MODULE-3

CHAPTER 6

The Django Administration 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 addition, editing, and deletion of site content. The interface you use to post to your blog, the back-end site managers use to moderate reader-generated comments, the tool your clients use to update the press releases on the Web site you built for them—these are all examples of admin interfaces.

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. This feature works by reading metadata in your model to provide a powerful and production-ready interface that site administrators can start using immediately. Here, we discuss how to activate, use, and customize this feature.

Activating the Admin Interface

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

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

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

    class Admin: pass
```

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

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

2. Install the admin application. Do this by adding `django.contrib.admin` to your `INSTALLED_APPS` setting and running `python manage.py syncdb`. This second step will install the extra database tables the admin interface uses.

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

---

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

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

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

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

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

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

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

Using the Admin Interface

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

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

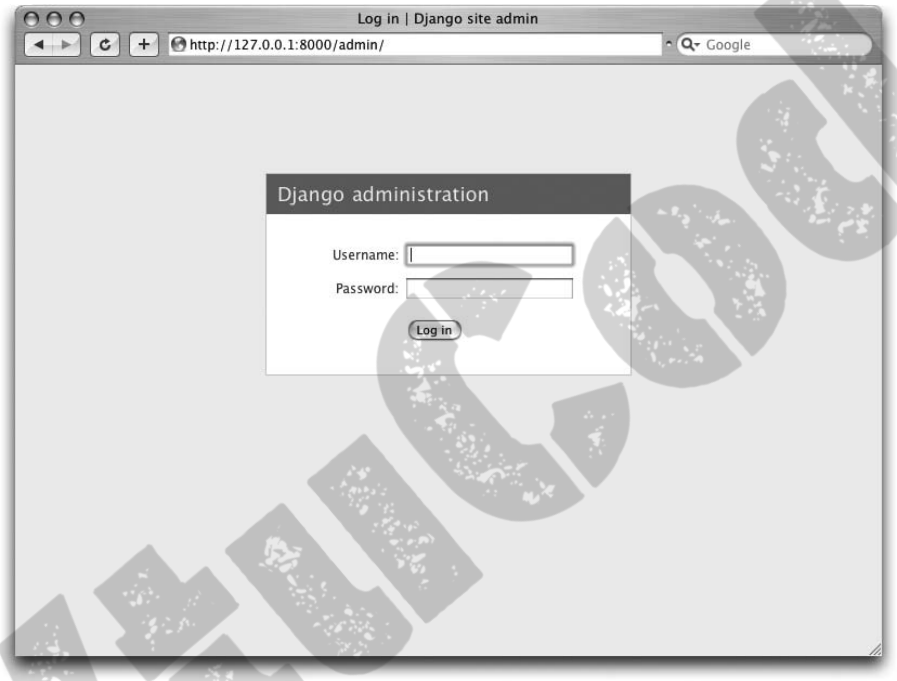

Figure 6-1. *Django's login screen*

You'll use the username and password you set up when you first added your superuser account. Once you're logged in, you'll see that you can manage users, groups, and permissions (more on that shortly).

Each object given an `Admin` declaration shows up on the main index page, as shown in Figure 6-2.

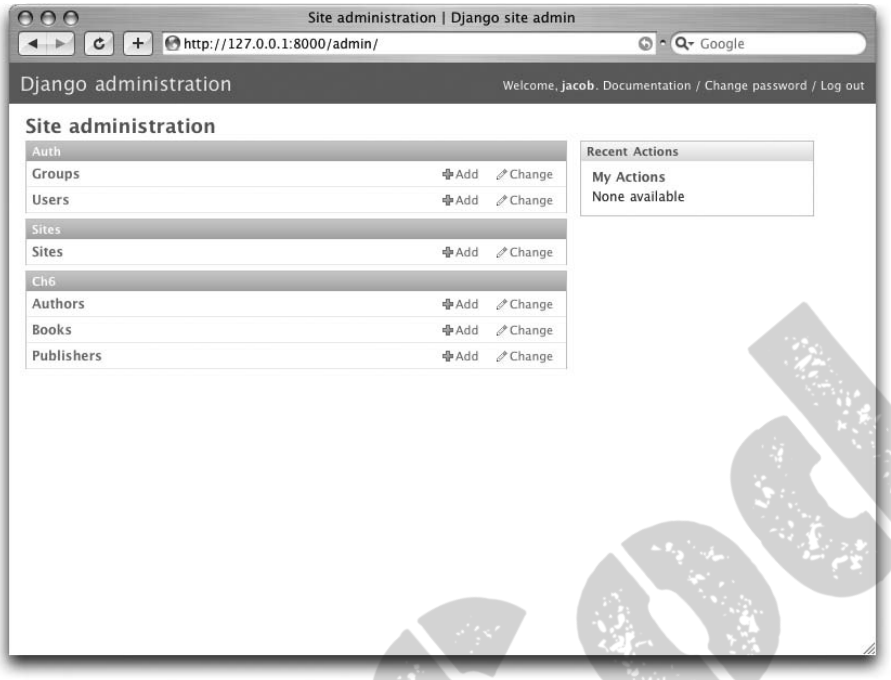

Figure 6-2. *The main Django admin index*

Links to add and change objects lead to two pages we refer to as object *change lists* and *edit forms*. Change lists are essentially index pages of objects in the system, as shown in Figure 6-3.

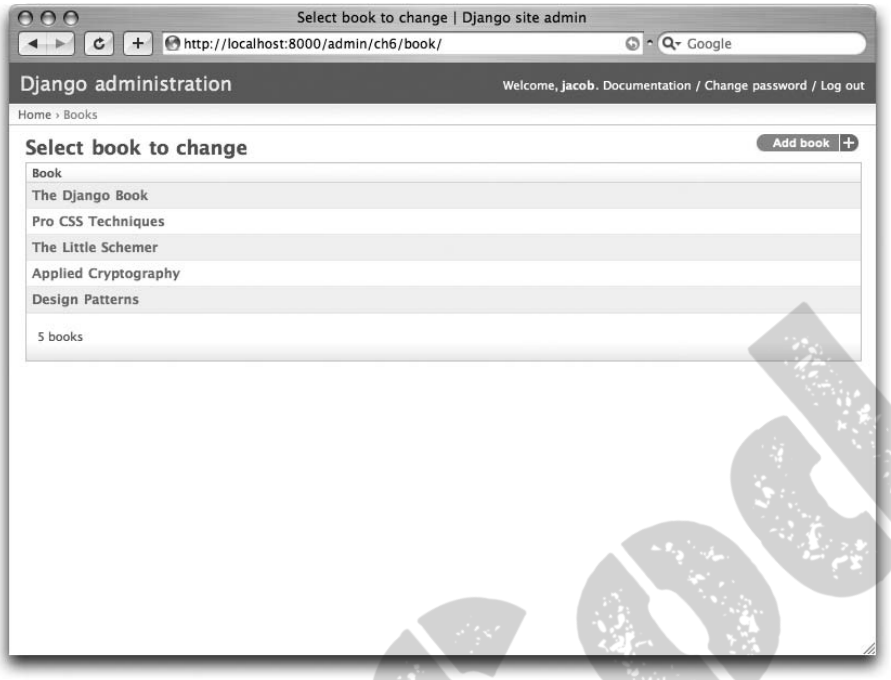

Figure 6-3. *A typical change list view*

A number of options control which fields appear on these lists and the appearance of extra features like date drill-downs, search fields, and filter interfaces. We discuss these features in more detail shortly.

