Django: Sane Testing Documentation

Release 0.5.11

Almad

March 26, 2015

Contents

Welcome to library that allows you to really test your Django applications. Read *introduction* if you're intrested in my motivations and an overview of what this library do; look at *test introduction* if you want to know my approach to web testing (fundamental to project design) or proceed directly to *Usage* to start using it, or *plugin description* if you're experienced testing/nose coder and just want the plugin reference.

Trac is available at my devel pages and downloads available from releases. Those interested can also always grab code directly from repository or track project at ohloh.

Feedback is always appreciated. Send it to bugs at almad.net, I'll be glad to hear from you.

CHAPTER 1

Introduction

Django: Sane Testing is a Python library for the Django framework that makes it possible to test.

As much as I like some Django concepts and as much as authors embrace testing in official documentation, I doubt that anyone really tried them in detail. Some basic testing techniques are impossible with Django and this library comes to fix them.

One of the great flaws is very simplistic testing system. By it's nature it's very slow (flushing database when not needed [1]), it does not allow some basic xUnit features and also do not allow testing via HTTP [2] and provides to extensibility model to fix the issues.

However, there is an excellent option available. Nose library is excellent test-dicovery framework, backward-compatible with PyUnit, that provides nice plugin interface to hook your own stuff. To make my life simpler, most of this library is made as a nose plugin.

And don't be afraid, we have TEST_RUNNER too, so your `./manage.py test` will still work.

1.1 Feature overview

Library supports following:

- Nose integration (load plugins and try `noserun`)
- Django integration (`./manage.py test` and boo ya)
- True unittests (no database interaction, no handling needed = SPEED)
- HTTP tests (has included wrapper for urlopen from urllib2)
- Transactional database tests (everything in one transaction, no database flush)
- Selenium RC integration (our pages needs to be tested with browser)
- CherryPy can be used instead of Django 's WSGI (when you need usable server)

1.2 Differences from standard Django test runner

If You are using older version than Django 1.3, you will be able to use unittest2 goodies.

Unlike Django's test runner, django-sane-testing do not modify your *DEBUG* behavior. You are free to run with *DEBUG* = *True*, which might be handy under some circumstances.

[1] At least transactional test cases are going to be implemented in Django 1.1, see #8138.
[2] Kinda buggy when we're talking about HTTP framework. See #2879.

CHAPTER 2

Test introduction

In this chapter, I'd like to present my view on testing, especially on web application testing. Through it will probably bring nothing new for most of the readers, I feel my view is very different from one that prevail in Django cmmunity and it's fundamental for library design, usage and further development.

When speaking about tests, I mean only tests written by developers themselves. QA has it's own tools and procedures that are out of scope of this document (and library).

2.1 Developer tests and xUnit

For past few years, unit-testing and test-driven development became nice buzzwords and together with agile and extreme programming created new style of programming. To take full advantage of testing, however, your precious test-suite must be handled properly, otherwise you will fail into futile world of slow, long test suites and constant need for suite refactor, which questions value of tests.

First of all, test suite must be deterministic. Under all supported configurations, tests suite must pass green; when some tests could not pass because of configured environment (like using some soft of application backend that do not support some features), tests must mark themselves as skipped.

This lead us to holy grail of agile methods, unit tests. Unit tests are fast and deterministic, because they do NOT:

- Talk to database
- Communicate over network
- Talk to 3rd party software

Instead, they use some sort of "dummy stub" (called mock) to replace real thing with dummy, predictable object that directly returns expected results and allows us to test proper response of our system. If you're doing this, inherit your cases from `UnitTestCase` and gain huge speedup provided.

For most Django-based apps, however, this is not practical; replacing database with dummy stubs would mean a lot of reimplementation (might be worh it sometimes, however). Still; one should bear in mind what unit test is and try to write them, because they lead to decoupled design and one might consider to create proper classes and functions (and test them separately) instead of binding it all to `Model()` instance methods.

Before digging into some of those challenges, let us take a brief look on tests we're talking about.

