Toast Driven

← Back to April 17, 2011

A Guide to Testing in Django #2

Viewable in Serbo-Croatian - Thanks to Vera Djuraskovic!

As part two in this series on testing in Django (part I if you missed it), we're going to continue expanding the test suite on the venerable "polls" app.

As with the previous post, the code can be found on Github with tags marking our progress along the way.

Where we left on the previous post was adding some view tests for basic HTTP GET requests. We're going to expand on that by diving into:

- Testing POST requests
- Testing forms
- Testing models

We're also going to start improving on the original code base, so that I can demonstrate how to get started testing other common aspects.

Testing POSTs

As a natural part of any web application, POST requests are a common (and good) way for the user to enter data into the application. In the current "polls" codebase, there's one view that handles POST. Ensuring that it does the right thing with both invalid & valid data is important.

Tag: 05-posts-round-i
We'll start off by testing good data, since that is oftentimes the easiest part of POSTs to test. Add the following method to the `PollsViewsTestCase`:

```
def test_good_vote(self):
      poll_1 = Poll.objects.get(pk=1)
      self.assertEqual(poll_1.choice_set.get(pk=1).votes, 1)

      resp = self.client.post('/polls/1/vote/', {'choice': 1})
      self.assertEqual(resp.status_code, 302)
      self.assertEqual(resp['Location'], 'http://testserver/polls/1/results/')

      self.assertEqual(poll_1.choice_set.get(pk=1).votes, 2)
```

We first load up the `Poll` we'll be working with from the database & ensure (as a sanity check) that the first `Choice` has the right number of votes. This helps ensure that, if a fixture or behavior is changed in the future, we'll know right away that something is wrong.

We then use a new method (`self.client.post`) to send a POST request to the desired URL. This method can take several kwargs, but for now, we're positionally providing the `data` argument, which is a simple dictionary simulating the **form-encoded** data that would be coming from a `<form>`. Since our template only populates `choice` with a `PK`, we just pass in that.

We then do a familiar check to see what the returned status code was, but this time we're looking for a redirect (`302`) instead of an OK (`200`). We also check the `Location` header being sent, to make sure we're being redirected to the right place. Django's testing infrastructure fills in a domain of `http://testserver/`, since there's no real domain to work with.

Finally, we load up another copy of the first `Choice` & ensure that the number of `votes` was incremented by one.

When you run the tests, you should get:

```
....
----------------------------------------------------------------------
Ran 4 tests in 0.374s

OK
```

However, as anyone who's ever done any user-oriented programming knows, things don't always go right. So we'll add another test method to check all the things that could go wrong. Add the following method to `PollsViewsTestCase`:

```
def test_bad_votes(self):
      # Ensure a non-existant PK throws a Not Found.
      resp = self.client.post('/polls/1000000/vote/')
      self.assertEqual(resp.status_code, 404)

      # Sanity check.
      poll_1 = Poll.objects.get(pk=1)
      self.assertEqual(poll_1.choice_set.get(pk=1).votes, 1)

      # Send no POST data.
      resp = self.client.post('/polls/1/vote/')
      self.assertEqual(resp.status_code, 200)
      self.assertEqual(resp.context['error_message'], "You didn't select a choice.")

      # Send junk POST data.
      resp = self.client.post('/polls/1/vote/', {'foo': 'bar'})
      self.assertEqual(resp.status_code, 200)
      self.assertEqual(resp.context['error_message'], "You didn't select a choice.")

      # Send a non-existant Choice PK.
      resp = self.client.post('/polls/1/vote/', {'choice': 300})
```

```
        self.assertEqual(resp.status_code, 200)
        self.assertEqual(resp.context['error_message'], "You didn't select a choice.")
```

The first & easiest thing to make sure of is making sure trying to vote on non-existant `Poll` objects doesn't work. We then do the same sanity check as before, just to make sure that if things change later on, we don't go on a wild goose chase.

We then try providing no POST data, which should return a 200 & an error message in the context. We follow that up with POST data, but not involving the data we're concerned with. Finally, we test a `Choice` primary key that wasn't in the form. Because it's HTTP POST, people can set up malicious scripts that may send random data. Ensuring that this doesn't break things is important.

In this case, the error message is the same regardless of what kind of data error there is. However, that behavior may change in the future. (FORESHADOWING!)

Running the tests should now give you:

```
.....
----------------------------------------------------------------------
Ran 5 tests in 0.475s

OK
```

A couple other notes on testing POSTs:

- Note that we didn't have to worry about CSRF. At this time, Django disables CSRF checks when running the test suite by default, though it's possible to turn them on.
- Testing HTTP headers is just a matter of adding the *Django* version of a header (i.e. `HTTP_REFERER` instead of `Referer`) as a kwarg to the `client.get` or `client.post` methods.
- Testing file uploads is also doable. The trick is to include a file-like object (via `open()`, `StringIO`, etc.) to the `data` dictionary.

