Django Testing Docs Documentation

Release 0.01

Eric Holscher

May 19, 2020

Contents

This is a place to help contribute to Django's docs about testing. It is hosted on Github.

If you want to help contribute, please fork the docs and help out! Hopefully this will make it's way back into the Django documentation at some point.

If you have any questions, please contact Eric Holscher at eric@ericholscher.com

Contents:

CHAPTER 1

Mostly Complete docs

1.1 Introduction to Python/Django testing: Basic Doctests

This is the first in a series of articles that will walk you through how to test your Django application. These posts will focus more on how to get things done in Django, but note that a lot of the content is applicable to pure Python as well. A lot of best practices are codified into Django's testing framework, so that we don't have to worry about them! I will try to point them out as we are using them through, because they are good things to know.

The currently planned structure for this series is below. Please comment if there is something that you think is missing, or something that I shouldn't do. This is subject to change, a lot, as well, so your feedback will help direct it. Also note that most or all of this content is available in the Django and Python documentation, and I will try and point there and not reinvent the wheel. I hope that these posts will take a more practical look, and try to point out some pit falls and other things that are useful.

1.1.1 Where to start

I'm assuming that you already have a project that you're working on that you would like to test. There are two different ways of putting tests inside of your django project. You can add a tests.py file and put your tests inside of there. You can also define a tests/ directory and put your tests in files inside of that. For these tutorials it is assumed that the second is the way things are done. It makes it a lot easier when you can break your tests out into logical files.

1.1.2 Doctests

These can go in two places inside your django project. You can put them in your models.py file, in the Docstring for your modules. This is a good way to show usage of your models, and to provide basic testing. The official docs have some great examples of this.

The other place your Doctests can go is inside your tests directory. A doctests file is usually pretty simple. A doctest is just a large string, so there isn't much else to put besides a string. Usually you want to use the triple quote, multi-line string delimiter to define them. That way your " and 's inside of your doctests don't break.:

```
"""
This is my worthless test.
>>> print "wee"
wee
>>> print False
False
"""
```

You can go ahead and put that in a file in your `tests/` directory, I named it `doctst.py`. I didn't name it doctest, because of the Python module with the same name. It's generally good to avoid possible name overlaps. My application that I'm writing tests for is `mine`, because it's the code for my website. Make sure that directory has an `__init__.py` as well, to signify that it is a Python module.

Now here is the tricky part; go ahead and try and run your test suite. In your project directory run `./manage.py test APPNAME`. It will show you that you have passed 0 tests. 0 tests? We just defined one.

You need to go into your `__init__.py` file and put some stuff in there.:

```
import doctest
__test__ = {
    'Doctest': doctest
}
```

You are importing the doc test module and then adding it to the `__test__` dictionary. You have to do this because of the way that Python handles looking for doc tests. It looks for a `__test__` dictionary inside of your module, and if that exists it looks through it, executing all docstrings as doctests. For more information look at the Python docs.

Now you should be able to go ahead and run the tests and see the magical `Ran 1 test in 0.003s OK` that all testers live for. This is little bit of overhead really threw me off when I was trying to break my tests.py out into the tests/ directory. Notice that the doc test runner sees all of your tests as one single test. This is one annoying thing that the doctests do.

So now we have a test suite that is worthless, but you know how to use doc tests. If you didn't notice, the doctest format is simply the output of your default Python shell, so when you are testing your code on the command line and it works, you can simple copy and paste it into your tests. This makes writing doc tests almost trivial. Note however, that they are somewhat fragile, and shouldn't be used for everything. In the next segment, we will talk about unit tests. Then we will compare the two and see what the use cases are for each.

1.1.3 Doctest Considerations

Keep in mind that when the test runner executes a doctest, it does not process the output in any way. For instance, this doctest will fail:

```
>>> {"abc": 1, "def": 2}
{"abc": 1, "def": 2}
```

The reason for the failure is that the Python interpreter always, when displaying dictionaries, converts double quotes into single quotes. This, on the other hand, will pass:

```
>>> {"abc": 1, "def": 2}
{'abc': 1, 'def': 2}
```

Also, if you are using the popular Python shell replacement IPython to aide in creating doctests, keep in mind it will not necessarily output objects the same way as the vanilla Python interactive interpreter. Example:

```
In [1]: {'key3': 'fff', 'key2': 123}
Out[1]: {'key2': 123, 'key3': 'fff'}
```

The same output with the vanilla Python shell:

```
>>> {'key3': 'fff', 'key2': 123}
{'key3': 'fff', 'key2': 123}
```

Notice the difference between the order of the keys.

