Django Test Utils Documentation

Release 0.2

Eric Holscher

January 07, 2014

Contents

Here you will find information on the utilities and other nice things located in this package. It is meant to make testing your Django applications easier. It also has things that will automate some parts of your testing.

CHAPTER 1

Source Code

The code for Django-test-utils is available at Github and Pypi.

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

There is a Bug Tracker at Github as well, if you find any defects with the code. Pull requests and patches are better than filing a bug.

CHAPTER 2

Contents

2.1 Django Testmaker

2.1.1 Source code

This projects is now a part of Django test utils. The 0.2 release is available at Pypi

2.1.2 What is does

Django testmaker is an application that writes tests for your Django views for you. You simply run a special development server, and it records tests for your application into your project for you. Tests will be in a Unit Test format, and it will create a separate test for each view that you hit.

2.1.3 Usage

Step 1: Add *test_utils* to your INSTALLED_APPS settings.

Step 2:

```
./manage.py testmaker -a APP
```

This will start the development server with testmaker loaded in. APP must be in installed apps, and it will use Django's mechanism for finding it. It should look a little something like this:

```
eric@Odin:~/EH$ ./manage.py testmaker mine
Handling app 'mine'
Logging tests to /home/eric/Python/EH/mine/tests/mine_testmaker.py
Appending to current log file
Inserting TestMaker logging server...
Validating models...
0 errors found

Django version 1.0.1 final, using settings 'EH.settings'
Development server is running at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

Then, as you browse around your site it will create unit test files for you, outputting the context variables and status code for each view that you visit. The test file used is in *APP/tests/APP_testmaker.py*. Once you have your tests written, you simply have to add them into your *__init__.py*, and then run your tests.

Step 3:

```
./manage.py test -a APP
```

2.1.4 Testmaker Internal Basics

Testmaker now has a pluggable backend architecture. It includes the concept of a *Processor* and a *Serializer*. A *Serializer* is responsible for serializing your request and responses, so that they can be run again later. A *Processor* is response for taking these request and response objects and turning them into the actual Unit Tests.

API

Both processors and serializers follow the standard API direction that django serializers started. For example, to grab instances of both of them, use the following code:

```
serializer = serializers.get_serializer('pickle')()
processor = processors.get_processor('django')()
```

Serializers

Testmaker ships with 2 default serializers currently. They are the *json* and *pickle* backends.

Processors

Testmaker currently ships with just one Processor, the *django* processor, which produces Django Testcase-style Unit Tests.

2.1.5 Extending Testmaker

Adding a new backend for a Processor or Serializer is easy. They both provide a similar interface, which can be located in the base.py file in their respective directories.

The only two functions that are required for each backend are the ability to save a request and a response. They have obvious defitions, and you should look in their directory for examples.

save_request(self, request)

save_response(self, request, response)

2.1.6 Options

-f –fixture

If you pass the *-f* option to testmaker, it will create fixtures for you. They will be saved in *APP/fixtures/APP_fixtures.FORMAT*. The default format is XML because I was having problems with JSON.

–format

Pass this in with a valid serialization format for Django. Options are currently json, yaml, or xml.

–addrport

This allows you to pass in the normal address and post options for runserver.

2.1.7 Future improvements

Force app filtering

I plan on having an option that allows you to restrict the views to the app that you passed in on the command line. This would inspect the URLConf for the app, and only output tests matching those URLs. This would allow you to fine tune your tests so that it is guaranteed to only test views in the app.

Better test naming scheme

The current way of naming tests is a bit hackish, and could be improved. It works for now, and keeps names unique, so it's achieving that goal. Suggestions welcome for a better way to name things.

2.2 Django Crawler

2.2.1 Core features

The Crawler at the beginning loops through all of your URLConfs. It then loads up all of the regular expressions from these URLConfs to examine later. Once the crawler is done crawling your site, it will tell you what URLConf entries are not being hit.

2.2.2 Usage

The crawler is implemented as a management command.

Step 1: Add *test_utils* to your *INSTALLED_APPS*

Step 2: The syntax for invoking the crawler looks like:

```
./manage.py crawlurls [options] [relative_start_url]
```

Relative start URLs are assumed to be relative to the site root and should look like 'some/path', 'home', or even '/'. The relative start URL will be normalized with leading and trailing slashes if they are not provided. The default relative start URL is '/'.

The crawler at the moment has 4 options implemented on it. It crawls your site using the Django Test Client (so no network traffic is required!) This allows the crawler to have intimate knowledge of your Django Code. This allows it to have features that other crawlers can't have.

2.2.3 Options

-v –verbosity [0,1,2]

Same as most django apps. Set it to 2 to get a lot of output. 1 is the default, which will only output errors.

-t –time

The *-t* option, as the help says: Pass -t to time your requests. This outputs the time it takes to run each request on your site. This option also tells the crawler to output the top 10 URLs that took the most time at the end of it's run. Here is an example output from running it on my site with *-t -v 2*:

```
Getting /blog/2007/oct/17/umw-blog-ring/ ({}) from (/blog/2007/oct/17/umw-blog-ring/)
Time Elapsed: 0.256254911423
Getting /blog/2007/dec/20/logo-lovers/ ({}) from (/blog/2007/dec/20/logo-lovers/)
Time Elapsed: 0.06906914711
Getting /blog/2007/dec/18/getting-real/ ({}) from (/blog/2007/dec/18/getting-real/)
Time Elapsed: 0.211296081543
Getting /blog/ ({u'page': u'5'}) from (/blog/?page=4)
Time Elapsed: 0.165636062622
NOT MATCHED: account/email/
NOT MATCHED: account/register/
NOT MATCHED: admin/doc/bookmarklets/
NOT MATCHED: admin/doc/tags/
NOT MATCHED: admin/(.*)
NOT MATCHED: admin/doc/views/
NOT MATCHED: account/signin/complete/
NOT MATCHED: account/password/
NOT MATCHED: resume/
/blog/2008/feb/9/another-neat-ad/ took 0.743204
/blog/2007/dec/20/browser-tabs/#comments took 0.637164
/blog/2008/nov/1/blog-post-day-keeps-doctor-away/ took 0.522269
```

-p –pdb

This option allows you to drop into pdb on an error in your site. This lets you look around the response, context, and other things to see what happened to cause the error. I don't know how useful this will be, but it seems like a neat feature to be able to have. I stole this idea from nose tests.

-s –safe

This option alerts you when you have escaped HTML fragments in your templates. This is useful for tracking down places where you aren't applying safe correctly, and other HTML related failures. This isn't implemented well, and might be buggy because I didn't have any broken pages on my site to test on :)

-r –response

This tells the crawler to store the response object for each site. This used to be the default behavior, but doing this bloats up memory. There isn't anything useful implemented on top of this feature, but with this feature you get a dictionary of request URLs with responses as their values. You can then go through and do whatever you want (including examine the Templates rendered and Contexts.

2.2.4 Considerations

At the moment, this crawler doesn't have a lot of end-user functionality. However, you can go in and edit the script at the end of the crawl to do almost anything. You are left with a dictionary of URLs crawled, and the time it took, and response (if you use the *-r* option).

2.2.5 Future improvements

There are a lot of future improvements that I have planned. I want to enable the test client to login as a user, passed in from the command line. This should be pretty simple, I just haven't implemented it yet.

Another thing that I want to do but isn't implemented is fixtures. I want to be able to output a copy of the data returned from the crawler run. This will allow for future runs of the crawler to diff against previous runs, creating a kind of regression test.

A third thing I want to implement is an option to only evaluate each URLConf entry X times. Where you could say "only hit /blog/[year]/[month]/ 10 times". This goes on the assumption that you are looking for errors in your views or templates, and you only need to hit each URL a couple of times. This also shouldn't be hard, but isn't implemented yet.

The big pony that I want to make is to use multiprocessing on the crawler. The crawler doesn't hit a network, so it is CPU-bound. However, running with CPUs with multiple cores, multiprocessing will speed this up. A problem with it is that some of the timing stuff and pdb things won't be as useful.

I would love to hear some people's feedback and thoughts on this. I think that this could be made into a really awesome tool. At the moment it works well for smaller sites, but it would be nice to be able to test only certain URLs in an app. There are lots of neat things I have planned, but I like following the release early, release often mantra.

2.3 Django Test Runner

This is a tool for running Django app tests standalone

2.3.1 Introduction

This script is fairly basic. Here is a quick example of how to use it:: django_test_runner.py [path-to-app]

You must have Django on the PYTHONPATH prior to running this script. This script basically will bootstrap a Django environment for you.

By default this script with use SQLite and an in-memory database. If you are using Python 2.5 it will just work out of the box for you.

2.4 Twill Runner

Integrates the twill web browsing scripting language with Django.

2.4.1 Introducation

Provides two main functions, `setup()` and `teardown`, that hook (and unhook) a certain host name to the WSGI interface of your Django app, making it possible to test your site using twill without actually going through TCP/IP.

It also changes the twill browsing behaviour, so that relative urls per default point to the intercept (e.g. your Django app), so long as you don't browse away from that host. Further, you are allowed to specify the target url as arguments to Django's `reverse()`.

Usage:

```
twill.setup()
try:
    twill.go('/')                    # --> Django WSGI
    twill.code(200)

    twill.go('http://google.com')
    twill.go('/services')            # --> http://google.com/services

    twill.go('/list', default=True)  # --> back to Django WSGI

    twill.go('proj.app.views.func',
             args=[1,2,3])
finally:
    twill.teardown()
```

For more information about twill, see: http://twill.idyll.org/

2.5 Persistent Database Test Runner

This code allows you to persist a database between test runs. It is really useful for running the same tests again and again, without incurring the cost of having to re-create the database each time.

2.5.1 Management Command

To call this function, simply use the `quicktest` management command, instead of the `test` command. If you need to alter the schema for your tests, simply run the normal `test` command, and the normal destroy/create cycle will take place.

2.5.2 Test Runner

The functionality is actually implemented in a Test Runner located at `test_utils.test_runners.keep_database`. If you want to use this as your default test runner, you can set the `TEST_RUNNER` setting to that value. This is basically all that the management command does, but in a temporary way.

2.6 Mock Objects

2.6.1 Mock Requests

This mock allows you to make requests against a view that isn't included in any URLConf.

RequestFactory

Usage:

```
rf = RequestFactory()
get_request = rf.get('/hello/')
post_request = rf.post('/submit/', {'foo': 'bar'})
```

This class re-uses the django.test.client.Client interface, docs here: http://www.djangoproject.com/documentation/testing/#the-test-client

Once you have a request object you can pass it to any view function, just as if that view had been hooked up using a URLconf.

Original Source

Taken from Djangosnippets.net, originally by Simon Willison.

2.7 Available Settings

This page contains the available settings for test utils. Broken down by the app that they affect.

2.7.1 Testmaker

TESTMAKER_SERIALIZER

This is the serializer that Testmaker should use. By default it is *pickle*.

TESTMAKER_PROCESSOR

This is the processor that Testmaker should use. By default it is *django*.

TEST_PROCESSOR_MODULES

This allows Testmaker to have access to your own custom processor modules. They are defined with the full path to the module import as the value.

To add your own processors, use the TEST_PROCESSOR_MODULES setting:

```
TEST_PROCESSOR_MODULES = {
    'awesome': 'my_sweet_app.processors.awesome',
}
```

TEST_SERIALIZATION_MODULES

The same as the above *TEST_PROCESSOR_MODULES*, allowing you to augment Testmakers default serializers.

To add your own serializers, use the TEST_SERIALIZATION_MODULES setting:

```
TEST_SERIALIZATION_MODULES = {
    'awesome': 'my_sweet_app.serializers.awesome',
}
```

2.8 How to run the tests

Test utils does contain some tests. Not as many as I would like, but it has enough to check the basic functionality of the things it does.

Running the tests is pretty simple. You just need to go into the test_project, and then run:

```
./manage.py test --settings=settings
```

In order to run just the tests for test utils do:

```
./manage.py test test_app --settings=settings
```

It is also possible to just run a single class of the tests if you so wish:

```
./manage.py test test_app.TestMakerTests --settings=settings
```

CHAPTER 3

Indices and tables

- *genindex*
- *search*