From here, we're going to start extending the Django tutorial code to use more advanced functionality & resemble what a modern pluggable Django app might look like.

Sidetrack: Named URLs

You'll note that the Django tutorial doesn't use named URLs, which can be a very useful part of a pluggable application. Named URLs allow you to hook up an app at different URLs without having to change the code/templates.

Integrating them is easy, in both the main code & our test suite.

Tag: 06-named-urls
First, we change the URLconf in `polls/urls.py` to incorporate the new names:

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

We then alter the `vote` view to change how the redirect URL is reversed:

```
return HttpResponseRedirect(reverse('polls_results', kwargs={'poll_id': p.id}))
```

Finally, we modify our templates to use `{% url polls_vote poll_id=poll.id %}` for the `<form>` action & `{% url polls_detail poll_id=poll.id %}` for the detail links.

Here's a place where our new test suite comes in handy. Without doing anything else, you should run your tests. You should get back:

```
.....
----------------------------------------------------------------------
Ran 5 tests in 0.253s

OK
```

This is **GREAT** news! It means that, despite us going in & making changes, we haven't broken our application! If we had made an error (say mistyped one of the `url` tags), we'd have gotten an error when running the suite saying so.

The last step is to move our tests over to the named URLs as well:

```
import datetime
from django.core.urlresolvers import reverse
from django.test import TestCase
from polls.models import Poll, Choice


class PollsViewsTestCase(TestCase):
    fixtures = ['polls_views_testdata.json']

    def test_index(self):
        resp = self.client.get(reverse('polls_index'))
        self.assertEqual(resp.status_code, 200)
        self.assertTrue('latest_poll_list' in resp.context)
        self.assertEqual([poll.pk for poll in resp.context['latest_poll_list']], [1])
        poll_1 = resp.context['latest_poll_list'][0]
        self.assertEqual(poll_1.question, 'Are you learning about testing in Django?')
        self.assertEqual(poll_1.choice_set.count(), 2)
        choices = poll_1.choice_set.all()
        self.assertEqual(choices[0].choice, 'Yes')
        self.assertEqual(choices[0].votes, 1)
        self.assertEqual(choices[1].choice, 'No')
        self.assertEqual(choices[1].votes, 0)
```

```
    def test_detail(self):
        resp = self.client.get(reverse('polls_detail', kwargs={'poll_id': 1}))
        self.assertEqual(resp.status_code, 200)
        self.assertEqual(resp.context['poll'].pk, 1)
        self.assertEqual(resp.context['poll'].question, 'Are you learning about testing in Django?')

        # Ensure that non-existent polls throw a 404.
        resp = self.client.get(reverse('polls_detail', kwargs={'poll_id': 2}))
        self.assertEqual(resp.status_code, 404)

    def test_results(self):
        resp = self.client.get(reverse('polls_results', kwargs={'poll_id': 1}))
        self.assertEqual(resp.status_code, 200)
        self.assertEqual(resp.context['poll'].pk, 1)
        self.assertEqual(resp.context['poll'].question, 'Are you learning about testing in Django?')

        # Ensure that non-existent polls throw a 404.
        resp = self.client.get(reverse('polls_results', kwargs={'poll_id': 2}))
        self.assertEqual(resp.status_code, 404)

    def test_good_vote(self):
        poll_1 = Poll.objects.get(pk=1)
        self.assertEqual(poll_1.choice_set.get(pk=1).votes, 1)

        resp = self.client.post(reverse('polls_vote', kwargs={'poll_id': 1}), {'choice': 1})
        self.assertEqual(resp.status_code, 302)
        self.assertEqual(resp['Location'], 'http://testserver/polls/1/results/')

        self.assertEqual(poll_1.choice_set.get(pk=1).votes, 2)

    def test_bad_votes(self):
        # Ensure a non-existant PK throws a Not Found.
        resp = self.client.post(reverse('polls_vote', kwargs={'poll_id': 1000000}))
        self.assertEqual(resp.status_code, 404)

        # Sanity check.
        poll_1 = Poll.objects.get(pk=1)
        self.assertEqual(poll_1.choice_set.get(pk=1).votes, 1)

        # Send no POST data.
        resp = self.client.post(reverse('polls_vote', kwargs={'poll_id': 1}))
        self.assertEqual(resp.status_code, 200)
        self.assertEqual(resp.context['error_message'], "You didn't select a choice.")

        # Send junk POST data.
        resp = self.client.post(reverse('polls_vote', kwargs={'poll_id': 1}), {'foo': 'bar'})
        self.assertEqual(resp.status_code, 200)
        self.assertEqual(resp.context['error_message'], "You didn't select a choice.")

        # Send a non-existant Choice PK.
        resp = self.client.post(reverse('polls_vote', kwargs={'poll_id': 1}), {'choice': 300})
        self.assertEqual(resp.status_code, 200)
        self.assertEqual(resp.context['error_message'], "You didn't select a choice.")
```

Now, running our tests again should still give a passing result, but it no longer matters where our URLconf is hooked up to.

