Django

Documentation

Writing your first Django app, part 5

This tutorial begins where Tutorial 4 left off. We've built a Web-poll application, and we'll now create some automated tests for it.

Introducing automated testing

What are automated tests?

Tests are simple routines that check the operation of your code.

Testing operates at different levels. Some tests might apply to a tiny detail (*does a particular model method return values as expected?*) while others examine the overall operation of the software (*does a sequence of user inputs on the site produce the desired result?*). That's no different from the kind of testing you did earlier in Tutorial 2, using the **`shell`** to examine the behavior of a method, or running the application and entering data to check how it behaves.

What's different in *automated* tests is that the testing work is done for you by the system. You create a set of tests once, and then as you make changes to your app, you can check that your code still works as you originally intended, without having to perform time consuming manual testing.

Why you need to create tests

So why create tests, and why now?

You may feel that you have quite enough on your plate just learning Python/Django, and having yet another thing to learn and do may seem overwhelming and perhaps unnecessary. After all, our polls application is working quite happily now; going through the trouble of creating automated tests is not going to make it work any better. If creating the polls application is the last bit of Django programming you will ever do, then true, you don't need to know how to create automated tests. But, if that's not the case, now is an excellent time to learn.

Tests will save you time

Up to a certain point, 'checking that it seems to work' will be a satisfactory test. In a more sophisticated application, you might have dozens of complex interactions between components.

A change in any of those components could have unexpected consequences on the application's behavior. Checking that it still 'seems to work' could mean running through your code's functionality with twenty different variations of your test data just to make sure you haven't broken something - not a good use of your time.

That's especially true when automated tests could do this for you in seconds. If something's gone wrong, tests will also assist in identifying the code that's causing the unexpected behavior.

Sometimes it may seem a chore to tear yourself away from your productive, creative programming work to face the unglamorous and unexciting business of writing tests, particularly when you know your code is working properly.

However, the task of writing tests is a lot more fulfilling than spending hours testing your application manually or trying to identify the cause of a newly-introduced problem.

Tests don't just identify problems, they prevent them

It's a mistake to think of tests merely as a negative aspect of development.

Without tests, the purpose or intended behavior of an application might be rather opaque. Even when it's your own code, you will sometimes find yourself poking around in it trying to find out what exactly it's doing.

Tests change that; they light up your code from the inside, and when something goes wrong, they focus light on the part that has gone wrong - *even if you hadn't even realized it had gone wrong*.

Tests make your code more attractive

You might have created a brilliant piece of software, but you will find that many other developers will simply refuse to look at it because it lacks tests; without tests, they won't trust it. Jacob Kaplan-Moss, one of Django's original developers, says "Code without tests is broken by design."

That other developers want to see tests in your software before they take it seriously is yet another reason for you to start writing tests.

Tests help teams work together

The previous points are written from the point of view of a single developer maintaining an application. Complex applications will be maintained by teams. Tests guarantee that

colleagues don't inadvertently break your code (and that you don't break theirs without knowing). If you want to make a living as a Django programmer, you must be good at writing tests!

Basic testing strategies

There are many ways to approach writing tests.

Some programmers follow a discipline called "test-driven development"; they actually write their tests before they write their code. This might seem counter-intuitive, but in fact it's similar to what most people will often do anyway: they describe a problem, then create some code to solve it. Test-driven development simply formalizes the problem in a Python test case.

More often, a newcomer to testing will create some code and later decide that it should have some tests. Perhaps it would have been better to write some tests earlier, but it's never too late to get started.

Sometimes it's difficult to figure out where to get started with writing tests. If you have written several thousand lines of Python, choosing something to test might not be easy. In such a case, it's fruitful to write your first test the next time you make a change, either when you add a new feature or fix a bug.

So let's do that right away.

Writing our first test

We identify a bug

Fortunately, there's a little bug in the **polls** application for us to fix right away: the **Question.was_published_recently()** method returns **True** if the **Question** was published within the last day (which is correct) but also if the **Question**'s **pub_date** field is in the future (which certainly isn't).