Edit forms are used to modify existing objects and create new ones (see Figure 6-4). Each field defined in your model appears here, and you'll notice that fields of different types get different widgets (e.g., date/time fields have calendar controls, foreign keys use a select box, etc.).

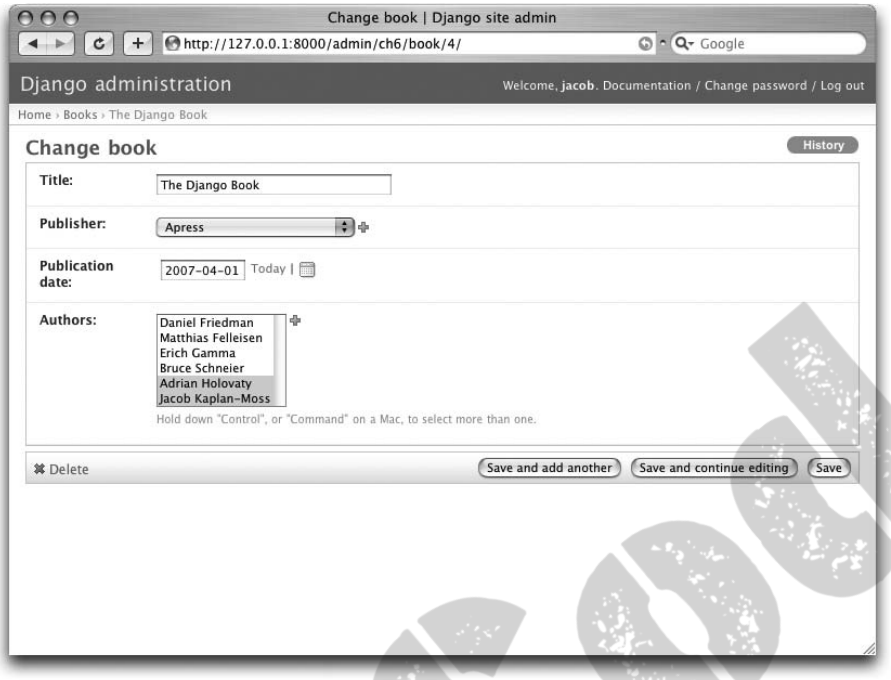

Figure 6-4. *A typical edit form*

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

When you edit an existing object, you'll notice a History button 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 button (see Figure 6-6).

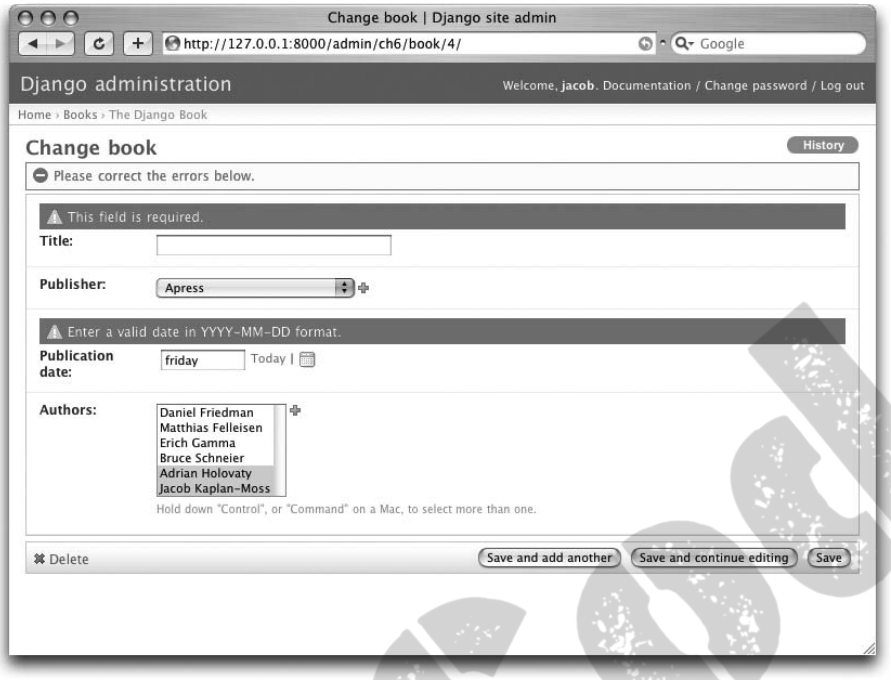

Figure 6-5. *An edit form displaying errors*

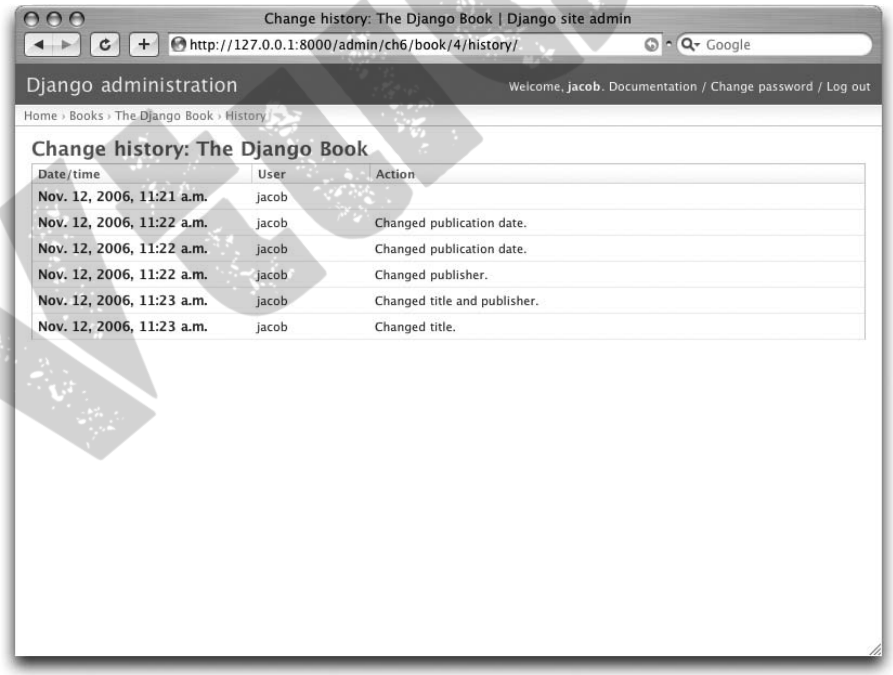

Figure 6-6. *Django's object history page*

When you delete an existing object, the admin interface asks you to confirm the delete action to avoid costly mistakes. Deletions also *cascade*; the deletion confirmation page shows you all the related objects that will be deleted as well (see Figure 6-7).

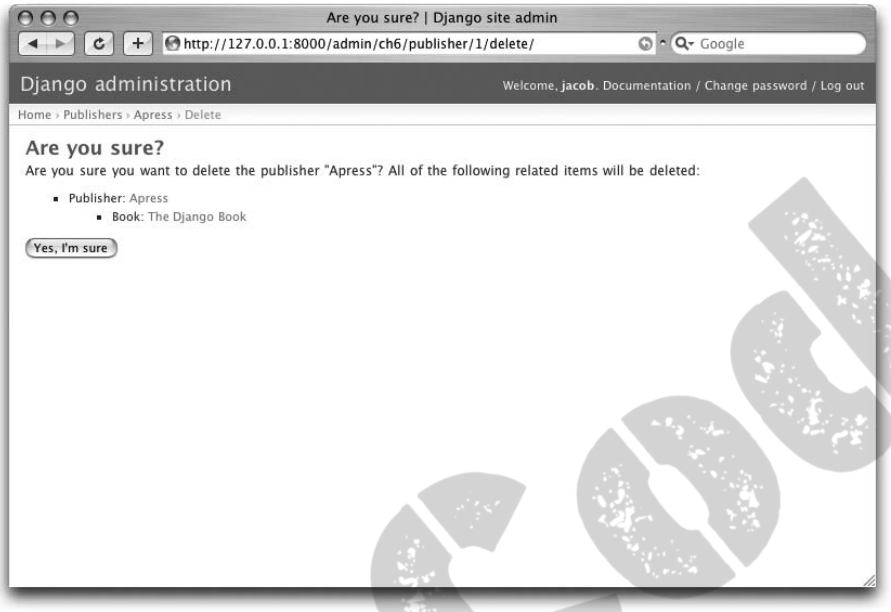

Figure 6-7. *Django's delete confirmation page*

Users, Groups, and Permissions

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

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

User objects have the standard username, password, 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 flags:

- The "is active" flag controls whether the user is active at all. If this flag is off, the user has no access to any URLs that require login.
- The "is staff" flag controls whether the user is allowed to log 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 12), this flag differentiates between public users and administrators.
- The "is superuser" flag gives the user full, unfettered access to every item in the admin interface; regular permissions are ignored.

"Normal" admin users—that is, active, nonsuperuser staff members—are granted access that depends on a set of assigned permissions. Each object editable through the admin interface has three permissions: a *create* permission, an *edit* permission, and a *delete* permission. Assigning permissions to a user grants the user access to do what is described by those permissions.

■ **Note** Access to edit users and permissions is also controlled by this permission 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!

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 large number of users.

Customizing the Admin Interface

You can customize the way the admin interface looks and behaves in a number of ways. We cover just a few of them in this section as they relate to our `Book` model; Chapter 17 covers customizing the admin interface in detail.

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

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

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

These four lines of code dramatically change our list interface, as shown in Figure 6-8.