Testing Forms

One eccentricity of the Django tutorial is the omission of the use of `Forms`. This is partly for historical reasons & partially to minimize early confusion. Especially given that the choice list constantly changes between requests to different `Poll` objects, you need to use a small amount of non-typical code to make it work.

But `Forms` provide some pretty powerful validation & reusability, so we're going to integrate them.

Tag: 07-initial-form
We start off by creating a new file (`polls/forms.py`) and dropping in the following code:

```
from django import forms
from polls.models import Choice


class PollForm(forms.Form):
    def __init__(self, *args, **kwargs):
        # We require an ``instance`` parameter.
        self.instance = kwargs.pop('instance')

        # We call ``super`` (without the ``instance`` param) to finish
        # off the setup.
        super(PollForm, self).__init__(*args, **kwargs)

        # We add on a ``choice`` field based on the instance we've got.
        # This has to be done here (instead of declaratively) because the
        # ``Poll`` instance will change from request to request.
        self.fields['choice'] = forms.ModelChoiceField(queryset=Choice.objects.filter(poll=self.instance.pk), empty_label=None, widget=forms.RadioSelect)
```

We then import the `PollForm` in our views & modify the `detail` & `vote` views to do the following:

```
def detail(request, poll_id):
    p = get_object_or_404(Poll, pk=poll_id)
    form = PollForm(instance=p)
    return render_to_response('polls/detail.html', {'poll': p, 'form': form},
                              context_instance=RequestContext(request))

# ...Then later...
```

```
def vote(request, poll_id):
    p = get_object_or_404(Poll, pk=poll_id)

    if request.method == 'POST':
        form = PollForm(request.POST, instance=p)

        if form.is_valid():
            choice = form.cleaned_data['choice']
            choice.votes += 1
            choice.save()

            return HttpResponseRedirect(reverse('polls_results', kwargs={'poll_id': p.id}))
    else:
        form = PollForm(instance=p)

    return render_to_response('polls/detail.html', {
        'poll': p,
        'form': form,
    }, context_instance=RequestContext(request))
```

We check to see if the HTTP method was POST. If not (the `else` case), we instantiate a basic `form` (with the `Poll` instance to flesh out the `choice` field) & return a regular page.

If it was `POST`, we instantiate the `form` with the posted data & our `Poll` instance. We let the form handle cleaning the data & checking to make sure the choice was valid. If the `form` is valid, we simply update the `choice.votes` as before & redirect. Otherwise, we let the `form` & the errors on the form get rendered to the template.

Finally, we update the `polls/detail.html` template, removing the manual `<input>` creation in favor of:

```
{{ form.as_p }}
```

If you run the tests, you should get:

```
E....
======================================================================
ERROR: test_bad_votes (polls.tests.PollsViewsTestCase)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/Users/daniel/Desktop/guide_to_testing/polls/tests.py", line 66, in test_bad_votes
    self.assertEqual(resp.context['error_message'], "You didn't select a choice.")
  File "/usr/local/Cellar/python/2.7/lib/python2.7/site-packages/django/template/context.py", line 60, in __getitem__
    raise KeyError(key)
KeyError: 'error_message'

----------------------------------------------------------------------
Ran 5 tests in 0.256s

FAILED (errors=1)
```

Uh-oh. We broke our tests, because the functionality changed. Interestingly, we can see that the `test_good_vote` method is still passing, meaning we didn't break the correct flow through the app. Since the form is now handling the error messages, we need to update our tests to match:

```
def test_bad_votes(self):
      # Ensure a non-existant PK throws a Not Found.
      resp = self.client.post(reverse('polls_vote', kwargs={'poll_id': 1000000}))
      self.assertEqual(resp.status_code, 404)

      # Sanity check.
      poll_1 = Poll.objects.get(pk=1)
      self.assertEqual(poll_1.choice_set.get(pk=1).votes, 1)

      # Send no POST data.
      resp = self.client.post(reverse('polls_vote', kwargs={'poll_id': 1}))
      self.assertEqual(resp.status_code, 200)
      self.assertEqual(resp.context['form']['choice'].errors, [u'This field is required.'])

      # Send junk POST data.
      resp = self.client.post(reverse('polls_vote', kwargs={'poll_id': 1}), {'foo': 'bar'})
      self.assertEqual(resp.status_code, 200)
      self.assertEqual(resp.context['form']['choice'].errors, [u'This field is required.'])

      # Send a non-existant Choice PK.
      resp = self.client.post(reverse('polls_vote', kwargs={'poll_id': 1}), {'choice': 300})
      self.assertEqual(resp.status_code, 200)
      self.assertEqual(resp.context['form']['choice'].errors, [u'Select a valid choice. That choice is not one of the available choices.'])
```

We now inspect the errors on the form elements to ensure the correct error messages are being presented. You can also see that changing over to a `Form` gave us better error messages for free to the user.