2.2 Test phases (and database)

Developer tests have, generally, four phases:

1. Prepare environment
2. Execute system interaction
3. Verify expected results [1]
4. Bring environment back to expected state

For true unittests, environment preparation (represented by `setUp` method in xUnit de-facto standard) consist mostly of mocking system components with those dummy thingies and replacing them back on `tearDown`.

Databases are, however, strange beasts. Most of tests using them need *generators* for unique, autogenerated fields (mostly id's) and tests are relying on them to be in expected states. Thus, suite must reinitialize database before doing such a test and that is a **very** costly operation that slow down your suite from running few hundred tests per second to few ones per second.

There is, however, compromise solution: transactions. Rolling back a transaction is not as costly as flushing whole database and thus can be used for testing with database object, where mocking is not a good time/profit solution. If it's your case, inherit from `DatabaseTestCase` and go. Beware, however, that you can't use it in multithreaded tests or when you have to commit during tests; then, you're stuck with `DestructiveDatabaseTestCase` that requests full cleanup. Otherwise, another tests will be strangely failing and interacting tests are really thing you don't want to have.

###FIXME: What follows is merely an extended feature intro, rewrite to follow in consistent way

2.3 HTTP Tests

While Django's TestClient (available for `DatabaseTestCase` and above) is cool, it's not usable for all cases (like, when you want to test your HTTP Basic/Digest view protection). If you want to test it, use `HttpTestCase` (which is sadly also `DestructiveDatabaseTestCase`) and framework will fire up multithreaded Django live server for you.

If this is not enough (and might not be, Django server is still kinda incomplete), you can have your Django served with CherryPy's production-ready, multi-threaded server. Just set `CHERRYPY_TEST_SERVER=True` in your settings and enjoy server you can repeatably connect to.

For HTTP requests, use included function urlopen (wrapper for eponymous function from urllib2), which can handles server-side traceback.

2.4 Web tests

Web tests are futile attempt to automatize acceptance tests, and also to test javascript between various browsers. Selenium is a great tool for that and we're providing support for it. Grab `SeleniumTestCase`, export Selenium tests in PyUnit format and enjoy `self.selenium`.

2.5 Developer's workflow

Common scenario when fixing a bug:

1. Write a failing regression tests
2. Debug to find wher test actually is

[1] To help defect localization, there should be only one condition tested. Rule of thumb is "one assert per test"

3. Write a failing unittest
4. Fix broken code
5. Run tests again to confirm we fixed the right thing
6. Run the whole suite to be sure we haven't broken anything

Single tests can be run with `./manage.py test package.package.module:TestCase.singleTest`. Using plugins, you can pipeline test runner to run only unittest and corresponding tests on developer machine and let your CI server to run full (and probably longer) suite to check your back.

2.6 Few words against Doctests

This library nor my thoughts dont cover doctests. It's for simple reason: they are lousy for testing. Doctest is excellent tool to verify your documentation and acceptable for making acceptance tests (thus write "system user stories").

However, for usual developers tests (and mainly unit tests), they are very bad idea. Few reasons:

1. You must flush database between them as there is no teardown to clear inconsistencies when test fail
2. Attempt for recovery when condition fails: One stares at hundred lines of traceback and must look for first condition that caused all the fails
3. Fixture support is lacking and must be done manually
4. They're hard to write: no support from editor, lot of >>>'s and ...'s

...but wait, isn't their so easy to write, because you just cut&paste your console output? If this is your case, then I'd say your development model is broken, and that is probably because your testing suite is broken. You are manually setting up your environment and friends and still feeling more productive then when using your suite, your suite is to blame (perhaps because you can't select only this one tes you are writing now? Well, you can do it with us). Fix it, because you're still doing all *four phases*, just wasting time doing it by hand instead of having it automatized.

CHAPTER 3

Usage

This part explains how to use django-sane-testing to create a workable test suite. For test writers, most important part is *about creating tests*, testers and CI admins may be more interested in *executing them*.

