Brome Documentation

Release 0.0.7

Jean-François Parent

December 07, 2015

Contents

Brome is a Framework for Selenium

CHAPTER 1

Documentation

1.1 Installation

1.1.1 Create the brome project using cookie-cutter

Install cookie-cutter:

```
$ [sudo] pip install cookiecutter
```

Create the brome project:

```
$ mkdir brome_project
$ cd brome_project
$ cookiecutter https://github.com/brome-hq/cookiecutter-brome -f
```

1.1.2 Create a virtual env *[optional]*

Install virtualenv and virtualenvwrapper:

```
$ pip install virtualenv virtualenvwrapper
```

Add this line to your .bash_profile:

```
source /usr/local/bin/virtualenvwrapper.sh
```

Reload your .bash_profile:

```
$ source ~/.bash_profile
```

Create your virtualenv:

```
$ mkvirtualenv venv
```

Activate the virtualenv:

```
$ workon venv
```

1.1.3 Install brome

```
$ pip install brome
```

1.2 Configuration

Here is all the configuration options that brome support. You have to provide a yaml configuration file to brome when you create it. This step is already done in the bro executable:

```
brome = Brome(
    config_path = os.path.join(HERE, "config", "brome.yml"), #<--- this file
    selector_dict = selector_dict,
    test_dict = test_dict,
    browsers_config_path = os.path.join(HERE, "config", "browsers_config.yml"),
    absolute_path = HERE
)
```

Inside the brome.yml configuration file is a bunch of section with options. You can control the behavior the runner, the proxy driver, the proxy element, put some custom project config, etc... Below is a break down of each of the brome configuration options.

1.2.1 sample configuration file

```
saucelabs:
  username: ''
  key: ''
browserstack:
  username: ''
  key: ''
browser:
  maximize_window: false
  window_height: 725
  window_width: 1650
  window_x_position: 0
  window_y_position: 0
database:
  sqlalchemy.url: 'sqlite:///unittest.db'
ec2:
  wait_after_instance_launched: 30
  wait_until_system_and_instance_check_performed: true
grid_runner:
  kill_selenium_server: true
  max_running_time: 7200
  selenium_hub_config: '/resources/hub-config.json'
  selenium_server_command: 'java -jar {selenium_server_jar_path} -role hub -hubConfig {selenium_hub_c
  selenium_server_ip: 'localhost'
  selenium_server_jar_path: '/resources/selenium-server-standalone.jar'
  selenium_server_port: 4444
  start_selenium_server: false
highlight:
  highlight_on_assertion_failure: true
  highlight_on_assertion_success: true
  highlight_when_element_is_clicked: true
  highlight_when_element_is_visible: true
  highlight_when_element_receive_keys: true
  style_on_assertion_failure: 'background: red; border: 2px solid black;'
```

```
  style_on_assertion_success: 'background: green; border: 2px solid black;'
  style_when_element_is_clicked: 'background: yellow; border: 2px solid red;'
  style_when_element_is_visible: 'background: purple; border: 2px solid black;'
  style_when_element_receive_keys: 'background: yellow; border: 2px solid red;'
  use_highlight: true
logger_runner:
  filelogger: false
  format: "[%(batchid)s]\e[32m%(message)s\e[0m"
  level: 'INFO'
  streamlogger: true
logger_test:
  filelogger: false
  format: "[%(batchid)s]\e[34m(%(testname)s)\e[0m:\e[32m%(message)s\e\[0m"
  level: 'INFO'
  streamlogger: true
project:
  test_batch_result_path: false
  url: 'http://localhost:7777'
proxy_driver:
  use_javascript_dnd: true
  default_timeout: 5
  intercept_javascript_error: true
  raise_exception: true
  take_screenshot_on_assertion_failure: true
  take_screenshot_on_assertion_success: false
  validate_css_selector: true
  validate_xpath_selector: true
  wait_until_not_present_before_assert_not_present: true
  wait_until_not_visible_before_assert_not_visible: true
  wait_until_present_before_assert_present: true
  wait_until_present_before_find: true
  wait_until_visible_before_assert_visible: true
  wait_until_visible_before_find: true
runner:
  cache_screenshot: true
  embed_on_assertion_failure: false
  embed_on_assertion_success: false
  embed_on_test_crash: false
  play_sound_on_assertion_failure: false
  play_sound_on_assertion_success: false
  play_sound_on_ipython_embed: false
  play_sound_on_pdb: false
  play_sound_on_test_crash: false
  play_sound_on_test_finished: false
  sound_on_assertion_failure: '{testid} failed'
  sound_on_assertion_success: '{testid} succeeded'
  sound_on_ipython_embed: 'Attention required'
  sound_on_pdb: 'Attention required'
  sound_on_test_crash: 'Crash'
  sound_on_test_finished: 'Test finished'
webserver:
  open_browser: false
  ASSETS_DEBUG: true
  CACHE_TYPE: 'simple'
  CLOSED_REGISTRATION: false
  DEBUG: false
  DEBUG_TB_ENABLED: false
  DEBUG_TB_INTERCEPT_REDIRECTS: false
```

```
  HOST: 'localhost'
  PORT: 5000
  REGISTRATION_TOKEN: ''
  SECRET_KEY: ''
  SHOW_TEST_INSTANCES: true
  SHOW_VIDEO_CAPTURE: true
  filelogger: false
  level: 'INFO'
  streamlogger: false
```

