Writing your first Django app, part 3**¶**

Part-3

Overview**¶**

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 blog 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:

* Question “index” page – displays the latest few questions.
* Question “detail” page – displays a question text, with no results but with a form to vote.
* Question “results” page – displays results for a particular question.
* Vote action – handles voting for a particular choice in a particular question.

In Django, web pages and other content are delivered by views. Each view is represented by a simple Python function (or method, in the case of class-based views). Django will choose a view by examining the URL that’s requested (to be precise, the part of the URL after the domain name).

Now in your time on the web you may have come across such beauties as “ME2/Sites/dirmod.asp?sid=&type=gen&mod=Core+Pages&gid=A6CD4967199A42D9B65B1B”. You will be pleased to know that Django allows us much more elegant *URL patterns* than that.

A URL pattern is simply the general form of a URL - for example: /newsarchive/<year>/<month>/.

To get from a URL to a view, Django uses what are known as ‘URLconfs’. A URLconf maps URL patterns to views.

This tutorial provides basic instruction in the use of URLconfs, and you can refer to URL dispatcher for more information.

Writing more views**¶**

Now let’s add a few more views to polls/views.py. These views are slightly different, because they take an argument:

polls/views.py

**def** detail(request, question\_id):

**return** HttpResponse("You're looking at question **%s**." % question\_id)

**def** results(request, question\_id):

response = "You're looking at the results of question **%s**."

**return** HttpResponse(response % question\_id)

**def** vote(request, question\_id):

**return** HttpResponse("You're voting on question **%s**." % question\_id)

Wire these new views into the polls.urls module by adding the following path() calls:

polls/urls.py

**from** **django.urls** **import** path

**from** **.** **import** views

urlpatterns = [

*# ex: /polls/*

path('', views.index, name='index'),

*# ex: /polls/5/*

path('<int:question\_id>/', views.detail, name='detail'),

*# ex: /polls/5/results/*

path('<int:question\_id>/results/', views.results, name='results'),

*# ex: /polls/5/vote/*

path('<int:question\_id>/vote/', views.vote, name='vote'),

]

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.

When somebody requests a page from your website – say, “/polls/34/”, Django will load the mysite.urls Python module because it’s pointed to by the ROOT\_URLCONF setting. It finds the variable named urlpatterns and traverses the patterns in order. After finding the match at 'polls/', it strips off the matching text ("polls/") and sends the remaining text – "34/" – to the ‘polls.urls’ URLconf for further processing. There it matches '<int:question\_id>/', resulting in a call to the detail() view like so:

detail(request=<HttpRequest object>, question\_id=34)

The question\_id=34 part comes from <int:question\_id>. Using angle brackets “captures” part of the URL and sends it as a keyword argument to the view function. The :question\_id> part of the string defines the name that will be used to identify the matched pattern, and the <int: part is a converter that determines what patterns should match this part of the URL path.

There’s no need to add URL cruft such as .html – unless you want to, in which case you can do something like this:

path('polls/latest.html', views.index),

But, don’t do that. It’s silly.

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 2. Here’s one stab at a new index() view, which displays the latest 5 poll questions in the system, separated by commas, according to publication date:

polls/views.py

**from** **django.http** **import** HttpResponse

**from** **.models** **import** Question

**def** index(request):

latest\_question\_list = Question.objects.order\_by('-pub\_date')[:5]

output = ', '.join([q.question\_text **for** q **in** latest\_question\_list])

**return** HttpResponse(output)

*# Leave the rest of the views (detail, results, vote) unchanged*

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 by creating a template that the view can use.

First, create a directory called templates in your polls directory. Django will look for templates in there.

Your project’s TEMPLATES setting describes how Django will load and render templates. The default settings file configures a DjangoTemplates backend whose APP\_DIRS option is set to True. By convention DjangoTemplates looks for a “templates” subdirectory in each of the INSTALLED\_APPS.

Within the templates directory you have just created, create another directory called polls, and within that create a file called index.html. In other words, your template should be at polls/templates/polls/index.html. Because of how the app\_directories template loader works as described above, you can refer to this template within Django simply as polls/index.html.

Template namespacing

Now we *might* be able to get away with putting our templates directly in polls/templates (rather than creating another polls subdirectory), but it would actually be a bad idea. Django will choose the first template it finds whose name matches, and if you had a template with the same name in a *different* application, Django would be unable to distinguish between them. We need to be able to point Django at the right one, and the easiest way to ensure this is by *namespacing* them. That is, by putting those templates inside *another* directory named for the application itself.