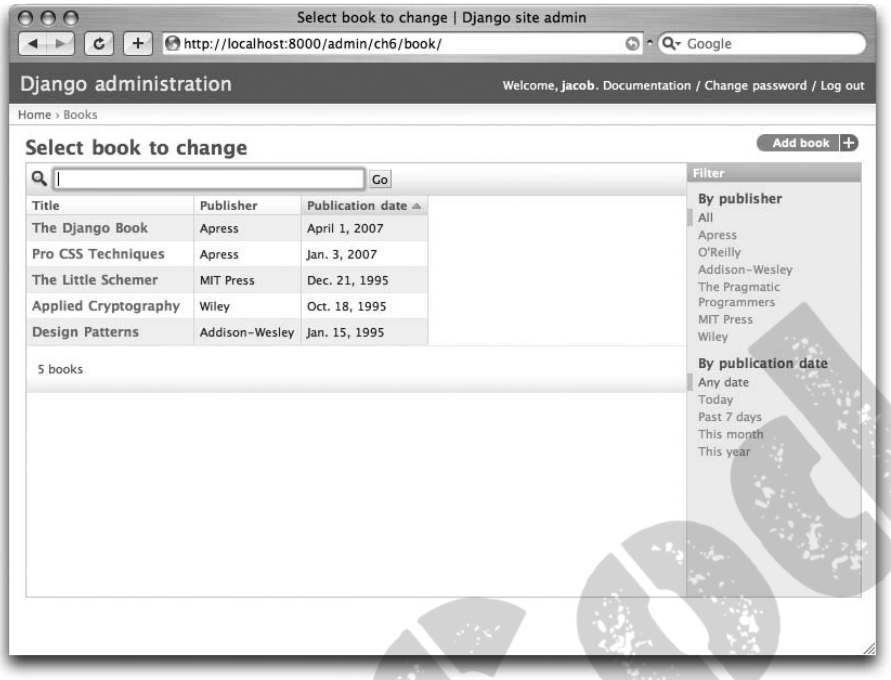

Figure 6-8. *Modified change list page*

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

- The `list_display` option controls which columns appear in the change list table. By default, the change list displays only a single column that contains the object's string representation. Here, we've changed that to show the title, publisher, and publication date.
- The `list_filter` option creates the filtering bar on the right side of the list. We've allowed filtering by date (which allows you to see only books published in the last week, month, etc.) and by publisher. The filters show up as long as there are at least two values to choose from.

  You can instruct the admin interface to filter by any field, but foreign keys, dates, Booleans, and fields with a `choices` attribute work best.
- The `ordering` option controls the order in which the objects are presented in the admin interface. It's simply a list of fields by which to order the results; prefixing a field with a minus sign reverses the given order. In this example, we're ordering by publication date, with the most recent first.
- Finally, the `search_fields` option creates a field that allows text searches. It allows searches by the `title` field (so you could type **Django** to show all books with "Django" in the title).

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

Customizing the Admin Interface's Look and Feel

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

It's easy to change, though, using Django's template system. The Django admin site is powered by Django itself, and its interfaces use Django's own template system. (Django's template system was covered in Chapter 4.)

As we explained in Chapter 4, the `TEMPLATE_DIRS` setting specifies a list of directories to check when loading Django templates. To customize Django's admin templates, simply copy the relevant stock admin template from the Django distribution into one of the directories pointed to by `TEMPLATE_DIRS`.

The admin site finds the "Django administration" header by looking for the template `admin/base_site.html`. By default, this template lives in the Django admin template directory, `django/contrib/admin/templates`, which you can find by looking in your Python `site-packages` directory, or wherever Django was installed. To customize this `base_site.html` template, copy that template into an `admin` subdirectory of whichever directory you're using in `TEMPLATE_DIRS`. For example, if your `TEMPLATE_DIRS` includes `"/home/mytemplates"`, then copy `django/contrib/admin/templates/admin/base_site.html` to `/home/mytemplates/admin/base_site.html`. Don't forget that `admin` subdirectory.

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

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