1.2.2 project

- **test_batch_result_path**: The test path where the test batch result file will be create. If you don't want to save any file when a test batch run, just set this options to False. *str [path] | bool (false only) (default: '')*
- **url**: The url of the server on which the test run (must include the protocol) e.g.:https://the-internet.herokuapp.com/ *str [url] (default: '')*

1.2.3 browserstack

- **username**: Browserstack username *str (default: '')*
- **key**: Browserstack key *str (default: '')*

1.2.4 saucelabs

- **username**: Saucelabs username *str (default: '')*
- **key**: Saucelabs key *str (default: '')*

1.2.5 proxy_driver

- **use_javascript_dnd**: Use javascript to perform drag and drop. If set to false then the ActionChains.drag_and_drop with be used instead. *bool (default: false)*
- **wait_until_visible_before_find**: If this options is set to true then each time you use the driver.find(selector) the proxy_driver will wait until the element is visible; if the element is not visible before the given timeout then it may wait_until_present(selector), raise an exception or return false. All of this is configurable from the brome.yml or provided to the function find(selector, wait_until_visible = (False | True)) directly via kwargs. *bool (default: false)*
- **intercept_javascript_error**: If set to true this options will execute some javascript code on each driver.get() that will intercept javascript error. *bool (default: false)*
- **validate_xpath_selector**: If set to true the proxy driver will raise an exception if the provided xpath selector is invalid. *bool (default: false)*
- **validate_css_selector**: If set to true the proxy driver will raise an exception if the provided css selector is invalid. *bool (default: false)*
- **default_timeout**: The default timeout in second. This will be used in a lot of the proxy driver function (wait_until_*); you can overwrite this default with the timeout kwargs. *int (second) (default: 5)*
- **raise_exception**: This options tell the brome driver to raise exception on failure (find_*, wait_until_*) or just return a bool instead. *bool (default: true)*

- **wait_until_present_before_assert_present**: Wait until not present before assert present. *bool (default: false)*
- **wait_until_not_present_before_assert_not_present**: Wait until not present before assert not present. *bool (default: false)*
- **wait_until_not_visible_before_assert_not_visible**: Wait until not visible before assert not visible. *bool (default: false)*
- **wait_until_visible_before_assert_visible**: Wait until visible before assert visible. *bool (default: false)*
- **wait_until_present_before_find**: Wait until visible before find. *bool (default: false)*
- **take_screenshot_on_assertion_success**: Take screenshot on assertion success *bool (default: false)*
- **take_screenshot_on_assertion_failure**: Take screenshot on assertion failure *bool (default: false)*