1.2 Introduction to Python/Django testing: Basic Unit Tests

Last post we talked about how to set up and use doc tests inside of Django. Today, in the second post of the series, we'll be talking about how to use the other testing framework that comes with Python, unittest. unittest is a xUnit type of testing system (JUnit from the Java world is another example) implemented in Python. It is a much more robust solution for testing than Doc tests, and allows for a lot more organization of code. We'll get into that in the next post in the series, comparing unit and doc tests.

So we're going to assume that you are picking up after the previous post in this series. If so, you should have a basic tests directory, with an `__init__.py` and a `doctst.py` file inside of it. Today we are going to write some very basic unit tests, and figure out how to wire those into your existing test suite.

1.2.1 Writing your first unit test

Making a unit test is a lot like making a Python class. As per usual, the Django docs have lots of great information and examples. They will show you how to do some easy stuff with your Django models. This tutorial will mostly be about how to use unit tests inside Django, regardless of the data at hand. So let's start with a very basic unit test.:

```
import unittest

class TestBasic(unittest.TestCase):
    "Basic tests"

    def test_basic(self):
        a = 1
        self.assertEqual(1, a)

    def test_basic_2(self):
        a = 1
        assert a == 1
```

This is a very basic unit test. You will notice it is just a normal Python class. You create a class that inherits from unittest.TestCase. This tells unittest that it is a test file. Then you simply go in and define some functions (Note: they need to start with test so that unittest will run them), in which you assert some conditions which are true. This allows you a lot more flexibility in the tests.

Now if you try to run these tests, you will again not see them showing up in your output! You need to go into your `__init__.py` in your tests directory. It should now look like the following (assuming you followed part 1 of this series):

```
from unittst import *

import doctst
```

(continues on next page)

(continued from previous page)

```
__test__ = {
    'Doctest': doctst
    }
```

Unit tests are a lot easier to import than doctests. You simply do a `from <filename> import <testnames>`. I named my unit test file `unittst.py`, and Python will import that from the current directory. You are importing the test classes that you defined in your file. So I could have as easily put `from unittest import TestBasic` and it would work. Using the `import *` syntax allows us to add more tests to the file later and not have to edit it.

You can go ahead and run your tests, and see if they're being properly imported.:

```
[eric@Odin:~/EH]$ ./manage.py test mine
Creating test database...
Creating table auth_permission
[Database stuff removed]
...
Ran 3 tests in 0.004s

OK
```

Awesome!

1.2.2 A couple neat features

There are some neat things you can do with basic unit tests. Below I'll show an addition to the above file, which is another test class, with a bit more functionality.:

```
class TestBasic2(unittest.TestCase):
    "Show setup and teardown"

    def setUp(self):
        self.a = 1

    def tearDown(self):
        del self.a

    def test_basic1(self):
        "Basic with setup"

        self.assertNotEqual(self.a, 2)

    def test_basic2(self):
        "Basic2 with setup"
        assert self.a != 2

    def test_fail(self):
        "This test should fail"
        assert self.a == 2
```

Here you see that you can define a docstring for the tests. These are used when you are running the tests, so you have a human readable name. You'll also notice that I've used some more assertions. The Python docs have a full list of assertions that you can make. The `setUp` and `tearDown` methods are run before and after every test respectively. This allows you to set up a basic context or environment inside of each of your tests. This also insures that each of your tests do not edit the data that other tests depend on. This is a basic tenet of testing, that each test should stand alone, and not affect the others.

This also seems like a good time to explicitly say that all of your test classes and files should start with test! If not, they will not be run! If you have a test not running and everything else looks right, this is probably your problem. Also note that they cannot be named the same thing! These will overwrite one another with the last one being imported into the file running. It is generally a good practice to name your tests something that is certain to be unique. I generally tend to follow whatever naming convention I've used for my named url patterns.

When you go ahead and run your tests, you should see one that fails (the last one).:

```
[eric@Odin:~/EH]$ ./manage.py test mine
Creating test database...
Creating table auth_permission
[Database stuff removed]
....F.
=====================================================
FAIL: This test should fail
Traceback (most recent call last):
  File "/home/eric/Python/EH/mine/tests/unittst.py", line 35, in
  test_fail
    assert self.a == 2
AssertionError

Ran 6 tests in 0.003s

FAILED (failures=1)
```

You can see the value of unit tests here. Each test is run seperately, so you get a nice human readable error message when it breaks. You can go ahead and make that test pass (`self.assertFalse(self.a == 2)`). You get an OK from your tests, and we can go on our merry way.

Now you can see for yourself that there are a lot of differences between Doc tests and unit tests. They each serve their own purpose, and in the next post in this series I will talk about when you should use each. Unit tests require a little bit more up front effort; you can't just paste something out of your Python shell and have it work. However, they give you a lot more flexibility.