3.1 Writing tests

Various test types were identified when taking look at *developer tests*. Every test type has it's corresponding class in `djangosanetesting.cases`, namely:

- `UnitTestCase`
- `DatabaseTestCase`
- `DestructiveDatabaseTestCase`
- `HttpTestCase`
- `SeleniumTestCase`
- `TemplateTagTestCase`

However, you are not *required* to inherit from those (except for *twill*), although it's much advised to keep test cases intent-revealing. Job of the library is to:

- Start manual transaction handling and roll it back after test
- Flush database
- Start live server frontend
- Setup selenium proxy

Those are done by *plugins*, and they dermine their jobs by looking at class attributes, namely:

- `database_single_transaction`
- `database_flush`
- `start_live_server`
- `selenium_start`

All of those are booleans. Because class attributes are currently used to determine behaviour, nor doctest nor function tests are supported (they will not break things, but you'll get no goodies from library).

To support proper test selection and error-handling, tests also has class atribute `required_sane_plugins`, which specifies list of plugins (from *those available*) that are required for this type of test; if it's not, test automatically skip itself.

Proper defaults are selected when using *library test cases*; however, if you have your own and complicated test inheritance model, you can integrate it on your own.

When writing tests, keep in mind limitations of the individual test cases to prevent interacting tests:

- `UnitTestCase` should not interact with database or server frontend.
- `DatabaseTestCase` must run in one transaction and thus cannot be multithreaded and must not call commit.
- `DestructiveDatabaseTestCase` is slow and do not have live server available (cannot test using urllib2 and friends).
- `HttpTestCase` provides all goodies except Selenium. When first encountered, live server is spawned; after that, it's as fast as `DestructiveDatabaseTestCase`.
- `SeleniumTestCase` has it all (except speed).
- `TemplateTagTestCase` provides helper methods for testing custom template tags and filters.

3.2 Running tests

Easiest way to run tests is to put `TEST_RUNNER="djangosanetesting.testrunner.DstNoseTestSuiteRunner"` into your `settings.py`. This allows You to use `manage.py test` command (all plugins are enabled by default).

You can still use standard `nosetests` command. However, keep in mind:

- There is no path handling done for you
- DJANGO_SETTINGS_VARIABLE is also not set by default

Most likely, you'll end up with something like `DJANGO_SETTINGS_MODULE="settings" PYTHONPATH=".:.." nosetests --with-django`; you can, however, flexibly add another nose modules (like `--with-coverage`).

Fine-grained test type selection is available via `SaneTestSelectionPlugin`.

3.3 Plugins

Provided plugins:

- *DjangoPlugin*
- *DjangoLiveServerPlugin*
- *CherryPyLiveServerPlugin*
- *SeleniumPlugin*
- *SaneTestSelectionPlugin*
- *DjangoTranslationPlugin*

3.3.1 `DjangoPlugin`

`DjangoPlugin` takes care about basic Django environment setup. It **must** be loaded in order to use other plugins with Django (obviously). This plugin takes care about `DatabaseTestCase` and `DestructiveDatabaseTestCase`:

- If `no_database_interaction` attribute is True, then whole database handling is skipped (this is to speed thing up for `UnitTestCase`)
- If `database_single_transaction` is True (`DatabaseTestCase`), manual transaction handling is enabled and things are rolled back after every case.
- If `database_flush` is True, then database if flushed before every case (and on the beginning of next one, if needed)

django.db.transaction is also available under self.transaction. Use at own discretion; you should only access it when using `DestructiveDatabaseTestCase` (to make data available for server thread), messing with it when using `database_single_transaction` can cause test interaction.

Since 0.6, You can use `--persist-test-database`. This is similar to quicktest command from django-test-utils: database is not flushed at the beginning if it exists and is not dropped at the end of the test run. Useful if You are debugging single test in flush-heavy applications.

Warning: By definition, strange things will happen if You'll change tests You're executing. Do not overuse this feature.

Warning: Tested by hand, not covered by automatic tests. Please report any bugs/testcases You'll encounter.

3.3.2 `DjangoLiveServerPlugin`

Responsible for starting HTTP server, sort of same as `./manage.py runserver`, however testing server is multithreaded (as if with patch from #3357, but always enabled: if you'll any problems with it, write me).

Server is first started when `start_live_server` attribute is first encountered, and is stopped after whole testsuite.