Running the tests now gives you a passing result:

```
.....
----------------------------------------------------------------------
Ran 5 tests in 0.240s

OK
```

All tests passing!

Refactoring

Tag: 08-posts-refactor
We also have a chance to refactor the code a bit. You'll note that the flow through the `detail` view & the flow through the `vote` when sent a GET are the same. Let's streamline our views. First, we remove the following URLconf item:

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

Next, we'll take the code from the `vote` view & put it in the `detail` view instead, overwriting what was there & removing the remnants of the `vote` view:

```
def detail(request, poll_id):
    p = get_object_or_404(Poll, pk=poll_id)

    if request.method == 'POST':
        form = PollForm(request.POST, instance=p)

        if form.is_valid():
            choice = form.cleaned_data['choice']
            choice.votes += 1
            choice.save()

            return HttpResponseRedirect(reverse('polls_results', kwargs={'poll_id': p.id}))
    else:
        form = PollForm(instance=p)

    return render_to_response('polls/detail.html', {
        'poll': p,
        'form': form,
    }, context_instance=RequestContext(request))
```

Lastly, we change up the `polls/detail.html` template to use:

```
<form action="." method="post">
```

If you hit the detail page in a browser, you'll see that it's working fine, but any tests that were pointing at the `vote` view are now out of date. We update all references to `reverse('polls_vote',...)` to `reverse('polls_detail', ...)`. When we run our tests, we should get:

```
.....
----------------------------------------------------------------------
Ran 5 tests in 0.239s

OK
```

Yay! We're back to green (all tests passing) & have successfully refactored our code. But there's one more improvement we can make. That vote increment really ought to get pushed down a little bit, since that'll be a common operation & perhaps not just restricted to that view.

We'll push it down to the model level, adding the following method to `Choice`:

```
class Choice(models.Model):
    # What's already there, then...

    def record_vote(self):
          self.votes += 1
          self.save()
```

We'll also update our `PollForm` to take advantage of the new method by adding a new method (`PollForm.save`) to it:

```
class PollForm(forms.Form):
    # What's already there, then...

    def save(self):
          choice = self.cleaned_data['choice']
          choice.record_vote()
          return choice
```

Finally, we'll update our `detail` view to call `form.save()` instead of the old logic:

```
def detail(request, poll_id):
    p = get_object_or_404(Poll, pk=poll_id)

    if request.method == 'POST':
        form = PollForm(request.POST, instance=p)

        if form.is_valid():
            form.save()
            return HttpResponseRedirect(reverse('polls_results', kwargs={'poll_id': p.id}))
    else:
        form = PollForm(instance=p)

    return render_to_response('polls/detail.html', {
        'poll': p,
        'form': form,
    }, context_instance=RequestContext(request))
```

Running the tests shows that we're still passing!

```
.....
----------------------------------------------------------------------
Ran 5 tests in 0.243s

OK
```

This is good, because now our view does as little as possible beyond tying together the needed elements & controlling flow. Our form presents an easy way to store the vote. And the logic for how votes get recorded is moved down to the `Choice` model, where other parts of our codebase can now take advantage of it.

In this case, it's almost so trivial it's not worth it, but should you need to change the way votes are recorded in the future (say logging it elsewhere or extending the data model to tie individual votes back to a user), you'd only need to update the model in order to accommodate.

Tag: 09-separate-forms-tests
A potential problem has cropped up though. Our form and our model have sprouted new methods not covered by Django's test suite. And while our view tests kind of make sure it is working right, they are more like integration tests (making sure the form/model/view/template all play nice together).

But first, we might want to reorganize our tests. We could just append them onto our `tests.py` file, but I prefer to split up the tests into a file structure that mirrors the app's setup.

To do this, we create a new `tests` directory then move the existing `tests.py` into `tests/views.py`. Lastly, we need to drop in an `tests/__init__.py` file (so that `tests/` is recognized as a Python module). Within that file, we need to put in the following code:

```
from polls.tests.views import *
```

This imports the test code we've rewritten so it gets run as part of the suite. Let's run our tests to double-check:

```
.....
----------------------------------------------------------------------
Ran 5 tests in 1.066s

OK
```

