Chapter 13: Generating Non-HTML Content

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

So far, we've focused on the common case of HTML production, but in this chapter we'll take a detour and look at using Django to produce other types of content.

Django has convenient built-in tools that you can use to produce some common non-HTML content:

- RSS/Atom syndication feeds
- Sitemaps (an XML format originally developed by Google that gives hints to search engines)

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

The basics: views and MIME-types

Recall from Chapter 3 that a view function is simply a Python function that takes a Web request and returns a Web response. This response can be the HTML contents of a Web page, or a redirect, or a 404 error, or an XML document, or an image...or anything, really.

More formally, a Django view function *must*

- Accept an `HttpRequest` instance as its first argument
- Return an `HttpResponse` instance

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

For example, let's look at a view that returns a PNG image. To keep things simple, we'll just read the file off the disk:

```
from django.http import HttpResponse


def my_image(request):

    image_data = open("/path/to/my/image.png",
"rb").read()

    return HttpResponse(image_data,
mimetype="image/png")
```

That's it! If you replace the image path in the `open()` call with a path to a real image, you can use this very simple view to serve an image, and the browser will display it correctly.

The other important thing to keep in mind is that `HttpResponse` objects implement Python's standard "file-like object" API. This means that you can use an `HttpResponse` instance in any place Python (or a third-party library) expects a file.

For an example of how that works, let's take a look at producing CSV with Django.

Producing CSV

CSV is a simple data format usually used by spreadsheet software. It's basically a series of table rows, with each cell in the row separated by a comma (CSV stands for *comma-separated values*). For example, here's some data on "unruly" airline passengers in CSV format:

```
Year,Unruly Airline Passengers

1995,146

1996,184

1997,235

1998,200

1999,226

2000,251

2001,299

2002,273

2003,281

2004,304

2005,203

2006,134

2007,147
```

Note

The preceding listing contains real numbers! They come from the U.S. Federal Aviation Administration.

Though CSV looks simple, its formatting details haven't been universally agreed upon. Different pieces of software produce and consume different variants of CSV, making it a bit tricky to use. Luckily, Python comes with a standard CSV library, `csv`, that is pretty much bulletproof.

Because the `csv` module operates on file-like objects, it’s a snap to use an `HttpResponse` instead:

```
import csv

from django.http import HttpResponse



# Number of unruly passengers each year 1995 - 2005. In
a real application

# this would likely come from a database or some other
back-end data store.

UNRULY_PASSENGERS =
[146,184,235,200,226,251,299,273,281,304,203]



def unruly_passengers_csv(request):

    # Create the HttpResponse object with the
appropriate CSV header.

    response = HttpResponse(mimetype='text/csv')

    response['Content-Disposition'] = 'attachment;
filename=unruly.csv'



    # Create the CSV writer using the HttpResponse as
the "file."

    writer = csv.writer(response)

    writer.writerow(['Year', 'Unruly Airline
Passengers'])

    for (year, num) in zip(range(1995, 2006),
UNRULY_PASSENGERS):

        writer.writerow([year, num])



    return response
```

The code and comments should be pretty clear, but a few things deserve special mention:

- The response is given the `text/csv` MIME type (instead of the default `text/html`). This tells browsers that the document is a CSV file.
- The response gets an additional `Content-Disposition` header, which contains the name of the CSV file. This header (well, the "attachment" part) will instruct the browser to prompt for a location to save the file instead of just displaying it. This file name is arbitrary; call it whatever you want. It will be used by browsers in the "Save As" dialog.

  To assign a header on an `HttpResponse`, just treat the `HttpResponse` as a dictionary and set a key/value.
- Hooking into the CSV-generation API is easy: just pass `response` as the first argument to `csv.writer`. The `csv.writer` function expects a file-like object, and `HttpResponse` objects fit the bill.
- For each row in your CSV file, call `writer.writerow`, passing it an iterable object such as a list or tuple.
- The CSV module takes care of quoting for you, so you don't have to worry about escaping strings with quotes or commas in them. Just pass information to `writerow()`, and it will do the right thing.

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

Let's look at a few more examples.

