The syndication feed framework

This document is for Django's SVN release, which can be significantly different than previous releases. Get old docs here: 0.96, 0.95.

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

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.

Django also comes with a lower-level feed-generating API. Use this if you want to generate feeds outside of a Web context, or in some other lower-level way.

The high-level framework

Overview

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

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

Initialization

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

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

This 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 myproject.feeds import LatestEntries, LatestEntriesByCategory

feeds = {
    'latest': LatestEntries,
    'categories': LatestEntriesByCategory,
}

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

```
        {'feed_dict': feeds}),
    # ...
)
```

The above 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 just need to define the `Feed` classes themselves.

Feed classes

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 codebase.

A simple example

This simple example, taken from chicagocrime.org, describes a feed of the latest five news items:

```
from django.contrib.syndication.feeds import Feed
from chicagocrime.models import NewsItem

class LatestEntries(Feed):
    title = "Chicagocrime.org site news"
    link = "/sitenews/"
    description = "Updates on changes and additions to chicagocrime.org."

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

Note:

- The class subclasses `django.contrib.syndication.feeds.Feed`.
- `title`, `link` and `description` correspond to the standard RSS `<title>`, `<link>` and `<description>` elements, respectively.
- `items()` is, simply, a method that returns a list of objects that should be included in the feed as `<item>` elements. Although this example returns `NewsItem` objects using Django's object-relational mapper, `items()` doesn't have to return model instances. Although you get a few bits of functionality "for free" by using Django models, `items()` can return any type of object you want.
- If you're creating an Atom feed, rather than an RSS feed, set the `subtitle` attribute instead of the `description` attribute. See Publishing Atom and RSS feeds in tandem, later, for an example.

One thing's left to do. 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 the `.html` extension is required. The RSS system renders that template for each item, passing it two template context variables:
  - `{{ obj }}` — The current object (one of whichever objects you returned in `items()`).
  - `{{ site }}` — A `django.models.core.sites.Site` object representing the current site. This is useful for `{{ site.domain }}` or `{{ site.name }}`.

  If you don't create a template for either the title or description, the framework will use the template `"{{ obj }}"` by default — that is, the normal string representation of the object. You can also change the names of these two templates by specifying `title_template` and `description_template` as attributes of your `Feed` class.
- To specify the contents of `<link>`, you have two options. For each item in `items()`, Django first tries executing a `get_absolute_url()` method on that object. If that method doesn't exist, it tries calling a method `item_link()` in the `Feed` class, passing it a single parameter, `item`, which is the object itself. Both `get_absolute_url()` and `item_link()` should return the item's URL as a normal Python string.
- For the LatestEntries example above, we could have very simple feed templates:
  - latest_title.html:

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

  - latest_description.html:

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

A complex example

The framework also supports more complex feeds, via parameters.

For example, chicagocrime.org offers an RSS feed of recent crimes for every police beat in Chicago. It'd be silly to create a separate `Feed` class for each police beat; that would violate the DRY principle and would couple data to programming logic. Instead, the syndication framework lets you make generic feeds that output items based on information in the feed's URL.

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

- `/rss/beats/0613/` — Returns recent crimes for beat 0613.
- `/rss/beats/1424/` — Returns recent crimes for beat 1424.

The slug here is `"beats"`. The syndication framework sees the extra URL bits after the slug — `0613` and `1424` — and gives you a hook to tell it what those URL bits mean, and how they should influence which items get published in the feed.

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

```
class BeatFeed(Feed):
    def get_object(self, bits):
        # In case of "/rss/beats/0613/foo/bar/baz/", or other such clutter,
```

