django-functest Documentation

Release 1.0.4

Luke Plant

Oct 29, 2018

Contents

Contents:

CHAPTER 1

Installation

At the command line:

```
$ easy_install django-functest
```

Or, if you have virtualenvwrapper installed:

```
$ mkvirtualenv django-functest
$ pip install django-functest
```

You will also need to add django-functest to your URLs. In your URLconf:

```
urlpatterns += patterns('',
    url(r'^django_functest/', include('django_functest.urls'))
)
```

or:

```
urlpatterns += [
    url(r'^django_functest/', include('django_functest.urls'))
]
```

This is only necessary for running tests, so the above can be done conditionally for test mode only, if possible.

When running tests, you will also need to have `localhost` in your `ALLOWED_HOSTS` setting.

1.1 Dependencies

django-webtest, WebTest and other dependencies are automatically installed. If you are using Django 1.11 or later, you should install django-functest 1.0.1 or later and django-webtest 1.9.1 or later.

Installing django-functest will install the Python `selenium` package automatically. However, due the nature of Selenium, if you are writing Selenium tests (i.e. if you are not simply using the WebTest wrapper), dependencies are quite complex.

Selenium uses a `WebDriver` protocol for talking to browsers that is more or less supported by different browsers. Please see the *`django_functest.FuncSeleniumMixin.driver_name`* attribute for selecting the browser to use, and note the following:

- Chrome can be used if chromedriver is installed.
- Firefox 45 and older can be used with Selenium < 3 without anything additional installed. Old versions of Firefox can be found here: https://ftp.mozilla.org/pub/firefox/releases/

  If you need to run your own tests with a different version of Firefox than the default one on your system, it is recommended you follow the pattern used by django-functest's own runtests.py script which allows you to pass a `--firefox-binary` option. This is then eventually returned by *`get_webdriver_options()`* as argument `firefox_binary` (see tests/base.py). You could also make `get_webdriver_options` look in `os.environ` if that is easier to arrange.
- Selenium >= 3 will not work with Firefox 45 or older.
- For newer versions of Firefox, you can use Selenium 3 or later, if you install the new Marionette driver, also known as `geckodriver` (download releases here).

  This implementation currently is incomplete and has bugs and various incompatibilities. However, with the most recent versions of Firefox (58.0), geckodriver (0.20.0) and Selenium (3.11), the django-functest suite passes fully.
- If installed PhantomJS can be used. PhantomJS is no longer officially supported - the test suite does not run against it and and bugs for it will not be fixed. This is because the project has been abandoned, and Selenium also no longer supports it.

CHAPTER 2

Usage

There are two main ideas behind django-functest:

1. Write functional tests for Django apps using a high level API.

   If you are using Selenium, you might see things like:

   ```
   self.driver.get(self.live_server_url + reverse("contact_form"))
   self.driver.find_element_by_css_selector('#id_email').send_keys('my@email.com')
   self.driver.find_element_by_css_selector('#id_message').send_keys('Hello')
   self.driver.find_element_by_css_selector('input[type=submit]').click()
   WebDriverWait(self.driver, 10).until(lambda driver: driver.find_element_by_css_
   ↪selector('body'))
   ```

   With django-webtest, it might look like:

   ```
   response = self.app.get(reverse("contact_form"))
   form = response.form
   form['email'] = 'my@email.com'
   form['message'] = 'Hello'
   response2 = form.submit().follow()
   ```

   Both of these are verbose, and much lower-level than you would like to write for testing a Django web app. (Tests that use Django test client are even worse, and additionally do a really bad job of modelling what happens when a user interacts with a web page.)

   With django-functest, the lower-level details are hidden, and instead you would write:

   ```
   self.get_url("contact_form")
   self.fill({'#id_email': 'my@email.com',
              '#id_message': 'Hello'})
   self.submit('input[type=submit]')
   ```

2. Write two sets of tests at once, by using an API that is unified as well as high level.

   For many web sites, you need them to work without Javascript, as well as having some Javascript functionality to enhance. This is good practice where possible, and it also means that you can write fast functional tests -

Selenium tests are often hundreds of times slower than tests that simply use HTTP.

django-functest provides wrappers for WebTest and Selenium that use the **same** API. This means you can write a test targetting both, and run in two different ways.

The fast WebTest tests can be used when you need to iterate quickly, but you can still run the full tests against a browser.