Generating PDFs

Portable Document Format (PDF) is a format developed by Adobe that's used to represent printable documents, complete with pixel-perfect formatting, embedded fonts, and 2D vector graphics. You can think of a PDF document as the digital equivalent of a printed document; indeed, PDFs are often used in distributing documents for the purpose of printing them.

You can easily generate PDFs with Python and Django thanks to the excellent open source ReportLab library (http://www.reportlab.org/rl_toolkit.html). The advantage of generating PDF files dynamically is that you can create customized PDFs for different purposes – say, for different users or different pieces of content.

For example, your humble authors used Django and ReportLab at KUSports.com to generate customized, printer-ready NCAA tournament brackets.

Installing ReportLab

Before you do any PDF generation, however, you'll need to install ReportLab. It's usually simple: just download and install the library from http://www.reportlab.org/downloads.html.

Note

If you're using a modern Linux distribution, you might want to check your package management utility before installing ReportLab. Most package repositories have added ReportLab.

For example, if you're using Ubuntu, a simple `apt-get install python-reportlab` will do the trick nicely.

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

Test your installation by importing it in the Python interactive interpreter:

```
>>> import reportlab
```

If that command doesn't raise any errors, the installation worked.

Writing Your View

Like CSV, generating PDFs dynamically with Django is easy because the ReportLab API acts on file-like objects.

Here's a "Hello World" example:

```
from reportlab.pdfgen import canvas

from django.http import HttpResponse



def hello_pdf(request):

    # Create the HttpResponse object with the
appropriate PDF headers.

    response = HttpResponse(mimetype='application/pdf')

    response['Content-Disposition'] = 'attachment;
filename=hello.pdf'



    # Create the PDF object, using the response object
as its "file."

    p = canvas.Canvas(response)



    # Draw things on the PDF. Here's where the PDF
generation happens.
```

```
    # See the ReportLab documentation for the full list
of functionality.

    p.drawString(100, 100, "Hello world.")


    # Close the PDF object cleanly, and we're done.

    p.showPage()

    p.save()

    return response
```

A few notes are in order:

- Here we use the `application/pdf` MIME type. This tells browsers that the document is a PDF file, rather than an HTML file. If you leave off this information, browsers will probably interpret the response as HTML, which will result in scary gobbledygook in the browser window.
- Hooking into the ReportLab API is easy: just pass `response` as the first argument to `canvas.Canvas`. The `Canvas` class expects a file-like object, and `HttpResponse` objects fit the bill.
- All subsequent PDF-generation methods are called on the PDF object (in this case, `p`), not on `response`.
- Finally, it's important to call `showPage()` and `save()` on the PDF file – or else, you'll end up with a corrupted PDF file.

Complex PDFs

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

Here's the previous "Hello World" example rewritten to use `cStringIO`:

```
from cStringIO import StringIO

from reportlab.pdfgen import canvas

from django.http import HttpResponse



def hello_pdf(request):

    # Create the HttpResponse object with the
appropriate PDF headers.
```

```
    response = HttpResponse(mimetype='application/pdf')

    response['Content-Disposition'] = 'attachment;
filename=hello.pdf'



    temp = StringIO()



    # Create the PDF object, using the StringIO object
as its "file."

    p = canvas.Canvas(temp)



    # Draw things on the PDF. Here's where the PDF
generation happens.

    # See the ReportLab documentation for the full list
of functionality.

    p.drawString(100, 100, "Hello world.")



    # Close the PDF object cleanly.

    p.showPage()

    p.save()



    # Get the value of the StringIO buffer and write it
to the response.

    response.write(temp.getvalue())

    return response
```

Other Possibilities

There’s a whole host of other types of content you can generate in Python. Here are a few more ideas and some pointers to libraries you could use to implement them:

- *ZIP files*: Python's standard library ships with the `zipfile` module, which can both read and write compressed ZIP files. You could use it to provide on-demand archives of a bunch of files, or perhaps compress large documents when requested. You could similarly produce TAR files using the standard library's `tarfile` module.
- *Dynamic images*: The Python Imaging Library (PIL; http://www.pythonware.com/products/pil/) is a fantastic toolkit for producing images (PNG, JPEG, GIF, and a whole lot more). You could use it to automatically scale down images into thumbnails, composite multiple images into a single frame, or even do Web-based image processing.
- *Plots and charts*: There are a number of powerful Python plotting and charting libraries you could use to produce on-demand maps, charts, plots, and graphs. We can't possibly list them all, so here are a couple of the highlights:
- `matplotlib` (http://matplotlib.sourceforge.net/) can be used to produce the type of high-quality plots usually generated with MatLab or Mathematica.
- `pygraphviz` (http://networkx.lanl.gov/pygraphviz/), an interface to the Graphviz graph layout toolkit (http://graphviz.org/), can be used for generating structured diagrams of graphs and networks.

In general, any Python library capable of writing to a file can be hooked into Django. The possibilities are immense.

Now that we've looked at the basics of generating non-HTML content, let's step up a level of abstraction. Django ships with some pretty nifty built-in tools for generating some common types of non-HTML content.

The Syndication Feed Framework

Django comes with a high-level syndication-feed-generating framework that makes creating RSS and Atom feeds easy.

What's RSS? What's Atom?

RSS and Atom are both XML-based formats you can use to provide automatically updating "feeds" of your site's content. Read more about RSS at http://www.whatisrss.com/, and get information on Atom at http://www.atomenabled.org/.

To create any syndication feed, all you have to do is write a short Python class. You can create as many feeds as you want.

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

To create a feed, you'll write a `Feed` class and point to it in your URLconf.

Initialization

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

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

),
```

This line tells Django to use the RSS framework to handle all URLs starting with `"feeds/"`. (You can change that `"feeds/"` prefix to fit your own needs.)

This URLconf line has an extra argument: `{'feed_dict': feeds}`. Use this extra argument to pass the syndication framework the feeds that should be published under that URL.

Specifically, `feed_dict` should be a dictionary that maps a feed's slug (short URL label) to its `Feed` class. You can define the `feed_dict` in the URLconf itself. Here's a full example URLconf:

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

from mysite.feeds import LatestEntries,
LatestEntriesByCategory



feeds = {

    'latest': LatestEntries,

    'categories': LatestEntriesByCategory,

}



urlpatterns = patterns('',

    # ...

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

        {'feed_dict': feeds}),

    # ...

)
```

The preceding example registers two feeds:

- The feed represented by `LatestEntries` will live at `feeds/latest/`.
- The feed represented by `LatestEntriesByCategory` will live at `feeds/categories/`.

Once that's set up, you'll need to define the `Feed` classes themselves.

A `Feed` class is a simple Python class that represents a syndication feed. A feed can be simple (e.g., a "site news" feed, or a basic feed displaying the latest entries of a blog) or more complex (e.g., a feed displaying all the blog entries in a particular category, where the category is variable).

`Feed` classes must subclass `django.contrib.syndication.feeds.Feed`. They can live anywhere in your code tree.

A Simple Feed

This simple example describes a feed of the latest five blog entries for a given blog:

```
from django.contrib.syndication.feeds import Feed
from mysite.blog.models import Entry


class LatestEntries(Feed):
    title = "My Blog"
    link = "/archive/"
    description = "The latest news about stuff."


    def items(self):
        return Entry.objects.order_by('-pub_date')[:5]
```

The important things to notice here are as follows:

- The class subclasses `django.contrib.syndication.feeds.Feed`.
- `title`, `link`, and `description` correspond to the standard RSS `<title>`, `<link>`, and `<description>`elements, respectively.
- `items()` is simply a method that returns a list of objects that should be included in the feed as `<item>`elements. Although this example returns `Entry` objects using Django's database API, `items()` doesn't have to return model instances.

  There's just one more step. In an RSS feed, each `<item>` has a `<title>`, `<link>`, and `<description>`. We need to tell the framework what data to put into those elements.

- To specify the contents of `<title>` and `<description>`, create Django templates called`feeds/latest_title.html` and `feeds/latest_description.html`, where `latest` is the `slug` specified in the URLconf for the given feed. Note that the `.html` extension is required.