```
        # check that bits has only one member.
        if len(bits) != 1:
            raise ObjectDoesNotExist
        return Beat.objects.get(beat__exact=bits[0])

    def title(self, obj):
        return "Chicagocrime.org: Crimes for beat %s" % obj.beat

    def link(self, obj):
        return obj.get_absolute_url()

    def description(self, obj):
        return "Crimes recently reported in police beat %s" % obj.beat

    def items(self, obj):
        return Crime.objects.filter(beat__id__exact=obj.id).order_by('-crime_date')[:30]
```

Here's the basic algorithm the RSS framework follows, given this class and a request to the URL `/rss/beats/0613/`:

- The framework gets the URL `/rss/beats/0613/` and notices there's an extra bit of URL after the slug. It splits that remaining string by the slash character (`"/"`) and calls the `Feed` class' `get_object()` method, passing it the bits. In this case, bits is `['0613']`. For a request to `/rss/beats/0613/foo/bar/`, bits would be `['0613', 'foo', 'bar']`.
- `get_object()` is responsible for retrieving the given beat, from the given `bits`. In this case, it uses the Django database API to retrieve the beat. Note that `get_object()` should raise `django.core.exceptions.ObjectDoesNotExist` if given invalid parameters. There's no `try`/`except` around the `Beat.objects.get()` call, because it's not necessary; that function raises `Beat.DoesNotExist` on failure, and `Beat.DoesNotExist` is a subclass of `ObjectDoesNotExist`. Raising `ObjectDoesNotExist` in `get_object()` tells Django to produce a 404 error for that request.
- To generate the feed's `<title>`, `<link>` and `<description>`, Django uses the `title()`, `link()` and `description()` methods. In the previous example, they were simple string class attributes, but this example illustrates that they can be either strings *or* methods. For each of `title`, `link` and `description`, Django follows this algorithm:
  - First, it tries to call a method, passing the `obj` argument, where `obj` is the object returned by `get_object()`.
  - Failing that, it tries to call a method with no arguments.
  - Failing that, it uses the class attribute.
- 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()`, then finally an `items` class attribute (which should be a list).

The `ExampleFeed` class below gives full documentation on methods and attributes of `Feed` classes.

Specifying the type of feed

By default, feeds produced in this framework use RSS 2.0.

To change that, add a `feed_type` attribute to your `Feed` class, like so:

```
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:

- `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, such as those used in creating podcast feeds, use the `item_enclosure_url`, `item_enclosure_length` and `item_enclosure_mime_type` hooks. See the `ExampleFeed` class below for usage examples.

Language

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

URLs

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

Atom feeds require a `<link rel="self">` that defines the feed's current location. The syndication framework populates this automatically, using the domain of the current site according to the SITE_ID setting.

Publishing Atom and RSS feeds in tandem

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

Here's a full example:

```
from django.contrib.syndication.feeds import Feed
from chicagocrime.models import NewsItem
from django.utils.feedgenerator import Atom1Feed

class RssSiteNewsFeed(Feed):
    title = "Chicagocrime.org site news"
    link = "/sitenews/"
```

```
    description = "Updates on changes and additions to chicagocrime.org."

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

class AtomSiteNewsFeed(RssSiteNewsFeed):
    feed_type = Atom1Feed
    subtitle = RssSiteNewsFeed.description
```

Note

In this example, the RSS feed uses a `description` while the Atom feed uses a `subtitle`. That's because Atom feeds don't provide for a feed-level "description," but they *do* provide for a "subtitle."

If you provide a `description` in your `Feed` class, Django will *not* automatically put that into the `subtitle` element, because a subtitle and description are not necessarily the same thing. Instead, you should define a `subtitle` attribute.

In the above example, we simply set the Atom feed's `subtitle` to the RSS feed's `description`, because it's quite short already.

