.org/1999/xhtml" xml:
lang="en-GB" lang="en-GB">{% endblock html_tag %}
```

will render:

```
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en-GB"
lang="en-GB">
```

And Russian may be declared in the same way:

```
{% block html_tag %}<html xmlns="http://www.w3.org/1999/xhtml" xml:
lang="ru-RU" lang="ru-RU">{% endblock html_tag %}
```

will render:

```
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="ru-RU"
lang="ru-RU">
```

The template as given does specify "en-US" more than once, but each of these is inside a block that can be overridden to specify another language.

We define initial blocks. First, the DTD:

```
{% block dtd %}<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//
EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
{% endblock dtd %}
```

Then, the HTML tag:

```
{% block html_tag %}<html xmlns="http://www.w3.org/1999/xhtml" xml:
lang="en-US" lang="en-US">{% endblock html_tag %}
```

Then we define the head, with the title and favicon:

```
{% block head %}<head>
<title>{% block title %}{{ page.title }}
{% endblock title %}</title>
{% block head_favicon %}<link rel="icon"
href="/static/favicon.ico" type="x-icon" />
<link rel="shortcut icon" href="/static/favicon.ico"
type="x-icon" />{% endblock head_favicon %}
```

Then we define hooks for meta tags in the head. We define a Content-Type of UTF-8; this is a basic point so that non-ASCII content will display correctly:

```
{% block head_meta %}
{% block head_meta_author%}{% endblock head_meta_author %}
{% block head_meta_charset %}
<meta http-equiv="Content-Type" content="text/html;
charset=UTF-8" />
{% endblock head_meta_charset %}
{% block head_meta_contentlanguage %}
<meta http-equiv="Content-Language" value="en-US" />
{% endblock head_meta_contentlanguage %}
{% block head_meta_description %}
{% endblock head_meta_description %}
{% block head_meta_keywords %}
{% endblock head_meta_keywords %}
{% block head_meta_othertags %}
{% endblock head_meta_othertags %}
{% block head_meta_refresh %}
{% endblock head_meta_refresh %}
{% block head_meta_robots %}
{% endblock head_meta_robots %}
{% endblock head_meta %}
```

We declare a block to specify an RSS feed for the page:

```
{% block head_rss %}{% endblock head_rss %}
```

We add hooks for CSS, both at the site level, and section, and page. This allows a fairly fine granularity of control:

```
{% block head_css %}
  {% block head_css_site %}
    <link rel="stylesheet" type="text/css"
        href="/static/css/style.css" />
  {% endblock head_css_site %}
  {% block head_css_section %}
  {% endblock head_css_section %}
  {% block head_css_page %}{% endblock head_css_page %}
{% endblock head_css %}
```

We add section- and page-specific header information:

```
{% block head_section %}{% endblock head_section %}
{% block head_page %}{% endblock head_page %}
```

Then we close the head and open the body:

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

We define a sidebar block, with a hook to populate it:

```
<div id="sidebar">
  {% block body_sidebar %}{% endblock body_sidebar %}
</div>
```

We define a block for the main content area:

```
<div id="content">
  {% block body_content %}
```

For the header of the main content area, we define a banner hook, and a header for the page's title, should such be provided. (If none is provided, there is no crash or error; the empty string is displayed for `{{ page.title }}`.)

```
<div id="header">
  {% block body_header %}
    {% block body_header_banner %}
    {% endblock body_header_banner %}
    {% block body_header_title %}<h1>
      {{ page.title }}</h1>
    {% endblock body_header_title %}
```

We define a breadcrumb, which is one of many small usability touches that can be desirable:

```
{% block body_header_breadcrumb %}
  {{ page.breadcrumb }}
{% endblock body_header_breadcrumb %}
{% endblock body_header %}
</div>
```

We add a slot for announcements, then the body's main area, and then close the block and div:

```
{% block body_announcements %}
{% endblock body_announcements %}
{% block body_main %}{% endblock body_main %}
{% endblock body_content %}
</div>
```

We define a footer div, with a footer breadcrumb, and a hook for anything our company's lawyers asked us to put:

```
<div id="footer">
{% block body_footer %}
  {% block body_footer_breadcrumb %}
    {{ page.breadcrumb }}
  {% endblock body_footer_breadcrumb %}
  {% block body_footer_legal %}
  {% endblock body_footer_legal %}
{% endblock body_footer %}
```

Now we close that div, the body, and the body block:

```
</div>
</body>{% endblock body %}
```

We add a footer, with JavaScript blocks, again at the site/section/page level of hooks:

```
{% block footer %}
  {% block footer_javascript %}
    {% block footer_javascript_site %}
      <script language="JavaScript" type="text/javascript"
        src="/static/js/jquery.js"></script>
    {% endblock footer_javascript_site %}
    {% block footer_javascript_section %}
    {% endblock footer_javascript_section %}
    {% block footer_javascript_page %}
```

```
        {% endblock footer_javascript_page %}
    {% endblock footer_javascript %}
    {% endblock footer %}
</html>
```

And that's it.

You can make as many layers of templates as you want. One suggested approach is to make three layers: one base template for your entire site, then more specific templates for sections of your site (whatever they may be), and then individual templates for end use. The template given is a base template, and it provides hooks as narrow as a specific meta tag or as broad as the main content area.

For our extended example, if it is named `base.html`, then we can create another template, `russian.html`, which will declare its content to be in the Russian language. (Ordinarily one would do more interesting things in overriding a template than merely replacing tags, but for illustration purposes we will do that, and only that:

```
{% extends "base.html" %}
{% block html_tag %}<html xmlns="http://www.w3.org/1999/xhtml" xml:
lang="ru-RU" lang="ru-RU">{% endblock html_tag %}
{% block head_meta_contentlanguage %}<meta http-equiv="Content-
Language" value="ru-RU" />{% endblock head_meta_contentlanguage %}
```

These block overrides may occur anywhere; while the `{% extends "base.html" %}` tag must be placed first, the two tags may be swapped. It would work just as well to create a language-agnostic `base.html` with only an empty hook:

```
{% block head_meta_contentlanguage %}
{% endblock head_meta_contentlanguage %}
```

and a default HTML tag of:

```
<html xmlns="http://www.w3.org/1999/xhtml">
```

And then create `english.html` and `russian.html`, or `en-US.html` and `ru-RU.html`, as base templates for those languages.

However, there are many other tags than those used for blocks and overriding. We will just barely sample them below, looking at how to display variables, and then other tags.

There are other tags that look like `{% ... %}`, but I would comment briefly on the variable tags such as `{{ page.title }}`. The Django templating language uses dotted references, but not in exactly the same way as Python's dotted references. Where Python requires a couple of different things to get values, dotted references provide one-stop shopping in Django's templating. When a reference to `{{ page.title }}` occurs, it will display `page[u'title']` if `page[u'title']` is available. If not, it will display `page.title` if `page.title` is available as an attribute, and if there is no such attribute, it will display `page.title()` if `page.title()` is available, and failing that, if the reference is a non-negative integer like 2, it will display `page[2]` if `page.2` is requested. If all of these fail, then Django defaults to the empty string because it's not acceptable for a professional site to crash because a programming error has the template asking for something that is not available. You can override the empty string by setting `TEMPLATE_STRING_IF_INVALID` in your `settings.py` file, and *in development* it may make sense to set `TEMPLATE_STRING_IF_INVALID` to something like `LOOKUP FAILED`, but you will want to set it back to the empty string for deployment in any production environment.

At this point, while there are explicit hooks to pull in multiple JavaScript and CSS files, the preferred practice, per Steve Souders's ground rules for client-side optimizations for high performance websites, is: for each page to load initially, you should have one HTML/XHTML page, one CSS file included at the top, and one JavaScript file included at the bottom. The hooks are more flexible than that, but this is intended more as "development leeway" than what the tightened final product should be.

If-then, if-then-else statements, and for loops are straightforward, and else clauses are optional:

```
{% if results %}
<ul>
  {% for result in results %}
    <li>{% result.title %}</li>
  {% endfor %}
</ul>
{% else %}
  <p>There were no results.</p>
{% endif %}
```

There are a number of convenience features and minor variations available; these are several of the major features.

Setting JavaScript and other static content in place

For the development server, putting static content, including images, CSS, and static content, is straightforward. For production use, the recommended best practice is to use a different implementation, and Django users are advised to use a separate server if possible, optimized for serving static media, such as a *stripped-down* build of Apache, or nginx. However, for development use, the following steps will serve up static content:

1. Create a directory named `static` within your project. (Note that other names may be used, but do not use `media`, as that can collide with administrative tools.)
2. Edit the `settings.py` file, and add the following at the top, after `import os`:

```
DIRNAME = os.path.abspath(os.path.dirname(__file__))
```
3. Change the settings of `MEDIA_ROOT` and `MEDIA_URL`:

```
MEDIA_ROOT = os.path.join(DIRNAME, 'static/')
...
MEDIA_URL = '/static/'
```
4. At the end of the `settings.py` file, add the following:

```
if settings.DEBUG:
    urlpatterns += patterns('django.views.static',
        (r'^%s(?P<path>.*)$' % (settings.MEDIA_URL[1:],), 'serve', {
            'document_root': settings.MEDIA_ROOT,
            'show_indexes': True })),)
```

This will turn off static media service when `DEBUG` is turned off, so that this code does not need to be changed when your site is deployed live, but the subdirectory `static` within your project should now serve up static content, like a very simplified Apache. We suggest that you create three subdirectories of `static`: `static/css`, `static/images`, and `static/js`, for serving up CSS, image, and JavaScript static content.

Summary

Guido van Rossum, the creator of the Python programming language, for one project asked about different Python frameworks and chose the Django templating engine for his purposes (<http://www.artima.com/weblogs/viewpost.jsp?thread=146606>). This chapter has provided an overview of Ajax and then provided a kickstart introduction to the Django templating engine. There's more to Django than its templating engine, but this should be enough to start exploring and playing.

In learning a new technology, a crucial threshold has been passed when there is enough of a critical mass of things you can do with a technology to begin tinkering, and taking one step often invites the question: "What can we do to take this one step further?". In this chapter, we have provided a kickstart to begin working with the Django templating engine.

In this chapter, we have looked at the idea of Pythonic problem solving, discussed Django and jQuery in relation to Pythonic problem solving, and discussed Ajax as not a single technology, but an *overlaid function* or technique that is overlaid on top of existing technologies. We have taken an overview of what the "usual suspect" technologies are for Ajax; given a kickstart to the Django templating engine, introducing some of its beauty and power; and addressed a minor but important detail: putting static content in place for Django's development server.

This is meant to serve as a point of departure for further discussion of jQuery Ajax in the next chapter, and building our sample application. Interested readers who want to know more of what they can do can read the official documentation at <http://docs.djangoproject.com/en/dev/topics/templates/>.

In the next chapter, we will push further and aim for a critical mass of things we can do. We will explore jQuery, the most common JavaScript library, and begin to see how we can use it to reach the point of tinkering, of having something that works and wondering, "What if we try this?", "What if we try that?", and being able to do it.

2

jQuery—the Most Common JavaScript Framework

JavaScript frameworks can offer at least two kinds of advantages, as we have discussed earlier. Firstly, they can offer considerable facilities when compared to building from scratch. TurboGears and Django are both server-side web frameworks in Python and both offer major advantages compared to web application development in Python using only the standard library, even with the CGI module included. Secondly, they can offer a more uniform virtual programming interface compared to writing unadorned JavaScript with enough detection and conditional logic to directly conform to disparate JavaScript environments so that code written in one browser and debugged in another browser stands a fighting chance of working without crashing in every common browser.

As regards the question, "Which is best?" if you only learn one client-side JavaScript framework, jQuery is an excellent choice. It is one of the most popular JavaScript frameworks, partly because it is (like Python) gentle to newcomers. However, comparing jQuery to Dojo is like comparing a hammer to a wrench. (A general JavaScript framework comparison is at http://en.wikipedia.org/wiki/Comparison_of_JavaScript_frameworks.) Unlike server-side web development frameworks such as Django and TurboGears, you don't necessarily ask, "Which one will we use for this project?" as there might be good reasons to use more than one framework in a web application. jQuery's developers know it isn't the only thing out there. It only uses two names in the global namespace: `$` and `jQuery`. Both do the same thing; you can always use `jQuery()` in lieu of `$()`, and if you call `jQuery.noConflict()`, jQuery will let go of `$` so that other libraries like Prototype (<http://www.prototypejs.org/>) are free to use that name.

Finally, we suggest that jQuery might be something like a "virtual higher-level language" built on top of JavaScript. A programmer's first reaction after viewing the JavaScript code, on seeing code using jQuery, might well be, "Is that really JavaScript? *It doesn't look like JavaScript!*" A programmer using jQuery can operate on element sets, and refine or expand, without ever typing out a JavaScript variable assignment or keyword such as `if`, `else`, or `for`, and even without cross-browser concerns, one runs into jQuery code that does the work of dozens of lines of "bare JavaScript" in a single line. But these are single lines that can be very easily read once you know the ground rules, not the cryptic one-liners of the Perl art form. This "virtual higher-level language" of jQuery is not Python and does not look like Python, but there is something "Pythonic in spirit" about how it works.

In this chapter, we will cover:

- How jQuery simplifies "Hello, world!" in Ajax
- The basic Ajax facilities jQuery offers
- How jQuery effectively offers a "virtual higher-level language"
- jQuery selectors, as they illustrate the kind of more Pythonic "virtual higher-level language facilities" that jQuery offers
- A sample "kickstart Django Ajax application," which covers every major basic feature except for server-side persistence management
- An overview of a more serious, in-depth Django application, to be covered in upcoming chapters

Let's explore the higher-level way to do things in jQuery.

jQuery and basic Ajax

Let's look at a "Hello, world!" in Ajax. A request containing one variable, `text`, is sent to the server, and the server responds with "Hello, [*contents of the variable text*]!" which is then put into the `innerHTML` of a paragraph with ID `result`:

```
if (typeof XMLHttpRequest == "undefined")
{
    XMLHttpRequest = function()
    {
        try
        {
            return new ActiveXObject("Msxml2.XMLHTTP.6.0");
        }
        catch(exception)
        {

```

```

        try
        {
            return new ActiveXObject("Msxml2.XMLHTTP.3.0");
        }
        catch(exception)
        {
            try
            {
                return new ActiveXObject("Msxml2.XMLHTTP");
            }
            catch(exception)
            {
                throw new Error("Could not construct
                                XMLHttpRequest");
            }
        }
    }
}

var xhr = new XMLHttpRequest();
xhr.open("GET", "/project/server.cgi?text=world");
callback = function()
{
    if (xhr.readyState == 4 && xhr.status >= 200 && xhr.status < 300)
    {
        document.getElementById("result").innerHTML =
            xhr.responseText;
    }
}
xhr.onreadystatechange = callback;
xhr.send(null);

```

With the simpler, "virtual high-level language" of jQuery, you accomplish more or less the same with only:

```
$("#result").load("/project/server.cgi", "text=world");
```

`$()` selects a wrapped set. This can be a CSS designation, a single HTML entity referenced by ID as we do here, or some other things. jQuery works on wrapped sets. Here, as is common in jQuery, we create a wrapped set and call a function on it. `$.load()` loads from a server URL, and we can optionally specify a query string.

How do we do things with jQuery? To start off, we create a **wrapped set**. The syntax is based on CSS and CSS2 selectors, with some useful extensions.

Here are some examples of creating a wrapped set:

```
$("#result");
$("p");
$("p.summary");
$("a");
$("p a");
$("p > a");
$("li > p");
$("p.summary > a");
$("table.striped tr:even");
```

They select nodes from the DOM more or less as one would expect:

- `$("#result#")` selects the item with HTML ID `result`
- `$("p")` selects all paragraph elements
- `$("p.summary")` selects all paragraph elements with class `summary`
- `$("a")` selects all anchor tags
- `$("p a")` selects all anchor tags in a paragraph
- `$("p > a")` selects all anchor tags whose immediate parent is a paragraph tag
- `$("li > p")` selects all `p` tags whose immediate parent is a `li` tag
- `$("p.summary > a")` selects all anchor tags whose immediate parent is a paragraph tag with class `summary`
- `$("table.striped tr:even")` selects even-numbered rows from all tables belonging to class `striped`

The difference between `$("p a")` and `$("p > a")` is that the first selects any tag contained in a `p` tag, whether its immediate parent is the `p` tag or whether there are intervening `span`, `em`, or `strong` tags, and the second selects only those tags whose immediate parent is a `p` tag; the latter yields a subset of the former. This last approach lends itself to a straightforward and clean way to address the chore of tiger-stripping tables:

```
$("table.striped tr:even").addClass("even");
```

The statements before this last example are not useful in themselves, but they lay a powerful foundation by creating a wrapped set. A wrapped set is an object that encapsulates both a set of DOM elements and a full complement of operations, such as `.addClass()`, which assigns a CSS class to all elements from a wrapped set, or `.load()`, which provided a one-line Ajax solution earlier.

One of the key features of a wrapped set is that these operations return the same wrapped set, unless they are one of a few exceptions, such as operations designed to change the set by adding or removing members. This means that operations can be **chained**; for a somewhat artificial example, our code to stripe items in a table could be extended to slowly hiding them, waiting for a second (specified as 1000 milliseconds), and then showing them again:

```
$("#table.striped tr:even").addClass("even").hide("slow").delay(1000).
show("slow");
```

In this statement, all of the operations return the same wrapped set. Dozens of further operations could be appended after the `.show()` call if desired. In chaining of operations, the additional functions can function as separate "virtual statements". The previous line of code, in pseudocode, could be written:

```
Select even-numbered table rows from tables having the class
"striped".
Add the class "even" to them.
Slowly hide them.
Wait 1000 milliseconds.
Slowly show them.
```

The final appearance of the page should be as if merely `$("#table.striped tr:even").addClass("even");` had been called.

jQuery Ajax facilities

jQuery provides good facilities both for Ajax and other JavaScript usage. The API is excellent and can be bookmarked from <http://api.jquery.com/>.

We will be going through some key facilities, Ajax and otherwise, and making comments on how these facilities might best be used.

\$.ajax()

The most foundational and low-level workhorse in jQuery is `$.ajax()`. It takes the arguments as shown in the following *Sample invocation*. Default values can be set with `$.ajaxSetup()`, as discussed below:

```
$.ajax({data: "surname=Smith&cartTotal=12.34", dataType: "text",
error: function(XMLHttpRequest, textStatus, errorThrown) {
    displayErrorMessage("An error has occurred: " + textStatus);
}, success: function(data, textStatus, XMLHttpRequest) {
    try
    {
```

```
        updatePage (JSON.parse (data)) ;
    }
    catch (error)
    {
        displayErrorMessage ("There was an error updating your
        shopping cart. Please call customer service at
        800-555-1212.");
    }
    }, type: "POST", url: "/update-user");
```

Some of the fields that can be passed are described in the following sections.

context

The context, available as the variable `this`, can be used to give access to variables in callbacks, even if they have fallen out of scope. The following code prompts for the user's name and e-mail address and provides them in a context, anonymously, so that the data are available to the callback function but are never a part of the global namespace.

```
$.ajax({success: function(data, textStatus, XMLHttpRequest)
{
    alert(this.name + ", your email address is " +
        this.email + ".");
    processData(data, this.name, this.email);
}, context:
{
    name: prompt("What is your name?", ""),
    email: prompt("What is your email address?", "")
}, ...
});
```

Closures

The principle at play is the principle of **closures**. The wrong way, perhaps, to learn about closures is to look for academic-style introductions to what you need to know in order to understand them. There's a lot there, and it's a lot harder to understand than learning by jumping in. Closures represent a key concept in core JavaScript, along with other things such as functions, objects, and prototypes. Closures are part of the conceptual landscape and not only because they are the basis on how to effectively create an object with private fields. So we will jump in and give a closure that serves as a proof of concept as, effectively, an object with private fields. To give a standard, "bare JavaScript" example of using a closure to create an object with private variables, which creates an object that stores an integer value, has a getter and setter, but in Java fashion ensures that the field can only have an integer value.

We define a function, which we will be immediately evaluating; `closureExample` stores not the anonymous function but its return value. The variable `field` is a local variable:

```
var closure_example = function()
{
  var field = 0;
  return {
```

The getter is an unadorned getter as in Java:

```
  get: function()
  {
    return field;
  },
```

The setter could be an unadorned setter, storing `newValue` in a field and doing nothing else. However, we provide a more discriminating behavior: we make a string of the object and then see if it can be parsed as an integer. This will be an integer if we obtain an integer and NaN (Not a Number) if it cannot be parsed as an integer. If the value is not a number, then we return false; if it gives us an integer, then we store the integer and return true:

```
    set: function(newValue)
    {
      var value = parseInt(newValue.toString());
      if (isNaN(value))
      {
        return false;
      }
      else
      {
        field = value;
        return true;
      }
    }
  }
} ();
```

This creates and evaluates an anonymous function. Local variables (in this case, `field`) do not enter the global namespace but also remain around, lurking, accessible to the object returned and its two members: functions that can still access `field`. As long as the object stored in `closure_example` is available, its members will have access to its local variables, which are not gone and garbage collected until the object in `closure_example` itself is gone and eligible for garbage collection.

We can use it as follows:

```
closure_example.set(3);  
var retrieved = closure_example.get();  
closure_example.set(1.2);  
var assignment_was_successful = closure_example.set(1.2)
```

Prototypes and prototypal inheritance

Prototypes and **prototypal inheritance** are a basis for object-oriented programming, with inheritance, but without classes. In Java, certain features of a class are fixed. In Python, an object inherits from a class but features that are fixed in Java cannot be overridden. In JavaScript, we go one step further and say that objects inherit from other objects. Class-based objects run in a Platonic fashion, where there is an ideal type and concrete shadows or copies of that ideal type. Prototypal inheritance is more like the evolutionary picture of single-celled organisms that can mutate, reproduce asexually by cell division, and pass on (accumulated) mutations when they divide. In JavaScript, an object's prototype is set by, for instance:

```
customer.prototype = employee;
```

Redefine members for a customer when one wants to change from the attributes of an employee. (Members that are not redefined default to the prototype's values, and if not found on the prototype, default to members on the prototype's prototype, going all the way up the chain to the object if need be.) Also note that this inheritance may or may not mean moving from more general to more specific; inheritance may better be seen as "*customer mutates from employee*" than "*customer is a more specific type of employee*."

Let us return to the parameters of `$.ajax()`.

data

This is the form data to pass, whether given as a string as it would appear in a GET URI, as follows:

```
query=pizza&page=2
```

or given as a dictionary, as follows:

```
{page: 2, query: "pizza"}
```


dataFilter

This is a reference to a function that will be passed two arguments: the raw response given by an `XMLHttpRequest`, and a type (`xml`, `json`, `script`, or `html`). This offers an important security hook: JSON can be passed to an `eval()`, but malicious JavaScript that is passed in place of JSON data can also be passed to an `eval()`.

One of the cardinal rules of security on both the client-side and server-side is to treat all input as guilty until proven innocent of being malicious. Even though it is a "double work" chore, user input should both be validated at the client-side and the server-side: client-side as a courtesy to the user to improve the user experience and *not* as trustworthy security, and on the server side as a security measure against malicious data even if it is malformed malicious data that no normal web browser would send.

We might comment that the Django principle of **Don't Repeat Yourself (DRY)**, is a major Django selling point but is *not* a reason to dodge handling both the user interface side of validation, and the security side. In practice, this means both validating from Django on the server-side and JavaScript on the client-side, for what we are doing. Needlessly repeating yourself in Django is a sign of bad code, but this is not *needless* repetition, even if it is a chore. It's *needed* repetition.

jQuery does not automatically include functionality to test whether something served up as JSON is malicious, and methods like `.getJSON()` trustingly execute what might contain malicious JavaScript. One serious alternative is the `JSON.parse()` method defined in <http://www.json.org/json2.js>. It will return a parsed object or throw a `SyntaxError` if it finds something suspicious.

dataType

This is something you should specify, and provide a default specified value via `$.ajaxSetup()`, for example:

```
$.ajaxSetup({dataType: "text"});
```

The possible values are `html`, `json`, `jsonp`, `script`, `text`, and `xml`. If you do *not* specify a value, jQuery will use an unsecure "intelligent guessing" that may trustingly execute any JavaScript or JSON it is passed without any attempt to determine if it is malicious. If you specify a `dataType` of `text`, your callback will be given the raw text which you can then test call `JSON.parse()` on. If you specify a `dataType` of `json`, you will get the parsed JSON object. So specify `text` even if you know you want JSON.

error(XMLHttpRequest, textStatus, errorThrown)

An error callback that takes up to three arguments. For example:

```
$.ajax({error: function(XMLHttpRequest, textStatus, errorThrown)
    {
        registerError(textStatus);
    }, ...
});
```

success(data, textStatus, XMLHttpRequest)

A callback for success that also takes up to three arguments, but in different order. For example:

```
$.ajax({success: function(data, textStatus, XMLHttpRequest) {
    processData(data);
}, ...
});
```

If you want to do something immediately after the request has completed, the recommended best practice is to specify a callback function that will pick up after the request has completed. If you want to make certain variables available to the callback function that would not otherwise remain available, you may specify the context as discussed above.

type

The type of the request, for example GET, POST, and so on. The default is GET, but this is only appropriate in very limited cases and it may make sense to simply always use POST. The more serious the work you are doing, the more likely it is that you should *only* use POST. GET is appropriate only when it doesn't matter how many extra times a form is submitted. In other words, GET is only appropriate when there are no significant side effects, and in particular no destructive side effects. In a shopping cart application, GET may be appropriate to view the contents of a shopping cart, although POST is also appropriate and has the side effect of guaranteeing a fresh load. GET is not, however, appropriate for creating, modifying, fulfilling, or deleting an order, and you should use POST when any or all of these are involved.

url

The URL to submit to; this defaults to the current page.

\$.ajaxSetup()

This takes the same arguments as `$.ajax()`. All arguments are optional, but you can use this to specify default values. In terms of DRY, if something can be appropriately offloaded to `$.ajaxSetup()`, it probably should be offloaded to `$.ajaxSetup()`.

Sample invocation

A sample invocation is as follows:

```
$.ajaxSetup({dataType: "text", type: "POST"});
```

\$.get() and \$.post()

These are convenience methods for `$.ajax()` that allow you to specify commonly used parameters without key-value hash syntax and are intended to simplify GET and POST operations. They both have the same signature.

The sample invocation is as follows:

```
$.get("/resources/update");
$.post("/resources/update");
$.post("/resources/update", "user=jsmith&product_id=112");
$.post("/resources/update", {user: "jsmith", product_id: 112});
$.post("/resources/update", function(data) { $("#result").html(data)
});
$.get("/resources/update", "user=jsmith&product_id=112",
function(data, textStatus, XMLHttpRequest) {
    $("#result").html(data);
    logStatus(textStatus);
});
$.post("/resources/update", "user=jsmith&product_id=112",
function(data, textStatus, XMLHttpRequest) {
    $("#result").html(data);
    logStatus(textStatus);
}, "text");
```

These are convenience methods for `$.ajax()` and make several common features from `$.ajax()` available, but notably not an error callback function. If you want a callback called in the case of an error for appropriate handling, as well as when everything goes perfectly well, register a global error handler or use `$.ajax()`.



For the real world, this is a substantial endorsement of not using the convenience methods alone, but either specifying a global error handler or using `$.ajax()`.

Something can go wrong in front of your boss, or worse, work perfectly when you show your boss and then proceed to blow up completely when your boss shows it to your customer. Heisenbugs, (subtle and hard-to-pin-down bugs that just show up under circumstances that are difficult to repeat), network errors, and server errors will occur, and unless a noop response is appropriate and production-ready behavior when an error prevents successful completion, you will want to specify an appropriate error callback.

Or, alternatively, it may be acceptable to specify a global error handler using `$.ajaxSetup({error: myErrorHandler})` if you can write something appropriate to generically but correctly handle all Ajax error conditions where you did not directly call `$.ajax()` and specify an error handler. This includes all calls to `$.get()`, `$.load()`, and `$.post()`.

.load()

This is a very convenient convenience method. If you're looking for a simple, higher-level alternative to `$.ajax()`, you can call this method on a wrapped set to load the results of an Ajax call into it. There are some caveats, however. Let's give some sample invocations, look at why `.load()` is attractive, and then give appropriate qualifications and warnings.

Sample invocations

The sample invocation is as follows:

```
$("#messages").load("/sitewide-messages");
$("#messages").load("/user-messages", "username=jsmith");
$("#hidden").load("/user-customizations", "username=jsmith",
function(responseText, textStatus, XMLHttpRequest) {
    performUserCustomizations(responseText);
});
```

On the surface, this looks like a good example of an alternative to doing JavaScript work that incorporates jQuery, and instead using the "virtual higher-level language" that jQuery provides. `.load()`, like many jQuery functions, is a function of a wrapped set and returns a wrapped set (as often, the wrapped set it was given). It is, therefore, *in principle* something that can be put in a chain.

Why "in principle"? It makes sense, upon some condition, to hide all of the paragraphs in a form containing a checkbox that is not checked:

```
$( "form p:has(input[type=checkbox]:not(:checked)) " ).hide("slow");
```

This says, "For all p elements in a form that have an unchecked checkbox, slowly hide them."

Furthermore, one could want to do several actions instead of one:

```
$("form p:has(input[type=checkbox]:not(:checked))").  
  addClass("strikethru").delay(500).hide("slow");
```

That is a more elaborate animation: "For all `p` elements in a form that have an unchecked checkbox, add the class `strikethru` (which can be defined in CSS to have a strikethrough line through text), then wait half a second (500 milliseconds), and then slowly hide it as was done before."

But for a wrapped set like this, it would not make much sense to insert a `.load()` after the wrapped set is generated:

```
$("form p:has(input[type=checkbox]:not(:checked))").load("/updates");
```

What that says is, "For all `p` elements in a form that have an unchecked box, load the contents of the relative URL `/updates` and replace whatever the selected `p` elements contain with what was loaded." That's *legal*, but it's not as clear *why* someone would want to do this. Ordinarily, if you are going to load something from Ajax to add to the web page, it will make sense to insert it in one place and not every element in a multi-element wrapped set. So the possibility of calling `.load()` on any wrapped set, instead of a single DOM element as can be encapsulated in a wrapped set like `$("#results")`, is not obviously such a terribly great improvement.

Furthermore, `.load()` will return immediately, not when things are loaded, so items following it in the chain should not be assumed to have the data loaded. But they also should not be assumed *not* to have the data loaded; we have a race condition, and we should only chain other items after `.load()` when race conditions about the order of execution do not matter. So the fact that we can chain other operations after `.load()`, which is arguably the glory of jQuery, does not mean that it is always wise to do so.

Another point to be made is as above: `.load()`, like `$.get()` and `$.post()`, effectively forces a noop error handler, and therefore should be used only when it is acceptable for nothing to be done when any of a number of things that can go wrong, do go wrong. There may be some cases where a noop is the best error handler, but usually best practices in Ajax are to give some feedback that something has gone wrong.

Convenience methods exist, but we recommend using `$.ajax()` because it provides callback facilities for error situations as well as for success, or writing an appropriate generic, all-purpose error callback to give to `$.ajaxSetup()` or equivalent (there are other alternatives, including `$.ajaxError()`). The best practices are to use a convenience method in conjunction with a global error handler, which can be made more sophisticated by examining the information in its arguments to tell what went wrong.

jQuery as a virtual higher-level language

There are some other basic functions that we have seen in jQuery besides direct Ajax. One example of other kinds of functions includes selectors.

The selectors

Selectors create a wrapped set. Some examples of selectors include:

- `$("*")`: Selects all DOM elements.
- `$(":animated")`: Selects all elements that are in the process of an animation at the time the selector is called.
- `$("id|=header")`: Selects all elements with attribute `id` (in this illustration), whose content matches `header` or `header-*`, like `header-image`. In this case, this would be an element ID. This and following selectors matching text are case sensitive, meaning that an ID of `"HEADER"` or even `"Header"` would not be matched.
- `$("value*=import")`: Selects all elements with attribute `value` containing the string `import`. This would include both strings like `"Then import the following class."` and `"This is important."`
- `$("value~=import")`: Matches elements having a value that contains the string `import`, but is delimited by spaces. This would include `"Then we import the following class."` but exclude `"This is important."`
- `$("id$=wrapper")`: Matches all elements having an ID that ends with `wrapper`. This would include an ID of `"comment-wrapper"` as well as `wrapper`.
- `$("http-equiv=refresh")`: Matches all elements where the `http-equiv` attribute exactly equals `refresh`.
- `$("id!=result")`: Matches all elements having IDs, where the ID does not equal `result`. This will not match elements that do not have IDs.
- `$("class^=main")`: Matches all elements having a class that begins with `main`.
- `$(":button")`: Selects all buttons, whether button elements directly or inputs of type `button`.
- `$(":checkbox")`: Selects all inputs of type `checkbox`.
- `$(":checked")`: Selects all checked inputs.
- `$(":contains('important')")`: Selects all elements containing the text `important`.
- `$(":disabled")`: Selects all inputs that are disabled.

- `$(":empty")`: Select all elements that have no children, not even a text node. This would include a node from `` or `<p></p>`, but not `<p>Hello.</p>` or the outer element of `<p></p>`.
- `$(":enabled")`: Selects all elements that are enabled.
- `$("p").eq(0)`: This gives two examples. First, all tags of a name may be found by calling their tag name; hence `$("li")` returns all `li` elements (regardless of the case in the source HTML). The `.eq()` function performs zero-based array indexing on the set. It is conceptually like how a JavaScript `$("p")[0]`. `$("p").eq(0)` returns a wrapped set containing one item: the first `p` element.
- `$(":even")`: Selects all even-numbered elements, but zero-based. Counter-intuitively, `$("table#directory tr:even")`, which says "Take the table with ID `directory` and return all even-numbered `tr` elements from it," will return the first, third, fifth, and so on, `tr` elements from the table if the table contains at least five table rows.
- `$(":file")`: Selects all inputs of type `file`.
- `$(":first-child")`: Selects all elements that are the first child of their parent. For example:

```
<html>
  <head>
    <link rel="stylesheet"
          type="text/css" href="/style.css" />
    <title>Welcome to our community!</title>
    <meta http-equiv="refresh" content="900" />
  </head>
  <body>
    <p>Welcome to our community! We offer:</p>
    <ul>
      <li>Experienced leadership.</li>
      <li>Well-furnished facilities.</li>
      <li>Good neighbors.</li>
    </ul>
  </body>
</html>
```

- `$(":first-child")` will return the `head` element, the `link` element, the `p` element, and the first `li` element. Note that this does not noisily and legalistically include every DOM text element. The text node containing "Good neighbors." is technically the first child of its `li` parent, but `$(":first-child")` returns a more useful, and conceptually cleaner to use, version of the first child.

- `. (" :gt (2) ")` will return all elements with (zero-based) index greater than two in the wrapped set. `$ ("p") . (" :gt (2) ")` will return the fourth and subsequent `p` elements, or an empty wrapped set if there are not at least four `p` elements.
- `$ (" :has (a) ")`: Returns all elements containing an anchor. `$ ("p :has (a) ")` returns all `p` elements containing an `a` element.
- `$ (" :header")`: Returns all `h1`, `h2`, `h3`, and so on, elements.
- `$ (" :hidden")`: Returns all elements that are hidden.
- `$ (" :image")`: Returns all images.
- `$ (" :input")`: Returns all buttons, inputs, selects, and text areas.
- `$ (" :last-child")`: Like `$ ("first-child")`, but selects the last instead of first element.
- `. (" :last")`: Returns the last element in a wrapped set if that set is non-empty; the Python equivalent to `$ ("p") . (" :last")` would be `paragraphs [-1]`.
- `. (" :lt")`: Like `. (" :gt")`, but selects elements less than the (zero-based) index. `$ ("p") . (" :lt (2) ")` would return the first two out of any `p` elements.
- `. (" :not (p a) ")`: `$ ("p :not (p a) ")` would return a matched set of all paragraphs that do not contain an `a` element.
- `$ (" :nth-child (3n) ")`: Would return every element that is the *one-based* third child of its parent. As well as `$ (" :nth-child (3n) ")`, `$ (" :nth-child (4n) ")`, and so on, being allowed, there is a more intuitive, one-based `$ (" :nth-child (even) ")` and `$ (" :nth-child (odd) ")`, which more predictably have odd `nth` children starting with the first child and even with the second.

The reason for this inconsistency is historical: other elements are zero-based in following JavaScript's and other languages' wide precedent, while this option is one-based in strictly following the CSS specification.

- `$ (" :odd")`: A selector that is counter-intuitively zero-based. `$ ("p :odd")` returns the second, fourth, sixth, and so on, paragraph elements if available.
- `$ (" :only-child")`: Selects all elements that are the only child of their parent.
- `$ (" :parent")`: Selects all nodes that are parents of other nodes, including text nodes. (Would return the `p` element for `<p>Hello, world!</p>`.)
- `$ (" :password")`: Selects all inputs of type password.
- `$ (" :radio")`: Selects all inputs of type radio.

- `$(":reset")`: Selects all inputs of type reset.
- `$(":selected")`: Returns all elements that are selected.
- `$(":submit")`: Returns all inputs of type submit.
- `$(":text")`: Returns all inputs of type text.
- `$(":visible")`: Returns all elements that are visible.

There are several remarks to be made here.

The first is that the selectors in these examples are (usually) given alone, but this is like words being listed alone in an old-fashioned paper dictionary. In English, there are a few things you can say with a single word, like "Stop!" but usually you say things by combining them, as we have done for a few examples. These selectors are powerful by themselves, but they are more powerful when viewed as words that make up sentences like `$("div.product ul:nth-child(odd)")`, which returns the first, third, fifth, and so on, `li` elements in each unordered list contained in a `div` of class `product`, such as one would use for tiger-stripping unordered lists. Many of these selectors are useful by themselves, but they are intended, like pipable Unix command-line tools, to work well together and to be assembled like Lego bricks.

One of the first remarks one might make to someone learning Perl is, "You are not really thinking Perl until you are thinking dictionaries/hashes/associative arrays." If you are solving problems like they do in a C class, from the basic data types such as `int`, `long[]`, `char**`, and `void*`, then you are missing one of the most important workhorses Perl has to offer (and Python, for that matter). And in similar fashion, *you are not really thinking jQuery until you are thinking wrapped sets as created by selectors like the examples above*. It's a foundational part of idiomatic use of the "virtual higher-level language" jQuery offers.

In some other libraries, and in unadorned JavaScript, you operate on the DOM one element at a time. If you want to operate on several elements, you still operate on them one at a time, but you do this several times in sequence. However, the basic unit of work in jQuery is the wrapped set; it may be a wrapped set of one, such as one may create by calling `$("#main:first-child")` or `$("div").eq(0)`, or one may end up getting it by calling `$("h2")` when the DOM contains exactly one `h2` header. However, even then it is missing something about jQuery to think of the set as simply a wrapper for an isolated, unitary element.

The list of the selectors above is something like a paper dictionary of nouns. We haven't yet discussed the verbs, or how to put them together in speech. Let's explore one example of including jQuery in simple Django Ajax. This example does not demonstrate jQuery's power and elegance yet. That is part of the goal in subsequent chapters, as they show Django Ajax put together using jQuery.

A closure-based example to measure clock skew

Let's put some things together in making a simple Ajax example. We will load a web page that will measure how long it takes to make a request from the server, and given a request that gives the time on the server, estimate clock skew between the server and the client. The server will give its answer in JSON, and we will do some DOM manipulations: we will use jQuery rather than an inline onclick-style attribute to register with the button's click event, and we will update the DOM without ever using innerHTML. And, for good measure, we will use a closure to make one incursion into the global namespace, so that if our code or some extension of it is reused, it will not overwrite other global variables.

First, let us make the template, for the sake of discussion starting from `base.html` as defined at the end of the last chapter. We will include it in the same directory as `base.html`, saved as `clockskew.html`:

```
{% extends "base.html" %}
{% block title %}Measure Clock Skew{% endblock title %}
{% block body_header_title %}Measure Clock Skew{% endblock body_
header_title %}
{% block head_css_page %}<style type="text/css">
<!--
.error
{
    color: red;
}
.success
{
    font-weight: bold;
}
// -->
</style>
{% endblock head_css_page %}
{% block footer_javascript_page %}<script language="JavaScript"
type="text/javascript" src="/static/js/json2.js"></script>
<script language="JavaScript" type="text/javascript"
src="/static/js/clock_skew.js"></script>
{% endblock footer_javascript_page %}
{% block body_content %}<ul id="results"></ul>
<button id="button">Measure Clock Skew</button>
{% endblock body_content %}
```

Now this page could stand to be refactored on a couple of grounds. Firstly, it includes as page-specific `/static/js/json2.js`, which as a library for safer parsing of JavaScript presented as JSON should probably be included sitewide, or to go one step further, it should be concatenated with `/static/js/jquery.js` at least for sitewide deployment, and any other sitewide includes, so that only one JavaScript HTTP request slows things down. (As far as HTTP requests go, a 2 KB download plus another 2 KB download, especially if they are 2 KB JavaScript downloads, add up to more slowness in page rendering than one 4 KB download.) Secondly, one of the major principles of Django is DRY, and the fact that this page repeats "Measure Clock Skew" three times is an invitation for refactoring. This example and what follows deliberately have room left for refinements. (*Can you spot any further improvements to make?*)

We will also define a couple of models, one to serve up this template and one to serve JSON, adapt and extend the `urls.py` file, and write the Ajax to add behavior to the page. Here is the JavaScript `clock_skew.js` file:

```
var MeasureClockSkew = function()
{
    var that = this;
    var lastButtonPress = new Date().getTime();
    var registerError = function(XMLHttpRequest, textStatus,
        errorThrown)
    {
        $( "#results" ).append("<li class='error'>Error: " +
            textStatus + "</li>");
        $( "#button" ).removeAttr("disabled");
    };
    var registerSuccess = function(data, textStatus, XMLHttpRequest)
    {
        try
        {
            var remote = JSON.parse(data).time;
            var halfway = (lastButtonPress +
                new Date().getTime()) / 2;
            var skew = (remote - halfway) / 1000;
            $( "#results" ).append("<li class='success'>Estimated clock
                skew: <span class='measurement'>" + skew + "</span>
                seconds.</li>");
        }
        catch(error)
        {
            $( "#results" ).append("<li class='error'>Error parsing
                JSON.</li>");
        }
    }
}
```

```
        $("#button").removeAttr("disabled");
    };
    var buttonPress = function()
    {
        lastButtonPress = new Date().getTime();
        $("#button").attr("disabled", "disabled");
        $.ajax({data: "", dataType: "text", error: registerError,
            success: registerSuccess, type: "POST",
            url: "/time/json"});
    };
    return {
        buttonPress: buttonPress
    }
} ( );
$("#button").click(MeasureClockSkew.buttonPress);
```

Before going further, we would like to make a few comments about double submission and race conditions. If our test script is deployed live on a faraway web server, with netlag we could fairly easily click the button twice before the response came back, and then it would calculate invalid data. The object, as a means of accounting for netlag, estimates that the server gave its timestamp halfway between when we submitted the click and when we received it, and if we clicked a button twice before the response came back, in addition to any inaccuracies in this estimation, it would combine the time the second click was made and when the first click's response came back, making the calculation corrupt. There are other ways this could have been dealt with; we could make a closure within a closure that would create a separate object for each click and allow overlapping trials with each end time matched to the corresponding start time. But here we have followed a much more generally applicable pattern from the e-commerce world, which is to disable the submit button so the user should not be able to generate overlapping clicks. This is another area where doing things right, even if it means doing double work, means that on the client side you prevent a second submission when that would not be in your visitor's best interests (you don't want your customers charged twice because they got impatient and pressed the button again), and on the server side you also take actions to prevent undesirable forms of double submission (remember, not all visitors have JavaScript enabled).

Furthermore, there is one other best practice worth mentioning: `this` is available only when an object is being constructed; but we can save a reference as `that`.

A Django model is a class that corresponds to a table in a database in Django's object-relational mapping. A Django model instance corresponds to a table row. The division of labor, or separation of concerns, is not exactly MVC, or model-view-controller, but MTV, model-template-view, where the model is a class of object that corresponds to a table in the database, a template is a designer-editable component that gets most HTML out of Python code, and a view is what renders a template or otherwise generates a loaded page. We will create two Django view methods to serve things up on the server side. One is like what we have seen before, and another is new but in principle self-explanatory. A view that serves up JSON is as follows:

```
#!/usr/bin/python/

import json
import time
from django.core import serializers
from django.http import HttpResponse
from django.shortcuts import render_to_response

def home(request):
    return render_to_response(u'clock_skew.html')

def timestamp(request):
    return HttpResponse(json.dumps({'u'time': 1000 * time.time()}),
        mimetype=u'application/json')
```

We save both views in `clock_skew.py`, and then edit `urls.py`, so that after

```
(r'^$', 'sample.views.home'),
```

we also have:

```
(r'^time$', 'sample.clock_skew.home'),
(r'^time/json$', 'sample.clock_skew.timestamp'),
```

This is a brief nutshell example of many, but not all, of the kinds of features we will use in our more in-depth case study. The astute reader may have noticed that this brief microcosm does not have the server storing information and saving the state for the user later. However, this does provide Ajax, jQuery, JSON, and some of the most foundational features that Django offers. As we move on we will take these features, incorporate Django models and database usage, and move into a more complex and more sophisticated usage of the features that are presented in this brief example.

Case study: A more in-depth application

In the following chapters, we will build a Django Ajax web application and use jQuery. The web application will be meant to work as a company's intranet employee photo directory, and we hope to put you in a position both to use our model application and customize it to your company's specific needs. In this test application, we will demonstrate both basic features and best practices in putting together a web application using our core technologies. The chapters in our case study will include the following sections.

Chapter 3: Validating Form Input on the Server Side

In this chapter, we will send an Ajax request to the server via jQuery, and validate it on the server side based on the principle that all input is guilty until proven innocent of being malicious, malformed, incomplete, or otherwise invalid. We will look at standard server validation approaches in light of usability practices and look for improvement.

Chapter 4: Server-side Database Search with Ajax

We will compare the merits of server-side and client-side searching, and look at the limitations of JavaScript, both in terms of language limitations, and in terms of client-side performance issues associated with doing too much in the client. We will look both at the merits of handling searching and other backend functions with the full power of a backend environment, and explore why, on the client side, we should work hard to be as lazy as possible in doing network-related work.

We might clarify that "lazy" here does not *specifically* refer to the programmer's virtue of a proactive laziness that tries to solve a problem once, correctly, rather than dash off a bad solution and then spend a lot of time cleaning up after a suboptimal solution. That is worth encouraging, but it is not what we are talking about here. "Lazy" refers, for instance, to Python's `xrange()`, a generator which yields integers one at a time as they are requested, rather than Python's `range()`, that builds a complete array immediately and takes significantly more memory for large ranges.

In our case, "lazy" refers in particular to not having the client try to anticipate user needs by fetching what might or might not be needed beforehand, but requesting the minimum necessary to meet user requests, only when requested. We will be exploring several approaches; "lazy" is one approach that we should definitely know about and be able to use.

Chapter 5: Signing Up and Logging into a Website Using Ajax

This chapter will introduce Django authentication facilities and account management, and explore how we can attractively handle the client side of authentication through Ajax client-side communication with the server and corresponding Ajax client-side updates.

Chapter 6: jQuery In-place Editing Using Ajax

In this chapter, we will show a way to use jQuery to make an in-place replacement of a table that allows in-place editing, which communicates with the server in the background, adding persistence to changes.

Chapter 7: Using jQuery UI Autocomplete in Django Templates

We will discuss jQuery's basic intention as having a useful core that's designed to invite plugins to the point of it not being uncommon for programmers to write plugins their first day doing jQuery. We will use autocomplete from the jQuery UI. We will then integrate this into Django templates and our project's user interface.

Chapter 8: Django ModelForm: a CSS Makeover

Django comes with a straightforward way to easily build forms from Django models. We will explore this feature and how to use it.

Chapter 9: Database and Search Handling

In this chapter, we will be showing "lazy" best practices in developing our application.

Chapter 10: Tinkering Around: Bugfixes, Friendlier Password Input, and a Directory That Tells Local Time

If you are interested in having an employee photo directory for your intranet, we not only want to provide an application but also help you have a starting point to create a customized application around your company's needs.

Chapter 11: Usability for Hackers

If you are reading this book, you may have some surprising strengths for usability. This chapter explores them. With this chapter we take a step back from our application and take a look at usability and the bedrock competencies hackers can leverage to do usability.

Appendix: Debugging Hard JavaScript Bugs

In this appendix, we take a look at the state of mind that is needed to debug difficult bugs.

Summary

The goal of these first two chapters has been to provide both a big picture and a sense for how things fit together. Ajax is a bit interdisciplinary; it involves server-side technologies (Django for us), client-side scripting and libraries (including jQuery for us), CSS, HTML, and the DOM, and the goal is very human in character. Ajax is interesting because it opens doors in user interface, usability, and user experience, and in that regard doing well with Ajax isn't just technical; it's also a bit like the arts and humanities.

7

Using jQuery UI Autocomplete in Django Templates

In this chapter, we will cover ground in two different dimensions. First of all, we will use jQuery UI's autocomplete, with Django templates and views, covering both the server-side and client-side aspects and explore Django Ajax a little more deeply. Second, we will use something more like a real-world process of discovery while we are doing this. That is to say, instead of starting immediately with a finished solution, we will show what it is like to meet roadblocks along the way and still deliver a working project.

In this chapter we will cover:

- jQuery UI's autocomplete and themeroller
- A "progressive enhancement" combobox
- What needs to be done on the server-side and client-side to do the following:
 - Using Django templating to dynamically create elements
 - Client-side event handling to send autocomplete-based selections to the server
 - DOM Level 0 and iframe-based alternatives on the client-side
 - Extending server-side Django Ajax views to handle updates from the client
 - Refining the working solution
- An example of practical problem solving when issues arise

In previous chapters, we have explored how to get a solution working under optimal conditions. Here we'll look at what we can do when tools don't always work.

Adding autocomplete: first attempt

For further development, we will be using a jQuery theme. What specific theme can be used is customizable, but autocomplete and other plugins require *some* theme such as jQuery UI Themeroller provides. jQuery UI Themeroller, which lets you customize and tweak a theme (or just download a default), is available at:

<http://jqueryui.com/themeroller/>

When you have made any customizations and downloaded a theme, you can unpack it under your static content directory. In our `base.html` template, after our site-specific stylesheet, we have added an include to the jQuery custom stylesheet (note that you may download a different version number than we have used here).

```
{% block head_css_site %}<link rel="stylesheet" type="text/css"
href="/static/css/style.css" />
<link rel="stylesheet" type="text/css"
href="/static/css/smoothness/jquery-ui-1.8.2.custom.css" />
{% endblock head_css_site %}
```

We will be using the jQuery UI combobox, which offers a "progressive enhancement" strategy by building a page that will still work with JavaScript off and will be more accessible than building a solution with nothing but Ajax.

Progressive enhancement, a best practice

"Progressive enhancement," in a nutshell, means that as much as possible you build a system that works without JavaScript or CSS, with semantic markup and similar practices, then add appearance with CSS, and then customize behavior with JavaScript. A textbook example of customizing behavior is to make a sortable table which is originally sortable, in Web 1.0 fashion, by clicking on a link in a table header which will load a version of the table sorted by that link. Then the links are "hijaxed" by using JavaScript to sort the table purely by JavaScript manipulations of the page, so that if a user does not have JavaScript on, clicking on the links loads a fresh page with the table sorted by that column, and if the user does have JavaScript, the same end result is achieved without waiting on a network hit. In this case, we mark up and populate a dropdown menu of available entities, which the JavaScript will hide and replace with an autocomplete box. The `option` tags follow a naming convention of `fieldname.id`; all the autocomplete fields are for fields of an entity, and the naming convention is not directly for the benefit of server-side code, but so that an event listener knows which field it has been given a value for.

Here we follow the same basic formula for department, location, and reports_to. We produce a list of all available options. The first entry is for no selected department/location/reports_to; as a courtesy to the user we add selected="selected" to the presently selected value so that the form is smart enough to remember the last selected value, rather than defaulting to a (presumably unwanted) choice each time the user visits it.

This much of the code is run once and creates the beginning of the containing paragraph, sets a strong tag, and creates the entry for no department selected (and selects it if appropriate):

```
<p>Department:
  <strong>
    <select name="department" id="department"
      class="autocomplete">
      <option
        {% if not entity.department.id %}
          selected="selected"
        {% endif %}
        value="department.-1">&mdash; Select &mdash;</option>
```

Then we loop through the list of departments, creating an option that has a value of "department." followed by the id of the entity provided as a department. Remember that earlier we simply used a list of all entities for departments. If you are interested in tinkering, you could add a checkbox to indicate whether an entity should be considered a department or a reports_to candidate. (Locations are a different data type, so no paring should be obviously helpful for them.)

```
      {% for department in departments %}
      <option
        {% if department.id == entity.department.id %}
          selected="selected"
        {% endif %}
        value="department.{{ department.id }}"
          {{ department.name}}
      </option>
    {% endfor %}
```

We then close the select and strong, and move on to the next line.

```
      </select>
    </strong>
  </p>
```

The location and reports_to are handled similarly:

```
Location:
    <select name="location" id="location" class="autocomplete">
```

We add the default, unselected option:

```
    <option
    {% if not entity.location.id %}
        selected="selected"
    {% endif %}
    value="location.-1">&mdash; Select &mdash;</option>
```

Then we loop through available locations and build their options.