1.3 Introduction to Python/Django tests: Fixtures

In the first two posts of this series, we talked about how to get the basic infrastructure for your tests up and running. You should have a file with doc tests and one with unit tests. They should be linked into your django project with an `__init__.py` file. If you were feeling adventurous you may have even added some real content to them. Today we're going to start going down the road of getting some data into your tests.

This post will cover fixtures. Fixtures are how you create a state for your database in your tests. This will be my first post in the comparison between unit tests and doc tests. I will focus on fixtures, but some other differences between the two may become relevant throughout the post, and I will address them in turn.

1.3.1 How to create a fixture

Fixtures can go into a couple places, but generally it is a good idea to put them in your applications `fixtures/` directory. This makes all of your test data self contained inside your app, so it can be run when it is distributed. The loaddata command discussed further down specifies where fixtures can be placed to be loaded, if you're curious.

Before we go about trying to figure out how to use fixtures, we need to know how to make them. Django's docs on this are pretty good. Basically if you have an app that has data in the database, you can use the `./manage.py dumpdata <app>` command to create a fixture from the current data in the database. It's handy to note that you can use the `--format` tag to specify the output format, and the `--indent` command to make the output prettier. My preferred command is:

```
#This assumes you are at the project level, right above your app.
 #and that APP/fixtures/ exists
 ./manage.py dumpdata APP --format=yaml --indent=4 >
 APP/fixtures/APP.yaml
```

This makes for a really nice, readable fixture, so if you need to edit it later you can. Go ahead and run this command in your project directory, substituting your app in the appropriate places. Open the fixture if you want and take a peak inside. It should be a nice readable version of your database, serialized into Yaml. **Note**: If you don't have PyYAML installed, it will say that your serialization format isn't valid, `sudo apt-get install python-yaml` gets you the package on Ubuntu. If not, you can remove the format option and it will default to JSON.

1.3.2 Testing your fixtures (how meta of us!)

Django also comes with a really neat tool to be able to test and update fixtures. The testserver command allows you to run the development server, passing a fixture to load before it launches. This allows you to run your code base against the fixture that you have, in a browser.

This seems really nice, but the killer feature of this command is that it keeps the database around after you kill the development server. This means that you can load up a fixture, edit it in the admin or frontend, and then kill the server; then run dumpdata against that database and get your updated fixture back out. Pretty neat! Note, your normal database name will be prefixed with `test_`, so it doesn't overwrite your normal DB. This is the one you want to get data out of. (You may have to define it in your `settings.py` file to get dumpdata to use it. This seems like a little bit of a hack, and maybe something could be done to make this easier.)

1.3.3 Fixtures in Doc tests

In what will become a recurring trend, doing fixtures in doc tests is a hack. Doc tests are made to be a simple answer to a relatively simple problem, and fixtures aren't a huge deal for them. So a lot of the functionality that we get for free with unit tests, has to be hacked into doc tests. I will just show how to do the basic things, because implementing anything beyond that isn't very useful for any of us.:

```
>>> from django.core.management import call_command
>>> call_command("loaddata", "' + 'fixturefile.json' + '",
verbosity=0)
```

In this snippet you are basically calling it the way it is called within Django. Normally when you are using loaddata, you would be calling it as `./manage.py loaddata FIXTURE`. Note that the loaddata docs talk about how to use loaddata normally. There are a couple of downsides to this method; The test is very fragile, if the fixture breaks, all of your tests fail. Also, you can really only call one fixture at a time because there is no setUp and tearDown that will make sure your environment is the same for every test. Doing things this way just makes writing tests a whole lot harder. It is indeed a hack, and one that shouldn't really be used unless you have a very good reason.

Generally in doc tests, you would create your content as if you were on the command line. This shows how doc tests are generally limited in their scope. You go ahead and create the objects that you care about in the test explicitly, and then run your tests against them. A simple example:

```
>>> from mine.models import Site
>>> s = Site.objects.create(url='http://google.com', query='test',
title='test', content='lots of stuff')

>>> s.query
```

(continues on next page)

(continued from previous page)

```
'test'
>>> s.save()
>>> pk_site = s.pk
>>> Site.objects.get(pk=pk_site)
<Site: test>
>>> Site.objects.get(pk=pk_site).delete()
```

This tests creating, retrieving and deleting an object. Not a lot of functionality, but if anything inside of the model saving code breaks you will know about it.