In addition, django-functest provides various helps to smooth things along:

- *get_url()* has Django URL reversing built-in, covering the common case.
- short-cuts for putting things into the session so that you can skip steps.
- For both Selenium and WebTest helpers, there are additional methods. For example, there is *click()*, which does element clicking in a browser, but takes care of many details like scrolling elements into view, using battle-hardened strategies.

2.1 Getting started

It is recommended for both Selenium and WebTest, you should create your own base classes. These can have *configuration*, helpers and functionality that are specific to your project as needed:

`yourproject.tests.base`:

```
from django.test import TestCase
from django.contrib.staticfiles.testing import StaticLiveServerTestCase
from django_functest import FuncSeleniumMixin, FuncWebTestMixin

class WebTestBase(FuncWebTestMixin, TestCase):
    def setUp(self):
        super(WebTestBase, self).setUp()  # Remember to call this!
        # Your custom stuff here etc.

class SeleniumTestBase(FuncSeleniumMixin, StaticLiveServerTestCase):
    driver_name = "Chrome"
```

Normally `StaticLiveServerTestCase` will be better than `LiveServerTestCase`.

django-functest deliberately does not provide the base classes as above, only the mixins, to make life easier especially in the case where you already have another base class you want to inherit from (for example, if you have custom needs instead of using `StaticLiveServerTestCase`).

Then:

`yourapp.tests`:

```
from yourproject.tests.base import SeleniumTestBase, WebTestBase
from django_functest import FuncBaseMixin

class ContactFormTestBase(FuncBaseMixin):
    def test_contact_form(self):
        self.get_url("contact_form")
        self.fill({'#id_email': 'my@email.com',
                   '#id_message': 'Hello'})
        self.submit('input[type=submit]')
        self.assertTextPresent('Thanks for your message!')

class ContactFormWebTests(ContactFormTestBase, WebTestBase):
```

(continues on next page)

(continued from previous page)

```
    pass

class ContactFormSeleniumTests(ContactFormTestBase, SeleniumTestBase):
    pass
```

You now have two tests for the price of one!

Of course:

- You don't have to use both - the high level API provided by django-functest is still useful for writing either kind of test.
- Sometimes you have pages that require Javascript to work for some parts. This can be handled by adding tests to the Selenium subclass only.

Sometimes you need different actions to be done if Javascript is enabled. In this case, there are several options:

1) Use an abstract method in the base class, and create different implementations of it in the subclasses:

```
class ContactFormTestBase(FuncBaseMixin):
    def test_foo(self):
        self.get_url('foo')
        self.do_thing()
        self.assertTextPresent('Success!')

class ContactFormWebTests(ContactFormTestBase, WebTestBase):
    def do_thing(self):
        pass # etc.

class ContactFormSeleniumTests(ContactFormTestBase, SeleniumTestBase):
    def do_thing(self):
        pass # etc.
```

2) Test the attribute `is_full_browser_test`. This is `True` for Selenium, and `False` for WebTest. For example:

```
def test_foo(self):
    self.get_url('foo')
    if self.is_full_browser_test:
        # Form is not visible until we click this button
        self.click('input.foo')
    self.fill_form()
    self.submit('input[type=submit]')
    self.assertTextPresent('Success!')
```

2.2 Tips

The following are various tips for writing reliable tests.

2.2.1 Use FuncBaseMixin

In the above example, `FuncBaseMixin` is not strictly needed at all - it provides method definitions which all raise `NotImplementedError` - so you could replace it with `object`. However, it can be very useful for editors that provide code autocompletion help, which be able to find the docstrings on `FuncBaseMixin` when you are writing

methods like `ContactFormTestBase.test_contact_form`. You may want to inherit from it in your own base class.

2.2.2 Avoid 404s

For Selenium tests, the browser will load not only the main page, but various other resources (Javascript, CSS etc.). It can be important to ensure that these resources will be served by your dev server. Requesting pages that don't exist will slow down your tests, and it can introduce unreliability. This can especially be true if your site has complex middleware, redirects etc. and things that affect the session. Unnecessary requests could trigger some of these actions and complicate things.

In particular, in the absence of a defined favicon location, browsers will request `/favicon.ico`. This will typically hit your app and produce 1) a redirect since it does not end with `/` and 2) a 404. Depending on your URLs it could also trigger other work, since it does not have the static URL prefix, and so it won't be handled by the normal staticfiles finder. To workaround this, it is recommended to put your favicon in the staticfiles folder, and specify its location.

