Writing your first Django app, part 3

This tutorial begins where Tutorial 2 left off. We're continuing the Web-poll application and will focus on creating the public interface – "views."

Philosophy

A view is a "type" of Web page in your Django application that generally serves a specific function and has a specific template. For example, in a Weblog application, you might have the following views:

- Blog homepage – displays the latest few entries.
- Entry "detail" page – permalink page for a single entry.
- Year-based archive page – displays all months with entries in the given year.
- Month-based archive page – displays all days with entries in the given month.
- Day-based archive page – displays all entries in the given day.
- Comment action – handles posting comments to a given entry.

In our poll application, we'll have the following four views:

- Poll "index" page – displays the latest few polls.
- Poll "detail" page – displays a poll question, with no results but with a form to vote.
- Poll "results" page – displays results for a particular poll.
- Vote action – handles voting for a particular choice in a particular poll.

In Django, each view is represented by a simple Python function.

Design your URLs

The first step of writing views is to design your URL structure. You do this by creating a Python module, called a URLconf. URLconfs are how Django associates a given URL with given Python code.

When a user requests a Django-powered page, the system looks at the `ROOT_URLCONF` setting, which contains a string in Python dotted syntax. Django loads that module and looks for a module-level variable called `urlpatterns`, which is a sequence of tuples in the following format:

```
(regular expression, Python callback function [, optional dictionary])
```

Django starts at the first regular expression and makes its way down the list, comparing the requested URL against each regular expression until it finds one that matches.

When it finds a match, Django calls the Python callback function, with an `HttpRequest` object as the first argument, any "captured" values from the regular expression as keyword arguments, and, optionally, arbitrary keyword arguments from the dictionary (an optional third item in the tuple).

For more on `HttpRequest` objects, see the Request and response objects. For more details on URLconfs, see the URL dispatcher.

When you ran `django-admin.py startproject mysite` at the beginning of Tutorial 1, it created a default URLconf in `mysite/urls.py`. It also automatically set your `ROOT_URLCONF`

setting (in `settings.py`) to point at that file:

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

Time for an example. Edit `mysite/urls.py` so it looks like this:

```
from django.conf.urls import patterns, include, url

from django.contrib import admin
admin.autodiscover()

urlpatterns = patterns('',
    url(r'^polls/$', 'polls.views.index'),
    url(r'^polls/(?P<poll_id>\d+)/$', 'polls.views.detail'),
    url(r'^polls/(?P<poll_id>\d+)/results/$', 'polls.views.results'),
    url(r'^polls/(?P<poll_id>\d+)/vote/$', 'polls.views.vote'),
    url(r'^admin/', include(admin.site.urls)),
)
```

This is worth a review. When somebody requests a page from your Web site -- say, "/polls/23/", Django will load this Python module, because it's pointed to by the `ROOT_URLCONF` setting. It finds the variable named `urlpatterns` and traverses the regular expressions in order. When it finds a regular expression that matches -- `r'^polls/(?P<poll_id>\d+)/$'` -- it loads the function `detail()` from `polls/views.py`. Finally, it calls that `detail()` function like so:

```
detail(request=<HttpRequest object>, poll_id='23')
```

The `poll_id='23'` part comes from `(?P<poll_id>\d+)`. Using parentheses around a pattern "captures" the text matched by that pattern and sends it as an argument to the view function; the `?P<poll_id>` defines the name that will be used to identify the matched pattern; and `\d+` is a regular expression to match a sequence of digits (i.e., a number).

Because the URL patterns are regular expressions, there really is no limit on what you can do with them. And there's no need to add URL cruft such as `.php` -- unless you have a sick sense of humor, in which case you can do something like this:

```
(r'^polls/latest\.php$', 'polls.views.index'),
```

But, don't do that. It's silly.

Note that these regular expressions do not search GET and POST parameters, or the domain name. For example, in a request to `http://www.example.com/myapp/`, the URLconf will look for `myapp/`. In a request to `http://www.example.com/myapp/?page=3`, the URLconf will look for `myapp/`.

If you need help with regular expressions, see Wikipedia's entry and the documentation of the `re` module. Also, the O'Reilly book "Mastering Regular Expressions" by Jeffrey Friedl is fantastic.

Finally, a performance note: these regular expressions are compiled the first time the URLconf module is loaded. They're super fast.

Write your first view

Well, we haven't created any views yet -- we just have the URLconf. But let's make sure

Django is following the URLconf properly.