And the accompanying URLconf:

```
from django.conf.urls.defaults import *
from myproject.feeds import RssSiteNewsFeed, AtomSiteNewsFeed

feeds = {
    'rss': RssSiteNewsFeed,
    'atom': AtomSiteNewsFeed,
}

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

Feed class reference

This example illustrates all possible attributes and methods for a `Feed` class:

```
from django.contrib.syndication.feeds import Feed
from django.utils import feedgenerator

class ExampleFeed(Feed):

    # FEED TYPE -- Optional. This should be a class that subclasses
    # django.utils.feedgenerator.SyndicationFeed. This designates which
    # type of feed this should be: RSS 2.0, Atom 1.0, etc.
    # If you don't specify feed_type, your feed will be RSS 2.0.
```

```
    # This should be a class, not an instance of the class.

    feed_type = feedgenerator.Rss201rev2Feed

    # TEMPLATE NAMES -- Optional. These should be strings representing
    # names of Django templates that the system should use in rendering the
    # title and description of your feed items. Both are optional.
    # If you don't specify one, or either, Django will use the template
    # 'feeds/SLUG_title.html' and 'feeds/SLUG_description.html', where SLUG
    # is the slug you specify in the URL.

    title_template = None
    description_template = None

    # TITLE -- One of the following three is required. The framework looks
    # for them in this order.

    def title(self, obj):
        """
        Takes the object returned by get_object() and returns the feed's
        title as a normal Python string.
        """

    def title(self):
        """
        Returns the feed's title as a normal Python string.
        """

    title = 'foo' # Hard-coded title.

    # LINK -- One of the following three is required. The framework looks
    # for them in this order.

    def link(self, obj):
        """
        Takes the object returned by get_object() and returns the feed's
        link as a normal Python string.
        """

    def link(self):
        """
        Returns the feed's link as a normal Python string.
        """

    link = '/foo/bar/' # Hard-coded link.

    # DESCRIPTION -- One of the following three is required. The framework
    # looks for them in this order.

    def description(self, obj):
        """
        Takes the object returned by get_object() and returns the feed's
        description as a normal Python string.
```

```
    """

def description(self):
    """
    Returns the feed's description as a normal Python string.
    """

description = 'Foo bar baz.' # Hard-coded description.

# AUTHOR NAME --One of the following three is optional. The framework
# looks for them in this order.

def author_name(self, obj):
    """
    Takes the object returned by get_object() and returns the feed's
    author's name as a normal Python string.
    """

def author_name(self):
    """
    Returns the feed's author's name as a normal Python string.
    """

author_name = 'Sally Smith' # Hard-coded author name.

# AUTHOR E-MAIL --One of the following three is optional. The framework
# looks for them in this order.

def author_email(self, obj):
    """
    Takes the object returned by get_object() and returns the feed's
    author's e-mail as a normal Python string.
    """

def author_email(self):
    """
    Returns the feed's author's e-mail as a normal Python string.
    """

author_email = 'test@example.com' # Hard-coded author e-mail.

# AUTHOR LINK --One of the following three is optional. The framework
# looks for them in this order. In each case, the URL should include
# the "http://" and domain name.

def author_link(self, obj):
    """
    Takes the object returned by get_object() and returns the feed's
    author's URL as a normal Python string.
    """

def author_link(self):
    """
```

```
    Returns the feed's author's URL as a normal Python string.
    """

author_link = 'http://www.example.com/' # Hard-coded author URL.

# CATEGORIES -- One of the following three is optional. The framework
# looks for them in this order. In each case, the method/attribute
# should return an iterable object that returns strings.

def categories(self, obj):
    """
    Takes the object returned by get_object() and returns the feed's
    categories as iterable over strings.
    """

def categories(self):
    """
    Returns the feed's categories as iterable over strings.
    """

categories = ("python", "django") # Hard-coded list of categories.

# COPYRIGHT NOTICE -- One of the following three is optional. The
# framework looks for them in this order.

def copyright(self, obj):
    """
    Takes the object returned by get_object() and returns the feed's
    copyright notice as a normal Python string.
    """

def copyright(self):
    """
    Returns the feed's copyright notice as a normal Python string.
    """

copyright = 'Copyright (c) 2007, Sally Smith' # Hard-coded copyright notice.