CHAPTER 3

Common WebTest/Selenium API

This page documents the methods and attributes that are provided by both *django_functest.FuncWebTestMixin* and *django_functest.FuncSeleniumMixin*.

Conventions:

unittest provides assertion methods that are `camelCased`. Django follows suit with assertion methods, but for other things defaults to the PEP8 recommendation of `name_with_underscores` e.g. `live_server_url`. We have followed the same pattern.

class django_functest.**FuncCommonApi**

Assertion methods

assertUrlsEqual(*url*, *other_url=None*)
Checks that the URLs are equal, with `other_url` defaulting to the current URL if not passed. The path and query are checked, and if both URLs contain a domain name and/or protocol, these are also checked. This means that relative URLs can be used, or protocol-relative URLs.

assertTextPresent(*text*)
Asserts that the text is present on the current page

assertTextAbsent(*text*)
Asserts that the text is not present on the current page

Other methods and attributes

back()
Go back in the browser.

For WebTest, this will not make additional requests. For Selenium tests, this may or may not make additional requests, depending on caching etc. and what happens when you press 'Back' in the browser being used.

current_url
The current full URL

follow_link(*css_selector*)
Follows the link specified in the CSS selector.

You will get an exception if no links match.

For *django_functest.FuncWebTestMixin*, you will get an exception if multiple links match and they don't have the same href.

fill(*data_dict*)
Fills form inputs using the values in `data_dict`. The keys are CSS selectors, and the values are the values for the inputs. Works for text inputs, radio boxes, check boxes, and select fields. Checkbox values can be specified using `True` and `False`. Radio button values should be specified using the `value` attribute that should be matched, and the radio button that matches that will be selected (even if the selector matched another button in that group).

To upload a file, pass an *Upload* instance as the value.

This will raise an exception if the fields can't be found. It will be a timeout exception for Selenium tests, so you will want to avoid attempting to fill in fields that don't exist.

If multiple fields match, you will get an exception for *FuncWebTestMixin* but not for *FuncSeleniumMixin* due to the way Selenium finds elements.

fill_by_id(*data_dict*)
Same as *fill()* except the keys are element IDs. **Deprecated** — instead of `fill_by_id({'foo': 'bar'})` you should do `fill({'#foo': 'bar'})`, because it is shorter and more flexible.

fill_by_name(*data_dict*)
Same as *fill()* except the keys are input names.

fill_by_text(*data_dict*)
Same as *fill()*, except the values are text captions. This can be used only for `select` elements.

get_url(*name*, **args*, ***kwargs*)
Gets the named URL, passing it through `django.core.urlresolvers.reverse` with `*args` and `**kwargs`.

e.g.:

```
self.get_url('admin:auth_user_change', object_id=1)
```

get_literal_url(*relative_url*, *auto_follow=True*, *expect_errors=False*)
Gets the URL given by the relative URL passed in.

For *FuncWebTestMixin*, pass `auto_follow=False` if you don't want redirects to be followed. This parameter is ignored by *FuncSeleniumMixin*.

For *FuncWebTestMixin*, pass `expect_errors=True` if you are expecting an error code e.g. a 404, otherwise you will get an exception. This parameter is ignored by *FuncSeleniumMixin*.

is_element_present(*css_selector*)
Returns `True` if the element specified by the CSS selector is present, `False` otherwise. See also *is_element_displayed()*.

is_full_browser_test
True for Selenium tests, False for WebTest tests.

set_session_data(*data_dict*)
Set data directly into the Django session from the supplied dictionary. This is useful for implementing setup/shortcuts needed for specific views.

get_session_data()
Get the Django session as a dictionary. This is useful for creating assertions.

new_browser_session()
Creates (and switches to) a new session that is separate from previous sessions. This can be used to simulate multiple devices/users accessing a site at the same time.

Returns a tuple (old_session_token, new_session_token). These values should be treated as opaque tokens that can be used with *switch_browser_session()*.

For Selenium tests, a new instance of the web driver is created, which results in a new browser instance with a separate profile being used. In this case, however, there are complications:

Django's `LiveServerTestCase` is currently single threaded. Some browsers keep (mutliple) connections open to a domain, and Chrome especially can lock up the test server when multiple sessions are open.

