Chapter 11: 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_pages(request, page):
    try:
        return direct_to_template(request, template="about/%s.html" % page)
    except TemplateDoesNotExist:
        raise Http404()
```

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

> **Is There a Security Vulnerability Here?**
>
> Sharp-eyed readers may have noticed a possible security hole: we're constructing the template name using interpolated content from the browser (`template="about/%s.html" % page`). At first glance, this looks like a classic *directory traversal* vulnerability (discussed in detail in Chapter 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` only accepts 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 a handful of patterns dealt with in the sections that follow.

Making "Friendly" Template Contexts

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

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

To pick a simple example, we might 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 actually work unless you add a `last_accessed` field to your `Author` model and create a `books/author_detail.html` template.

We can use a similar idiom to alter the response returned by the generic view. If we wanted to provide a downloadable plain-text version of the list of authors, we could use a view like this:

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

This works because the generic views return simple `HttpResponse` objects that can be treated like dictionaries to set HTTP headers. This `Content-Disposition` business, by the way, instructs the browser to download and save the page instead of displaying it in the browser.

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.