  The RSS system renders that template for each item, passing it two template context variables:

- `obj`: The current object (one of whichever objects you returned in `items()`).

- `site`: A `django.models.core.sites.Site` object representing the current site. This is useful for`{{ site.domain }}` or `{{ site.name }}`.

  If you don't create a template for either the title or description, the framework will use the template`"{{ obj }}"` by default – that is, the normal string representation of the object. (For model objects, this will be the `__unicode__()` method.

  You can also change the names of these two templates by specifying `title_template` and`description_template` as attributes of your `Feed` class.

- To specify the contents of `<link>`, you have two options. For each item in `items()`, Django first tries executing a `get_absolute_url()` method on that object. If that method doesn't exist, it tries calling a method `item_link()` in the `Feed` class, passing it a single parameter, `item`, which is the object itself.

  Both `get_absolute_url()` and `item_link()` should return the item's URL as a normal Python string.

- For the previous `LatestEntries` example, we could have very simple feed templates.`latest_title.html` contains:

```
{{ obj.title }}
```

and `latest_description.html` contains:

```
{{ obj.description }}
```

It's almost *too* easy...

A More Complex Feed

The framework also supports more complex feeds, via parameters.

For example, say your blog offers an RSS feed for every distinct "tag" you've used to categorize your entries. It would be silly to create a separate `Feed` class for each tag; that would violate the Don't Repeat Yourself (DRY) principle and would couple data to programming logic.

Instead, the syndication framework lets you make generic feeds that return items based on information in the feed's URL.

Your tag-specific feeds could use URLs like this:

- `http://example.com/feeds/tags/python/`: Returns recent entries tagged with "python"
- `http://example.com/feeds/tags/cats/`: Returns recent entries tagged with "cats"

The slug here is `"tags"`. The syndication framework sees the extra URL bits after the slug – `'python'` and`'cats'` – and gives you a hook to tell it what those URL bits mean and how they should influence which items get published in the feed.

An example makes this clear. Here's the code for these tag-specific feeds:

```
from django.core.exceptions import ObjectDoesNotExist

from mysite.blog.models import Entry, Tag


class TagFeed(Feed):

    def get_object(self, bits):

        # In case of "/feeds/tags/cats/dogs/mice/", or 
other such

        # clutter, check that bits has only one member.

        if len(bits) != 1:

            raise ObjectDoesNotExist

        return Tag.objects.get(tag=bits[0])


    def title(self, obj):

        return "My Blog: Entries tagged with %s" % 
obj.tag


    def link(self, obj):

        return obj.get_absolute_url()


    def description(self, obj):

        return "Entries tagged with %s" % obj.tag


    def items(self, obj):

        entries = 
Entry.objects.filter(tags__id__exact=obj.id)

        return entries.order_by('-pub_date')[:30]
```

Here's the basic algorithm of the RSS framework, given this class and a request to the URL/feeds/tags/python/:

1. The framework gets the URL /feeds/tags/python/ and notices there's an extra bit of URL after the slug. It splits that remaining string by the slash character ("/") and calls the Feed class's get_object() method, passing it the bits.

   In this case, bits is ['python']. For a request to /feeds/tags/python/django/, bits would be['python', 'django'].

2. get_object() is responsible for retrieving the given Tag object, from the given bits.

   In this case, it uses the Django database API to retrieve the Tag. Note that get_object() should raisedjango.core.exceptions.ObjectDoesNotExist if given invalid parameters. There's no try/except around the Tag.objects.get() call, because it's not necessary. That function raises Tag.DoesNotExist on failure, and Tag.DoesNotExist is a subclass of ObjectDoesNotExist. Raising ObjectDoesNotExist inget_object() tells Django to produce a 404 error for that request.