Perfect. Now we'll create a new `tests/forms.py` file & start by adding unit tests for our form. We don't need to test every aspect of the form (as Django has good test coverage of forms). What we *do* need to test is places we've extended the form, specifically the `__init__` & `save` methods. Our code should look something like:

```
from django.test import TestCase
from polls.forms import PollForm
from polls.models import Poll, Choice


class PollFormTestCase(TestCase):
    fixtures = ['polls_forms_testdata.json']

    def setUp(self):
        super(PollFormTestCase, self).setUp()
        self.poll_1 = Poll.objects.get(pk=1)
        self.poll_2 = Poll.objects.get(pk=2)

    def test_init(self):
        # Test successful init without data.
        form = PollForm(instance=self.poll_1)
        self.assertTrue(isinstance(form.instance, Poll))
        self.assertEqual(form.instance.pk, self.poll_1.pk)
        self.assertEqual([c for c in form.fields['choice'].choices], [(1, u'Yes'), (2, u'No')])

        # Test successful init with data.
        form = PollForm({'choice': 3}, instance=self.poll_2)
        self.assertTrue(isinstance(form.instance, Poll))
        self.assertEqual(form.instance.pk, self.poll_2.pk)
        self.assertEqual([c for c in form.fields['choice'].choices], [(3, u'Alright.'), (4, u'Meh.'), (5, u'Not so good.')])

        # Test a failed init without data.
        self.assertRaises(KeyError, PollForm)

        # Test a failed init with data.
        self.assertRaises(KeyError, PollForm, {})
```

Very similar to what we've seen before, with some new bits. We've added a `setUp` method, which lets us do things that will affect all the test methods within the class. Since we're going to need `Poll` objects in several places, we load them up there.

We then manually test the `__init__` method. We try both with & without data, to ensure that no errors are raised & the choices are fleshed out correctly. We also have added a new `Poll`+`Choices` to our fixture data, so we can actually ensure that something isn't broken & pulling in wrong/all data, which is important to be able to assert on the `choice` field.

We've also introduced a new method (`assertRaises`) when testing the missing `instance` kwarg. This method ensures that a certain exception is raised in the course of executing the provided callable. Using it feels odd, because you don't use the parenthetical version (`PollForm({})`), instead expanding arguments out within the `assertRaises` call.

Note that we're also using a new fixture (`fixtures/polls_forms_testdata.json`). We're doing this because if we changed our `polls_views_testdata.json`, we'd break our existing view tests.

Fixtures can be slightly fragile in that if all your tests share a single fixture, it's easy to break a lot of tests with a small change. However, adding to fixture can help find bugs in other places or unveil fragile tests. You need to weigh the benefits/costs & decide what works for your situation.

Our new fixture should look something like the following & incorporates a new `Poll` with more `Choice` options, so we can verify that choices change between whichever instance is used.

```
[
    {
        "pk": 1,
        "model": "polls.poll",
        "fields": {
            "pub_date": "2011-04-09 23:44:02",
            "question": "Are you learning about testing in Django?"
        }
    },
    {
        "pk": 2,
        "model": "polls.poll",
        "fields": {
```

```
            "pub_date": "2011-04-16 14:42:52",
            "question": "How do you feel today?"
        }
    },
    {
        "pk": 1,
        "model": "polls.choice",
        "fields": {
            "votes": 1,
            "poll": 1,
            "choice": "Yes"
        }
    },
    {
        "pk": 2,
        "model": "polls.choice",
        "fields": {
            "votes": 0,
            "poll": 1,
            "choice": "No"
        }
    },
    {
        "pk": 3,
        "model": "polls.choice",
        "fields": {
            "votes": 1,
            "poll": 2,
            "choice": "Alright."
        }
    },
    {
        "pk": 4,
        "model": "polls.choice",
        "fields": {
            "votes": 0,
            "poll": 2,
            "choice": "Meh."
        }
    },
    {
        "pk": 5,
        "model": "polls.choice",
        "fields": {
            "votes": 0,
            "poll": 2,
            "choice": "Not so good."
        }
    }
]
```

One last detail. We need to add to the `tests/__init__.py` file to pull in the new forms tests. All that's needed is adding:

```
from polls.tests.forms import *
```

Now, running our tests should yield:

```
......
----------------------------------------------------------------------
Ran 6 tests in 0.926s

OK
```

Now, to test our custom `save` method. We'll start a new method (`test_save`) and drop in now-familiar code:

```
def test_save(self):
    self.assertEqual(self.poll_1.choice_set.get(pk=1).votes, 1)
    self.assertEqual(self.poll_1.choice_set.get(pk=2).votes, 0)

    # Test the first choice.
    form_1 = PollForm({'choice': 1}, instance=self.poll_1)
    form_1.save()
    self.assertEqual(self.poll_1.choice_set.get(pk=1).votes, 2)
    self.assertEqual(self.poll_1.choice_set.get(pk=2).votes, 0)
```

Let's run our tests & see what happens:

```
.E.....
======================================================================
ERROR: test_save (polls.tests.forms.PollFormTestCase)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/Users/daniel/Desktop/guide_to_testing/polls/tests/forms.py", line 39, in test_save
    form_1.save()
  File "/Users/daniel/Desktop/guide_to_testing/polls/forms.py", line 20, in save
    choice = self.cleaned_data['choice']
AttributeError: 'PollForm' object has no attribute 'cleaned_data'

----------------------------------------------------------------------
Ran 7 tests in 0.328s

FAILED (errors=1)
```

Uh oh. What happened? The traceback indicates that `cleaned_data` isn't present on the form. The only times this can happen is either when the form is unbound (no data provided), when there's a validation error or when validation hasn't been performed. The last reason is the cause, as nothing is calling `form.full_clean` or `form.is_valid`.