1.3.4 Django's Testcase

The fixture story in unit tests is much better, as you would expect. However, before we go into how unit tests use fixtures, there is something that I need to explain. Because of the fact that unit tests are classes, they can be subclassed just like any other Python class. This means that Django has provided it's own Testcase class that we can inherit from and get some nice extra Django functionality. The official docs has it really well documented.

You'll notice that Django's Testcase has a section for the Test Client and URLConf configuration. We can safely skip those for the moment because they are geared towards testing views. The relevant sections for us at the moment are Fixture loading and Assertions. I recommend that you actually read the entire testing doc, it isn't that long, and is packed full of useful information. However, knowing about all of the assertions that are available to you will make testing a little bit easier.

1.3.5 Fixtures in Unit Tests

The big thing that the Django Testcase does for you in regards to fixtures is that it maintains a consistent state for all of your tests. Before each test is run, the database is flushed: returning it to a pristine state (like after your first syncdb). Then your fixture gets loaded into the DB, then setUp() is called, then your test is run, then tearDown() is called. Keeping your tests insulated from each other is incredibly important when you are trying to make a good test suite. Having tests altering each others data, or having tests that depend on another test altering data are inherently fragile.

Now lets talk about how you're actually going to use these fixtures. We're going to go ahead and recreate the simple doc test above. It simply loads up a Site object into the database, checks for some data in it, then deletes it. The fixture handling will handle all of the loading and deleting for us, so all we need to worry about is testing our logic! This makes the test a lot easier to read, and makes its intention a lot clearer.:

```
from django.test import TestCase
from mine.models import Site

class SiteTests(TestCase):
    #This is the fixture:
    #-   fields: {content: lots of stuff, query: test, title:
    test, url: 'http://google.com'}
    #model: mine.site
    #pk: 1
    fixtures = ['mine']

    def testFluffyAnimals(self):
        s = Site.objects.get(pk=1)
        self.assertEquals(s.query, 'test')
        s.query = 'who cares'
        s.save()
```

As you can see, this test is a lot simpler than the above one. It is also neat that we can edit the object and save it, and it doesn't matter. No other tests (if they existed) would be affected by this change. Notice that in my fixtures list, I only had mine and not mine.yaml or mine.json. If you don't add a file extension to your fixture, it will search for all fixtures with that name, of any extension. You can define an extension if you only want it to search for those types of files.

I hope that you can see already how unit tests give you a lot more value when working with fixtures than doc tests. Having all of the loading, unloading, and flushing handled for you means that it will be done correctly. Once you get a moderately complicated testing scheme, trying to handle that all yourself inside of a doc test will lead to fragile and buggy code.

1.4 Practical Django Testing Examples: Views

This is the fourth in a series of Django testing posts. Today is the start of a sub-series, which is practical examples. This series will be going through each of the different kinds of tests in Django, and showing how to do them. I will also try to point out what you want to be doing to make sure you're getting good code coverage and following best practices.

Instead of picking some contrived models and views, I figured I would do something a little bit more useful. I'm going to take one of my favorite open source Django apps, and write some tests for it! Everyone loves getting patches, and patches that are tests are a god send. So I figured that I might as well do a tutorial and give back to the community at the same time.

So I'm going to be writing some tests for Nathan Borror's Basic Blog. It also happens to be the code that powers my blog here, with some slight modifications. So this is a win-win-win for everyone involved, just as it should be.

Nathan's app has some basic view testing already done on it. He was gracious enough to allow me to publicly talk about his tests. He claims to be a designer, and not a good coder, but I know he's great at both. So we're going to talk about his view testing today, and then go ahead and make some Model and Template Tag tests later.

1.4.1 Basic philosophy

Usually when I go about testing a Django application, there are 3 major parts that I test. Models, Views, and Template Tags. Templates are hard to test, and are generally more about aesthetics than code, so I tend not to think about actually testing Templates. This should cover most of the parts of your application that are standard. Of course, if your project has utils, forms, feeds, and other things like that, you can and should probably test those as well!

1.4.2 Views

So lets go ahead and take a look to see what the tests used to look like. He has already updated the project with my new tests, so you can check them out, and break them at your leisure.:

```
"""
>>> from django.test import Client
>>> from basic.blog.models import Post, Category
>>> import datetime

>>> from django.urls import reverse
>>> client = Client()

>>> response = client.get(reverse('blog_index'))
>>> response.status_code
200
```

(continues on next page)

(continued from previous page)

```
>>> response = client.get(reverse('blog_category_list'))
>>> response.status_code
200
>>> category = Category(title='Django', slug='django')

>>> category.save()
>>> response = client.get(category.get_absolute_url())
>>> response.status_code
200

>>> post = Post(title='My post', slug='my-post', body='Lorem ipsum
dolor sit amet', status=2, publish=datetime.datetime.now())
>>> post.save()
>>> post.categories.add(category)


>>> response = client.get(post.get_absolute_url())
>>> response.status_code
200
"""
```

Notice how he is using reverse() when referring to his URLs, this makes tests a lot more portable, because if you change your URL Scheme, the tests will still work. A good thing to note is that a lot of best practices that apply to coding apply to testing too! Then the tests go on to create a Category, save it, and then test it's view and get_absolute_url() method. This is a really clever way of testing a view and a model function (get_absolute_url) at the same time.