```
    {% for location in locations %}
        <option
        {% if location.id == entity.location.id %}
            selected="selected"
        {% endif %}
        value="location.{{ location.id }}"
            {{ location.identifier }}</option>
    {% endfor %}
</select>
</p>
```

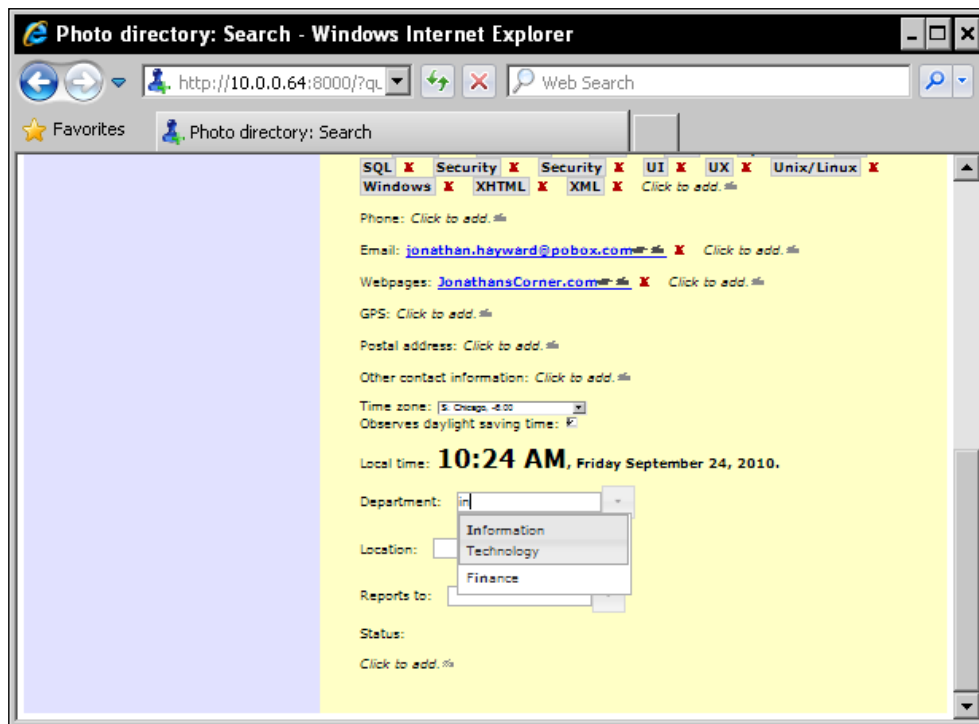
And likewise in Reports to:

```
<p>
Reports to:
    <select name="reports_to" id="reports_to" class="autocomplete">
        <option
        {% if not entity.reports_to.id %}
            selected="selected"
        {% endif %}
        value="reports_to.-1">&mdash; Select &mdash;</option>
        {% for reports_to in reports_to_candidates %}
            <option
            {% if reports_to.id == entity.reports_to.id %}
                selected="selected"
            {% endif %}
            value="reports_to_{{ reports_to.id }}"
                {{ reports_to.name }}</option>
        {% endfor %}
    </select>
</p>
```

Our profile page is modified to provide more variables to the template. Later on, we might identify which entities can be an entity's departments and which entities can be reported to, thus improving the available options by paring away irrelevant entities, but for now we simply provide all entities.

```
@login_required
def profile(request, id):
    entity = directory.models.Entity.objects.get(pk = id)
    emails = directory.models.EntityEmail.objects.filter(
        entity__exact = id).all()
    all_entities = directory.models.Entity.objects.all()
    all_locations = directory.models.Location.objects.all()
    return HttpResponse(get_template(u'profile.html').render(Context(
        {
            u'entity': entity,
            u'emails': emails,
            u'departments': all_entities,
            u'reports_to_candidates': all_entities,
            u'locations': all_locations,
        })))
```

This provides nice-looking autocomplete functionality like:



However, there's a bit of a quirk. When we use the code as implemented earlier, it does not seem to save our changes; they are not persistent across page views.

A real-world workaround

In the course of writing this book, a difficulty was encountered. We will take advantage of that opportunity to look at finding alternatives when things do not work. For those interested in jQuery UI complete specifically, the StackOverflow question at <http://stackoverflow.com/questions/188442/whats-a-good-ajax-autocomplete-plugin-for-jquery> is recommended, and a response with nine upvotes as of this writing said autocomplete doesn't work as of roughly jQuery UI 1.6. <http://pastie.org/362706> is reported as a working version. However, let's look at a workaround when things don't work—because even a recommended plugin for a good library may not work, or may not work for us. However, we will succeed later on working with the provided plugin.

The difficulty in question is as follows. jQuery UI includes autocomplete functionality. As downloaded in our case, the autocomplete functionality appeared to work from the user interface perspective, but did not appear to be saving data on the server. A callback function had been created and (attemptedly) registered, but it was not being called. Inserting `alert()` in the beginning of the function did not trigger any alert boxes, and no error appeared in the browser console. Consulting forums at <http://forum.jquery.com/> and <http://stackoverflow.com/>, which are excellent resources, did not seem to turn up results. A bug was filed at <http://dev.jqueryui.com/>, but that did not generate any response quickly. Supposing that we have deadlines looming, what should we do in a case like this?

"Interest-based negotiation": a power tool for problem solving when plan A doesn't work

At this point we might discuss a tangent from *interest-based negotiation*.

Fisher and Ury's negotiation classic *Getting to Yes* discusses two basic kinds of negotiation: hard and soft. Hard negotiation tries to bend as little as possible from its stated position; soft negotiation, such as often occurs in informal and friendly settings, is much more flexible. But they both suffer from an Achilles heel: when both sides start by defining a position, and the only question is who is going to give how much from their initial positions, the result is almost guaranteed to be suboptimal. If you play that game in the first place, you lose because you are playing the wrong game.

The alternative, interest-based negotiation, which involves finding what interests exist on both sides and doing creative problem solving based on those interests, is much more likely to produce a winner for all involved. *Getting to Yes* discusses interest-based negotiation as a power tool for hostile negotiations where the other side has the upper hand, and perhaps it may be. But some of the best mileage you can get out of interest-based negotiations is in friendly negotiations.

One such situation that keeps coming up at work is when someone has basically figured out what they consider a solution to a problem, and then asks for you to attend to the implementation. For example, a manager might have come to a system administrator in the days when pagers were the hot new thing and said, "Our disk filled up last night! In order to prevent this from happening again, is there any way you can set up an automatic process to send the output of a `df` to a pager every five minutes?" And almost the worst response in that situation is, "Yes, I'll get right on it." In cases like these, if the solution was envisioned, architected, and designed by someone nontechnical, and the technical person is asked only to handle implementation details, the solution is almost guaranteed to be wrong.

The correct solution is not to negotiate on a level of positions, but of interests. This particular solution uses a program whose output is designed for a full terminal window, which would be quite painful to scroll through on an early pager even once. It is a "boy who cried wolf" solution that means to the system administrator, "Here's some spam you have to scroll through every five minutes to find out if there is interesting data." Not, necessarily, that it is wrong to send something to a pager. It might well be an appropriate solution to periodically check and send a brief message to the system administrators if the disk is fuller than some threshold percentage, or if the disk is being filled up beyond some threshold percentage per unit of time. But in many cases the correct response is to politely receive the stated position and then get on to identifying the interests involved and trying as best you can to craft a position.

In this case, we are applying the principle of interest-based negotiation to negotiation with the computer. Our initial position, "Follow the jQuery UI instructions" has not produced the desired results, at least not yet. So the next thing we can do is identify our interests. Our interest here, without degrading anything else, is to provide autocomplete functionality to the user interface. This allows at least four potential ways to get past the obstacle:

- Resolve the problem and complete the intended solution
- Work around the bug using the same framework
- Find another jQuery plugin to handle autocomplete
- Use a standalone solution, or another library

We don't need to stay stuck; in fact, we have several options. The first option cannot be ruled out, but right now we have not succeeded. Let's say that's not an option in our case. The third and fourth possibilities almost certainly have multiple options, and multiple live options, but in this case we can bend the rules a little bit, comment out our event-handling code, and give a little nudge to let our jQuery UI-based solution work with all of the niceties of using jQuery UI. We will go with this workaround, but we are not just a workaround away from being blocked by a brick wall. There are presumably several live options, and the more we think in terms of interests rather than positions, and identify interests to feed into problem solving, the more a brick wall fades into a cornucopia of possibilities.

A first workaround

The workaround, like many workarounds, is itself an example of interest-based negotiation with the computer. What we want to happen, that isn't happening yet, is for the data to be saved. The obvious way for us to save the data is by an XMLHttpRequest based call but it is not strictly an interest to say that we go through XMLHttpRequest or jQuery for the submission. It would also work to have a form that submitted the same data to an iframe. This is not a first choice solution, but we should be much more cautious about ruling out positions altogether than identifying our interests. Now we don't want a spurious iframe on the page, but we can set it to `display: none;`, and treat it as a bit bucket, or a Unix `/dev/null`. And the graceful degradation solution provided by jQuery UI uses a `select`, so we can specify an `onchange` for the `select` that will submit the form whenever the `select` is changed, that is whenever an autocomplete value is selected. jQuery UI allows us to display the `select`, and we will do this, both to allow a person to see available options, and to provide a choice about means of input.

Our revised code is as follows in `static/style.css`:

```
bitbucket, #bitbucket
{
    display: none;
}
```

In the top of the `body_main` block in `templates/profile.html` we add:

```
<iframe class="bitbucket" id="bitbucket" name="bitbucket"
        src="about:blank"></iframe>
```


One design consideration from our implementation is that it wraps each of the `select` tags (that populate an autocomplete) in its own form element. This means that semantically we cannot have all of them in different `br` separated lines of the same `p`, but this gives an opportunity to make the design better. If we put each field in its own paragraph, it will make a more readable use of whitespace. So we give each field its own paragraph, wrap the paragraphs with `select` tags in form elements which submit via `POST` to our Ajax-gearred URL and have a target of `bitbucket`, add a hidden form element which will receive special treatment on the server side as will be discussed later, and specify an `onchange` form submission for `select`. This workaround uses what is informally called "DOM Level 0," and is not a first choice. It does, however, allow us to keep a "close to jQuery" solution, with a workaround that is readily replaced by a more preferable solution if a bug is fixed.

The previous code is largely the same, but is wrapped in a `form` tag, and preceded by a hidden `input` designed to ensure that the view has all the information it needs.

```
<form action="/ajax/save" method="POST" name="department_form"
      id="department_form" target="bitbucket">
  <input type="hidden" name="id"
        value="Entity_department_{{ entity.id }}" />
  <p>Department:
```

The `select` has an `onchange` set to submit the form wrapping it. This kind of approach, informally referred to as DOM Level 0 scripting, is not a first choice solution; it is not exactly semantic markup. But it has been said, "In theory, theory and practice are the same. In practice, theory and practice are different." Especially if you have a deadline, given a choice between purist semantic markup that doesn't work, and a less pure solution that works, choose the solution that works. (But, of course, if you have a choice between a purist solution that works and a less pure solution that works, choose the purist solution that works, even if it involves more learning or more work up front.)

```
<select name="department" id="department" class="autocomplete"
      onchange="this.form.submit();">
  <option
    {% if not entity.department.id %}
      selected="selected"
    {% endif %}
```

We changed an option value of `—` `Select —`, such as is a common practice for `select` used for their own sake, to `None`. The reason has to do with why we are building this `select`: we wish to populate an autocomplete, and for autocomplete purposes, if someone wants to set a value to `None`, it is unlikely that they will start typing "Select".

But these are changes of relatively small details. The select is still essentially like it was before. And that is a benefit: if we are able to get things working in pure semantic markup using best practices, it helps to have a page that's still basically like a "best practices" implementation.

