
بسم الله

TEST

HOSSEIN FORGHANI

MAKTAB SHARIF

Contents

- Writing Django tests
- What is test-driven development
- How to write tests for Django models and views?
- Introduction to Test Client in order to test views
- Integration test by Selenium

Introduction

- Why to write tests?
  - When you're writing new code, you can use tests to validate your code works as expected
  - When you're refactoring or modifying old code, you can use tests to ensure your changes haven't affected your application's behavior unexpectedly
- The preferred way to write tests in Django is using the unittest module
- You can also use any other Python test framework

Writing Tests

- The default startapp template creates a tests.py file in the new application
- This might be fine if you only have a few tests
- As your test suite grows you'll likely want to restructure it into a tests package so you can split your tests into different submodules such as test_models.py, test_views.py, test_forms.py, etc

Writing Tests – cont.

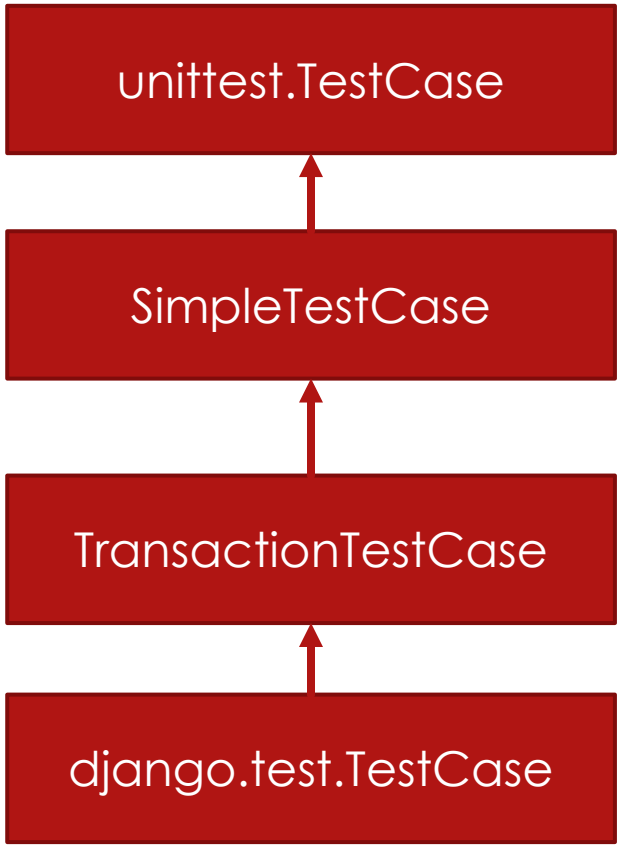
- Put something similar to this in tests.py :
- Subclass from django.test.TestCase, which is a subclass of unittest.TestCase
- Runs each test inside a transaction to provide isolation (Fixtures do not persist in DB after test)

```python
from django.test import TestCase
from myapp.models import Animal

class AnimalTestCase(TestCase):
    def setUp(self):
        Animal.objects.create(name="lion", sound="roar")
        Animal.objects.create(name="cat", sound="meow")

    def test_animals_can_speak(self):
        """Animals that can speak are correctly identified"""
        lion = Animal.objects.get(name="lion")
        cat = Animal.objects.get(name="cat")
        self.assertEqual(lion.speak(), 'The lion says "roar"')
        self.assertEqual(cat.speak(), 'The cat says "meow"')
```

Class Inheritance



A test for our polls example:

polls/tests.py

```
import datetime

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

from .models import Question


class QuestionModelTests(TestCase):

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

Running tests

```
$ ./manage.py test
```

- By default, this will discover tests in any file named "test*.py" under the current working directory

```
# Run all the tests in the animals.tests module
$ ./manage.py test animals.tests

# Run all the tests found within the 'animals' package
$ ./manage.py test animals

# Run just one test case
$ ./manage.py test animals.tests.AnimalTestCase

# Run just one test method
$ ./manage.py test
animals.tests.AnimalTestCase.test_animals_can_speak
```

The Test Database

- Tests that require a database, will not use your real database. Blank databases are created for the tests.
- The test databases are destroyed when all the tests have been executed
- You can keep it by using: test –keepdb
- If the DB is kept, on the next run, you'll be asked whether you want to reuse or destroy the database
- Use the test --noinput option to suppress that prompt and automatically destroy the database
- The test database is created by the user specified by USER, so you'll need to make sure that the given user account has sufficient privileges to create a new database on the system

Execution Order of Tests

- In order to guarantee that all TestCase code starts with a clean database, Django reorders tests in the following way:
  1. All TestCase subclasses
  2. All other Django-based tests (test cases based on SimpleTestCase, including TransactionTestCase) with no particular ordering guaranteed
  3. Any other unittest.TestCase tests that may alter the database without restoring it to its original state are run

Tests Run in Production Mode

- Regardless of the value of the DEBUG setting in your configuration file, all Django tests run with DEBUG=False
- This is to ensure that the observed output of your code matches what will be seen in a production setting

Understanding the Test Output

- You can control the level of detail of output with the --verbosity

```
Creating test database...
Creating table myapp_animal
Creating table myapp_mineral
```

- At last, If everything goes well, you'll see something like this:

```
----------------------------------
Ran 22 tests in 0.221s

OK
```

Understanding the Test Output – cont.

- If there are test failures, however, you'll see full details about which tests failed:

```
======================================================================
FAIL: test_was_published_recently_with_future_poll
(polls.tests.PollMethodTests)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/dev/mysite/polls/tests.py", line 16, in
test_was_published_recently_with_future_poll
    self.assertIs(future_poll.was_published_recently(), False)
AssertionError: True is not False

----------------------------------------------------------------------
Ran 1 test in 0.003s

FAILED (failures=1)
```

Test-driven Development

- We repeatedly write the test and then the code to fix it
- This is test-driven development
- But it doesn't really matter in which order we do the work

Fixing the bug

- Remember the test for our polls example
- Question.was_published_recently() should return False if its pub_date is in the future:

polls/models.py

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

polls/tests.py

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

def test_was_published_recently_with_recent_question(self):
    """
    was_published_recently() returns True for questions whose
pub_date
    is within the last day.
    """
    time = timezone.now() - datetime.timedelta(hours=23, minutes=59,
seconds=59)
    recent_question = Question(pub_date=time)
    self.assertIs(recent_question.was_published_recently(), True)
```

More tests for our polls example in QuestionModelTests test case class

Test a view

- Django provides a test Client to simulate a user interacting with the code at the view level
- We can access it by self.client in TestCase methods
- To simulate a GET request to a URL:

```
response = self.client.get(reverse('polls:index'))
```

- We can then check the status_code, context, … of the response
- Also we have post() method

Testing our Index View

- Remember our class-based IndexView:

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]
```

Testing our Index View – cont.

- In tests.py we'll create a shortcut function to create questions:

polls/tests.py

```
def create_question(question_text, days):
    """
    Create 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)
```

Testing our Index View – cont.

And a new test class:

```
class QuestionIndexViewTests(TestCase):
    def test_no_questions(self):
        """
        If no questions exist, an appropriate message is 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_past_question(self):
        """
        Questions with a pub_date in the past are 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.>']
        )
```

Testing our Index View – cont.

```
    def test_future_question(self):
        """
        Questions with a pub_date in the future aren't 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.")
        self.assertQuerysetEqual(response.context['latest_question_list'],
[])

    def test_future_question_and_past_question(self):
        """
        Even if both past and future questions exist, only past questions
        are 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.>']
        )
```

These tests will fail because our view lists future questions as well as past questions

Testing our Index View – cont.

```
def test_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.>']
    )
```

Fixing the Bug

- Now we'll fix the bug:
  - The list of polls shows polls that aren't published yet (i.e. those that have a pub_date in the future)

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]
```

Testing the DetailView

- Remember our DetailView:

```
class DetailView(generic.DetailView):
    model = Question
    template_name = 'polls/detail.html'
```

- We should then 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 QuestionDetailViewTests(TestCase):
    def test_future_question(self):
        """
        The detail view of a question with a pub_date in the future
        returns a 404 not found.
        """
        future_question = create_question(question_text='Future question.',
days=5)
        url = reverse('polls:detail', args=(future_question.id,))
        response = self.client.get(url)
        self.assertEqual(response.status_code, 404)

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


Will fail!

Fixing the Bug

- So we need a change in 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())
```

When testing, more is better

- It might seem that our tests are growing out of control
- It doesn't matter!
- For the most part, you can write a test once and then forget about it
- in testing redundancy is a good thing

Django Test Rules-of-thumb

1. A separate TestClass for each model or view
2. A separate test method for each set of conditions you want to test
3. Test method names that describe their function

Further testing

- We introduced some of the basics of testing
- you can use an "in-browser" framework such as Selenium to test the way your HTML actually renders in a browser
- to check not just the behavior of your Django code, but also, for example, of your JavaScript
- Django includes LiveServerTestCase to facilitate integration with tools like Selenium

LiveServerTestCase

- First install Selenium:

```
$ python -m pip install selenium
```

- Then write a test like what you see in the next page

```python
from django.contrib.staticfiles.testing import StaticLiveServerTestCase
from selenium.webdriver.firefox.webdriver import WebDriver

class MySeleniumTests(StaticLiveServerTestCase):
    fixtures = ['user-data.json']

    @classmethod
    def setUpClass(cls):
        super().setUpClass()
        cls.selenium = WebDriver()
        cls.selenium.implicitly_wait(10)

    @classmethod
    def tearDownClass(cls):
        cls.selenium.quit()
        super().tearDownClass()

    def test_login(self):
        self.selenium.get('%s%s' % (self.live_server_url, '/login/'))
        username_input = self.selenium.find_element_by_name("username")
        username_input.send_keys('myuser')
        password_input = self.selenium.find_element_by_name("password")
        password_input.send_keys('secret')
        self.selenium.find_element_by_xpath('//input[@value="Log in"]').click()
```

LiveServerTestCase – cont.

- Finally, you may run the test as follows:

```
$ ./manage.py test myapp.tests.MySeleniumTests.test_login
```

References

- https://docs.djangoproject.com/en/3.1/intro/tutorial05/
- https://docs.djangoproject.com/en/3.1/topics/testing/overview/
- https://docs.djangoproject.com/en/3.1/topics/testing/tools/

Any Question?