Next a post is created, and saved, then a category is added to it, the one created above. That is all that these tests do, but it covers a really good subsection of the code. It's always good to test if you can save your objects because a lot of bugs are found in that operation. So for the length of the code it is remarkably well done.

This is a pretty simple test suite at the moment. But the fact that he has tests is better than 90% of other open source projects! I'm sure if we asked Nathan, he would tell us that even this simple test suite helps a ton. Most of the bugs people make break in very loud and obvious ways. Which just goes to emphasize my point that everything should have tests, even if they're simplistic.

So how are we going to improve this testing of views? First of all, note that these tests are hardly touching models, and not testing any template tags; this will be addressed later. In regard to views, these tests aren't checking the context of the responses, they are simply checking status code. This isn't really testing the functionality of the view, just testing if it doesn't break. There are also some views that aren't being touched, like search, pagination, and the date archive views. We aren't going to test pagination because we don't have enough data to paginate. This brings me to a meta subject, slight tangent time.

1.4.3 Do I test Django stuff?

So we have some Generic views in our application, should we test them? I don't think that there is a correct answer to this question, but I have an opinion. I think that you should test generic views, but only in the ways that they can break based on how you define them. This doesn't look much different than normal tests that you should be writing anyway.

For the date-based generic view for example, you are passing in a QuerySet and DateField in the URLConf; and the parts of the date you're using in the actual URLs. So what is the easiest way to test that all of these things are being set correctly? Find the most specific example and test for it. So you would test the context and response code of blog_detail page, because it has to use the query set, the date field, and the full path for the URLs. Assuming that your code isn't broken in some horrible way, that means that all the other parts of the date URLs should work.

1.4.4 Let's write some tests

So we need to add some stuff to the tests. We need to get some data into the tests, in order to use the date-based archives, and search stuff. So we're going to take the stuff that was previously at the bottom of the test, and move it up to the top. Also need to add 2 posts and categories, so that we know that our filtering functionality is working.:

```
>>> category = Category(title='Django', slug='django')
>>> category.save()

>>> category2 = Category(title='Rails', slug='rails')
>>> category2.save()
>>> post = Post(title='DJ Ango', slug='dj-ang', body='Yo DJ! Turn
that music up!', status=2, publish=datetime.datetime(2008,5,5,16,20))
>>> post.save()

>>> post2 = Post(title='Where my grails at?', slug='where', body='I
Can haz Holy plez?', status=2, publish=datetime.datetime(2008,4,2,11,11))
>>> post2.save()
>>> post.categories.add(category)
>>> post2.categories.add(category2)
```

Pretty obvious what this test is doing. If these tests were going to be much more complicated than this, it would make a lot of sense to write a fixture to store the data. However I'm trying to test the saving functionality (which is technically a model thing), so it's good to make the objects inline.

So now we have our data, and we need to do something with it. Let's go ahead and run the test suite to make sure that we haven't done anything stupid. It's a tenet of Test Driven Development to test after every change, and one that I picked up from that philosophy. It's really handy. I don't do it on a really granular level like it suggests, but I try to do it after any moderately important change.

1.4.5 Getting into context

So we have the tests that were there before, and they're fine. They perform a great function, so we should keep them around, we just need to add some stuff to them. This is one of the reasons I really don't like doctests. Using unit tests you can just throw an `import pdb; pdb.set_trace()` in your code and it will drop you into a prompt, and you can easily use this to write new tests. Doctests however hijack the STDOUT during the tests, so when I drop into pdb with a `>>> import pdb; pdb.set_trace()` in the test, i can't see the output, so it's hard for me to get testing information.

Note: You can also do this by changing your settings file database (because otherwise these objects would be created in your real DB), running syncdb, running `s/>>> //` on your test, adding a setup_test_environment() import and call to the test, and running `python -i testfile`, if you want. But do you really want to do that?