1.2.6 proxy_element

- **use_touch_instead_of_click**: Use touch instead of click. *bool (default: false)*

1.2.7 browser

- **window_x_position**: Window x position. *int (default: 0)*
- **window_y_position**: Window y position. *int (default: 0)*
- **window_height**: Window height. *int (default: 725)*
- **window_width**: Window width. *int (default: 1650)*
- **maximize_window**: Maximize window. *Note: this may not work in a xvfb environment; so set the width and height manually in this case. bool (default: false)*

1.2.8 highlight

- **highlight_on_assertion_success**: Highlight on assertion success. *bool (default: false)*
- **highlight_on_assertion_failure**: Highlight on assertion failure. *bool (default: false)*
- **highlight_when_element_is_clicked**: Highlight when element is clicked. *bool (default: false)*
- **highlight_when_element_receive_keys**: Highlight when element received keys. *bool (default: false)*
- **highlight_when_element_is_visible**: Highlight when element is visible. *bool (default: false)*
- **style_when_element_is_clicked**: Style when element is clicked. *str 'background: yellow; border: 2px solid red;'*
- **style_when_element_receive_keys**: Style when element receive keys. *str 'background: yellow; border: 2px solid red;'*
- **style_on_assertion_failure**: Style on assertion failure. *str 'background: red; border: 2px solid black;'*
- **style_on_assertion_success**: Style on assertion success. *str 'background: green; border: 2px solid black;'*
- **style_when_element_is_visible**: Style when element is visible. *str 'background: purple; border: 2px solid black;'*
- **use_highlight**: Use highlight. *bool (default: false)*

1.2.9 runner

- **embed_on_assertion_success**: Embed on assertion success. *bool (default: false)*
- **embed_on_assertion_failure**: Embed on assertion failure. *bool (default: false)*
- **embed_on_test_crash**: Embed on test crash. *bool (default: false)*
- **play_sound_on_test_crash**: Play sound on test crash. *bool (default: false)*
- **play_sound_on_assertion_success**: Play sound on assertion success. *bool (default: false)*
- **play_sound_on_assertion_failure**: Play sound on assertion failure. *bool (default: false)*
- **play_sound_on_test_finished**: Play sound on test batch finished. *bool (default: false)*
- **play_sound_on_ipython_embed**: Play sound on ipython embed. *bool (default: false)*
- **play_sound_on_pdb**: Play sound on pdb. *bool (default: false)*
- **sound_on_test_crash**: Sound on test crash. *str Crash*
- **sound_on_assertion_success**: sound on assertion success. *str {testid} succeeded*
- **sound_on_assertion_failure**: Sound on assertion failure. *str {testid} failed*
- **sound_on_test_finished**: Sound on test batch finished. *str Test finished*
- **sound_on_ipython_embed**: Sound on ipython embed. *str Attention required*
- **sound_on_pdb**: Sound on pdb. *str Attention required*
- **cache_screenshot**: Use the cache screenshot. *bool (default: true)*

1.2.10 database

- **sqlalchemy.url**: the database url *str (default: '')*

1.2.11 logger_runner

- **level**: *'DEBUG'* | *'INFO'* | *'WARNING'* | *'ERROR'* | *'CRITICAL' (default: INFO)*
- **streamlogger**: The logger with output to the sdtout. *bool (default: true)*
- **filelogger**: The logger with output to a file in the test batch result directory. *bool (default: true)*
- **format**: Logger format. *str (default: [%(batchid)s]%(message)s)*

1.2.12 logger_test

- **level**: *'DEBUG'* | *'INFO'* | *'WARNING'* | *'ERROR'* | *'CRITICAL' (default: INFO)*
- **streamlogger**: The logger with output to the sdtout. *bool (default: true)*
- **filelogger**: The logger with output to a file in the test batch result directory. *bool (default: true)*
- **format**: Logger format. *str (default: [%(batchid)s](%(testname)s):%(message)s)*