Since this could happen in real code, we've got a bug on our hands.

To fix this, we'll add a check to the `save` method to check that it's been validated:

```
class PollForm(forms.Form):
    # Existing code then...
    def save(self):
        if not self.is_valid():
            raise forms.ValidationError("PollForm was not validated first before trying to call 'save'.")

        choice = self.cleaned_data['choice']
        choice.record_vote()
        return choice
```

The important thing we're trying to prevent against here is an invalid access of the data or, worse, writing bad data to the database. Now running the tests should get us:

```
.......
----------------------------------------------------------------------
Ran 7 tests in 0.323s

OK
```

Much better. Let's finish testing the `save` method. Revise the code to look like:

```
def test_save(self):
    self.assertEqual(self.poll_1.choice_set.get(pk=1).votes, 1)
    self.assertEqual(self.poll_1.choice_set.get(pk=2).votes, 0)

    # Test the first choice.
    form_1 = PollForm({'choice': 1}, instance=self.poll_1)
    form_1.save()
    self.assertEqual(self.poll_1.choice_set.get(pk=1).votes, 2)
    self.assertEqual(self.poll_1.choice_set.get(pk=2).votes, 0)

    # Test the second choice.
    form_2 = PollForm({'choice': 2}, instance=self.poll_1)
    form_2.save()
    self.assertEqual(self.poll_1.choice_set.get(pk=1).votes, 2)
    self.assertEqual(self.poll_1.choice_set.get(pk=2).votes, 1)

    # Test the other poll.
    self.assertEqual(self.poll_2.choice_set.get(pk=3).votes, 1)
    self.assertEqual(self.poll_2.choice_set.get(pk=4).votes, 0)
    self.assertEqual(self.poll_2.choice_set.get(pk=5).votes, 0)

    form_3 = PollForm({'choice': 5}, instance=self.poll_2)
    form_3.save()
    self.assertEqual(self.poll_2.choice_set.get(pk=3).votes, 1)
    self.assertEqual(self.poll_2.choice_set.get(pk=4).votes, 0)
    self.assertEqual(self.poll_2.choice_set.get(pk=5).votes, 1)
```

We do a bunch of different assertions here on a variety of data, just to make sure the proper counts are being saved in the right places. One more run of the tests yields:

```
.......
----------------------------------------------------------------------
Ran 7 tests in 0.323s

OK
```

And our form is tested! If your forms involve other things, such as custom `clean_FOO` methods, those are also good things to test, in the same style as the `save` method, making sure they returned the right cleaned data or raised an exception like they should.

Testing Models

Tag: 10-model-tests
Finally, on to testing the model. The good news is that our tests will look very familiar, strongly resembling the flow & content of the forms tests.

Again, we won't worry about testing things that Django should have tests for. We'll test only our extensions to the models as well as testing our usage of them.

First, drop the following in the `tests/__init__.py` file, so that we don't forget to import our tests later:

```
from polls.tests.models import *
```

We'll then create a new file called `tests/models.py`. After reviewing our models, there are several immediate things we should be testing. They are:

- The `was_published_today` method on `Poll`.
- The `record_vote` method on `Choice`.

Additionally, given the skeletal nature of the models, there are some improvements we can make:

- Default datetime in the `Poll.pub_date` field.
- A sane default number of `votes` on ``Choice.
- `Poll` objects with a future `pub_date` might accidentally get published in our list view. Having a list of published `Polls` would be useful.

Let's start with the initial tests to cover our methods. Drop in the following code in `tests/models.py`:

```
import datetime
from django.test import TestCase
from polls.models import Poll, Choice
```

```
class PollTestCase(TestCase):
    fixtures = ['polls_forms_testdata.json']

    def setUp(self):
        super(PollTestCase, self).setUp()
        self.poll_1 = Poll.objects.get(pk=1)
        self.poll_2 = Poll.objects.get(pk=2)

    def test_was_published_today(self):
        # Because unless you're timetraveling, they weren't.
        self.assertFalse(self.poll_1.was_published_today())
        self.assertFalse(self.poll_2.was_published_today())

        # Modify & check again.
        now = datetime.datetime.now()
        self.poll_1.pub_date = now
        self.poll_1.save()
        self.assertTrue(self.poll_1.was_published_today())
```

I'm cheating & reusing the forms fixture for brevity. We do the same `setUp` dance as we did in the `PollFormTestCase`, then do some basic testing on the `was_published_today` method. We make sure the test data wasn't created today, then modify one of the polls to use the current date/time. Rechecking that method should now give us a `True`.

