pytest-selenium Documentation

Release latest

Dave Hunt

Aug 01, 2017

Contents

pytest-selenium is a plugin for pytest that provides support for running Selenium based tests.

CHAPTER 1

Installation

Requirements

pytest-selenium will work with Python 3.6 and 2.7.

Install pytest-selenium

To install pytest-selenium using pip:

```
$ pip install pytest-selenium
```

To install from source:

```
$ python setup.py develop
```

CHAPTER 2

User Guide

Contents

Quick Start

The pytest-selenium plugin provides a method scoped selenium fixture for your tests. This means that any test with selenium as an argument will cause a browser instance to be invoked. The browser may run locally or remotely depending on your configuration, and may even run headless.

Here's a simple example test that opens a website using Selenium:

```
def test_example(selenium):
    selenium.get('http://www.example.com')
```

To run the above test you will need to specify the browser instance to be invoked. For example, to run it using Firefox installed in a default location:

```
pytest --driver Firefox
```

For full details of the Selenium API you can refer to the documentation.

Configuration Files

There are a number of options and values that can be set in an INI-style configuration file. For details of the expected name, format, and location of these configuration files, check the pytest documentation.

Specifying a Base URL

To specify a base URL, refer to the documentation for the pytest-base-url plugin.

Note: By default, any tests using a base URL will be skipped. This is because all tests are considered destructive, and all environments are considered sensitive. See *Sensitive Environments* for further details.

Sensitive Environments

To avoid accidental changes being made to sensitive environments such as your production instances, all tests are assumed to be destructive. Any destructive tests attempted to run against a sensitive environment will be skipped.

Nondestructive Tests

To explicitly mark a test as nondestructive, you can add the appropriate marker as shown here:

```
import pytest
@pytest.mark.nondestructive
def test_nondestructive(selenium):
    selenium.get('http://www.example.com')
```

Indicating Sensitive Environments

Sensitive environments are indicated by a regular expression applied to the base URL or any URLs discovered in the history of redirects when retrieving the base URL. By default this matches all URLs, but can be configured by setting the `SENSITIVE_URL` environment variable, using a *configuration file*, or by using the command line.

An example using a *configuration file*:

```
[pytest]
sensitive_url = example\.com
```

An example using the command line:

```
pytest --sensitive-url "example\.com"
```

Specifying a Browser

To indicate the browser you want to run your tests against you will need to specify a driver and optional capabilties.

Firefox

To run your automated tests with Firefox version 47 or earlier, simply specify `Firefox` as your driver:

```
pytest --driver Firefox
```

For Firefox version 48 onwards, you will need to download GeckoDriver and `selenium` 3.0 or later. If the driver executable is not available on your path, you can use the `--driver-path` option to indicate where it can be found:

```
pytest --driver Firefox --driver-path /path/to/geckodriver
```

See the GeckoDriver documentation for more information.

Configuration

A `firefox_options` fixture is available to configure various options for Firefox. The following example demonstrates specifying a binary path, preferences, and a command line argument:

```
import pytest
@pytest.fixture
def firefox_options(firefox_options):
    firefox_options.binary = '/path/to/firefox-bin'
    firefox_options.add_argument('-foreground')
    firefox_options.set_preference('browser.anchor_color', '#FF0000')
    return firefox_options
```

See the Firefox options API documentation for full details of what can be configured.

Chrome

To use Chrome, you will need to download ChromeDriver and specify `Chrome` for the `--driver` command line option. If the driver executable is not available on your path, you can use the `--driver-path` option to indicate where it can be found:

```
pytest --driver Chrome --driver-path /path/to/chromedriver
```

See the ChromeDriver documentation for more information.

Configuration

A `chrome_options` fixture is available to configure various options for Chrome. The following example demonstrates specifying a binary path, adding an extension, and passing an argument to start Chrome in kiosk mode:

```
import pytest
@pytest.fixture
def chrome_options(chrome_options):
    chrome_options.binary_location = '/path/to/chrome'
    chrome_options.add_extension('/path/to/extension.crx')
    chrome_options.add_argument('--kiosk')
    return chrome_options
```

See the Chrome options API documentation for full details of what can be configured.

The ChromeDriver supports various command line arguments. These can be passed by implementing a `driver_args` fixture and returning a list of the desired arguments. The following example specifies the log file path:

```
import pytest
@pytest.fixture
def driver_args():
    return ['--log-path=chromedriver.log']
```

For a full list of supported command line arguments, run `chromedriver --help` in your terminal.