Let's go poking around inside of response.context, which is a dictionary of contexts for the response. We only care about [-1], because that is where our context will be (except for generic views, annoying right?). So go down to the first view, `blog_index`, and put:

```
>>> response = client.get(reverse('blog_index'))

>>> response.context[-1]['object_list']
[test]
```

In your tests. We know [test] won't match, but we just want to know what the real output is. When you go ahead and run the tests your should find some output like this:

```
Expected:
    [test]
Got:

    [<Post: DJ Ango>, <Post: Where my grails at?>]
```

So go ahead and put in the correct information in where [test] was. This is a really annoying way of testing, and I'm going to repeat that this is why doc tests are evil, but we're already this far, so let's push on. Writing tests this way requires the tester to be vigilant, because you're trusting that the code is outputting the correct value. This is kind of nice actually, because it forces you to mentally make sure that your tests are correct, and if you're code isn't outputting what you expect, then you've already found bugs, just by writing the tests ;) But if you're testing code that's complex, this method breaks down, because you don't know if the output is correct!

If you look in the context, you'll see lots of other things that we could test for as well. Some that Django (oh so nicely) gave us, and other stuff that is user defined. Things like pagination, results per page, and some other stuff that we really don't care about. The object_list on the page is really what we're after, so we can move on. Run your tests to be sure, and lets move on.

1.4.6 Updating current tests

Now that we have our hackjob way of getting data out of the tests, we can move on to writing more tests. Go down to the next view test of `blog_category_list`, and pull the old object_list trick. You should get the following back out once you run your tests:

```
Expected:
     [test]
 Got:
     [<Category: Django>, <Category: Rails>]
```

This looks correct, so lets go ahead and put that in the test. As you can see, for this simple stuff, it isn't really a huge deal doing testing this way. The test suite runs in about 3 seconds on my machine, so it's not a huge hurdle.

Let's go ahead and do it for the category and post detail pages. When I don't remember or don't know what variables we'll be looking for in the context, I usually just put `>>> request.context[-1]` to output all of it, and see what it is that I want. For the `category.get_absolute_url()` we need `object_list` again. For the `post.get_absolute_url()` we just want `object`.:

```
>>> response = client.get(category.get_absolute_url())
>>> response.context[-1]['object_list']
[<Post: DJ Ango>]
>>> response.status_code
200

>>> response = client.get(post.get_absolute_url())
>>> response.context[-1]['object']

<Post: DJ Ango>
>>> response.status_code
```

We can consider those views tested now.

1.4.7 Creating new tests

So now we've improved on the tests that were already there. Let's go ahead and write some new ones for search and the date-based views. Starting with search, because it will be interesting. Search requires some GET requests with the test client, which should be fun.:

```
>>> response = client.get(reverse('blog_search'), {'q': 'DJ'})

>>> response.context[-1]['object_list']
[<Post: DJ Ango>]

>>> response.status_code
200
>>> response = client.get(reverse('blog_search'), {'q': 'Holy'})

>>> response.context[-1]['object_list']
[<Post: Where my grails at?>]

>>> response.status_code
200
>>> response = client.get(reverse('blog_search'), {'q': ''})

>>> response.context[-1]['message']
'Search term was too vague. Please try again.'
```

As you can see, we're testing to make sure that search works. We're also testing the edge case of a blank search, and making sure this does what we want. A blank search could return everything, nothing, or an error. The correct output is an error, so we go ahead and check for that. Notice that you pass GET parameters in the test client as a dictionary after the URL, and passing them as `?q=test` on the URL wouldn't work. Russ is working on fixing that, and by the time you read this, it might not be true.

Next, on to testing the generic date views. You should be in the hang of it by now.:

```
>>> response = client.get(reverse('blog_detail', args=[2008, 'apr', 2, 'where']))

>>> response.context[-1]['object']
<Post: Where my grails at?>

>>> response.status_code
200
```

Notice here that we're using the args on reverse, and not using get parameters. We're passing those arguments as positional into the view. You can also use kwargs={'year': '2008'} if you want to be more explicit. As talked about above, I feel that this is enough of testing for the generic views.

Wow! That was a long post. I'm glad I decided to split the testing up into separate posts! I hope this has been enlightening for everyone, and I'm sure that I'm doing it wrong in some places. I would love some feedback, and to hear how you work around and solve some of the problems above. Also your thoughts on this kind of stuff.

Nathan has graciously included my new tests in his project, if you want to see them live, or check them out.

1.5 Serializers and Deserializers

The `manage.py dumpdata` or `manage.py loaddata` commands can translate the contents of your database into YAML, XML, or some other format using a **serializer**. There may come a time when the built-in serializers do not meet your needs, and you find yourself writing a custom serializer (and probably a custom **deserializer** too). How can you go about testing these?