Fire up the Django development Web server:

```
python manage.py runserver
```

Now go to "http://localhost:8000/polls/" on your domain in your Web browser. You should get a pleasantly-colored error page with the following message:

```
ViewDoesNotExist at /polls/

Could not import polls.views.index. View does not exist in module polls.views.
```

This error happened because you haven't written a function `index()` in the module `polls/views.py`.

Try "/polls/23/", "/polls/23/results/" and "/polls/23/vote/". The error messages tell you which view Django tried (and failed to find, because you haven't written any views yet).

Time to write the first view. Open the file `polls/views.py` and put the following Python code in it:

```
from django.http import HttpResponse

def index(request):
    return HttpResponse("Hello, world. You're at the poll index.")
```

This is the simplest view possible. Go to "/polls/" in your browser, and you should see your text.

Now lets add a few more views. These views are slightly different, because they take an argument (which, remember, is passed in from whatever was captured by the regular expression in the URLconf):

```
def detail(request, poll_id):
    return HttpResponse("You're looking at poll %s." % poll_id)

def results(request, poll_id):
    return HttpResponse("You're looking at the results of poll %s." % poll_id)

def vote(request, poll_id):
    return HttpResponse("You're voting on poll %s." % poll_id)
```

Take a look in your browser, at "/polls/34/". It'll run the *detail()* method and display whatever ID you provide in the URL. Try "/polls/34/results/" and "/polls/34/vote/" too -- these will display the placeholder results and voting pages.

Write views that actually do something

Each view is responsible for doing one of two things: Returning an `HttpResponse` object containing the content for the requested page, or raising an exception such as `Http404`. The rest is up to you.

Your view can read records from a database, or not. It can use a template system such as Django's -- or a third-party Python template system -- or not. It can generate a PDF file, output XML, create a ZIP file on the fly, anything you want, using whatever Python libraries you want.

All Django wants is that `HttpResponse`. Or an exception.

Because it's convenient, let's use Django's own database API, which we covered in Tutorial 1. Here's one stab at the `index()` view, which displays the latest 5 poll questions in the system, separated by commas, according to publication date:

```
from polls.models import Poll
from django.http import HttpResponse

def index(request):
    latest_poll_list = Poll.objects.all().order_by('-pub_date')[:5]
    output = ', '.join([p.question for p in latest_poll_list])
    return HttpResponse(output)
```

There's a problem here, though: The page's design is hard-coded in the view. If you want to change the way the page looks, you'll have to edit this Python code. So let's use Django's template system to separate the design from Python:

```
from django.template import Context, loader
from polls.models import Poll
from django.http import HttpResponse

def index(request):
    latest_poll_list = Poll.objects.all().order_by('-pub_date')[:5]
    t = loader.get_template('polls/index.html')
    c = Context({
        'latest_poll_list': latest_poll_list,
    })
    return HttpResponse(t.render(c))
```

That code loads the template called "polls/index.html" and passes it a context. The context is a dictionary mapping template variable names to Python objects.

Reload the page. Now you'll see an error:

```
TemplateDoesNotExist at /polls/
polls/index.html
```

Ah. There's no template yet. First, create a directory, somewhere on your filesystem, whose contents Django can access. (Django runs as whatever user your server runs.) Don't put them under your document root, though. You probably shouldn't make them public, just for security's sake. Then edit `TEMPLATE_DIRS` in your `settings.py` to tell Django where it can find templates -- just as you did in the "Customize the admin look and feel" section of Tutorial 2.

When you've done that, create a directory `polls` in your template directory. Within that, create a file called `index.html`. Note that our `loader.get_template('polls/index.html')` code from above maps to "[template_directory]/polls/index.html" on the filesystem.

Put the following code in that template:

```
{% if latest_poll_list %}
    <ul>
    {% for poll in latest_poll_list %}
        <li><a href="/polls/{{ poll.id }}/">{{ poll.question }}</a></li>
    {% endfor %}
    </ul>
```

```
{% else %}
    <p>No polls are available.</p>
{% endif %}
```

Load the page in your Web browser, and you should see a bulleted-list containing the "What's up" poll from Tutorial 1. The link points to the poll's detail page.

A shortcut: render_to_response()

It's a very common idiom to load a template, fill a context and return an `HttpResponse` object with the result of the rendered template. Django provides a shortcut. Here's the full `index()` view, rewritten:

```
from django.shortcuts import render_to_response
from polls.models import Poll

def index(request):
    latest_poll_list = Poll.objects.all().order_by('-pub_date')[:5]
    return render_to_response('polls/index.html', {'latest_poll_list': latest_poll_lis
```

Note that once we've done this in all these views, we no longer need to import `loader`, `Context` and `HttpResponse`.