Edge

To use Edge, you will need to download Edge WebDriver and specify `Edge` for the `--driver` command line option. If the driver executable is not available on your path, you can use the `--driver-path` option to indicate where it

can be found:

```
pytest --driver Edge --driver-path \path\to\MicrosoftWebDriver.exe
```

Internet Explorer

To use Internet Explorer, you will need to download and configure the Internet Explorer Driver and specify `IE` for the `--driver` command line option. If the driver executable is not available on your path, you can use the `--driver-path` option to indicate where it can be found:

```
pytest --driver IE --driver-path \path\to\IEDriverServer.exe
```

PhantomJS

To use PhantomJS, you will need download it and specify `PhantomJS` for the `--driver` command line option. If the driver executable is not available on your path, you can use the `--driver-path` option to indicate where it can be found:

```
pytest --driver PhantomJS --driver-path /path/to/phantomjs
```

See the PhantomJS documentation for more information.

Configuration

PhantomJS supports various command line arguments. These can be passed by implementing a `driver_args` fixture and returning a list of the desired arguments. The following example specifies the log file path:

```
import pytest
@pytest.fixture
def driver_args():
    return ['--webdriver-logfile=phantomjs.log']
```

For a full list of supported command line arguments, run `phantomjs --help` in your terminal.

Safari

To use Safari, you will need to have at least Safari 10 running on OS X El Capitan or later, and `selenium` 3.0 or later. Once you have these prerequisites, simply specify `Safari` for the `--driver` command line option:

```
pytest --driver Safari
```

Selenium Server/Grid

To run your automated tests against a Selenium server or a Selenium Grid you must have a server running and know the host and port of the server.

By default Selenium will listen on host 127.0.0.1 and port 4444. This is also the default when running tests against a remote driver.

To run your automated tests, simply specify `Remote` as your driver. Browser selection is determined using capabilities. Check the desired capabilities documentation for details of accepted values. There are also a number of browser

specific capabilities that can be set. Be sure to also check the documentation for your chosen driver, as the accepted capabilities may differ:

```
pytest --driver Remote --capability browserName firefox
```

Note that if your server is not running locally or is running on an alternate port you will need to specify the `--host` and `--port` command line options:

```
pytest --driver Remote --host selenium.hostname --port 5555 --capability browserName␣
↪firefox
```

Sauce Labs

To run your automated tests using Sauce Labs, you must provide a valid username and API key. This can be done either by using a `.saucelabs` configuration file in the working directory or your home directory, or by setting the `SAUCELABS_USERNAME` and `SAUCELABS_API_KEY` environment variables.

Alternatively, when using Jenkins CI declarative pipelines, credentials can be set as environment variables as follows:

```
environment {
  SAUCELABS = credentials('SAUCELABS')
}
```

For more information, see using environment variables in Jenkins pipelines.

Configuration

Below is an example `.saucelabs` configuration file:

```
[credentials]
username = username
key = secret
```

Running tests

To run your automated tests, simply specify `SauceLabs` as your driver:

```
pytest --driver SauceLabs --capability browserName Firefox
```

See the supported platforms to help you with your configuration. Additional capabilities can be set using the `--capability` command line arguments. See the test configuration documentation for full details of what can be configured.

BrowserStack

To run your automated tests using BrowserStack, you must provide a valid username and access key. This can be done either by using a `.browserstack` configuration file in the working directory or your home directory, or by setting the `BROWSERSTACK_USERNAME` and `BROWSERSTACK_ACCESS_KEY` environment variables.

Alternatively, when using Jenkins CI declarative pipelines, credentials can be set as environment variables as follows:

```
environment {
  BROWSERSTACK = credentials('BROWSERSTACK')
}
```

For more information, see using environment variables in Jenkins pipelines.

Configuration

Below is an example `.browserstack` configuration file:

```
[credentials]
username = username
key = secret
```

Running tests

To run your automated tests, simply specify `BrowserStack` as your driver:

```
pytest --driver BrowserStack --capability browserName Firefox
```

See the capabilities documentation for additional configuration that can be set using `--capability` command line arguments.

TestingBot

To run your automated tests using TestingBot, you must provide a valid key and secret. This can be done either by using a `.testingbot` configuration file in the working directory or your home directory, or by setting the `TESTINGBOT_KEY` and `TESTINGBOT_SECRET` environment variables.

Alternatively, when using Jenkins CI declarative pipelines, credentials can be set as environment variables as follows:

```
environment {
  TESTINGBOT = credentials('TESTINGBOT')
}
```

Note that for TestingBot, `username` corresponds to `key` and `password` to `secret`.

For more information, see using environment variables in Jenkins pipelines.

Configuration

Below is an example `.testingbot` configuration file:

```
[credentials]
key = key
secret = secret
```

Running tests

To run your automated tests, simply specify `TestingBot` as your driver:

```
pytest --driver TestingBot --capability browserName firefox --capability version 39 --
↪capability platform WIN8
```

See the list of available browsers to help you with your configuration. Additional capabilities can be set using the `--capability` command line arguments. See the test options for full details of what can be configured.