Plugin uses following setttings variables:

- `LIVE_SERVER_PORT` - to which port live server is bound to. Default to 8000.
- `LIVE_SERVER_ADDRESS` - to which IP address/interface server is bound to. Default to 0.0.0.0, meaning "all interfaces".

Warning: Because application logic is always executed in another thread (even when server would be single-threaded), it's not possible to use `HttpTestCase`'s with in-memory databases (well, theoretically, we could do database setup in each thread and have separate databases, but that will be really nasty).
Thus, if encountered with in-memory database, server is not started and `SkipTest` is raised instead.

Warning: Because of *twill integration*, if non-empty `_twill` attribute is encountered, twill's reset_browser is called. This might be a problem if You, for whatever reason, set this attribute without interacting with it.
If it annoys You, write me and I might do something better. Until then, it's at least documented.

3.3.3 `CherryPyLiveServerPlugin`

Responsible for starting HTTP server, in similar way to `DjangoLiveServerPlugin`. However, CherryPy WSGI is used instead, as it's much more mature and considered to be production-ready, unlike Django's development server.

Use when in need of massive parallel requests, or when encountering a bug (like #10117).

Plugin uses following setttings variables:

- `LIVE_SERVER_PORT` - to which port live server is bound to. Default to 8000.

- `LIVE_SERVER_ADDRESS` - to which IP address/interface server is bound to. Default to 0.0.0.0, meaning "all interfaces".

Note: When using `./manage.py test`, Django server is used by default. You can use CherryPy's by setting `CHERRYPY_TEST_SERVER = True` in settings.py.

Warning: `DjangoLiveServerPlugin` (`--with-djangoliveserver`) and `CherryPyLiveServerPlugin` (`--with-cherrypyliveserver`) are mutually exclusive. Using both will cause errors, and You're responsible for choosing one when running tests with `nosetests` (see *Running tests* for details).

3.3.4 SeleniumPlugin

Selenium is excellent tool for regression (web application) testing. `SeleniumPlugin` easily allows you to use xUnit infrastructure together with Selenium RC and enjoy unified, integrated infrastructure.

Selenium proxy server must be set up and running, there is no support for auto-launching (yet).

`SeleniumPlugin` recognizes following configuration variables in settings.py:

- `SELENIUM_BROWSER_COMMAND` - which browser command should be send to proxy server to launch. Default to "*opera" and may require some more complicated adjusting on some configurations, take a look at experimental launchers.
- `SELENIUM_HOST` - where Selenium proxy server is running. Default to "localhost"
- `SELENIUM_PORT` - to which port Selenium server is bound to. Default to 4444.
- `SELENIUM_URL_ROOT` - where is (from proxy server's point of view) application running. Default to "http://URL_ROOT_SERVER_ADDRESS:LIVE_SERVER_PORT/" (There is a difference between `LIVE_SERVER_ADDRESS` and `URL_ROOT_SERVER_ADDRESS`, as `LIVE_SERVER_ADDRESS` is where server is bound to and `URL_ROOT_SERVER_ADDRESS` is which address is visible to client. Important when server is bound to all interfaces, as 0.0.0.0 is not a viable option for browser.)
- `FORCE_SELENIUM_TESTS` changes running behavior, see below.

When plugin encounters `selenium_start` attribute (set to True), it tries to start browser on selenium proxy. If exception occurs (well, I'd catch socket errors, but this seems to be impossible on Windows), it assumes that proxy is not running, thus environment conditions are not met and `SkipTest` is raised. If `FORCE_SELENIUM_TESTS` is set to True, then original exceptin is raised instead, causing test to fail (usable on web testing CI server to ensure tests are runnig properly and are not mistakenly skipped).

3.3.5 SaneTestSelectionPlugin

Test cases varies in their speed, in order:

1. Unit tests
2. Database tests
3. Destructive database tests and HTTP tests
4. Selenium webtests

As your test suite will grow, you'll probably want to do *test pipelining*: run your tests in order, from fastest to slowest, and if one of the suites will break, you'll stop running slower tests to save time and resources.