```
        value="department.-1">None</option>
    {% for department in departments %}
        <option
            {% if department.id == entity.department.id %}
                selected="selected"
            {% endif %}
            value="department.{{ department.id }}"
                {{ department.name }}</option>
        {% endfor %}
    </select>
</p>
</form>
```

The other fields are changed, if slightly. They are in their own paragraphs, which is semantically at least as good as having them one per line in a large paragraph, and lends itself to display with better use of whitespace—a major benefit if people will be using our directory a lot!

We define a link for the homepage:

```
<p>
    Homepage:
    {% if entity.homepage %}
        <a href="{{ entity.homepage }}">
    {% endif %}
```

Within the link we define `strong`, which is marked up for (right-click) editing, and display the URL inside the link:

```
<strong class="edit_rightclick"
    id="entity_homepage_{{ entity.id }}">{{ entity.homepage }}
</strong>
```

Then we close the link and paragraph:

```
    {% if entity.homepage %}
        </a>
    {% endif %}
</p>
```

We define similar, but not identical, handling for the e-mail:

```
<p>
  Email:
  <strong>
    {% for email in emails %}
      <a id="EntityEmail_email_{{ email.id }}"
        class="edit_rightclick"
        href="mailto:{{ email.email }}">
```

We eliminate some whitespace so that there will not be a space between the link and the separating comma:

```
      {{ email.email }}</a>{% if not forloop.last %},
    {% endif %}
  {% endfor %}
  <span class="edit" id="EntityEmail_new_{{ entity.id }}">
    Click to add email.
  </span>
</strong>
</p>
```

The location fields, as the `reports_to` field in the following snippet, follow the same pattern as the department previously seen.

We define a form:

```
<form action="/ajax/save" method="POST" name="location_form"
  id="location_form" target="bitbucket">
```

We define a hidden input so the server-side code has a parsable identifier:

```
<input type="hidden" name="id"
  value="Entity_location_{{ entity.id }}" />
<p>
  Location:
```

We define a select with a DOM Level 0 submit:

```
<select name="location" id="location" class="autocomplete"
  onchange="this.form.submit();">
```

We build the first option for when nothing has been selected:

```
<option
  {% if not entity.location.id %}
    selected="selected"
  {% endif %}
  value="location.-1">None</option>
```

We populate the rest of the list:

```
{% for location in locations %}
  <option
    {% if locationi.id == entity.locationi.id %}
      selected="selected"
    {% endif %}
    value="location.{{ location.id }}"
    {{ location.identifier }}>
{% endfor %}
```

And we close tags that need to be closed:

```
</select>
</p>
</form>
```

The Phone field is an in-place edit field. It works as a regular in-place edit field, not an autocomplete:

```
<p>
  Phone:
  <strong class="edit" id="entity_phone_{{ entity.id }}">
    {% if entity.phone %}
      {{ entity.phone }}
    {% else %}
      Click to change.
    {% endif %}
  </strong>
</p>
```

And finally, we have our third and last autocomplete, which works like the first two:

```
<form action="/ajax/save" method="POST" name="reports_to_form"
  id="reports_to_form" target="bitbucket">
  <input type="hidden" name="id"
    value="Entity_reports_to_{{ entity.id }}" />
  <p>
    Reports to:
    <select name="reports_to" id="reports_to" class="autocomplete"
      onchange="this.form.submit();">
      <option
        {% if not entity.reports_to.id %}
          selected="selected"
        {% endif %}
        value="reports_to_-1">None</option>
```

```

        {% for reports_to in reports_to_candidates %}
        <option
        {% if reports_to.id == entity.reports_to.id %}
            selected="selected"
        {% endif %}
        value="reports_to_{{ reports_to.id }}">
            {{ reports_to.name }}</option>
        {% endfor %}
    </select>
</p>
</form>
<p>
    Start date:
    <strong>
        {{ entity.start_date }}
    </strong>
</p>

```

The handler follows the signature for event handlers registered, as we wish to register them. While we do not use the event object, we keep it in the signature. `context.item` should be the option that was selected. Its value will look like `department.1` and is meant to carry all the information needed for this function. The data submitted follows the same `id-value` structure as we have already used for in-place editing in the previous chapter, and submits it to the same view. We will in fact have different server-side code handling these selections. The code that stores text in a field will not take an ID and look up the right instance of the intended field, at least not without significant refactoring. However, we are developing with an analogous interface in mind: submitted data in the `id-value` format, with the ID following the *ModelName_fieldname_instanceID* naming convention.

```

function update_autocomplete(event, context)
{
    var split_value = context.item.value.split(".");
    if (split_value.length == 2 && !isNaN(split_value[1]))
    {
        var field = split_value[0];
        var id = split_value[1];
        $.ajax({
            data:
            {
                id: "Entity_" + field + "_" + {{ entity.id }},
                value: id,
            },

```

```
        url: "/ajax/save",
      });
    };
  }
```

Boilerplate code from jQuery UI documentation

It is commonplace when using software to include adding boilerplate code. Here is an example. We insert boilerplate code from the documentation pages for jQuery UI at <http://jqueryui.com/demos/autocomplete/#combobox>:

```
(function($) {
  $.widget("ui.combobox", {
    _create: function() {
      var self = this;
      var select = this.element.hide();
      var input = $("")
        .insertAfter(select)
        .autocomplete({
          source: function(request, response) {
            var matcher = new RegExp(request.term,
                                     "i");
            response(select.children("option").map(function() {
              var text = $(this).text();
              if (this.value && (!request.term ||
                               matcher.test(text)))
                return {
                  id: this.value,
                  label: text.replace(new
            RegExp("(?!^[^&;]+;)(?!<[^>]*) (" +
            $.ui.autocomplete.escapeRegex(request.term) +
            ")(?!<[^>]*>)(?!^[^&;]+;)", "gi"), "<strong>$1</strong>"),
                  value: text
                };
            }));
          },
          delay: 0,
          change: function(event, ui) {
            if (!ui.item) {
              // remove invalid value,
              // as it didn't match anything
              $(this).val("");
              return false;
            }
          }
        });
    }
  });
});
```

```

        }
        select.val(ui.item.id);
        self._trigger("selected", event, {
            item: select.find("[value='" +
                ui.item.id + "']")
        });

    },
    minLength: 0
})
.addClass(
    "ui-widget ui-widget-content ui-corner-left");
$("<button>&nbsp;</button>")
.attr("tabIndex", -1)
.attr("title", "Show All Items")
.insertAfter(input)
.button({
    icons: {
        primary: "ui-icon-triangle-1-s"
    },
    text: false
}).removeClass("ui-corner-all")
.addClass("ui-corner-right ui-button-icon")
.click(function() {
    // close if already visible
    if (input.autocomplete("widget").is(":visible")) {
        input.autocomplete("close");
        return;
    }
    // pass empty string as value to search for,
    // displaying all results
    input.autocomplete("search", "");
    input.focus();
});
}
});
})(jQuery);

```

Turning on Ajax behavior (or trying to)

We make autocompletes out of the relevant selects, and also display the selects, which the `combobox()` call hides by default. The user now has a choice between the select and an autocomplete box.

```
$(function()
{
    $(".autocomplete").combobox();
    $(".autocomplete").toggle();
});
```

Here we have code which, from the documentation, might be expected to call the `update_autocomplete()` event handler when a selection or change is made. However, at this point we encounter a bend in the road.

The bend in the road is this: the following commented code, when uncommented, doesn't seem to be able to trigger the handler being called. When it was uncommented, and an `alert()` placed at the beginning of `update_autocomplete()`, the `alert()` was not triggered even once. And the usual suspects in terms of forums and even filing a bug did not succeed in getting the handler to be called. The `alert()` was still not called.

After this code, let's look at our updated code on the server side, and then see how this bend in the road can be addressed.

```
/*
    $(".autocomplete").autocomplete({select: update_autocomplete});
    $(".autocomplete").bind({"autocompleteselect": update_
autocomplete});
    $(".autocomplete").bind({"autocompletechange": update_
autocomplete});
*/
});
```

Now let us turn our attention to the server side.

Code on the server side

Here we have the updated `save()` view which has been expanded to address its broader scope. We accept either GET or POST requests, although requests that alter data should only be made by POST for production purposes, and save the dictionary for exploration.

```
@ajax_login_required
def save(request):
    try:
        html_id = request.POST[u'id']
```

```

        dictionary = request.POST
    except:
        html_id = request.GET[u'id']
        dictionary = request.GET

```

If we have one of the autocomplete values, we have a hidden field named `id` which guarantees that any submission will have that field in its dictionary, either `request.POST` or `request.GET`. However, the `department`, `location`, and `reports_to` fields are not all named `value`, and we manually check for them and use `value` as a default:

```

if html_id.startswith(u'Entity_department_'):
    value = dictionary[u'department']
elif html_id.startswith(u'Entity_location_'):
    value = dictionary[u'location']
elif html_id.startswith(u'Entity_reports_to_'):
    value = dictionary[u'reports_to']
else:
    value = dictionary[u'value']

```

We perform some basic validation. Our code's HTML ID should only consist of word characters:

```

if not re.match(ur'^\w+$', html_id):
    raise Exception("Invalid HTML id.")

```

Then we handle several special cases before the general-purpose code that handles most `/ajax/save` requests. If there is a new `EntityEmail`, we create it and save it:

```

match = re.match(ur'EntityEmail_new_(\d+)', html_id)
if match:
    model = int(match.group(1))
    email = directory.models.EntityEmail(email = value, entity =
        directory.models.Entity.objects.get(pk = model))
    email.save()
    directory.functions.log_message(u'EntityEmail for Entity ' +
        str(model) + u' added by: ' + request.user.username + u',
        value: ' + value + u'\n')
    return HttpResponse(
        u'<a class="edit_rightclick"
            id="EntityEmail_email_' + str(email.id)
        + u'" href="mailto:' + value + u'">' + value + u'</a>' +
        u'"<span class="edit" id="EntityEmail_new_%s">
            Click to add email.</span>' % str(email.id))

```

We have the added code to look up the entity, look up the appropriate department (if any), assign the updated department, and save the entity. This technique is repeated, with slight variation, for the location and reports_to fields. We still need to return an `HttpResponse`, even if the value is ignored. On the client-side, the following code we develop will report a Django error page for development purposes but even then will discard the reported value if the update runs without error.

Here we have the special case of an Entity's department being set. While we create the illusion on the client-side that this is just the same sort of submission as (say) an Entity's description, we need to handle a few things on the server side to give the client-side a simple appearance of "Mark it up right and save it, and it will be saved."