There are two basic scenarios:

Serialization Dump database objects to whatever format you want `dumpdata` to output

Deserialization Loading database objects from your custom format, like `loaddata` does

You could test each of these separately, or you could do a full round-trip test that mimics `dumpdata` followed by `loaddata`. We'll test them separately in this example. We won't be testing the `manage.py` command itself; this is strictly a unit test on the custom serializer and/or deserializer.

There are many different levels of detail you could get into; we'll cover only a very simple scenario, involving a single model with a single string field. A more comprehensive test would include multiple models, relationships between and within models, and a wide variety of model field types and data. This example covers only a single model, with a single string field; it's intended only to give you a basic starting point. It's up to you how rigorously you want to test your serializer and deserializer.

1.5.1 What we're testing

If you've written your own serializer, you've probably added something like this to your `settings.py`:

```
SERIALIZATION_MODULES = {
    'yaml': 'myapp.custom_yaml',
}
```

In this example, we're using a customized YAML serializer. Anytime Django needs to dump or load `yaml` format, it'll use `myapp/custom_yaml.py` to do it.

This example serializer uses a slightly different structure than Django's built-in YAML serializer. It's designed to be significantly less verbose, while still capturing all the relevant data. You may be familiar with Django's usual YAML output, which looks something like this:

```
- fields: {name: green}
  model: myapp.color
  pk: 1
- fields: {name: blue}
  model: myapp.color
  pk: 2
- fields: {name: red}
  model: myapp.color
  pk: 3
- fields: {name: squishy}
  model: myapp.texture
  pk: 1
- fields: {name: crumbly}
  model: myapp.texture
  pk: 2
```

This serializer uses a more condensed format:

```
myapp.color:
  1: {name: green}
  2: {name: blue}
  3: {name: red}
myapp.texture:
  1: {name: squishy}
  2: {name: crumbly}
```

We won't go into the details of the serializer itself; all we're concerned with here is how make sure it produces correct output, and that the corresponding deserializer loads it correctly afterwards.

1.5.2 Test the serializer

To test the serializer, we will create some model instances, then serialize them and make sure the output is correct. The `django.core.serializers` module defines a `serialize` function that takes the name of the format you want to serialize, along with a `QuerySet` of objects to serialize, and returns a string of serialized data. That's what we'll use in our test:

```
from django.test import TestCase
from django.core import serializers
from myapp.models import Color

class YamlSerializerTest (TestCase):
    def test_serializer(self):
        # Stuff to serialize
        Color(name='green').save()
        Color(name='blue').save()
        Color(name='red').save()

        # Expected output
        expect_yaml = \
            'myapp.color:\n' \
            '  1: {name: green}\n' \
            '  2: {name: blue}\n' \
            '  3: {name: red}\n'

        # Do the serialization
        actual_yaml = serializers.serialize('yaml', Color.objects.all())

        # Did it work?
        self.assertEqual(actual_yaml, expect_yaml)
```

Notice that we pass `'yaml'` as the first argument to `serialize`; ordinarily this would use the default YAML serializer, but since we've overridden that in `SERIALIZATION_MODULES`, it'll use our custom one instead.

Since we're working with standard YAML, another way to verify the result is to parse it using `yaml.load`, and check that the resulting Python data structure (in this case, a `dict`) matches expectations:

```
class YamlSerializerTest (TestCase):
    def test_serializer(self):
        # ...

        self.assertEqual(
            yaml.load(actual_yaml), {
                'myapp.color': {
                    1: {'name': 'green'},
                    2: {'name': 'blue'},
                    3: {'name': 'red'},
                }
            }
        )
```

Of course, if you're serializing to your own made-up custom format, you may not have a standalone parser for that format so readily available. In that case, you may simply choose to rely on your deserializer tests to ensure that the output is parsed correctly.

1.5.3 Test the deserializer

Unless your serializer is designed for one-way conversion, you'll want to include some tests for your deserializer as well. Starting with the serialized text output, we'll make sure that it gets loaded into the database and correctly builds the original models.

The `django.core.serializers` module includes a counterpart to the `serialize` function called (you guessed it) `deserialize`. This function accepts a format (like `yaml`), along with a chunk of text to deserialize. It returns a generator that yields each of the objects as they are parsed. We'll convert these into a list, then verify that the deserialized objects have the correct values in their fields:

```
class YamlSerializerTest (TestCase):
    def test_deserializer(self):
        # Input text
        input_yaml = \
            'myapp.color:\n' \
            '  1: {name: green}\n' \
            '  2: {name: blue}\n' \
            '  3: {name: red}\n'

        # Deserialize into a list of objects
        objects = list(serializers.deserialize('yaml', input_yaml))

        # Were three objects deserialized?
        self.assertEqual(len(objects), 3)

        # Did the objects deserialize correctly?
        self.assertEqual(objects[0].object.name, 'green')
        self.assertEqual(objects[1].object.name, 'blue')
        self.assertEqual(objects[2].object.name, 'red')
```

Perhaps this isn't the most elegant way to do it, but it gets the job done.