# ITEMS -- One of the following three is required. The framework looks
# for them in this order.

def items(self, obj):
    """
    Takes the object returned by get_object() and returns a list of
    items to publish in this feed.
    """

def items(self):
    """
    Returns a list of items to publish in this feed.
    """

items = ('Item 1', 'Item 2') # Hard-coded items.
```

```
# GET_OBJECT -- This is required for feeds that publish different data
# for different URL parameters. (See "A complex example" above.)

def get_object(self, bits):
    """
    Takes a list of strings gleaned from the URL and returns an object
    represented by this feed. Raises
    django.core.exceptions.ObjectDoesNotExist on error.
    """

# ITEM LINK -- One of these three is required. The framework looks for
# them in this order.

# First, the framework tries the get_absolute_url() method on each item
# returned by items(). Failing that, it tries these two methods:

def item_link(self, item):
    """
    Takes an item, as returned by items(), and returns the item's URL.
    """

def item_link(self):
    """
    Returns the URL for every item in the feed.
    """

# ITEM AUTHOR NAME --One of the following three is optional. The
# framework looks for them in this order.

def item_author_name(self, item):
    """
    Takes an item, as returned by items(), and returns the item's
    author's name as a normal Python string.
    """

def item_author_name(self):
    """
    Returns the author name for every item in the feed.
    """

item_author_name = 'Sally Smith' # Hard-coded author name.

# ITEM AUTHOR E-MAIL --One of the following three is optional. The
# framework looks for them in this order.
#
# If you specify this, you must specify item_author_name.

def item_author_email(self, obj):
    """
    Takes an item, as returned by items(), and returns the item's
    author's e-mail as a normal Python string.
    """
```

```
def item_author_email(self):
    """
    Returns the author e-mail for every item in the feed.
    """

item_author_email = 'test@example.com' # Hard-coded author e-mail.

# ITEM AUTHOR LINK --One of the following three is optional. The
# framework looks for them in this order. In each case, the URL should
# include the "http://" and domain name.
#
# If you specify this, you must specify item_author_name.

def item_author_link(self, obj):
    """
    Takes an item, as returned by items(), and returns the item's
    author's URL as a normal Python string.
    """

def item_author_link(self):
    """
    Returns the author URL for every item in the feed.
    """

item_author_link = 'http://www.example.com/' # Hard-coded author URL.

# ITEM ENCLOSURE URL -- One of these three is required if you're
# publishing enclosures. The framework looks for them in this order.

def item_enclosure_url(self, item):
    """
    Takes an item, as returned by items(), and returns the item's
    enclosure URL.
    """

def item_enclosure_url(self):
    """
    Returns the enclosure URL for every item in the feed.
    """

item_enclosure_url = "/foo/bar.mp3" # Hard-coded enclosure link.

# ITEM ENCLOSURE LENGTH -- One of these three is required if you're
# publishing enclosures. The framework looks for them in this order.
# In each case, the returned value should be either an integer, or a
# string representation of the integer, in bytes.

def item_enclosure_length(self, item):
    """
    Takes an item, as returned by items(), and returns the item's
    enclosure length.
    """
```

```
def item_enclosure_length(self):
    """
    Returns the enclosure length for every item in the feed.
    """

item_enclosure_length = 32000 # Hard-coded enclosure length.

# ITEM ENCLOSURE MIME TYPE -- One of these three is required if you're
# publishing enclosures. The framework looks for them in this order.

def item_enclosure_mime_type(self, item):
    """
    Takes an item, as returned by items(), and returns the item's
    enclosure mime type.
    """

def item_enclosure_mime_type(self):
    """
    Returns the enclosure length, in bytes, for every item in the feed.
    """

item_enclosure_mime_type = "audio/mpeg" # Hard-coded enclosure mime-type.