This can be done with `SaneTestSelectionPlugin`. When enabled by `--with-sanetestselection`, you can pass additional parameters to enable respecitve types of tests:

- `--select-unittests` (or `-u`)
- `--select-databasetests`
- `--select-destructivedatabasetests` and `--select-httptests`
- `--select-seleniumtests`

Only selected test types will be run. Test type is determined from class attribute `test_type`; when not found, test is assumed to be unittest.

Note: You're still responsible for loading required plugins for respective test cases. Unlike test selection with usual plugins, selection plugin enables you to run slower tests without faster (i.e. HTTP tests without unittests), and also skipping is faster (Selection plugin is run before others, thus skip is done without any unneccessary database handling, which may not be true for usual skips).

Warning: This plugin relies on setUp from `SaneTestCase`. Thus, it will work only with tests inheriting from it. Also, if You are overwriting setUp, You have to behave nicely and call `super(YourTestClass, self).setUp()`.

3.3.6 `DjangoTranslationPlugin`

If `make_translation` is True (default for every test), django.utils.translation.activate() is called before every test. If `translation_language_code` is set, it's passed to activate(); otherwise settings.LANGUAGE_CODE or 'en-us' is used.

This allows you to use translatable string taking usage of ugettest_lazy in tests.

Warning: It looks like Django is not switching back to "null" translations once any translation has been selected. make_translations=False will thus return lastly-activated translation.

3.4 Messing with syncdb

syncdb alone is now rarely used, as excellent South entered mainstream. Thus, by default, *migrate* is called every time database is recreated. This behavior can be adjuted using *DST_RUN_SOUTH_MIGRATIONS settings variable*

You may be doing something irresponsible like, say, referencing `ContentType` ID from fixtures, working around their dynamic creation by having own content type fixture. This, however, prevents you from specifying those in fixtures attribute, as flush emits post-sync signal causing ContentTypes to be created.

By specifying `TEST_DATABASE_FLUSH_COMMAND`, you can reference a function for custom flushing (you can use resetdb instead).

Note: You must specify function object directly (it takes one argument, test_case object). Recognizing objects from string is not yet supported as it's not needed for me - patches and tests welcomed.

Note: When using `TEST_DATABASE_FLUSH_COMMAND`, please note that *migrate* runs before `TEST_DATABASE_FLUSH_COMMAND`. This behavior will change in future releases.

Also, create_test_db (which is needed to be run at the very beginning) emits post_sync signal. Thus, you also probably want to set `FLUSH_TEST_DATABASE_AFTER_INITIAL_SYNCDB` to True.

3.5 Twill integration

Twill is simple browser-like library for page browsing and tests. For `HttpTestCase` and all inherited TestCases, `self.twill` is available with twill's `get_browser()`. It's setted up lazily and is resetted and purged after test case.

Browser has patched `go()` method: You can pass relative paths to it. Besides, it will throw `HTTPError` (from urllib2), if server serves page with status 500.

Also, use can use `go_xpath()` to use lxml-based XPath to specify hyperlink on page to visit.

`self.twill` also has `commands` attribute, equal to `twill.commands`.

`self.assert_code` is same as `twill.commands.code("number")`, except it raises usual `AssertionError`.

Note: Twill is using standard HTTP instead of WSGI intercept. This might be available in the future as an option, if there is a demand or patch written.

3.6 Settings & configuration

Behavior of django-sane-testing can be configured to match your needs.

TODO: List of whatever settings you can play with.

CHAPTER 4

django-sane-testing changelog

4.1 0.5.11 (planned for 2011-05-17)

- Scraping for unittest2 assertions (#9)
- TemplateTagTestCase
- manage.py test (runner) can now take usual nose arguments

4.2 0.5.7 (released 2010-12-13)

- Helper function for handling Django HTTP 500 tracebacks (thanks to Jiri Suchan)

4.3 0.5.8 - 0.5.9

- Releases revoked because of packaging issues

4.4 0.5.7 (released 2010-12-02)

- Support for Django 1.2 and multidb
- Support for Django 1.0 dropped
- Test client avaliable in UnitTestCase
- (Undocumented) south support, see #36
- Twill support improvements
- *–persist-test-database option*
- Switched to github as primary development place