```
elif html_id.startswith(u'Entity_department_') :
    entity_id = int(html_id[len(u'Entity_department_'):])
    department_id = int(value[len(u'department.'):])
    entity = directory.models.Entity.objects.get(pk = entity_id)
    if department_id == -1:
        entity.department = None
    else:
        entity.department = directory.models.Entity.objects.get(
            pk = department_id)
    entity.save()
    return HttpResponse(value)
```

The location code works the same way as reports_to and department:

```
elif html_id.startswith(u'Entity_location_') :
    entity_id = int(html_id[len(u'Entity_location_'):])
    location_id = int(value[len(u'location.'):])
    if location_id == -1:
        entity.location = None
    else:
        entity.location = directory.models.Location.objects.get(
            pk == location_id)
    entity.save()
    return HttpResponse(value)
elif html_id.startswith(u'Entity_reports_to_') :
    entity_id = int(html_id[len(u'Entity_reports_to_'):])
    reports_to_id = int(value[len(u'reports_to.'):])
    entity = directory.models.Entity.objects.get(pk = entity_id)
    if reports_to_id == -1:
        entity.reports_to = None
    else:
```

```

        entity.reports_to = directory.models.Entity.objects.get(
            pk == reports_to_id)
    entity.save()
    return HttpResponse(value)

```

Although it is last, this is the mainstream and most generic handler, handling the usual cases of text fields supporting in-place editing:

```

else:
    match = re.match(ur'^(.*?)_(.*)_(\d+)$', html_id)
    model = match.group(1)
    field = match.group(2)
    id = int(match.group(3))
    selected_model = get_model(u'directory', model)
    instance = selected_model.objects.get(pk = id)
    setattr(instance, field, value)
    instance.save()
    directory.functions.log_message(model + u'.' + field +
        u'(' + str(id) + u') changed by: ' +
        request.user.username + u' to: ' + value + u'\n')
    return HttpResponse(escape(value))

```

Refining our solution further

This approach works, but there is room to further clean it up. First of all, we can set things up in the base template so that, for development, Django's informative error pages are displayed; we also set form submissions to POST by default. We make a target div for the notifications:

```

{% block body_site_announcements %}
{% endblock body_site_announcements %}
{% block body_notifications %}<div
    id="notifications"></div>
{% endblock body_notifications %}

```

Then we add, to the `footer_javascript_site` block, a slightly tweaked `send_notifications()`. Compared to our earlier code, instead of delaying five seconds, it delays five seconds plus two milliseconds per character of the message. This does not have a noticeably different effect for normal, short notifications, but it means that if a 60k Django error page is served up, you have more time to inspect the error. We could tweak it further so that above a threshold length, or on some other conditions, the notification is only dismissed by explicitly pressing a button, but we will stop here.

```

<script language="JavaScript" type="text/javascript">
function send_notification(message)
{

```

```
$("#notifications").html("<p>" + message + "</p>");
setTimeout("$.('#notifications').show('slow').delay(" + (5000 +
    message.length * 2) + ").hide('slow');", 0);
}
```

Our notifications area has several different messages, not all of which need to be visually labeled as errors, so we move from a red-based styling to one that is silver and grey in `static/css/style.css`:

```
#notifications
{
    background-color: #c0c0c0;
    border: 3px solid #808080;
    display: none;
    padding: 20px;
}
```

We call, on page load, `$.ajaxSetup()` to specify a default error handler, and also specify form submission via `POST`. This will need to be changed for deployment, but together this means that a valuable Django error page is displayed as a notification whenever an Ajax error occurs, which is a kind of "best of both worlds" solution for development.

```
$(function()
{
    $.ajaxSetup(
    {
        error: function(XMLHttpRequest, textStatus, errorThrown)
        {
            send_notification(XMLHttpRequest.responseText);
        },
        type: "POST",
    });
});
</script>
```

In the profile template, we will remove the containing form elements and the hidden inputs, and replace the contents of the `onchange` attributes. We will also rename the original `update_autocomplete()` to `update_autocomplete_handler()`, leaving it available should the originally intended approach work. The new `update_autocomplete()` will make the Ajax call, submitting the same information as the (now) `update_autocomplete_handler()`. We will remove the bit bucket `iframe`, although we leave the CSS in, in case a bit bucket is desired later on.

We are in a position technically to make one big paragraph out of several fields as we did originally, but breaking them into their own paragraphs was a fortunate change, and we will retain it.

The Department paragraph now looks like a cross between the previous two entries. It is its own paragraph, but the form is gone. There is an onchange attribute set, although its contents are different from earlier. It calls `update_autocomplete()` with an ID following the name convention and the present value of the select.

```
<p>Department:
  <select name="department" id="department" class="autocomplete"
    onchange="update_autocomplete(
      'Entity_department_{ { entity.id} }', this.value);">
    <option
      {% if not entity.department.id %}
        selected="selected"
      {% endif %}
      value="department.-1">None</option>
    {% for department in departments %}
      <option
        {% if department.id == entity.department.id %}
          selected="selected"
        {% endif %}
        value="department.{ { department.id } }">
          {{ department.name }}</option>
        {% endfor %}
      </select>
</p>
```

The location and reports_to areas follow suit:

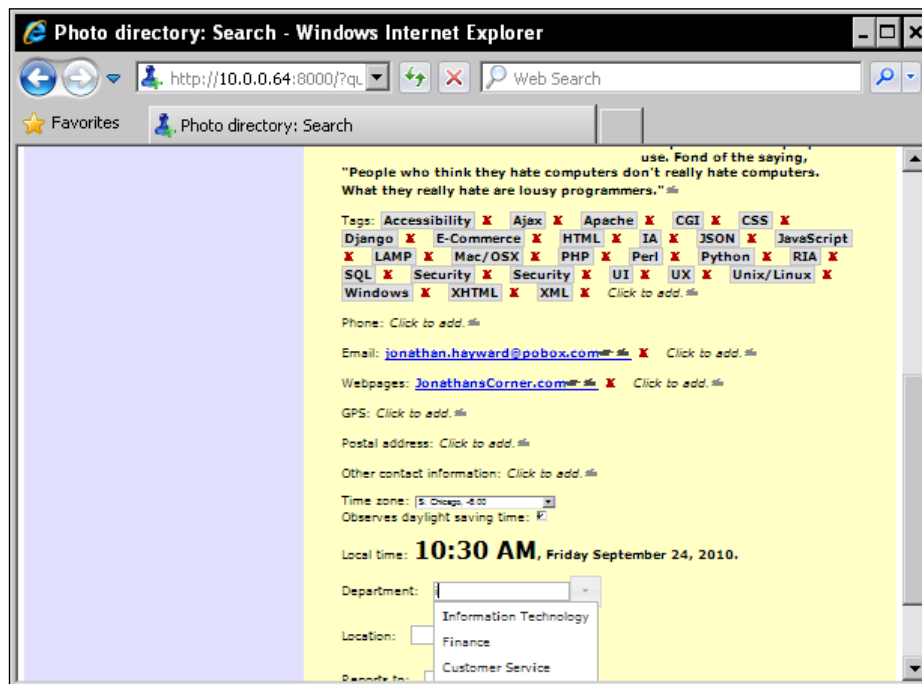
```
<p>Location:
  <select name="location" id="location" class="autocomplete"
    onchange="update_autocomplete('Entity_location_{ { entity.id} }',
      this.value);">
    <option
      {% if not entity.location.id %}
        selected="selected"
      {% endif %}
      value="location.-1">None</option>
    {% for location in locations %}
      <option
        {% if location.id == entity.location.id %}
          selected="selected"
        {% endif %}
        value="location.{ { location.id } }">
          {{ location.identifier }}</option>
      {% endfor %}
    </select>
</p>
```

The Reports to field also follows suit:

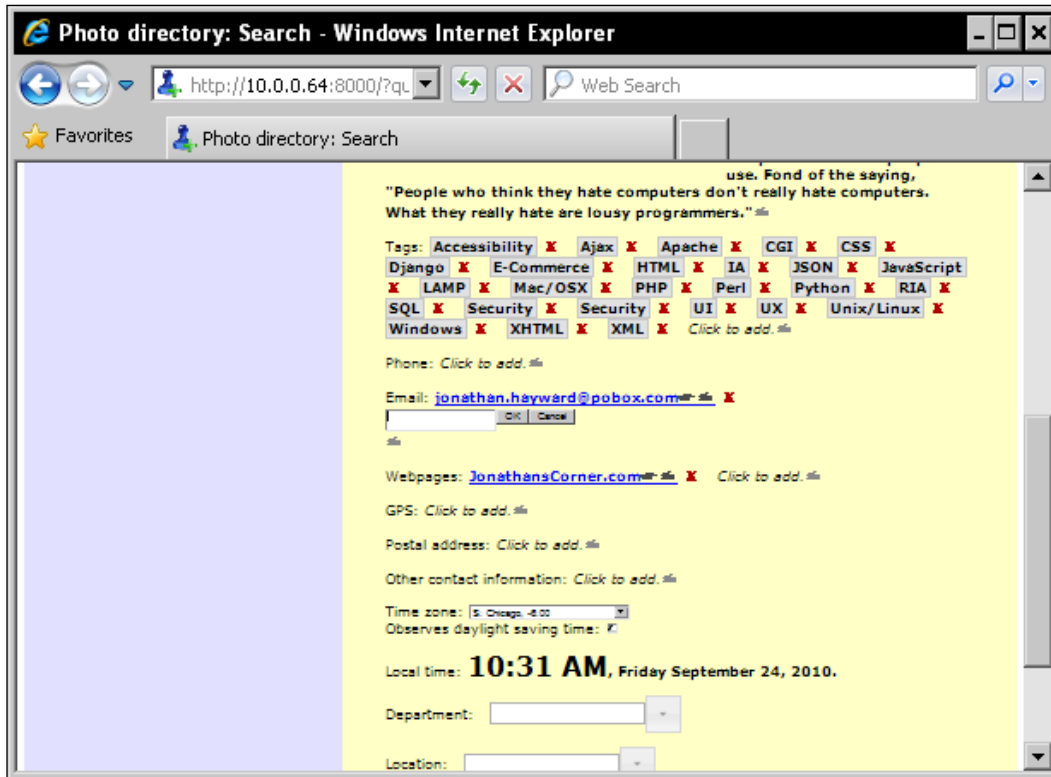
```
<p>Reports to:
  <select name="reports_to" id="reports_to" class="autocomplete"
    onchange="update_autocomplete(
      'Entity_reports_to_{{ entity.id }}', this.value);">
  <option
    {% if not entity.reports_to.id %}
      selected="selected"
    {% endif %}
    value="reports_to_-1">None</option>
    {% for reports_to in reports_to_candidates %}
      <option
        {% if reports_to.id == entity.reports_to.id %}
          selected="selected"
        {% endif %}
        value="reports_to_{{ reports_to.id }}">
        {{ reports_to.name }}</option>
    {% endfor %}
  </select>
</p>
```

And that's it. We now have a working, and slightly more polished internally, implementation that supports autocomplete and in-place editing like the following.

For the autocomplete:



Or, for another of several examples of user input that was allowed, here is an in-place edit used to add a new email address.



Summary

Again, we have moved in two different dimensions in this chapter. The first dimension is the obvious one: what we need on the client-side and server-side to get autocomplete working with jQuery UI. The second dimension has to do with creative problem solving when something goes wrong.

We have covered the nuts and bolts of jQuery UI's autocomplete, and where plugins can be obtained. We continue with the concept of "progressive enhancement," and a concrete example. We looked at what tools we have on the server-side and client-side to do this. We have continued to get our hands dirty with Django templating to build the desired pages.