A fix for this is to add *django_functest.MultiThreadedLiveServerMixin* to any test class that needs this functionality, especially if run against Chrome. However, please note the issues documented for that mixin.

switch_browser_session(*session_token*)
Switch to the browser session indicated by the supplied token. The token must be an object returned from a previous call to *new_browser_session()* or *switch_browser_session()*.

Returns a tuple (old_session_token, new_session_token).

submit(*css_selector*, *wait_for_reload=True*, *auto_follow=True*, *window_closes=False*)
Submits a form via the button specified in `css_selector`.

For *FuncSeleniumMixin*, `wait_for_reload=True` causes it to wait until a whole new page is loaded (which always happens with *FuncWebTestMixin*). If you are expecting an AJAX submission or Javascript code to stop a new page from actually being loaded, pass `wait_for_reload=False`.

For Selenium tests, if you are expecting the window to close, pass `window_closes=False` and then use *switch_window()*, or you may experience long timeouts with Chrome. This implies `wait_for_reload=False` and other tweaks. It does nothing when running WebTest tests.

For *FuncWebTestMixin*, `auto_follow=True` causes redirects to be followed automatically (which always happens with *FuncSeleniumMixin*). Pass `False` to allow intermediate responses (i.e. 3XX redirect responses) to be inspected via *last_response*.

value(*css_selector*)
Returns the value of the form input specified in CSS selector.

The types of the values correspond to those that are passed to *fill()*:

- For check boxes, it will return `True` or `False`.
- For text inputs, returns the text value.
- For selects, returns the internal `value` attribute of the selected item.

class django_functest.**Upload**

__init__(*filename*, *content=data*)
Construct an object for uploading in a normal file upload field. The *content* parameter must be a bytestring (*str* on Python 2, *bytes* on Python 3)

CHAPTER 4

Selenium wrapper

The class `FuncSeleniumMixin` has some Selenium/full browser specific methods, as well as the *Common WebTest/Selenium API*.

class django_functest.**FuncSeleniumMixin**

Configuration

These are class attributes and class methods that determine how the browser will be set up and run. It is easiest to override the class attribute, but the class method exists also for more involved needs. Note that most of these are used when `setUpClass` is called, and most of the related methods are therefore classmethods. Some are called within `setUp`

It is usually a good idea to set up a `FuncSeleniumMixin` sub-class in your project, to be used as a base-class for your tests, and set these configuration values on it.

browser_window_size
If set, should be a tuple containing `(width, height)` in pixels. This will be used inside `setUp` to set the browser window size. Defaults to `None`.

default_timeout
Controls most Selenium timeouts, defaults to `10` (seconds).

display
Controls whether browser window is displayed or not, defaults to `False`.

Note that PhantomJS is always invisible, regardless of this value.

driver_name
Controls which Selenium 'driver' i.e. browser will be used. Defaults to `"Firefox"`. You can also use `"Chrome"` if Chrome and chromedriver are installed, and `"PhantomJS"` if PhantomJS is installed.

display_browser_window()
classmethod. Returns boolean that determines if the browser window should be shown. Defaults to *display*.

get_browser_window_size()
Returns *browser_window_size* by default.

get_default_timeout()
classmethod. Returns the time in seconds for Selenium to wait for the browser to respond etc. Defaults to *default_timeout*.

get_driver_name()
classmethod. Returns the driver name i.e. the browser to use. Defaults to *driver_name*.

get_page_load_timeout()
classmethod. Returns the time in seconds for Selenium to wait for the browser to return a page. Defaults to *page_load_timeout*.

page_load_timeout
Controls Selenium timeouts for loading page, defaults to `20` (seconds).

get_webdriver_options()
Returns options to pass to the WebDriver class. Defaults to `{}`. This can be used to pass `capabilities`, `firefox_binary` or `firefox_options` if you are using the Firefox driver, for example.

Other attributes and methods

click(*css_selector=None*, *xpath=None*, *text=None*, *text_parent_id=None*, *wait_for_reload=False*, *double=False*, *scroll=True*, *window_closes=False*)
Clicks the button or control specified by the CSS selector e.g.:

```
self.click("input.default")
```

Alternatively, `xpath` or `text` can be provided as keyowrd arguments, instead of a CSS selector e.g.:

```
self.click(xpath='//a[contains(text(), "kitten")]')
self.click(text="kitten")
```

Additionally, `text_parent_id` can be used in combination with `text` to limit the search to descendent elements of the one with the supplied id.