Put the following code in that template:

polls/templates/polls/index.html

{% **if** latest\_question\_list %}

<**ul**>

{% **for** question **in** latest\_question\_list %}

<**li**><**a** href="/polls/{{ question.id }}/">{{ question.question\_text }}</**a**></**li**>

{% **endfor** %}

</**ul**>

{% **else** %}

<**p**>No polls are available.</**p**>

{% **endif** %}

Now let’s update our index view in polls/views.py to use the template:

polls/views.py

**from** **django.http** **import** HttpResponse

**from** **django.template** **import** loader

**from** **.models** **import** Question

**def** index(request):

latest\_question\_list = Question.objects.order\_by('-pub\_date')[:5]

template = loader.get\_template('polls/index.html')

context = {

'latest\_question\_list': latest\_question\_list,

}

**return** HttpResponse(template.render(context, request))

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.

Load the page by pointing your browser at “/polls/”, and you should see a bulleted-list containing the “What’s up” question from Tutorial 2. The link points to the question’s detail page.

**A shortcut: render()¶**

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:

polls/views.py

**from** **django.shortcuts** **import** render

**from** **.models** **import** Question

**def** index(request):

latest\_question\_list = Question.objects.order\_by('-pub\_date')[:5]

context = {'latest\_question\_list': latest\_question\_list}

**return** render(request, 'polls/index.html', context)

Note that once we’ve done this in all these views, we no longer need to import loader and HttpResponse (you’ll want to keep HttpResponse if you still have the stub methods for detail, results, and vote).

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

Raising a 404 error**¶**

Now, let’s tackle the question detail view – the page that displays the question text for a given poll. Here’s the view:

polls/views.py

**from** **django.http** **import** Http404

**from** **django.shortcuts** **import** render

**from** **.models** **import** Question

*# ...*

**def** detail(request, question\_id):

**try**:

question = Question.objects.get(pk=question\_id)

**except** Question.DoesNotExist:

**raise** Http404("Question does not exist")

**return** render(request, 'polls/detail.html', {'question': question})

The new concept here: The view raises the Http404 exception if a question 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, a file containing just:

polls/templates/polls/detail.html

{{ question }}

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:

polls/views.py

**from** **django.shortcuts** **import** get\_object\_or\_404, render

**from** **.models** **import** Question

*# ...*

**def** detail(request, question\_id):

question = get\_object\_or\_404(Question, pk=question\_id)

**return** render(request, 'polls/detail.html', {'question': question})

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. Some controlled coupling is introduced in the django.shortcuts module.

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.

Use the template system**¶**

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

polls/templates/polls/detail.html

<**h1**>{{ question.question\_text }}</**h1**>

<**ul**>

{% **for** choice **in** question.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 {{ question.question\_text }}, first Django does a dictionary lookup on the object question. 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: question.choice\_set.all is interpreted as the Python code question.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.

Removing hardcoded URLs in templates**¶**

Remember, when we wrote the link to a question in the polls/index.html template, the link was partially hardcoded like this:

<**li**><**a** href="/polls/{{ question.id }}/">{{ question.question\_text }}</**a**></**li**>

The problem with this hardcoded, tightly-coupled approach is that it becomes challenging to change URLs on projects with a lot of templates. However, since you defined the name argument in the path() functions in the polls.urls module, you can remove a reliance on specific URL paths defined in your url configurations by using the {% url %} template tag:

<**li**><**a** href="{% **url** 'detail' question.id %}">{{ question.question\_text }}</**a**></**li**>

The way this works is by looking up the URL definition as specified in the polls.urls module. You can see exactly where the URL name of ‘detail’ is defined below:

...

*# the 'name' value as called by the {% url %} template tag*

path('<int:question\_id>/', views.detail, name='detail'),

...

If you want to change the URL of the polls detail view to something else, perhaps to something like polls/specifics/12/ instead of doing it in the template (or templates) you would change it in polls/urls.py:

...

*# added the word 'specifics'*

path('specifics/<int:question\_id>/', views.detail, name='detail'),

...

Namespacing URL names**¶**

The tutorial project has just one app, polls. In real Django projects, there might be five, ten, twenty apps or more. How does Django differentiate the URL names between them? For example, the polls app has a detail view, and so might an app on the same project that is for a blog. How does one make it so that Django knows which app view to create for a url when using the {% url %} template tag?

The answer is to add namespaces to your URLconf. In the polls/urls.py file, go ahead and add an app\_name to set the application namespace:

polls/urls.py

**from** **django.urls** **import** path

**from** **.** **import** views

app\_name = 'polls'

urlpatterns = [

path('', views.index, name='index'),

path('<int:question\_id>/', views.detail, name='detail'),

path('<int:question\_id>/results/', views.results, name='results'),

path('<int:question\_id>/vote/', views.vote, name='vote'),

]

Now change your polls/index.html template from:

polls/templates/polls/index.html

<**li**><**a** href="{% **url** 'detail' question.id %}">{{ question.question\_text }}</**a**></**li**>

to point at the namespaced detail view:

polls/templates/polls/index.html

<**li**><**a** href="{% **url** 'polls:detail' question.id %}">{{ question.question\_text }}</**a**></**li**>

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