To check if the bug really exists, using the Admin create a question whose date lies in the future and check the method using the **shell**:

```
>>> import datetime
>>> from django.utils import timezone
>>> from polls.models import Question
>>> # create a Question instance with pub_date 30 days in the future
>>> future_question = Question(pub_date=timezone.now() + datetime.timedelta(days=30))
>>> # was it published recently?
>>> future_question.was_published_recently()
True
```

Since things in the future are not 'recent', this is clearly wrong.

Create a test to expose the bug

What we've just done in the **shell** to test for the problem is exactly what we can do in an automated test, so let's turn that into an automated test.

A conventional place for an application's tests is in the application's **tests.py** file; the testing system will automatically find tests in any file whose name begins with **test**.

Put the following in the **tests.py** file in the **polls** application:

polls/tests.py

```
import datetime
from django.utils import timezone
from django.test import TestCase

from .models import Question


class QuestionMethodTests(TestCase):

    def test_was_published_recently_with_future_question(self):
        """
        was_published_recently() should return False for questions whose
        pub_date is in the future.
        """
        time = timezone.now() + datetime.timedelta(days=30)
        future_question = Question(pub_date=time)
        self.assertEqual(future_question.was_published_recently(), False)
```

What we have done here is created a **django.test.TestCase** subclass with a method that creates a **Question** instance with a **pub_date** in the future. We then check the output of **was_published_recently()** - which *ought* to be False.

Running tests

In the terminal, we can run our test:

```
$ python manage.py test polls
```

and you'll see something like:

```
Creating test database for alias 'default'...
F
======================================================================
FAIL: test_was_published_recently_with_future_question (polls.tests.QuestionMethodTests)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/path/to/mysite/polls/tests.py", line 16, in test_was_published_recently_with_future_question
    self.assertEqual(future_question.was_published_recently(), False)
AssertionError: True != False

----------------------------------------------------------------------
Ran 1 test in 0.001s

FAILED (failures=1)
Destroying test database for alias 'default'...
```

What happened is this:

- **python manage.py test polls** looked for tests in the **polls** application
- it found a subclass of the **django.test.TestCase** class
- it created a special database for the purpose of testing
- it looked for test methods - ones whose names begin with **test**
- in **test_was_published_recently_with_future_question** it created a **Question** instance whose **pub_date** field is 30 days in the future
- ... and using the **assertEqual()** method, it discovered that its **was_published_recently()** returns **True**, though we wanted it to return **False**

The test informs us which test failed and even the line on which the failure occurred.

Fixing the bug

We already know what the problem is: **Question.was_published_recently()** should return **False** if its **pub_date** is in the future. Amend the method in **models.py**, so that it will only return **True** if the date is also in the past:

polls/models.py

```
def was_published_recently(self):
    now = timezone.now()
    return now - datetime.timedelta(days=1) <= self.pub_date <= now
```

and run the test again:

```
Creating test database for alias 'default'...
.
----------------------------------------------------------------------
Ran 1 test in 0.001s

OK
Destroying test database for alias 'default'...
```

After identifying a bug, we wrote a test that exposes it and corrected the bug in the code so our test passes.

Many other things might go wrong with our application in the future, but we can be sure that we won't inadvertently reintroduce this bug, because simply running the test will warn us immediately. We can consider this little portion of the application pinned down safely forever.

More comprehensive tests

While we're here, we can further pin down the **`was_published_recently()`** method; in fact, it would be positively embarrassing if in fixing one bug we had introduced another.

Add two more test methods to the same class, to test the behavior of the method more comprehensively:

polls/tests.py

```
def test_was_published_recently_with_old_question(self):
    """
    was_published_recently() should return False for questions whose
    pub_date is older than 1 day.
    """
    time = timezone.now() - datetime.timedelta(days=30)
    old_question = Question(pub_date=time)
    self.assertEqual(old_question.was_published_recently(), False)

def test_was_published_recently_with_recent_question(self):
    """
    was_published_recently() should return True for questions whose
    pub_date is within the last day.
    """
    time = timezone.now() - datetime.timedelta(hours=1)
    recent_question = Question(pub_date=time)
    self.assertEqual(recent_question.was_published_recently(), True)
```

And now we have three tests that confirm that **`Question.was_published_recently()`** returns sensible values for past, recent, and future questions.

