Part-4

Write a simple form**¶**

Let’s update our poll detail template (“polls/detail.html”) from the last tutorial, so that the template contains an HTML <form> element:

polls/templates/polls/detail.html

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

{% **if** error\_message %}<**p**><**strong**>{{ error\_message }}</**strong**></**p**>{% **endif** %}

<**form** action="{% **url** 'polls:vote' question.id %}" method="post">

{% **csrf\_token** %}

{% **for** choice **in** question.choice\_set.all %}

<**input** type="radio" name="choice" id="choice{{ forloop.counter }}" value="{{ choice.id }}" />

<**label** for="choice{{ forloop.counter }}">{{ choice.choice\_text }}</**label**><**br** />

{% **endfor** %}

<**input** type="submit" value="Vote" />

</**form**>

A quick rundown:

* The above template displays a radio button for each question choice. The value of each radio button is the associated question choice’s ID. The name of each radio button is "choice". That means, when somebody selects one of the radio buttons and submits the form, it’ll send the POST data choice=# where # is the ID of the selected choice. This is the basic concept of HTML forms.
* We set the form’s action to {% url 'polls:vote' question.id %}, and we set method="post". Using method="post" (as opposed to method="get") is very important, because the act of submitting this form will alter data server-side. Whenever you create a form that alters data server-side, use method="post". This tip isn’t specific to Django; it’s just good Web development practice.
* forloop.counter indicates how many times the for tag has gone through its loop
* Since we’re creating a POST form (which can have the effect of modifying data), we need to worry about Cross Site Request Forgeries. Thankfully, you don’t have to worry too hard, because Django comes with a very easy-to-use system for protecting against it. In short, all POST forms that are targeted at internal URLs should use the {% csrf\_token %} template tag.

Now, let’s create a Django view that handles the submitted data and does something with it. Remember, in Tutorial 3, we created a URLconf for the polls application that includes this line:

polls/urls.py

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

We also created a dummy implementation of the vote() function. Let’s create a real version. Add the following to polls/views.py:

polls/views.py

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

**from** **django.http** **import** HttpResponseRedirect, HttpResponse

**from** **django.urls** **import** reverse

**from** **.models** **import** Choice, Question

*# ...*

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

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

**try**:

selected\_choice = question.choice\_set.get(pk=request.POST['choice'])

**except** (**KeyError**, Choice.DoesNotExist):

*# Redisplay the question voting form.*

**return** render(request, 'polls/detail.html', {

'question': question,

'error\_message': "You didn't select a choice.",

})

**else**:

selected\_choice.votes += 1

selected\_choice.save()

*# Always return an HttpResponseRedirect after successfully dealing*

*# with POST data. This prevents data from being posted twice if a*

*# user hits the Back button.*

**return** HttpResponseRedirect(reverse('polls:results', args=(question.id,)))

This code includes a few things we haven’t covered yet in this tutorial:

* request.POST is a dictionary-like object that lets you access submitted data by key name. In this case, request.POST['choice'] returns the ID of the selected choice, as a string. request.POST values are always strings.

Note that Django also provides request.GET for accessing GET data in the same way – but we’re explicitly using request.POST in our code, to ensure that data is only altered via a POST call.

* request.POST['choice'] will raise KeyError if choice wasn’t provided in POST data. The above code checks for KeyError and redisplays the question form with an error message if choice isn’t given.
* After incrementing the choice count, the code returns an HttpResponseRedirect rather than a normal HttpResponse. HttpResponseRedirect takes a single argument: the URL to which the user will be redirected (see the following point for how we construct the URL in this case).

As the Python comment above points out, you should always return an HttpResponseRedirect after successfully dealing with POST data. This tip isn’t specific to Django; it’s just good Web development practice.

* We are using the reverse() function in the HttpResponseRedirect constructor in this example. This function helps avoid having to hardcode a URL in the view function. It is given the name of the view that we want to pass control to and the variable portion of the URL pattern that points to that view. In this case, using the URLconf we set up in Tutorial 3, this reverse() call will return a string like
* '/polls/3/results/'

where the 3 is the value of question.id. This redirected URL will then call the 'results' view to display the final page.

As mentioned in Tutorial 3, request is an HttpRequest object. For more on HttpRequest objects, see the request and response documentation.

After somebody votes in a question, the vote() view redirects to the results page for the question. Let’s write that view:

polls/views.py

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

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

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

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

This is almost exactly the same as the detail() view from Tutorial 3. The only difference is the template name. We’ll fix this redundancy later.