Local tunnel

To run the tests using TestingBot's local tunnel you'll also need to set the `--host` and `--port` command line arguments.

CrossBrowserTesting

To run your automated tests using CrossBrowserTesting, you must provide a valid username and auth key. This can be done either by using a `.crossbrowsertesting` configuration file in the working directory or your home directory, or by setting the `CROSSBROWSERTESTING_USERNAME` and `CROSSBROWSERTESTING_AUTH_KEY` environment variables.

Alternatively, when using Jenkins CI declarative pipelines, credentials can be set as environment variables as follows:

```
environment {
  CROSSBROWSERTESTING = credentials('CROSSBROWSERTESTING')
}
```

For more information, see using environment variables in Jenkins pipelines.

Configuration

Below is an example `.crossbrowsertesting` configuration file:

```
[credentials]
username = username
key = secret
```

Running tests

To run your automated tests, simply specify `CrossBrowserTesting` as your driver:

```
pytest --driver CrossBrowserTesting --capability os_api_name Win10 --capability␣
↪browser_api_name FF46
```

Additional capabilities can be set using the `--capability` command line arguments. See the automation capabilities for full details of what can be configured.

Specifying Capabilities

Configuration options are specified using a capabilities dictionary. This is required when using a Selenium server to specify the target environment, but can also be used to configure local drivers.

Command Line Capabilities

Simple capabilities can be set or overridden on the command line:

```
pytest --driver Remote --capability browserName Firefox
```

Capabilities Files

To specify capabilities, you can provide a JSON file on the command line using the pytest-variables plugin. For example if you had a `capabilties.json` containing your capabilities, you would need to include `--variables capabilities.json` on your command line.

The following is an example of a variables file including capabilities:

```
{ "capabilities": {
    "browserName": "Firefox",
    "platform": "MAC" }
}
```

Capabilities Fixtures

The `session_capabilities` fixture includes capabilities that apply to the entire test session (including any command line or file based capabilities). Any changes to these capabilities will apply to every test. These capabilities are also reported at the top of the HTML report.

```
import pytest
@pytest.fixture(scope='session')
def session_capabilities(session_capabilities):
    session_capabilities['tags'] = ['tag1', 'tag2', 'tag3']
    return session_capabilities
```

The `capabilities` fixture contains all of the session capabilities, plus any capabilities specified by the capabilities marker. Any changes to these capabilities will apply only to the tests covered by scope of the fixture override.

```
import pytest
@pytest.fixture
def capabilities(capabilities):
    capabilities['public'] = 'private'
    return capabilities
```

Capabilities Marker

You can add or change capabilities using the `capabilities` marker:

```
import pytest
@pytest.mark.capabilities(foo='bar')
def test_capabilities(selenium):
    selenium.get('http://www.example.com')
```

Common Selenium Setup

If you have common setup that you want to apply to your tests, such as setting the implicit timeout or window size, you can override the `selenium` fixture:

```
import pytest
@pytest.fixture
def selenium(selenium):
    selenium.implicitly_wait(10)
    selenium.maximize_window()
    return selenium
```

HTML Report

A custom HTML report is generated when the `--html` command line option is given. By default this will include additional debug information for failures.

Debug Types

The following debug information is gathered by default when a test fails:

- **URL** - The current URL open in the browser.
- **HTML** - The HTML source of the page open in the browser.
- **LOG** - All logs available. Note that this will vary depending on the browser and server in use. See logging for more details.
- **SCREENSHOT** - A screenshot of the page open in the browser.

Capturing Debug

To change when debug is captured you can either set `selenium_capture_debug` in a *configuration file*, or set the `SELENIUM_CAPTURE_DEBUG` environment variable. Valid options are: `never`, `failure` (the default), and `always`. Note that always capturing debug will dramatically increase the size of the HTML report.

Excluding Debug

You may need to exclude certain types of debug from the report. For example, log files can contain sensitive information that you may not want to publish. To exclude a type of debug from the report, you can either set `selenium_exclude_debug` in a *configuration file*, or set the `SELENIUM_EXCLUDE_DEBUG` environment variable to a list of the *Debug Types* to exclude.

For example, to exclude HTML, logs, and screenshots from the report, you could set `SELENIUM_EXCLUDE_DEBUG` to `html:logs:screenshot`.

CHAPTER 3

Development

Automated Testing

All pull requests and merges are tested in Travis CI based on the `.travis.yml` file.

Usually, a link to your specific travis build appears in pull requests, but if not, you can find it on the pull requests page

The only way to trigger Travis CI to run again for a pull request, is to submit another change to the pull branch.