3. To generate the feed's <title>, <link>, and <description>, Django uses the title(), link(), anddescription() methods. In the previous example, they were simple string class attributes, but this example illustrates that they can be either strings *or* methods. For each of title, link, anddescription, Django follows this algorithm:

1. It tries to call a method, passing the obj argument, where obj is the object returned byget_object().
2. Failing that, it tries to call a method with no arguments.
3. Failing that, it uses the class attribute.
4. Finally, note that items() in this example also takes the obj argument. The algorithm for items is the same as described in the previous step – first, it tries items(obj), then items(), and then finally anitems class attribute (which should be a list).

   Full documentation of all the methods and attributes of the Feed classes is always available from the official Django documentation (http://docs.djangoproject.com/en/dev/ref/contrib/syndication/).

Specifying the Type of Feed

By default, the syndication framework produces RSS 2.0. To change that, add a feed_type attribute to yourFeed class:

```
from django.utils.feedgenerator import Atom1Feed


class MyFeed(Feed):

    feed_type = Atom1Feed
```

Note that you set feed_type to a class object, not an instance. Currently available feed types are shown in Table 11-1.

Table 11-1. Feed Types

Feed Class	Format
django.utils.feedgenerator.Rss201rev2Feed	RSS 2.01 (default)
django.utils.feedgenerator.RssUserland091Feed	RSS 0.91
django.utils.feedgenerator.Atom1Feed	Atom 1.0

Enclosures

To specify enclosures (i.e., media resources associated with a feed item such as MP3 podcast feeds), use the item_enclosure_url, item_enclosure_length,
and item_enclosure_mime_type hooks, for example:

```
from myproject.models import Song


class MyFeedWithEnclosures(Feed):

    title = "Example feed with enclosures"

    link = "/feeds/example-with-enclosures/"


    def items(self):

        return Song.objects.all()[:30]


    def item_enclosure_url(self, item):

        return item.song_url


    def item_enclosure_length(self, item):

        return item.song_length


    item_enclosure_mime_type = "audio/mpeg"
```

This assumes, of course, that you've created a `Song` object with `song_url` and `song_length` (i.e., the size in bytes) fields.

Language

Feeds created by the syndication framework automatically include the appropriate `<language>` tag (RSS 2.0) or `xml:lang` attribute (Atom). This comes directly from your `LANGUAGE_CODE` setting.

URLs

The `link` method/attribute can return either an absolute URL (e.g., `"/blog/"`) or a URL with the fully qualified domain and protocol (e.g., `"http://www.example.com/blog/"`). If `link` doesn't return the domain, the syndication framework will insert the domain of the current site, according to your `SITE_ID` setting. (See Chapter 16 for more on `SITE_ID` and the sites framework.)

Atom feeds require a `<link rel="self">` that defines the feed's current location. The syndication framework populates this automatically.

Publishing Atom and RSS Feeds in Tandem

Some developers like to make available both Atom *and* RSS versions of their feeds. That's easy to do with Django: just create a subclass of your `feed` class and set the `feed_type` to something different. Then update your URLconf to add the extra versions. Here's a full example:

```
from django.contrib.syndication.feeds import Feed
from django.utils.feedgenerator import Atom1Feed
from mysite.blog.models import Entry


class RssLatestEntries(Feed):
    title = "My Blog"
    link = "/archive/"
    description = "The latest news about stuff."


    def items(self):
        return Entry.objects.order_by('-pub_date')[:5]


class AtomLatestEntries(RssLatestEntries):
    feed_type = Atom1Feed
```

And here's the accompanying URLconf:

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

from myproject.feeds import RssLatestEntries,
AtomLatestEntries



feeds = {

    'rss': RssLatestEntries,

    'atom': AtomLatestEntries,

}



urlpatterns = patterns('',

    # ...

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

        {'feed_dict': feeds}),

    # ...

)
```

The Sitemap Framework

A *sitemap* is an XML file on your Web site that tells search engine indexers how frequently your pages change and how "important" certain pages are in relation to other pages on your site. This information helps search engines index your site.

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

```
<?xml version="1.0" encoding="UTF-8"?>

<urlset
xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">

  <url>
```

```
<loc>http://www.djangoproject.com/documentation/</loc>
    <changefreq>weekly</changefreq>
    <priority>0.5</priority>
  </url>
  <url>