Be careful with date-based tests, especially things that use `datetime.datetime.now`. What passes tests one day might fail the next. A common way of handling that is to use mocking of one sort or another, which we'll cover at a later point.

Running the tests should now return:

```
........
----------------------------------------------------------------------
Ran 8 tests in 0.353s

OK
```

Now we'll test the `Choice.record_vote` method. We'll add a new class to our `tests/models.py` file like so:

```
class ChoiceTestCase(TestCase):
    fixtures = ['polls_forms_testdata.json']

    def test_record_vote(self):
        choice_1 = Choice.objects.get(pk=1)
        choice_2 = Choice.objects.get(pk=2)

        self.assertEqual(Choice.objects.get(pk=1).votes, 1)
        self.assertEqual(Choice.objects.get(pk=2).votes, 0)

        choice_1.record_vote()
        self.assertEqual(Choice.objects.get(pk=1).votes, 2)
        self.assertEqual(Choice.objects.get(pk=2).votes, 0)

        choice_2.record_vote()
        self.assertEqual(Choice.objects.get(pk=1).votes, 2)
        self.assertEqual(Choice.objects.get(pk=2).votes, 1)

        choice_1.record_vote()
        self.assertEqual(Choice.objects.get(pk=1).votes, 3)
        self.assertEqual(Choice.objects.get(pk=2).votes, 1)
```

Nothing crazy here, just checking a couple calls to the `Choice.record_vote` method to make sure increments are done correctly. You'll note that we're continually using `Choice.objects.get(pk=1)` instead of the `choice_1` object we've created at the top. That's because we want to make sure the database is really getting updated (not silently failing & only updating the count on the model instance in memory).

Running the tests yields a happy result:

```
----------------------------------------------------------------------
Ran 9 tests in 0.373s

OK
```

Tag: 11-model-improvements Now, on to our improvements. For a change, we'll use the **test-driven** style, starting with the tests then building out the functionality in the models.

We start by adding the following method to the `PollTestCase`:

```
def test_better_defaults(self):
    now = datetime.datetime.now()
    poll = Poll.objects.create(
        question="A test question."
    )
    self.assertEqual(poll.pub_date.date(), now.date())
```

Running the tests gives us a "red" (test failure):

```
...E......
======================================================================
ERROR: test_better_defaults (polls.tests.models.PollTestCase)
----------------------------------------------------------------------
Traceback (most recent call last):
  # Snipped for brevity...
IntegrityError: polls_poll.pub_date may not be NULL

----------------------------------------------------------------------
```

```
Ran 10 tests in 0.661s

FAILED (errors=1)
```

Ugh, `IntegrityError`. Let's fix that by changing the model definition (``polls):

```
class Poll(models.Model):
    question = models.CharField(max_length=200)
    pub_date = models.DateTimeField('date published', default=datetime.datetime.now)
```

Re-running the tests turns them "green" (test success):

```
..........
----------------------------------------------------------------------
Ran 10 tests in 0.378s

OK
```

WARNING - When we're making model changes, the tests will pick this up (due to freshly rebuilding DB structure on each run) **BUT** your live setup will still be using the old schema. Please use a tool like South or nashvegas to handle live migrations.

Let's improve `Choice` next by giving it a sane default number of votes. Add the following method to `ChoiceTestCase`:

```
def test_better_defaults(self):
    poll = Poll.objects.create(
        question="Are you still there?"
    )
    choice = Choice.objects.create(
        poll=poll,
        choice="I don't blame you."
    )

    self.assertEqual(poll.choice_set.all()[0].choice, "I don't blame you.")
    self.assertEqual(poll.choice_set.all()[0].votes, 0)
```

Running the tests yields red:

```
E..........
======================================================================
ERROR: test_better_defaults (polls.tests.models.ChoiceTestCase)
----------------------------------------------------------------------
Traceback (most recent call last):
  # Snipped for brevity...
IntegrityError: polls_choice.votes may not be NULL

----------------------------------------------------------------------
Ran 11 tests in 0.563s

FAILED (errors=1)
```

Let's spruce up `Choice`:

```
class Choice(models.Model):
    poll = models.ForeignKey(Poll)
    choice = models.CharField(max_length=200)
    votes = models.IntegerField(default=0)
```