You might wonder how, if `TEMPLATE_DIRS` was empty by default, Django found the default admin templates. The answer is that, by default, Django automatically looks for templates within a `templates/` subdirectory in each application package as a fallback. See the "Writing Custom Template Loaders" section in Chapter 10 for more information about how this works.

Customizing the Admin Index Page

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

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

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

For full details on customizing the look and feel of the Django admin site in general, see Chapter 17.

When and Why to Use the Admin Interface

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

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

Django's admin interface 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, development of a typical online feature—a special report on water quality in the municipal supply, say—goes something like this:

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

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

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

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

CHAPTER 7

Form Processing

Guest author: Simon Willison

After following along with the last chapter, you should now have a fully functioning if somewhat simple site. In this chapter, we'll deal with the next piece of the puzzle: building views that take input from readers.

We'll start by making a simple search form "by hand" and looking at how to handle data submitted from the browser. From there, we'll move on to using Django's forms framework.

Search

The Web is all about search. Two of the Net's biggest success stories, Google and Yahoo, built their multibillion-dollar businesses around search. Nearly every site sees a large percentage of traffic coming to and from its search pages. Often the difference between a site's success or failure is the quality of its search. So it looks like we'd better add some searching to our fledgling books site, no?

We'll start by adding the search view to our URLconf (`mysite.urls`). Recall that this means adding something like (`r'^search/$', 'mysite.books.views.search'`) to the set of URL patterns.

Next, we'll write this `search` view into our view module (`mysite.books.views`):

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

def search(request):
    query = request.GET.get('q', '')
    if query:
        qset = (
            Q(title__icontains=query) |
            Q(authors__first_name__icontains=query) |
            Q(authors__last_name__icontains=query)
        )
        results = Book.objects.filter(qset).distinct()
```

```
    else:
        results = []
    return render_to_response("books/search.html", {
        "results": results,
        "query": query
    })
```