Again, **`polls`** is a simple application, but however complex it grows in the future and whatever other code it interacts with, we now have some guarantee that the method we have written tests for will behave in expected ways.

Test a view

The polls application is fairly undiscriminating: it will publish any question, including ones whose **`pub_date`** field lies in the future. We should improve this. Setting a **`pub_date`** in the future should mean that the Question is published at that moment, but invisible until then.

A test for a view

When we fixed the bug above, we wrote the test first and then the code to fix it. In fact that was a simple example of test-driven development, but it doesn't really matter in which order we do the work.

In our first test, we focused closely on the internal behavior of the code. For this test, we want to check its behavior as it would be experienced by a user through a web browser.

Before we try to fix anything, let's have a look at the tools at our disposal.

The Django test client

Django provides a test **`Client`** to simulate a user interacting with the code at the view level. We can use it in **`tests.py`** or even in the **`shell`**.

We will start again with the **`shell`**, where we need to do a couple of things that won't be necessary in **`tests.py`**. The first is to set up the test environment in the **`shell`**:

```
>>> from django.test.utils import setup_test_environment
>>> setup_test_environment()
```

`setup_test_environment()` installs a template renderer which will allow us to examine some additional attributes on responses such as **`response.context`** that otherwise wouldn't be available. Note that this method *does not* setup a test database, so the following will be run against the existing database and the output may differ slightly depending on what questions you already created.

Next we need to import the test client class (later in **`tests.py`** we will use the **`django.test.TestCase`** class, which comes with its own client, so this won't be required):

```
>>> from django.test import Client
>>> # create an instance of the client for our use
>>> client = Client()
```

With that ready, we can ask the client to do some work for us:

```
>>> # get a response from '/'
>>> response = client.get('/')
>>> # we should expect a 404 from that address
>>> response.status_code
404
>>> # on the other hand we should expect to find something at '/polls/'
>>> # we'll use 'reverse()' rather than a hardcoded URL
>>> from django.core.urlresolvers import reverse
>>> response = client.get(reverse('polls:index'))
>>> response.status_code
200
>>> response.content
b'\n\n\n    <p>No polls are available.</p>\n\n'
>>> # note - you might get unexpected results if your ``TIME_ZONE``
>>> # in ``settings.py`` is not correct. If you need to change it,
>>> # you will also need to restart your shell session
>>> from polls.models import Question
>>> from django.utils import timezone
>>> # create a Question and save it
>>> q = Question(question_text="Who is your favorite Beatle?", pub_date=timezone.now())
>>> q.save()
>>> # check the response once again
>>> response = client.get('/polls/')
>>> response.content
b'\n\n\n    <ul>\n    \n        <li><a href="/polls/1/">Who is your favorite Beatle?</a></li>\n    \n    </ul>\n\n'
>>> # If the following doesn't work, you probably omitted the call to
>>> # setup_test_environment() described above
>>> response.context['latest_question_list']
[<Question: Who is your favorite Beatle?>]
```

Improving our view

The list of polls shows polls that aren't published yet (i.e. those that have a **`pub_date`** in the future). Let's fix that.

In Tutorial 4 we introduced a class-based view, based on **`ListView`**:

polls/views.py

```
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]
```

We need to amend the **get_queryset()** method and change it so that it also checks the date by comparing it with **timezone.now()**. First we need to add an import:

polls/views.py

```
from django.utils import timezone
```

and then we must amend the **get_queryset** method like so:

polls/views.py

```
def get_queryset(self):
    """
    Return the last five published questions (not including those set to be
    published in the future).
    """
    return Question.objects.filter(
        pub_date__lte=timezone.now()
    ).order_by('-pub_date')[:5]
```

Question.objects.filter(pub_date__lte=timezone.now()) returns a queryset containing **Question**s whose **pub_date** is less than or equal to - that is, earlier than or equal to - **timezone.now**.