The `render_to_response()` function takes a template name as its first argument and a dictionary as its optional second argument. It returns an `HttpResponse` object of the given template rendered with the given context.

Raising 404

Now, let's tackle the poll detail view -- the page that displays the question for a given poll. Here's the view:

```
from django.http import Http404
# ...
def detail(request, poll_id):
    try:
        p = Poll.objects.get(pk=poll_id)
    except Poll.DoesNotExist:
        raise Http404
    return render_to_response('polls/detail.html', {'poll': p})
```

The new concept here: The view raises the `Http404` exception if a poll with the requested ID doesn't exist.

We'll discuss what you could put in that `polls/detail.html` template a bit later, but if you'd like to quickly get the above example working, just:

```
{{ poll }}
```

will get you started for now.

A shortcut: get_object_or_404()

It's a very common idiom to use `get()` and raise `Http404` if the object doesn't exist. Django provides a shortcut. Here's the `detail()` view, rewritten:

```
from django.shortcuts import render_to_response, get_object_or_404
# ...
def detail(request, poll_id):
    p = get_object_or_404(Poll, pk=poll_id)
    return render_to_response('polls/detail.html', {'poll': p})
```

The `get_object_or_404()` function takes a Django model as its first argument and an arbitrary number of keyword arguments, which it passes to the `get()` function of the model's manager. It raises `Http404` if the object doesn't exist.

> **Philosophy**
>
> Why do we use a helper function `get_object_or_404()` instead of automatically catching the `ObjectDoesNotExist` exceptions at a higher level, or having the model API raise `Http404` instead of `ObjectDoesNotExist`?
>
> Because that would couple the model layer to the view layer. One of the foremost design goals of Django is to maintain loose coupling.

There's also a `get_list_or_404()` function, which works just as `get_object_or_404()` -- except using `filter()` instead of `get()`. It raises `Http404` if the list is empty.

Write a 404 (page not found) view

When you raise `Http404` from within a view, Django will load a special view devoted to handling 404 errors. It finds it by looking for the variable `handler404` in your root URLconf (and only in your root URLconf; setting `handler404` anywhere else will have no effect), which is a string in Python dotted syntax -- the same format the normal URLconf callbacks use. A 404 view itself has nothing special: It's just a normal view.

You normally won't have to bother with writing 404 views. If you don't set `handler404`, the built-in view `django.views.defaults.page_not_found()` is used by default. Optionally, you can create a `404.html` template in the root of your template directory. The default 404 view will then use that template for all 404 errors when `DEBUG` is set to `False` (in your settings module).

A couple more things to note about 404 views:

- If `DEBUG` is set to `True` (in your settings module) then your 404 view will never be used (and thus the `404.html` template will never be rendered) because the traceback will be displayed instead.
- The 404 view is also called if Django doesn't find a match after checking every regular expression in the URLconf.

Write a 500 (server error) view

Similarly, your root URLconf may define a `handler500`, which points to a view to call in case of server errors. Server errors happen when you have runtime errors in view code.

Use the template system

Back to the `detail()` view for our poll application. Given the context variable `poll`, here's what the "polls/detail.html" template might look like:

```
<h1>{{ poll.question }}</h1>
<ul>
{% for choice in poll.choice_set.all %}
    <li>{{ choice.choice_text }}</li>
{% endfor %}
</ul>
```

The template system uses dot-lookup syntax to access variable attributes. In the example of `{{ poll.question }}`, first Django does a dictionary lookup on the object `poll`. Failing that, it tries an attribute lookup -- which works, in this case. If attribute lookup had failed, it would've tried a list-index lookup.

Method-calling happens in the `{% for %}` loop: `poll.choice_set.all` is interpreted as the Python code `poll.choice_set.all()`, which returns an iterable of `Choice` objects and is suitable for use in the `{% for %}` tag.

See the template guide for more about templates.

Simplifying the URLconfs

Take some time to play around with the views and template system. As you edit the URLconf, you may notice there's a fair bit of redundancy in it:

```
urlpatterns = patterns('',
    url(r'^polls/$', 'polls.views.index'),
    url(r'^polls/(?P<poll_id>\d+)/$', 'polls.views.detail'),
    url(r'^polls/(?P<poll_id>\d+)/results/$', 'polls.views.results'),
    url(r'^polls/(?P<poll_id>\d+)/vote/$', 'polls.views.vote'),
)
```

Namely, `polls.views` is in every callback.