There are a couple of things going on here that you haven't yet seen. First, there's `request.GET`. This is how you access GET data from Django; POST data is accessed through a similar `request.POST` object. These objects behave exactly like standard Python dictionaries with some extra features covered in Appendix H.

WHAT'S GET AND POST DATA?

GET and POST are the two methods that browsers use to send data to a server. Most of the time, you'll see them in HTML form tags:

```
<form action="/books/search/" method="get">
```

This instructs the browser to submit the form data to the URL `/books/search/` using the GET method.

There are important differences between the semantics of GET and POST that we won't get into right now, but see `http://www.w3.org/2001/tag/doc/whenToUseGet.html` if you want to learn more.

So the line

```
query = request.GET.get('q', '')
```

looks for a GET parameter named `q` and returns an empty string if that parameter wasn't submitted.

Note that we're using the `get()` method on `request.GET`, which is potentially confusing. The `get()` method here is the one that every Python dictionary has. We're using it here to be careful: it is *not* safe to assume that `request.GET` contains a `'q'` key, so we use `get('q', '')` to provide a default fallback value of `''` (the empty string). If we merely accessed the variable using `request.GET['q']`, that code would raise a `KeyError` if `q` wasn't available in the GET data.

Second, what about this `Q` business? `Q` objects are used to build up complex queries—in this case, we're searching for any books where either the title or the name of one of the authors matches the search query. Technically, these `Q` objects comprise a `QuerySet`, and you can read more about them in Appendix C.

In these queries, `icontains` is a case-insensitive search that uses the SQL `LIKE` operator in the underlying database.

Since we're searching against a many-to-many field, it's possible for the same book to be returned more than once by the query (e.g., a book with two authors who both match the search query). Adding `.distinct()` to the filter lookup eliminates any duplicate results.

There's still no template for this search view, however. This should do the trick:

```
<!DOCTYPE HTML PUBLIC "-//W3C/DTD HTML 4.01//EN">
<html lang="en">
<head>
  <title>Search {% if query %}Results{% endif %}</title>
</head>
<body>
  <h1>Search</h1>
  <form action="." method="GET">
    <label for="q">Search: </label>
    <input type="text" name="q" value="{{ query|escape }}">
    <input type="submit" value="Search">
  </form>

  {% if query %}
    <h2>Results for "{{ query|escape }}":</h2>

    {% if results %}
      <ul>
      {% for book in results %}
        <li>{{ book }}</l1>
      {% endfor %}
      </ul>
    {% else %}
      <p>No books found</p>
    {% endif %}
  {% endif %}
</body>
</html>
```

Hopefully by now what this does is fairly obvious. However, there are a few subtleties worth pointing out:

- The form's action is `.`, which means "the current URL." This is a standard best practice: don't use separate views for the form page and the results page—use a single one that serves the form and search results.
- We reinsert the value of the query back into the `<input>`. This lets readers easily refine their searches without having to retype what they searched for.
- Everywhere `query` is used, we pass it through the `escape` filter to make sure that any potentially malicious search text is filtered out before being inserted into the page.

  It's *vital* that you do this with any user-submitted content! Otherwise you open your site up to cross-site scripting (XSS) attacks. Chapter 19 discusses XSS and security in more detail.
- However, we don't need to worry about harmful content in your database lookups—we can simply pass the query into the lookup as is. This is because Django's database layer handles this aspect of security for you.

Now we have a working search. A further improvement would be putting a search form on every page (i.e., in the base template); we'll let you handle that one yourself.

Next, we'll look at a more complex example. But before we do, let's discuss a more abstract topic: the "perfect form."

The "Perfect Form"

Forms can often be a major cause of frustration for the users of your site. Let's consider the behavior of a hypothetical perfect form:

- It should ask the user for some information, obviously. Accessibility and usability matter here, so smart use of the HTML `<label>` element and useful contextual help are important.
- The submitted data should be subjected to extensive validation. The golden rule of Web application security is "never trust incoming data," so validation is essential.
- If the user has made any mistakes, the form should be redisplayed with detailed, informative error messages. The original data should be prefilled, to save the user from having to reenter everything.
- The form should continue to redisplay until all of the fields have been correctly filled.

Constructing the perfect form seems like a lot of work! Thankfully, Django's forms framework is designed to do most of the work for you. You provide a description of the form's fields, the validation rules, and a simple template, and Django does the rest. The result is a "perfect form" with very little effort.

Creating a Feedback Form

The best way to build a site that people love is to listen to their feedback. Many sites appear to have forgotten this; they hide their contact details behind layers of FAQs, and they seem to make it as difficult as possible to get in touch with an actual human being.

When your site has millions of users, this may be a reasonable strategy. When you're trying to build up an audience, though, you should actively encourage feedback at every opportunity. Let's build a simple feedback form and use it to illustrate Django's forms framework in action.

We'll start by defining our form. Forms in Django are created in a similar way to models: declaratively, using a Python class. Here's the class for our simple form. By convention, we'll insert it into a new `forms.py` file within our application directory.

```
from django import newforms as forms

TOPIC_CHOICES = (
    ('general', 'General enquiry'),
    ('bug', 'Bug report'),
    ('suggestion', 'Suggestion'),
)
```

```
class ContactForm(forms.Form):
    topic = forms.ChoiceField(choices=TOPIC_CHOICES)
    message = forms.CharField()
    sender = forms.EmailField(required=False)
```

"NEW" FORMS? WHAT?

When Django was first released to the public, it had a complicated, confusing forms system. It made producing forms far too difficult, so it was completely rewritten and is now called "newforms." However, there's still a fair amount of code that depends on the "old" form system, so for the time being Django ships with two form packages.

As we write this book, Django's old form system is still available as `django.forms` and the new form package as `django.newforms`. At some point that will change and `django.forms` will point to the new form package. However, to make sure the examples in this book work as widely as possible, all the examples will refer to `django.newforms`.

A Django form is a subclass of `django.newforms.Form`, just as a Django model is a subclass of `django.db.models.Model`. The `django.newforms` module also contains a number of `Field` classes; a full list is available in Django's documentation at `http://www.djangoproject.com/documentation/0.96/newforms/`.

Our `ContactForm` consists of three fields: a topic, which is a choice among three options; a message, which is a character field; and a sender, which is an email field and is optional (because even anonymous feedback can be useful). There are a number of other field types available, and you can write your own if they don't cover your needs.