Running Tests

You will need Tox installed to run the tests against the supported Python versions.

```
$ pip install tox
$ tox
```

CHAPTER 4

Release Notes

1.11.0 (2017-06-22)

- Add Chrome and Firefox options to capabilities for remote servers.
- Avoid unnecessarily sending Firefox profile to remote servers.
- Add `firefox_arguments` and `firefox_preferences` markers to specify arguments and preferences to pass to the `firefox_options` fixture. Run `pytest --markers` for details.
- Restore host and port in HTML report when using defaults.
- Warn in pytest header when the sensitive URL matches the base URL.
  - Thanks to @Jenselme for the PR
- Use a separate log file for each driver instance.

1.10.0 (2017-05-04)

- Add alternate credentials environment variables for Jenkins declarative pipelines.
  - Thanks to @BeyondEvil for the PR
- Deprecate `--firefox-extension`, `--firefox-path`, `--firefox-preference`, and `--firefox-profile` command line options. The preferred way to set these is now through the `firefox_options` fixture.
- Only create a Firefox profile if `--firefox-extension`, `--firefox-preference`, or `--firefox-profile` is specified.
- Add `chrome_options` fixture for configuring Google Chrome.
- Add `driver_args` fixture for adding command line arguments to the driver services. Currently only used by Chrome and PhantomJS.
- Add support for TestingBot local tunnel via `--host` and `--port` command line options.

  - Thanks to @micheletest for the report and to @BeyondEvil for the PR
- Add support for Microsoft Edge.
  - Thanks to @birdsarah for the PR
- Add driver logs to HTML report.
  - Thanks to @jrbenny35 for the PR

1.9.1 (2017-03-01)

- Add capabilities to metadata during `pytest_configure` hook instead of the `session_capabilities` fixture to make them available to other plugins earlier.

1.9.0 (2017-02-27)

- Add driver and session capabilities to metadata provided by pytest-metadata

1.8.0 (2017-01-25)

- **BREAKING CHANGE:** Moved cloud testing provider credentials into separate files for improved security.
  - If you are using the environment variables for specifying cloud testing provider credentials, then you will not be affected.
  - If you are storing credentials from any of the cloud testing providers in one of the default configuration files then they will no longer be used. These files are often checked into source code repositories, so it was previously very easy to accidentally expose your credentials.
  - Each cloud provider now has their own configuration file, such as `.browserstack`, `.crossbrowsertesting`, `.saucelabs`, `.testingbot` and these can be located in the working directory or in the user's home directory. This provides a convenient way to set up these files globally, and override them for individual projects.
  - To migrate, check `pytest.ini`, `tox.ini`, and `setup.cfg` for any keys starting with `browserstack_`, `crossbrowsertesting_`, `saucelabs_`, or `testingbot_`. If you find any, create a new configuration file for the appropriate cloud testing provider with your credentials, and remove the entries from the original file.
  - The configuration keys can differ between cloud testing providers, so please check the *User Guide* for details.
  - See #60 for for original issue and related patch.

1.7.0 (2016-11-29)

- Introduced a `firefox_options` fixture.
- Switched to Firefox options for speciying binary and profile.

1.6.0 (2016-11-17)

- Added support for CrossBrowserTesting.

1.5.1 (2016-11-03)

- Fix issues with Internet Explorer driver.

1.5.0 (2016-10-13)

- Replaced driver fixtures with generic `driver_class` fixture.
- Introduced a `driver_kwargs` fixture.

1.4.0 (2016-09-30)

- Added support for Safari.

1.3.1 (2016-07-13)

- Made `firefox_path` a session scoped fixture.

1.3.0 (2016-07-12)

- Moved retrieval of Firefox path to `firefox_path` fixture.
- Added driver and sensitive URL to report header.
- Moved base URL implementation to the pytest-base-url plugin.

1.2.1 (2016-02-25)

- Fixed regression with Chrome, PhantomJS, and Internet Explorer drivers.

1.2.0 (2016-02-25)

- Added support for Python 3.
- Introduced a new capabilities fixture to combine session and marker capabilities.
- **BREAKING CHANGE:** Renamed session scoped capabilities fixture to session_capabilities.
  - If you have any `capabilities` fixture overrides, they will need to be renamed to `session_capabilities`.

- Move driver implementations into fixtures and plugins.

1.1 (2015-12-14)

- Consistently stash the base URL in the configuration options.
- Drop support for pytest 2.6.
- Avoid deprecation warnings in pytest 2.8.
- Report warnings when gathering debug fails. (#40)

1.0 (2015-10-26)

- Official release

1.0b5 (2015-10-20)

- Assign an initial value to log_types. (#38)

1.0b4 (2015-10-19)

- Use strings for HTML to support serialization when running multiple processes.
- Catch exception if driver has not implemented log types.

1.0b3 (2015-10-14)

- Allow the sensitive URL regex to be specified in a configuration file.

1.0b2 (2015-10-06)

- Added support for non ASCII characters in log files. (#33)
- Added support for excluding any type of debug.

1.0b1 (2015-09-08)

- Initial beta