This method will attempt to scroll the window to make the element visible if `scroll=True` is passed (the default) - this is usually necessary for browsers to click controls correctly.

If `double=True` is passed, a double click will be performed. Note, this will simply be two clicks, like a user would, rather than the Selenium `double_click` action chain, which doesn't actually trigger single click events.

See also the notes in *submit()* regarding `wait_for_reload` and `window_closes` (noting that the default values are different).

execute_script(*script*, **args*)
Executes the suppplied Javascript in the browser and returns the results.

If you need to pass arguments, you can receive them in the script using `arguments` e.g.:

```
self.execute_script("return arguments[0] + arguments[1];", 1, 2)
```

Arguments and return values are serialized and deserialized by Selenium.

hover(*css_selector*)
Peform a mouse hover over the element specified by the CSS selector.

is_element_displayed(*css_selector*)
Returns `True` if the element specified by the CSS selector is both present (see *is_element_present()*) and visible on the page (e.g. does not have `display: none;`), `False` otherwise.

save_screenshot(*dirname="./"*, *filename=None*)
Saves a screenshot of the browser window. By default, it is saved with a filename that includes a timestamp and the current test being run, into the current working directory, but this can be overridden by passing in a directory path and/or a filename. The full filename of the screenshot is returned.

set_window_size(*width*, *height*)
Sets the browser window size to the specified width and height in pixels.

For PhantomJS browser, this sets the document size - there isn't really a window.

switch_window(*handle=None*)
Switches the browser window that has focus.

If there are only 2 windows, it can work out which window to switch to. Otherwise, you must pass in the window handle as the `handle` kwarg.

The method returns a tuple of `(old_window_handle, new_window_handle)` which can be used in subsequent calls to `switch_window`.

wait_for_page_load()
Waits until the page has finished loading. You may want to override this to add extra things if a page has specific requirements.

wait_until(*callback*, *timeout=None*)
Waits until the callback returns `True`, with a timeout that defaults to the *`default_timeout`*. The callback must accept a single parameter which will be the driver instance.

wait_until_loaded(*css_selector*)
Waits until an element matching the CSS selector appears.

CHAPTER 5

WebTest wrapper

The class `FuncWebTestMixin` has some WebTest specific methods/attributes, as well as the *Common WebTest/Selenium API*.

class django_functest.**FuncWebTestMixin**

last_response
This is an object containing the last HTTP response, as documented in the WebTest docs.

CHAPTER 6

Utilities

class django_functest.**AdminLoginMixin**
This provides helpers for logging into the admin interface using the standard login page. It assumes logging in with username and password.

This works with both `FuncWebTestMixin` and `FuncSeleniumMixin`.

do_login(*username=None*, *password=None*, *shortcut=True*)
Do an admin login for the provided username and password. However, it will actually do a shortcut by default, unless *shortcut=False* is passed, using the *shortcut_login()*.

do_logout(*shortcut=True*)
Do a log out for the current user, using *shortcut_logout()* if `shortcut=True` (the default), or using the admin logout page if `shortcut=False`

class django_functest.**ShortcutLoginMixin**
This provides a method for doing a login without actually doing HTTP-level work, as far as possible.

This works with both `FuncWebTestMixin` and `FuncSeleniumMixin`. These methods do *not* simulate exactly what happens when a user logs in and out with real HTTP requests — they only do enough to get you to "logged in user", or "logged out user" state. In particular, other side effects on the session that normally happen (such as session key rotation, or anything that responds to the `user_logged_in` signal etc.) will not be done.

shortcut_login(***credentials*)
Pass credentials (typically `username` and `password`), as accepted by `django.contrib.auth.authenticate`, and if they are valid, you will get a session where the user is logged in. Otherwise an exception is raised.

Manipulates the session and cookies directly.

shortcut_logout()
Logs out the user from the current session

Manipulates the session and cookies directly.

class django_functest.**MultiThreadedLiveServerMixin**
Add this as a mixin to any test class (or test class base) to enable a multi-threaded live server.

This makes it possible to use some browsers (e.g. Chrome) in combination with test methods like *new_browser_session()*.

Note that there are some limitations:

- You cannot use this with an in-memory SQLite test database. You will need to set a NAME parameter for the test database to force it to be a non-in-memory database.

On Django 2.0 and greater, this is class does nothing since the Django 2.0 `LiveServerTestCase` already has this behaviour builtin.