The form object itself knows how to do a number of useful things. It can validate a collection of data, it can generate its own HTML "widgets," it can construct a set of useful error messages and, if we're feeling lazy, it can even draw the entire form for us. Let's hook it into a view and see it in action. In `views.py`:

```
from django.shortcuts import render_to_response
from mysite.books.forms import ContactForm

def contact(request):
    form = ContactForm()
    return render_to_response('contact.html', {'form': form})
```

and in `contact.html`:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
<html lang="en">
<head>
    <title>Contact us</title>
</head>
<body>
    <h1>Contact us</h1>
    <form action="." method="POST">
```

```
        <table>
            {{ form.as_table }}
        </table>
        <p><input type="submit" value="Submit"></p>
    </form>
</body>
</html>
```

The most interesting line here is `{{ form.as_table }}`. `form` is our ContactForm instance, as passed to `render_to_response`. `as_table` is a method on that object that renders the form as a sequence of table rows (`as_ul` and `as_p` can also be used). The generated HTML looks like this:

```
<tr>
    <th><label for="id_topic">Topic:</label></th>
    <td>
        <select name="topic" id="id_topic">
            <option value="general">General enquiry</option>
            <option value="bug">Bug report</option>
            <option value="suggestion">Suggestion</option>
        </select>
    </td>
</tr>
<tr>
    <th><label for="id_message">Message:</label></th>
    <td><input type="text" name="message" id="id_message" /></td>
</tr>
<tr>
    <th><label for="id_sender">Sender:</label></th>
    <td><input type="text" name="sender" id="id_sender" /></td>
</tr>
```

Note that the `<table>` and `<form>` tags are not included; you need to define those yourself in the template, which gives you control over how the form behaves when it is submitted. Label elements *are* included, making forms accessible out of the box.

Our form is currently using a `<input type="text">` widget for the message field. We don't want to restrict our users to a single line of text, so we'll swap in a `<textarea>` widget instead:

```
class ContactForm(forms.Form):
    topic = forms.ChoiceField(choices=TOPIC_CHOICES)
    message = forms.CharField(widget=forms.Textarea())
    sender = forms.EmailField(required=False)
```

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.

At the moment, submitting the form doesn't actually do anything. Let's hook in our validation rules:

```
def contact(request):
    if request.method == 'POST':
        form = ContactForm(request.POST)
    else:
        form = ContactForm()
    return render_to_response('contact.html', {'form': form})
```

A form instance can be in one of two states: bound or unbound. A *bound* instance is attached to a dictionary (or dictionary-like object) and knows how to validate and redisplay the data from it. An *unbound* form has no data associated with it and simply knows how to display itself.

Try clicking Submit on the blank form. The page should redisplay, showing a validation error that informs us that our message field is required.

Try entering an invalid email address as well. The `EmailField` knows how to validate email addresses, at least to a reasonable level of doubt.

SETTING INITIAL DATA

Passing data directly to the form constructor binds that data and indicates that validation should be performed. Often, though, we need to display an initial form with some of the fields prefilled—for example, an "edit" form. We can do this with the `initial` keyword argument:

```
form = CommentForm(initial={'sender': 'user@example.com'})
```

If our form will *always* use the same default values, we can configure them in the form definition itself:

```
message = forms.CharField(initial="Replace with your feedback"
```

Processing the Submission

Once the user has filled the form to the point that it passes our validation rules, we need to do something useful with the data. In this case, we want to construct and send an email containing the user's feedback. We'll use Django's email package to do this.

First, though, we need to tell if the data is indeed valid, and if it is, we need access to the validated data. The forms framework does more than just validate the data; it also converts it into Python types. Our contact form only deals with strings, but if we were to use an `IntegerField` or `DateTimeField`, the forms framework would ensure that we got back a Python integer or `datetime` object, respectively.

To tell whether a form is bound to valid data, call the `is_valid()` method:

```
form = ContactForm(request.POST)
if form.is_valid():
    # Process form data
```

Now we need access to the data. We could pull it straight out of `request.POST`, but if we did, we'd miss out on the type conversions performed by the forms framework. Instead, we use `form.clean_data`:

```
if form.is_valid():
    topic = form.clean_data['topic']
    message = form.clean_data['message']
    sender = form.clean_data['sender']
    # ...
```

Finally, we need to record the user's feedback. The easiest way to do this is to email it to a site administrator. We can do that using the `send_mail` function:

```
from django.core.mail import send_mail

# ...

send_mail(
    'Feedback from your site, topic: %s' % topic,
    message, sender,
    ['administrator@example.com']
)
```

The `send_mail` function has four required arguments: the email subject, the email body, the "from" address, and a list of recipient addresses. `send_mail` is a convenient wrapper around Django's `EmailMessage` class, which provides advanced features such as attachments, multi-part emails, and full control over email headers.

Having sent the feedback email, we'll redirect our user to a static confirmation page. The finished view function looks like this:

```
from django.http import HttpResponseRedirect
from django.shortcuts import render_to_response
from django.core.mail import send_mail
from forms import ContactForm

def contact(request):
    if request.method == 'POST':
        form = ContactForm(request.POST)
        if form.is_valid():
            topic = form.cleaned_data['topic']
            message = form.cleaned_data['message']
            sender = form.cleaned_data.get('sender', 'noreply@example.com')
            send_mail(
                'Feedback from your site, topic: %s' % topic,
                message, sender,
                ['administrator@example.com']
            )
            return HttpResponseRedirect('/contact/thanks/')
    else:
        form = ContactForm()
    return render_to_response('contact.html', {'form': form})
```

■ **Note** If a user selects Refresh on a page that was displayed by a POST request, that request will be repeated. This can often lead to undesired behavior, such as a duplicate record being added to the database. Redirect after POST is a useful pattern that can help avoid this scenario: after a successful POST has been processed, redirect the user to another page rather than returning HTML directly.

Custom Validation Rules

Imagine we've launched our feedback form, and the emails have started tumbling in. There's just one problem: some of the emails contain just one or two words, hardly enough for a detailed missive. We decide to adopt a new validation policy: four words or more, please.

There are a number of 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 need to add a `clean_message` method to our form:

```
class ContactForm(forms.Form):
    topic = forms.ChoiceField(choices=TOPIC_CHOICES)
    message = forms.CharField(widget=forms.Textarea())
    sender = forms.EmailField(required=False)

    def clean_message(self):
        message = self.clean_data.get('message', '')
        num_words = len(message.split())
        if num_words < 4:
            raise forms.ValidationError("Not enough words!")
        return message
```

This new method will be called after the default field validator (in this case, the validator for a required `CharField`). Because the field data has already been partially processed, we need to pull it out of the form's `clean_data` dictionary.

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 `ValidationError`. The string attached to this exception will be displayed to the user as an item in the error list.

It is important that we explicitly return the 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 and then `None` will be returned, and the original value will be lost.

A Custom Look and Feel

The quickest way to customize the form's presentation is with CSS. The list of errors in particular could do with some visual enhancement, and the `<ul>` has a class attribute of `errorlist` for that exact purpose. 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>
```

While it's convenient to have our form's HTML generated for us, in many cases the default rendering won't be right for our application. `{{ form.as_table }}` and friends are useful shortcuts while we develop our application, but everything about the way a form is displayed can be overridden, mostly within the template itself.

Each field widget (`<input type="text">`, `<select>`, `<textarea>`, or similar) can be rendered individually by accessing `{{ form.fieldname }}`. Any errors associated with a field are available as `{{ form.fieldname.errors }}`. We can use these form variables to construct a custom template for our contact form:

```
<form action="." method="POST">
    <div class="fieldWrapper">
        {{ form.topic.errors }}
        <label for="id_topic">Kind of feedback:</label>
        {{ form.topic }}
    </div>
    <div class="fieldWrapper">
        {{ form.message.errors }}
        <label for="id_message">Your message:</label>
        {{ form.message }}
    </div>
    <div class="fieldWrapper">
        {{ form.sender.errors }}
        <label for="id_sender">Your email (optional):</label>
        {{ form.sender }}
    </div>
    <p><input type="submit" value="Submit"></p>
</form>
```

`{{ form.message.errors }}` will display as 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, for example:

```
<div class="fieldWrapper{% if form.message.errors %} errors{% endif %}">
    {% if form.message.errors %}
        <ol>
        {% for error in form.message.errors %}
            <li><strong>{{ error|escape }}</strong></li>
        {% endfor %}
        </ol>
    {% endif %}
    {{ 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 ordered list.

Creating Forms from Models

Let's build something a little more interesting: a form that submits a new publisher to our book application from Chapter 5.

An important principle in software development that Django tries to adhere to is Don't Repeat Yourself (DRY). Andy Hunt and Dave Thomas in *The Pragmatic Programmer* (Addison-Wesley, 1999) define this as follows:

> *Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.*

Our `Publisher` model class says that a publisher has a `name`, `address`, `city`, `state_province`, `country`, and `website`. Duplicating this information in a form definition would break the DRY rule. Instead, we can use a useful shortcut: `form_for_model()`:

```
from models import Publisher
from django.newforms import form_for_model

PublisherForm = form_for_model(Publisher)
```

`PublisherForm` is a `Form` subclass, just like the `ContactForm` class we created manually earlier on. We can use it in much the same way:

```
def add_publisher(request):
    if request.method == 'POST':
        form = PublisherForm(request.POST)
        if form.is_valid():
            form.save()
            return HttpResponseRedirect('/add_publisher/thanks/')
    else:
        form = PublisherForm()
    return render_to_response('add_publisher.html', {'form': form})
```

The `add_publisher.html` file is almost identical to our original `contact.html` template, so it has been omitted.

There's one more shortcut being demonstrated here. Since forms derived from models are often used to save new instances of the model to the database, the form class created by `form_for_model` includes a convenient `save()` method. This deals with the common case; you're welcome to ignore it if you want to do something a bit more involved with the submitted data.

`form_for_instance()` is a related method that can create a preinitialized form from an instance of a model class. This is useful for creating "edit" forms.

CHAPTER 8

Advanced Views and URLconfs

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

URLconf Tricks

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 current_datetime, hours_ahead, hours_behind,
now_in_chicago, now_in_london

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

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

But as a Django application grows in complexity, its URLconf grows, too, and keeping those imports can be tedious to manage. (For each new view function, you 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'^now/$', views.current_datetime),
    (r'^now/plus(\d{1,2})hours/$', views.hours_ahead),
    (r'^now/minus(\d{1,2})hours/$', views.hours_behind),
    (r'^now/in_chicago/$', views.now_in_chicago),
    (r'^now/in_london/$', views.now_in_london),
)
```

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

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

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

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

Note that you don't put a trailing dot (`"."`) in the prefix, nor do you put a leading dot in the view strings. Django puts 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.
- 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'^/?$', 'mysite.views.archive_index'),
    (r'^(\d{4})/([a-z]{3})/$', 'mysite.views.archive_month'),
    (r'^tag/(\w+)/$', 'weblog.views.tag'),
)
```

New:

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

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

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

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

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

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

if settings.DEBUG:
    urlpatterns += patterns('',
        (r'^debuginfo$', 'mysite.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 for 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 template they use:

```
# urls.py

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

urlpatterns = patterns('',
    (r'^foo/$', views.foo_view),
    (r'^bar/$', views.bar_view),
)

# views.py

from django.shortcuts import render_to_response
from mysite.models import MyModel

def foo_view(request):
    m_list = MyModel.objects.filter(is_new=True)
    return render_to_response('template1.html', {'m_list': m_list})

def bar_view(request):
    m_list = MyModel.objects.filter(is_new=True)
    return render_to_response('template2.html', {'m_list': m_list})
```

We're repeating ourselves in this code, and that's inelegant. At first, you may think to remove the redundancy by using the same view for both URLs, putting 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)/$', views.foobar_view),
    (r'^(bar)/$', views.foobar_view),
)

# views.py

from django.shortcuts import render_to_response
from mysite.models import MyModel

def foobar_view(request, url):
    m_list = MyModel.objects.filter(is_new=True)
    if url == 'foo':
        template_name = 'template1.html'
    elif url == 'bar':
        template_name = 'template2.html'
    return render_to_response(template_name, {'m_list': m_list})
```

The problem with that solution, though, is that it couples your URLs to your code. If you decide to rename `/foo/` to `/fooey/`, you'll have to remember to change the view code.