Now, create a polls/results.html template:

polls/templates/polls/results.html

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

<**ul**>

{% **for** choice **in** question.choice\_set.all %}

<**li**>{{ choice.choice\_text }} -- {{ choice.votes }} vote{{ choice.votes|pluralize }}</**li**>

{% **endfor** %}

</**ul**>

<**a** href="{% **url** 'polls:detail' question.id %}">Vote again?</**a**>

Now, go to /polls/1/ in your browser and vote in the question. You should see a results page that gets updated each time you vote. If you submit the form without having chosen a choice, you should see the error message.

Note

The code for our vote() view does have a small problem. It first gets the selected\_choice object from the database, then computes the new value of votes, and then saves it back to the database. If two users of your website try to vote at *exactly the same time*, this might go wrong: The same value, let’s say 42, will be retrieved for votes. Then, for both users the new value of 43 is computed and saved, but 44 would be the expected value.

This is called a *race condition*. If you are interested, you can read Avoiding race conditions using F() to learn how you can solve this issue.

Use generic views: Less code is better**¶**

The detail() (from Tutorial 3) and results() views are very simple – and, as mentioned above, redundant. The index() view, which displays a list of polls, is similar.

These views represent a common case of basic Web development: getting data from the database according to a parameter passed in the URL, loading a template and returning the rendered template. Because this is so common, Django provides a shortcut, called the “generic views” system.

Generic views abstract common patterns to the point where you don’t even need to write Python code to write an app.

Let’s convert our poll app to use the generic views system, so we can delete a bunch of our own code. We’ll just have to take a few steps to make the conversion. We will:

1. Convert the URLconf.
2. Delete some of the old, unneeded views.
3. Introduce new views based on Django’s generic views.

Read on for details.

Why the code-shuffle?

Generally, when writing a Django app, you’ll evaluate whether generic views are a good fit for your problem, and you’ll use them from the beginning, rather than refactoring your code halfway through. But this tutorial intentionally has focused on writing the views “the hard way” until now, to focus on core concepts.

You should know basic math before you start using a calculator.

**Amend URLconf¶**

First, open the polls/urls.py URLconf and change it like so:

polls/urls.py

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

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

app\_name = 'polls'

urlpatterns = [

path('', views.IndexView.as\_view(), name='index'),

path('<int:pk>/', views.DetailView.as\_view(), name='detail'),

path('<int:pk>/results/', views.ResultsView.as\_view(), name='results'),

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

]

Note that the name of the matched pattern in the path strings of the second and third patterns has changed from <question\_id> to <pk>.

**Amend views¶**

Next, we’re going to remove our old index, detail, and results views and use Django’s generic views instead. To do so, open the polls/views.py file and change it like so:

polls/views.py

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

**from** **django.http** **import** HttpResponseRedirect

**from** **django.urls** **import** reverse

**from** **django.views** **import** generic

**from** **.models** **import** Choice, Question

**class** **IndexView**(generic.ListView):

template\_name = 'polls/index.html'

context\_object\_name = 'latest\_question\_list'

**def** get\_queryset(self):

*"""Return the last five published questions."""*

**return** Question.objects.order\_by('-pub\_date')[:5]

**class** **DetailView**(generic.DetailView):

model = Question

template\_name = 'polls/detail.html'

**class** **ResultsView**(generic.DetailView):

model = Question

template\_name = 'polls/results.html'

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

... *# same as above, no changes needed.*

We’re using two generic views here: ListView and DetailView. Respectively, those two views abstract the concepts of “display a list of objects” and “display a detail page for a particular type of object.”

* Each generic view needs to know what model it will be acting upon. This is provided using the model attribute.
* The DetailView generic view expects the primary key value captured from the URL to be called "pk", so we’ve changed question\_id to pk for the generic views.

By default, the DetailView generic view uses a template called <app name>/<model name>\_detail.html. In our case, it would use the template "polls/question\_detail.html". The template\_name attribute is used to tell Django to use a specific template name instead of the autogenerated default template name. We also specify the template\_name for the results list view – this ensures that the results view and the detail view have a different appearance when rendered, even though they’re both a DetailView behind the scenes.

Similarly, the ListView generic view uses a default template called <app name>/<model name>\_list.html; we use template\_name to tell ListView to use our existing "polls/index.html" template.

In previous parts of the tutorial, the templates have been provided with a context that contains the question and latest\_question\_list context variables. For DetailView the question variable is provided automatically – since we’re using a Django model (Question), Django is able to determine an appropriate name for the context variable. However, for ListView, the automatically generated context variable is question\_list. To override this we provide the context\_object\_name attribute, specifying that we want to use latest\_question\_list instead. As an alternative approach, you could change your templates to match the new default context variables – but it’s a lot easier to just tell Django to use the variable you want.

Run the server, and use your new polling app based on generic views.