# ITEM PUBDATE -- It's optional to use one of these three. This is a
# hook that specifies how to get the pubdate for a given item.
# In each case, the method/attribute should return a Python
# datetime.datetime object.

def item_pubdate(self, item):
    """
    Takes an item, as returned by items(), and returns the item's
    pubdate.
    """

def item_pubdate(self):
    """
    Returns the pubdate for every item in the feed.
    """

item_pubdate = datetime.datetime(2005, 5, 3) # Hard-coded pubdate.

# ITEM CATEGORIES -- It's optional to use one of these three. This is
# a hook that specifies how to get the list of categories for a given
# item. In each case, the method/attribute should return an iterable
# object that returns strings.

def item_categories(self, item):
    """
    Takes an item, as returned by items(), and returns the item's
    categories.
    """
```

```
    def item_categories(self):
        """
        Returns the categories for every item in the feed.
        """

    item_categories = ("python", "django") # Hard-coded categories.

    # ITEM COPYRIGHT NOTICE (only applicable to Atom feeds) -- One of the
    # following three is optional. The framework looks for them in this
    # order.

    def item_copyright(self, obj):
        """
        Takes an item, as returned by items(), and returns the item's
        copyright notice as a normal Python string.
        """

    def item_copyright(self):
        """
        Returns the copyright notice for every item in the feed.
        """

    item_copyright = 'Copyright (c) 2007, Sally Smith' # Hard-coded copyright notice.
```

The low-level framework

Behind the scenes, the high-level RSS framework uses a lower-level framework for generating feeds' XML. This framework lives in a single module: django/utils/feedgenerator.py.

Feel free to use this framework on your own, for lower-level tasks.

The `feedgenerator` module contains a base class `SyndicationFeed` and several subclasses:

- `RssUserland091Feed`
- `Rss201rev2Feed`
- `Atom1Feed`

Each of these three classes knows how to render a certain type of feed as XML. They share this interface:

`__init__(title, link, description, language=None, author_email=None, author_name=None, author_link=None, subtitle=None, categories=None, feed_url=None)`

Initializes the feed with the given metadata, which applies to the entire feed (i.e., not just to a specific item in the feed).

All parameters, if given, should be Unicode objects, except `categories`, which should be a sequence of Unicode objects.

`add_item(title, link, description, author_email=None, author_name=None, pubdate=None, comments=None, unique_id=None, enclosure=None, categories=())`

Add an item to the feed with the given parameters. All parameters, if given, should be

Unicode objects, except:

- `pubdate` should be a Python datetime object.
- `enclosure` should be an instance of `feedgenerator.Enclosure`.
- `categories` should be a sequence of Unicode objects.

`write(outfile, encoding)`

Outputs the feed in the given encoding to outfile, which is a file-like object.

`writeString(encoding)`

Returns the feed as a string in the given encoding.

Example usage

This example creates an Atom 1.0 feed and prints it to standard output:

```
>>> from django.utils import feedgenerator
>>> f = feedgenerator.Atom1Feed(
...     title=u"My Weblog",
...     link=u"http://www.example.com/",
...     description=u"In which I write about what I ate today.",
...     language=u"en")
>>> f.add_item(title=u"Hot dog today",
...     link=u"http://www.example.com/entries/1/",
...     description=u"<p>Today I had a Vienna Beef hot dog. It was pink, plump and perfect.</p>")
>>> print f.writeString('utf8')
<?xml version="1.0" encoding="utf8"?>
<feed xmlns="http://www.w3.org/2005/Atom" xml:lang="en"><title>My Weblog</title>
<link href="http://www.example.com/"></link><id>http://www.example.com/</id>
<updated>Sat, 12 Nov 2005 00:28:43 -0000</updated><entry><title>Hot dog today</title>
<link>http://www.example.com/entries/1/</link><id>tag:www.example.com/entries/1/</id>
<summary type="html">&lt;p&gt;Today I had a Vienna Beef hot dog. It was pink, plump and perfect.&lt;/p&gt;</summary>
</entry></feed>
```

© 2005-2007 Lawrence Journal-World unless otherwise noted. Django is a registered trademark of Lawrence Journal-World.