The elegant solution involves 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/$', views.foobar_view, {'template_name': 'template1.html'}),
    (r'^bar/$', views.foobar_view, {'template_name': 'template2.html'}),
)

# views.py

from django.shortcuts import render_to_response
from mysite.models import MyModel
```

```
def foobar_view(request, template_name):
    m_list = MyModel.objects.filter(is_new=True)
    return render_to_response(template_name, {'m_list': m_list})
```

As you can see, the URLconf in this example specifies `template_name` in the URLconf. The view function treats it as just another parameter.

This extra URLconf options technique is a nice way of sending additional information to your view functions with minimal fuss. As such, it's used by a couple of Django's bundled applications, most notably its generic views system, which we cover in Chapter 9.

The following sections contain a couple of ideas about 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 will 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 the next chapter.

Giving a View Configuration Options

If you're distributing a Django application, chances are that your users will want some degree of configuration. In this case, it's a good idea to add hooks to your views for any configuration

options you think people may want to change. You can use extra URLconf parameters for this purpose.

A common bit of an application to make configurable is the template name:

```
def my_view(request, template_name):
    var = do_something()
    return render_to_response(template_name, {'var': var})
```

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 *

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 *

urlpatterns = patterns('',
    (r'^blog/$', views.page),
    (r'^blog/page(?P<num>\d+)/$', views.page),
)

# views.py

def page(request, num="1"):
    # Output the appropriate page of blog entries, according to num.
    # ...
```

Here, both URL patterns point to the same view—`views.page`—but the first pattern doesn't capture anything from the URL. If the first pattern matches, the `page()` function will use its default argument for `num`, `"1"`. If the second pattern matches, `page()` will use whatever `num` value was captured by the regular expression.

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, the "add an object" pages in Django's admin site are represented by this URLconf line:

```
urlpatterns = patterns('',
    # ...
    ('^([^/]+)/([^/]+)/add/$', 'django.contrib.admin.views.main.add_stage'),
    # ...
)
```

This matches URLs such as `/myblog/entries/add/` and `/auth/groups/add/`. However, the "add" page for a user object (`/auth/user/add/`) is a special case—it doesn't display all of the form fields, it displays two password fields, 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/$', 'django.contrib.admin.views.auth.user_add_stage'),
    ('^([^/]+)/([^/]+)/add/$', 'django.contrib.admin.views.main.add_stage'),
    # ...
)
```

With this in place, a request to `/auth/user/add/` will be handled by the `user_add_stage` view. Although that URL matches the second pattern, it matches the top one first. (This is short-circuit logic.)

Capturing Text in URLs

Each captured argument is sent to the view as a plain Python string, regardless of what sort of match the regular expression makes. For example, in the following URLconf line:

```
(r'^articles/(?P<year>\d{4})/$', views.year_archive),
```

the `year` argument to `views.year_archive()` will be a string, not an integer, even though `\d{4}` will only match integer strings.

This is important to keep in mind when you're writing view code. Many built-in Python functions are fussy (and rightfully so) about accepting only objects of a certain type. A common error is to attempt to create a `datetime.date` object with string values instead of integer values:

```
>>> import datetime
>>> datetime.date('1993', '7', '9')
Traceback (most recent call last):
    ...
TypeError: an integer is required
>>> datetime.date(1993, 7, 9)
datetime.date(1993, 7, 9)
```

Translated to a URLconf and view, the error looks like this:

```
# urls.py

from django.conf.urls.defaults import *

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

# views.py

import datetime

def day_archive(request, year, month, day)
    # The following statement raises a TypeError!
    date = datetime.date(year, month, day)
```

Instead, `day_archive()` can be written correctly like this:

```
def day_archive(request, year, month, day)
    date = datetime.date(int(year), int(month), int(day))
```

Note that `int()` itself raises a `ValueError` when you pass it a string that is not 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 normal Python string (not as a Unicode string). This does not include `GET` or `POST`

parameters, or the domain name. It also does not include the leading slash, because every URL has a leading slash.

For example, in a request to `http://www.example.com/myapp/`, Django will try to match `myapp/`. In a request to `http://www.example.com/myapp/?page=3`, Django will try to match `myapp/`.

The request method (e.g., `POST`, `GET`, `HEAD`) is *not* taken into account when traversing the URLconf. In other words, all request methods will be routed to the same function for the same URL. It's the responsibility of a view function to perform branching based on request method.

Including Other URLconfs

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'),
)
```

There's an important gotcha here: the regular expressions in this example that point to an `include()` do *not* have a `$` (end-of-string match character) but *do* include a trailing slash. Whenever Django encounters `include()`, it chops off whatever part of the URL matched up to that point and sends the remaining string to the included URLconf for further processing.

Continuing this example, here's the URLconf `mysite.blog.urls`:

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

urlpatterns = patterns('',
    (r'^(\d\d\d\d)/$', 'mysite.blog.views.year_detail'),
    (r'^(\d\d\d\d)/(\d\d)/$', 'mysite.blog.views.month_detail'),
)
```

With these two URLconfs, here's how a few sample requests would be handled:

- `/weblog/2007/`: In the first URLconf, the pattern `r'^weblog/'` matches. Because it is an `include()`, Django strips all the matching text, which is `'weblog/'` in this case. The remaining part of the URL is `2007/`, which matches the first line in the `mysite.blog.urls` URLconf.
- `/weblog//2007/`: In the first URLconf, the pattern `r'^weblog/'` matches. Because it is an `include()`, Django strips all the matching text, which is `'weblog/'` in this case. The remaining part of the URL is `/2007/` (with a leading slash), which does not match any of the lines in the `mysite.blog.urls` URLconf.
- `/about/`: 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 1:

```
# 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 2:

```
# 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.