<loc>http://www.djangoproject.com/documentation/0_90/</
loc>
    <changefreq>never</changefreq>
    <priority>0.1</priority>
  </url>
  ...
</urlset>
```

For more on sitemaps, see http://www.sitemaps.org/.

The Django sitemap framework automates the creation of this XML file by letting you express this information in Python code. To create a sitemap, you just need to write a `Sitemap` class and point to it in your URLconf.

Installation

To install the sitemap application, follow these steps:

1. Add `'django.contrib.sitemaps'` to your `INSTALLED_APPS` setting.
2. Make sure `'django.template.loaders.app_directories.load_template_source'` is in your`TEMPLATE_LOADERS` setting. It's in there by default, so you'll need to change this only if you've changed that setting.
3. Make sure you've installed the sites framework (see Chapter 16).

Note

The sitemap application doesn't install any database tables. The only reason it needs to go into `INSTALLED_APPS` is so the `load_template_source` template loader can find the default templates.

Initialization

To activate sitemap generation on your Django site, add this line to your URLconf:

```
(r'^sitemap\.xml$',
'django.contrib.sitemaps.views.sitemap', {'sitemaps':
sitemaps})
```

This line tells Django to build a sitemap when a client accesses `/sitemap.xml`. Note that the dot character in`sitemap.xml` is escaped with a backslash, because dots have a special meaning in regular expressions.

The name of the sitemap file is not important, but the location is. Search engines will only index links in your sitemap for the current URL level and below. For instance, if `sitemap.xml` lives in your root directory, it may reference any URL in your site. However, if your sitemap lives at `/content/sitemap.xml`, it may only reference URLs that begin with `/content/`.

The sitemap view takes an extra, required argument: `{'sitemaps': sitemaps}`. `sitemaps` should be a dictionary that maps a short section label (e.g., `blog` or `news`) to its `Sitemap` class (e.g., `BlogSitemap` or`NewsSitemap`). It may also map to an *instance* of a `Sitemap` class (e.g., `BlogSitemap(some_var)`).

Sitemap Classes

A `Sitemap` class is a simple Python class that represents a "section" of entries in your sitemap. For example, one `Sitemap` class could represent all the entries of your weblog, while another could represent all of the events in your events calendar.

In the simplest case, all these sections get lumped together into one `sitemap.xml`, but it's also possible to use the framework to generate a sitemap index that references individual sitemap files, one per section (as described shortly).

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

For example, let's assume you have a blog system, with an `Entry` model, and you want your sitemap to include all the links to your individual blog entries. Here's how your `Sitemap` class might look:

```
from django.contrib.sitemaps import Sitemap

from mysite.blog.models import Entry



class BlogSitemap(Sitemap):

    changefreq = "never"

    priority = 0.5



    def items(self):
        return Entry.objects.filter(is_draft=False)