1.5.4 References

Several custom serializers are available on djangosnippets.org, including json and csv serializers. The slightly better YAML serializer was used as the basis for the examples above. For a much more thorough serializer test suite, please consult Django's regression tests.

1.6 Measuring Coverage

After you've written some tests for your Django app, and gotten them all to pass, you may wonder "Do I have enough tests? Am I missing anything?" One way to help yourself answer that question is to measure the **coverage** of your tests–that is, how thoroughly your tests exercise the application's code.

Perhaps the most popular tool for measuring coverage in Python is simply called coverage. While your tests are running, it keeps track of which lines of application code are executed, which ones are skipped (like comments), and which ones are never reached. At the end, it spits out a report that indicates which lines of code were not executed–this points directly to holes in your test coverage.

The nose testing tool integrates nicely with coverage, and django-nose ties it all into Django. This chapter will give an overview of how to get it working.

1.6.1 Configure django-nose

The first thing to do is install django-nose using `pip`:

```
$ pip install django-nose
```

Then make these additions to your project's `settings.py`:

```
INSTALLED_APPS = (
    # ...
    'django_nose',
)

# Use nose to run all tests
TEST_RUNNER = 'django_nose.NoseTestSuiteRunner'

# Tell nose to measure coverage on the 'foo' and 'bar' apps
NOSE_ARGS = [
    '--with-coverage',
    '--cover-package=foo,bar',
]
```

Here, we're setting a couple of command-line arguments to be included every time we run `python manage.py test`. The `--with-coverage` option says we want a coverage report, and the `--cover-package` option lists all of the modules we are hoping to cover (these are the names of your Django apps). For a complete list of other available options, run `python manage.py help test`.

1.6.2 Coverage reports

When running test cases with coverage enabled, a report is printed at the end looking something like this:

```
Name                 Stmts   Miss  Cover   Missing
-----------------------------------------------
foo.models              30      5    85%   10-12, 16, 19
bar.models              10      1    90%   4
-----------------------------------------------
TOTAL                   40      6    87%
```

This says the `foo.models` module has 30 lines of executable code, and 5 of those lines were not evaluated during testing. The specific lines that aren't covered are listed at the end.

Why would certain lines not be executed? Perhaps those lines define a function that was never called, which means we need to add some tests for that function. Maybe those lines are inside an `if / else` block that only executed the `if` part, so we need to add tests for the `else` part also. It could be an exception handler that never encountered an exception, in which case we could add tests that purposely cause that exception (and verify that the correct exception was raised).

Try adding the `--cover-html` option to your `NOSE_ARGS` if you'd like a nice HTML report that highlights the missing lines in your source code.

1.6.3 Unreachable code

It's possible (though rare) that some lines of code are missed because they are simply unreachable. For example, the line inside this `if` statement can never be executed:

```
if 2 + 2 == 5:
    print("Unusually large value of 2")
```

Or, have you ever seen code like this?:

```
try:
    do_something(x)
# This should never happen, but just in case
except SomeError:
    do_something_else(x)
```

With sufficient testing and coverage analysis, you can determine with near-certainty whether "This should never happen" is a true statement. If there is no possible way for `do_something(x)` to raise `SomeError`, then there's no reason to keep the extra code around.

1.6.4 Further reading

So far, what you're getting out of this is **statement coverage**, which is the most basic kind of code coverage, and also arguably the weakest. It only tells you which lines of code were evaluated, but it does not tell you anything about all the possible ways that each of those lines could be evaluated. Those alternatives can be measured using **branch** and **condition** coverage, which is beyond the scope of this example.

Statement coverage is a good first step, and can point you towards obvious gaps in your test suite. It may be insufficient in the long run, but it's the easiest place to start if you've never measured coverage before.

See What is Wrong with Statement Coverage for more insight, and refer to Test coverage analysis for a Python-specific introduction to more detailed coverage techniques.

CHAPTER 2

Unfinished docs

2.1 Testing Django Models

2.1.1 Should I use doctests or unit tests?

2.2 Testing Using Mock Objects

2.2.1 Which mock library should I use?

2.3 Testing Template Tags

2.3.1 Should I use doctests or unit tests?

2.4 Django Testing Philosophy

2.4.1 What is the philosophy behind testing in Django?

2.5 Django Tutorial Part 5: Testing

Now that you have a working polls application, lets make sure that it will continue to work.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search