Re-running the tests stays red but gives a different error:

```
E..........
======================================================================
ERROR: test_better_defaults (polls.tests.models.ChoiceTestCase)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/Users/daniel/Desktop/guide_to_testing/polls/tests/models.py", line 64, in test_better_defaults
    self.assertEqual(poll.choices.all()[0].choice, "I don't blame you.")
AttributeError: 'Poll' object has no attribute 'choices'

----------------------------------------------------------------------
Ran 11 tests in 0.500s

FAILED (errors=1)
```

Re-running the tests give us:

```
...........
----------------------------------------------------------------------
Ran 11 tests in 0.435s

OK
```

And we're back to green. Let's tackle our final improvement, granting `Poll` a better way to fetch `published` polls. We'll take the approach of adding a new `Manager` to the `Poll` class. Tests first!

```
def test_no_future_dated_polls(self):
    # Create the future-dated ``Poll``.
    poll = Poll.objects.create(
        question="Do we have flying cars yet?",
        pub_date=datetime.datetime.now() + datetime.timedelta(days=1)
    )
    self.assertEqual(list(Poll.objects.all().values_list('id', flat=True)), [1, 2, 3])
    self.assertEqual(list(Poll.published.all().values_list('id', flat=True)), [1, 2])
```

Note that we're using `.values_list` to select the list of primary keys, **BUT** we're wrapping that call in `list(...)`. This is because, despite the `__repr__` looking like it's a `list`, `.values_list` actually returns a `ValuesListQuerySet`, which won't test as being "equal" to the vanilla list of integers.

Running the tests yields a failure as expected:

```
.....E......
======================================================================
ERROR: test_no_future_dated_polls (polls.tests.models.PollTestCase)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/Users/daniel/Desktop/guide_to_testing/polls/tests/models.py", line 39, in test_no_future_dated_polls
    self.assertEqual(list(Poll.published.all().values_list('id', flat=True)), [1, 2])
AttributeError: type object 'Poll' has no attribute 'published'

----------------------------------------------------------------------
Ran 12 tests in 0.418s

FAILED (errors=1)
```

So we modify the `Poll` model setup like so, adding a new, non-default manager to the model:

```
class PollManager(models.Manager):
    def get_query_set(self):
        now = datetime.datetime.now()
        return super(PollManager, self).get_query_set().filter(pub_date__lte=now)


class Poll(models.Model):
    question = models.CharField(max_length=200)
    pub_date = models.DateTimeField('date published', default=datetime.datetime.now)

    objects = models.Manager()
    published = PollManager()
```

And we run our tests again, getting green once again:

```
............
----------------------------------------------------------------------
Ran 12 tests in 0.574s

OK
```

Unhappily, those tests passing are actually masking a bug. We've added that new `Poll.published` manager, but not using it anywhere outside of the test suite, meaning the live views still will serve future dated items. If we alter `fixtures/polls_views_testdata.json` to include an extra, future-dated `Poll`, we'll expose the bug:

```
# ...Somewhere within the ``[]``s...
{
    "pk": 2,
    "model": "polls.poll",
    "fields": {
        "pub_date": "2031-04-09 23:44:02",
        "question": "IT'S THE FUTURE..."
    }
},
```

Now running the tests yields several failures:

```
........F.FF
======================================================================
FAIL: test_detail (polls.tests.views.PollsViewsTestCase)
----------------------------------------------------------------------
# Snipped.

======================================================================
FAIL: test_index (polls.tests.views.PollsViewsTestCase)
----------------------------------------------------------------------
# Snipped.

======================================================================
FAIL: test_results (polls.tests.views.PollsViewsTestCase)
----------------------------------------------------------------------
# Snipped.

----------------------------------------------------------------------
Ran 12 tests in 0.484s

FAILED (failures=3)
```

We can fix these by updating our views to the following code:

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


def detail(request, poll_id):
    p = get_object_or_404(Poll.published.all(), pk=poll_id)

    if request.method == 'POST':
        form = PollForm(request.POST, instance=p)

        if form.is_valid():
            form.save()
            return HttpResponseRedirect(reverse('polls_results', kwargs={'poll_id': p.id}))
```

```
    else:
        form = PollForm(instance=p)

    return render_to_response('polls/detail.html', {
        'poll': p,
        'form': form,
    }, context_instance=RequestContext(request))


def results(request, poll_id):
    p = get_object_or_404(Poll.published.all(), pk=poll_id)
    return render_to_response('polls/results.html', {'poll': p})
```

Note that `get_object` can take either a `Model` (using the default manager) or a `QuerySet`, in this case the `Poll.published.all()`. Running the tests brings us back to green:

```
............
----------------------------------------------------------------------
Ran 12 tests in 0.436s

OK
```

What's Next?

We covered a lot more ground, getting more familiar with the flow of tests, using tests to help sanity-check refactorings & seen a little TDD in action. In the next post, we'll look at other common Django components:

- Testing template tags & filters
- Testing email
- Testing syndication

By on April 17, 2011.