Because this is a common case, the URLconf framework provides a shortcut for common prefixes. You can factor out the common prefixes and add them as the first argument to `patterns()`, like so:

```
urlpatterns = patterns('polls.views',
    url(r'^polls/$', 'index'),
    url(r'^polls/(?P<poll_id>\d+)/$', 'detail'),
    url(r'^polls/(?P<poll_id>\d+)/results/$', 'results'),
    url(r'^polls/(?P<poll_id>\d+)/vote/$', 'vote'),
)
```

This is functionally identical to the previous formatting. It's just a bit tidier.

Since you generally don't want the prefix for one app to be applied to every callback in your URLconf, you can concatenate multiple `patterns()`. Your full `mysite/urls.py` might now look like this:

```
from django.conf.urls import patterns, include, url

from django.contrib import admin
admin.autodiscover()

urlpatterns = patterns('polls.views',
    url(r'^polls/$', 'index'),
```

```
    url(r'^polls/(?P<poll_id>\d+)/$', 'detail'),
    url(r'^polls/(?P<poll_id>\d+)/results/$', 'results'),
    url(r'^polls/(?P<poll_id>\d+)/vote/$', 'vote'),
)

urlpatterns += patterns('',
    url(r'^admin/', include(admin.site.urls)),
)
```

Decoupling the URLconfs

While we're at it, we should take the time to decouple our poll-app URLs from our Django project configuration. Django apps are meant to be pluggable -- that is, each particular app should be transferable to another Django installation with minimal fuss.

Our poll app is pretty decoupled at this point, thanks to the strict directory structure that `python manage.py startapp` created, but one part of it is coupled to the Django settings: The URLconf.

We've been editing the URLs in `mysite/urls.py`, but the URL design of an app is specific to the app, not to the Django installation -- so let's move the URLs within the app directory.

Copy the file `mysite/urls.py` to `polls/urls.py`. Then, change `mysite/urls.py` to remove the poll-specific URLs and insert an `include()`, leaving you with:

```
from django.conf.urls import patterns, include, url

from django.contrib import admin
admin.autodiscover()

urlpatterns = patterns('',
    url(r'^polls/', include('polls.urls')),
    url(r'^admin/', include(admin.site.urls)),
)
```

`include()` simply references another URLconf. Note that the regular expression doesn't have a `$` (end-of-string match character) but has the 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.

Here's what happens if a user goes to "/polls/34/" in this system:

- Django will find the match at `'^polls/'`
- Then, Django will strip off the matching text (`"polls/"`) and send the remaining text -- `"34/"` -- to the 'polls.urls' URLconf for further processing.

Now that we've decoupled that, we need to decouple the `polls.urls` URLconf by removing the leading "polls/" from each line, removing the lines registering the admin site, and removing the `include` import which is no longer used. Your `polls/urls.py` file should now look like this:

```
from django.conf.urls import patterns, url

urlpatterns = patterns('polls.views',
    url(r'^$', 'index'),
    url(r'^(?P<poll_id>\d+)/$', 'detail'),
```

```
    url(r'^(?P<poll_id>\d+)/results/$', 'results'),
    url(r'^(?P<poll_id>\d+)/vote/$', 'vote'),
)
```

The idea behind `include()` and URLconf decoupling is to make it easy to plug-and-play URLs. Now that polls are in their own URLconf, they can be placed under "/polls/", or under "/fun_polls/", or under "/content/polls/", or any other path root, and the app will still work.

All the poll app cares about is its relative path, not its absolute path.

Removing hardcoded URLs in templates

Remember, when we wrote the link to a poll in our template, the link was partially hardcoded like this:

```
<li><a href="/polls/{{ poll.id }}/">{{ poll.question }}</a></li>
```

To use the decoupled URLs we've just introduced, replace the hardcoded link with the `url` template tag:

```
<li><a href="{% url 'polls.views.detail' poll.id %}">{{ poll.question }}</a></li>
```

> **Note**
>
> If `{% url 'polls.views.detail' poll.id %}` (with quotes) doesn't work, but `{% url polls.views.detail poll.id %}` (without quotes) does, that means you're using a version of Django < 1.5. In this case, add the following declaration at the top of your template:
>
> ```
> {% load url from future %}
> ```

When you're comfortable with writing views, read part 4 of this tutorial to learn about simple form processing and generic views.