Testing our new view

Now you can satisfy yourself that this behaves as expected by firing up the runserver, loading the site in your browser, creating **Questions** with dates in the past and future, and checking that only those that have been published are listed. You don't want to have to do that *every single time you make any change that might affect this* - so let's also create a test, based on our **shell** session above.

Add the following to **polls/tests.py**:

polls/tests.py

```
from django.core.urlresolvers import reverse
```

and we'll create a shortcut function to create questions as well as a new test class:

polls/tests.py

```
    def create_question(question_text, days):
        """
        Creates a question with the given `question_text` and published the
        given number of `days` offset to now (negative for questions published
        in the past, positive for questions that have yet to be published).
        """
        time = timezone.now() + datetime.timedelta(days=days)
        return Question.objects.create(question_text=question_text,
                                       pub_date=time)


class QuestionViewTests(TestCase):
    def test_index_view_with_no_questions(self):
        """
        If no questions exist, an appropriate message should be displayed.
        """
        response = self.client.get(reverse('polls:index'))
        self.assertEqual(response.status_code, 200)
        self.assertContains(response, "No polls are available.")
        self.assertQuerysetEqual(response.context['latest_question_list'], [])

    def test_index_view_with_a_past_question(self):
        """
        Questions with a pub_date in the past should be displayed on the
        index page.
        """
        create_question(question_text="Past question.", days=-30)
        response = self.client.get(reverse('polls:index'))
        self.assertQuerysetEqual(
            response.context['latest_question_list'],
            ['<Question: Past question.>']
        )

    def test_index_view_with_a_future_question(self):
        """
        Questions with a pub_date in the future should not be displayed on
        the index page.
        """
        create_question(question_text="Future question.", days=30)
        response = self.client.get(reverse('polls:index'))
        self.assertContains(response, "No polls are available.",
                            status_code=200)
        self.assertQuerysetEqual(response.context['latest_question_list'], [])

    def test_index_view_with_future_question_and_past_question(self):
        """
        Even if both past and future questions exist, only past questions
        should be displayed.
        """
        create_question(question_text="Past question.", days=-30)
        create_question(question_text="Future question.", days=30)
        response = self.client.get(reverse('polls:index'))
        self.assertQuerysetEqual(
            response.context['latest_question_list'],
            ['<Question: Past question.>']
        )

    def test_index_view_with_two_past_questions(self):
        """
        The questions index page may display multiple questions.
        """
        create_question(question_text="Past question 1.", days=-30)
        create_question(question_text="Past question 2.", days=-5)
        response = self.client.get(reverse('polls:index'))
        self.assertQuerysetEqual(
            response.context['latest_question_list'],
            ['<Question: Past question 2.>', '<Question: Past question 1.>']
        )
```

Let's look at some of these more closely.

First is a question shortcut function, **`create_question`**, to take some repetition out of the process of creating questions.

`test_index_view_with_no_questions` doesn't create any questions, but checks the message: "No polls are available." and verifies the **`latest_question_list`** is empty. Note that the **`django.test.TestCase`** class provides some additional assertion methods. In these examples, we use **`assertContains()`** and **`assertQuerysetEqual()`**.

In **`test_index_view_with_a_past_question`**, we create a question and verify that it appears in the list.

In **`test_index_view_with_a_future_question`**, we create a question with a **`pub_date`** in the future. The database is reset for each test method, so the first question is no

longer there, and so again the index shouldn't have any questions in it.

And so on. In effect, we are using the tests to tell a story of admin input and user experience on the site, and checking that at every state and for every new change in the state of the system, the expected results are published.