    def lastmod(self, obj):
        return obj.pub_date
```

Declaring a `Sitemap` should look very similar to declaring a `Feed`. That's by design.

Like `Feed` classes, `Sitemap` members can be either methods or attributes. See the steps in the earlier "A Complex Example" section for more about how this works.

A `Sitemap` class can define the following methods/attributes:

- `items` (**required**): Provides list of objects. The framework doesn't care what *type* of objects they are; all that matters is that these objects get passed to the `location()`, `lastmod()`, `changefreq()`, and`priority()` methods.
- `location` (optional): Gives the absolute URL for a given object. Here, "absolute URL" means a URL that doesn't include the protocol or domain. Here are some examples:
- Good: `'/foo/bar/'`
- Bad: `'example.com/foo/bar/'`
- Bad: `'http://example.com/foo/bar/'`

If `location` isn't provided, the framework will call the `get_absolute_url()` method on each object as returned by `items()`.

- `lastmod` (optional): The object's "last modification" date, as a Python `datetime` object.
- `changefreq` (optional): How often the object changes. Possible values (as given by the Sitemaps specification) are as follows:
- `'always'`
- `'hourly'`
- `'daily'`
- `'weekly'`
- `'monthly'`
- `'yearly'`
- `'never'`
- `priority` (optional): A suggested indexing priority between `0.0` and `1.0`. The default priority of a page is `0.5`; see the http://sitemaps.org/ documentation for more about how `priority` works.

Shortcuts

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

FlatPageSitemap

The `django.contrib.sitemaps.FlatPageSitemap` class looks at all flat pages defined for the current site and creates an entry in the sitemap. These entries include only the `location` attribute – not `lastmod`,`changefreq`, or `priority`.

See Chapter 16 for more about flat pages.

GenericSitemap

The `GenericSitemap` class works with any generic views (see Chapter 11) you already have.

To use it, create an instance, passing in the same `info_dict` you pass to the generic views. The only requirement is that the dictionary have a `queryset` entry. It may also have a `date_field` entry that specifies a date field for objects retrieved from the `queryset`. This will be used for the `lastmod` attribute in the generated sitemap. You may also pass `priority` and `changefreq` keyword arguments to the `GenericSitemap`constructor to specify these attributes for all URLs.

Here's an example of a URLconf using both `FlatPageSitemap` and `GenericSiteMap` (with the hypothetical`Entry` object from earlier):

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

from django.contrib.sitemaps import FlatPageSitemap,
GenericSitemap

from mysite.blog.models import Entry


info_dict = {

    'queryset': Entry.objects.all(),

    'date_field': 'pub_date',

}


sitemaps = {

    'flatpages': FlatPageSitemap,

    'blog': GenericSitemap(info_dict, priority=0.6),

}
```

```
urlpatterns = patterns('',
    # some generic view using info_dict
    # ...

    # the sitemap
    (r'^sitemap\.xml$',
     'django.contrib.sitemaps.views.sitemap',
     {'sitemaps': sitemaps})
)
```

Creating a Sitemap Index

The sitemap framework also has the ability to create a sitemap index that references individual sitemap files, one per each section defined in your `sitemaps` dictionary. The only differences in usage are as follows:

- You use two views in your URLconf: `django.contrib.sitemaps.views.index` and`django.contrib.sitemaps.views.sitemap`.
- The `django.contrib.sitemaps.views.sitemap` view should take a `section` keyword argument.

Here is what the relevant URLconf lines would look like for the previous example:

```
(r'^sitemap.xml$',
 'django.contrib.sitemaps.views.index',
 {'sitemaps': sitemaps}),


(r'^sitemap-(?P<section>.+).xml$',
 'django.contrib.sitemaps.views.sitemap',
 {'sitemaps': sitemaps})
```

This will automatically generate a `sitemap.xml` file that references both `sitemap-flatpages.xml` and`sitemap-blog.xml`. The `Sitemap` classes and the `sitemaps` dictionary don’t change at all.

Pinging Google

You may want to “ping” Google when your sitemap changes, to let it know to reindex your site. The framework provides a function to do just that: `django.contrib.sitemaps.ping_google()`.

`ping_google()` takes an optional argument, `sitemap_url`, which should be the absolute URL of your site’s sitemap (e.g., `'/sitemap.xml'`). If this argument isn’t provided, `ping_google()` will attempt to figure out your sitemap by performing a reverse lookup on your URLconf.

`ping_google()` raises the exception `django.contrib.sitemaps.SitemapNotFound` if it cannot determine your sitemap URL.

One useful way to call `ping_google()` is from a model’s `save()` method:

```
from django.contrib.sitemaps import ping_google


class Entry(models.Model):
    # ...
    def save(self, *args, **kwargs):
        super(Entry, self).save(*args, **kwargs)
        try:
            ping_google()
        except Exception:
            # Bare 'except' because we could get a
variety
            # of HTTP-related exceptions.
            pass
```

A more efficient solution, however, would be to call `ping_google()` from a `cron` script or some other scheduled task. The function makes an HTTP request to Google’s servers, so you may not want to introduce that network overhead each time you call `save()`.

Finally, if `'django.contrib.sitemaps'` is in your `INSTALLED_APPS`, then your `manage.py` will include a new command, `ping_google`. This is useful for command-line access to pinging. For example:

```
python manage.py ping_google /sitemap.xml
```