CHAPTER 7

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

7.1 Types of Contributions

7.1.1 Report Bugs

Report bugs at https://github.com/django-functest/django-functest/issues.

If you are reporting a bug, please include:

- Your Django version.
- Browser name and version if applicable.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

7.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" is open to whoever wants to implement it.

7.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with "feature" is open to whoever wants to implement it.

7.1.4 Write Documentation

django-functest could always use more documentation, whether as part of the official django-functest docs, in docstrings, or even on the web in blog posts, articles, and such.

7.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/django-functest/django-functest/issues

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

7.2 Get Started!

Ready to contribute? Here's how to set up `django-functest` for local development.

1. Fork the *django-functest* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/django-functest.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv django-functest
$ cd django-functest/
$ python setup.py develop
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ flake8 django_functest
% isort -rc -c .
$ ./runtests.py
$ tox
```

To get flake8/tox/isort, just `pip install` them into your virtualenv.

To run the full test suite, you will need to install:

- Firefox and geckodriver.
- chromedriver

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

7.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 2.7, 3.3, 3.4 and 3.5. Check https://travis-ci.org/django-functest/django-functest/pull_requests and make sure that the tests pass for all supported Python versions.

7.4 Tips

To run a subset of tests:

```
$ ./runtests.py  django_functest.tests.SomeTestCase
```

7.5 Conduct

Contributors of any kind are expected to act with politeness to all other contributors, in pull requests, issue trackers etc., and harassing behaviour will not be tolerated.

CHAPTER 8

Credits

Wolf & Badger, for whom the code was originally developed, and who allowed us to Open Source it. Thanks!

Luke Plant <L.Plant.98@cantab.net> Frankie Robertson <frankie@robertson.name>

CHAPTER 9

History

9.1 1.0.4

- Fixed bug with setting checkboxes if a form with multiple checkboxes of the same name
- Enabled installation on PyPy (doesn't necessarily work completely).
- Test against Django 2.1
- Removed tests and official support for PhantomJS. (No actual functionality was changed regarding PhantomJS).

9.2 1.0.3

- Deprecated `fill_by_id`. Instead of `fill_by_id({'foo': 'bar'})` you should do `fill({'#foo': 'bar'})`, because it is shorter and more flexible.
- Test against latest Firefox
- Django 2.0 compatibility
- Fix for Django 1.11.2 and later for MultiThreadedLiveServerMixin

9.3 1.0.2

- Fixes to cope with WebTest 2.0.28. We now require django-webtest 1.9.2 or later, and only test against the latest WebTest.
- Fixed some deprecation warnings

9.4 1.0.1

- Fixed incompatibility with django-webtest 1.9.0 and later

9.5 1.0

- Added Django 1.11 support.
- Dropped official Django 1.7 support (may still work).

9.6 0.2.1

- Made *get_literal_url()* accept absolute URLs for Selenium (WebTest already worked by accident).

9.7 0.2.0

- Added *new_browser_session()* and *switch_browser_session()* to the common API. These can be used to simulate multiple devices or users accessing the site. See the docs for important usage information.

9.8 0.1.9

- Fix for scrolling to exactly the right place.
- Added docstrings everywhere, and a base class you can inherit from for the purpose of providing autocomplete help.

9.9 0.1.8

- Django 1.10 compatibility

9.10 0.1.7

- Fixed performance/reliability issue caused by browsers attempting to retrieve `/favicon.ico` after visiting `emptypage`.

9.11 0.1.6

- Fixed bug where elements wouldn't scroll into view if html height is set to 100%
- New method *get_webdriver_options()* for customizing WebDriver behaviour.

9.12 0.1.5

- Added get_session_data()
- Improved reliability of `FuncSeleniumMixin.get_literal_url()`
- Allow `<select>` elements to be set using integers for values.
- Fixed issues with `.value()` for radio buttons and text areas
- Fixed bug with setting radio buttons when there are more than one set of radio buttons in the form.

9.13 0.1.4

- Added support for file uploads

9.14 0.1.3

- Support for filling radio buttons
- More convenient support for quotes and apostrophes (" ') in text assertion methods.

9.15 0.1.2

- Fixed wheel building - again!

9.16 0.1.1

- Fixed packaging bug that caused wheels to fail on Python 3.

9.17 0.1.0

- First release on PyPI.

Index

Symbols

A

B

C

D

E

F

G

H

I