Testing the `DetailView`

What we have works well; however, even though future questions don't appear in the *index*, users can still reach them if they know or guess the right URL. So we need to add a similar constraint to **`DetailView`**:

polls/views.py

```
class DetailView(generic.DetailView):
    ...
    def get_queryset(self):
        """
        Excludes any questions that aren't published yet.
        """
        return Question.objects.filter(pub_date__lte=timezone.now())
```

And of course, we will add some tests, to check that a **`Question`** whose **`pub_date`** is in the past can be displayed, and that one with a **`pub_date`** in the future is not:

polls/tests.py

```
class QuestionIndexDetailTests(TestCase):
    def test_detail_view_with_a_future_question(self):
        """
        The detail view of a question with a pub_date in the future should
        return a 404 not found.
        """
        future_question = create_question(question_text='Future question.',
                                          days=5)
        response = self.client.get(reverse('polls:detail',
                                   args=(future_question.id,)))
        self.assertEqual(response.status_code, 404)

    def test_detail_view_with_a_past_question(self):
        """
        The detail view of a question with a pub_date in the past should
        display the question's text.
        """
        past_question = create_question(question_text='Past Question.',
                                        days=-5)
        response = self.client.get(reverse('polls:detail',
                                   args=(past_question.id,)))
        self.assertContains(response, past_question.question_text,
                            status_code=200)
```

Ideas for more tests

We ought to add a similar **`get_queryset`** method to **`ResultsView`** and create a new test class for that view. It'll be very similar to what we have just created; in fact there will be a lot of repetition.

We could also improve our application in other ways, adding tests along the way. For example, it's silly that **`Questions`** can be published on the site that have no **`Choices`**. So, our views could check for this, and exclude such **`Questions`**. Our tests would create a **`Question`** without **`Choices`** and then test that it's not published, as well as create a similar **`Question`** *with* **`Choices`**, and test that it *is* published.

Perhaps logged-in admin users should be allowed to see unpublished **`Questions`**, but not ordinary visitors. Again: whatever needs to be added to the software to accomplish this should be accompanied by a test, whether you write the test first and then make the code pass the test, or work out the logic in your code first and then write a test to prove it.

At a certain point you are bound to look at your tests and wonder whether your code is suffering from test bloat, which brings us to:

When testing, more is better

It might seem that our tests are growing out of control. At this rate there will soon be more code in our tests than in our application, and the repetition is unaesthetic, compared to the elegant conciseness of the rest of our code.

It doesn't matter. Let them grow. For the most part, you can write a test once and then forget about it. It will continue performing its useful function as you continue to develop your program.

Sometimes tests will need to be updated. Suppose that we amend our views so that only **`Questions`** with **`Choices`** are published. In that case, many of our existing tests will fail - *telling us exactly which tests need to be amended to bring them up to date*, so to that extent tests help look after themselves.

At worst, as you continue developing, you might find that you have some tests that are now redundant. Even that's not a problem; in testing redundancy is a *good* thing.

As long as your tests are sensibly arranged, they won't become unmanageable. Good rules-of-thumb include having:

- a separate **`TestClass`** for each model or view
- a separate test method for each set of conditions you want to test
- test method names that describe their function

Further testing

This tutorial only introduces some of the basics of testing. There's a great deal more you can do, and a number of very useful tools at your disposal to achieve some very clever things.

For example, while our tests here have covered some of the internal logic of a model and the way our views publish information, you can use an "in-browser" framework such as Selenium to test the way your HTML actually renders in a browser. These tools allow you to check not just the behavior of your Django code, but also, for example, of your JavaScript. It's quite something to see the tests launch a browser, and start interacting with your site, as if a human being were driving it! Django includes **`LiveServerTestCase`** to facilitate integration with tools like Selenium.

If you have a complex application, you may want to run tests automatically with every commit for the purposes of continuous integration, so that quality control is itself - at least partially - automated.

A good way to spot untested parts of your application is to check code coverage. This also helps identify fragile or even dead code. If you can't test a piece of code, it usually means that code should be refactored or removed. Coverage will help to identify dead code. See Integration with coverage.py for details.

Testing in Django has comprehensive information about testing.

What's next?

For full details on testing, see Testing in Django.

When you're comfortable with testing Django views, read part 6 of this tutorial to learn about static files management.

Writing your first Django app, part 4 **Writing your first Django app, part 6**

Contribute to Django

Submit a Bug

Report a Security Issue

Follow Us

GitHub

Twitter

News RSS

Django Users Mailing List