1.2.13 ec2

- **wait_after_instance_launched**: Wait X seconds after the instances are launched. *int [second] (default: 30)*
- **wait_until_system_and_instance_check_performed**: Wait until system and instance checks are performed. *bool (default: true)*

1.2.14 grid_runner

- **max_running_time**: This is the time limit the grid runner can run before raising a TimeoutException. This is to prevent a test batch from running forever using up precious resources. *(int [second]) (default: 7200)*
- **start_selenium_server**: Start selenium server automatically. *bool (default: true)*
- **selenium_server_ip**: Selenium server ip address. *str (default: 'localhost')*
- **selenium_server_port**: Selenium port. *int (default: 4444)*
- **selenium_server_command**: Selenium server command. *str (default: '')*
- **selenium_server_jar_path**: Selenium server jar path. *str [path] (default: '')*
- **selenium_hub_config**: Selenium server hub config path. *str [path] (default: '')*
- **kill_selenium_server**: Kill selenium server when the test batch finished. *bool (default: true)*

1.2.15 webserver

- **open_browser**: Open the webserver index in a new tab on start. *bool (default: false)*
- **level**: *'DEBUG'* | *'INFO'* | *'WARNING'* | *'ERROR'* | *'CRITICAL' (default: INFO)*
- **streamlogger**: The logger with output to the sdtout. *bool (default: true)*
- **filelogger**: The logger with output to a file in the test batch result directory. *bool (default: true)*
- **CLOSED_REGISTRATION**: This options will required the user to enter a token if he want to register in the brome webserver. *bool (default: false)*
- **REGISTRATION_TOKEN**: The token used to register in the brome webserver. *str (default: '')*
- **HOST**: [FLASK CONFIG]
- **PORT**: [FLASK CONFIG]
- **DEBUG**: [FLASK CONFIG]
- **CACHE_TYPE**: [FLASK CONFIG]
- **ASSETS_DEBUG**: [FLASK CONFIG]
- **DEBUG_TB_INTERCEPT_REDIRECTS**: [FLASK CONFIG]
- **DEBUG_TB_ENABLED**: [FLASK CONFIG]
- **SECRET_KEY**: [FLASK CONFIG]

1.3 Brome CLI

In order to use Brome you must first create a brome object. The project template come with a *bro* python script that do just that:

```
#!/usr/bin/env python

import sys
import os

from brome import Brome

from model.selector import selector_dict
from model.test_dict import test_dict

if __name__ == '__main__':

    HERE = os.path.abspath(os.path.dirname(__file__))

    brome = Brome(
        config_path = os.path.join(HERE, "config", "brome.yml"),
        selector_dict = selector_dict,
        test_dict = test_dict,
        browsers_config_path = os.path.join(HERE, "config", "browsers_config.yml"),
        absolute_path = HERE
    )

    brome.execute(sys.argv)
```

You provide Brome with a few things:

- *config_path*: this is the path to the brome yaml config (see *Configuration*).
- *selector_dict* **[optional]**: this is the dictionary holding your selector (see *Selector variable dictionary*).
- *test_dict* **[optional]**: this is the dictionary holding your test description (see *Assertion*).
- *browser_config_path*: this is the path to the browser yaml config (see *Browsers*).
- *absolute_path*: the path of your project.

1.3.1 Execute

To get started:

```
$ ./bro -h
>$ ./bro admin | run | webserver | list | find
```

To get help for one specific command:

```
$ ./bro admin -h
>usage: bro [-h] [--generate-config] [--reset] [--create-database]
       [--delete-test-states] [--delete-test-results] [--reset-database]
       [--delete-database] [--update-test]

>Brome admin

>optional flags:
>  -h, --help            show this help message and exit
```

```
>  --generate-config     Generate the default brome config
>  --reset               Reset the database + delete the test batch results +
>                        update the test table + delete all the test state
>  --create-database     Create the project database
>  --delete-test-states  Delete all the test states
>  --delete-test-results Delete all the test batch results
>  --reset-database      Reset the project database
>  --delete-database     Delete the project database
>  --update-test         Update the test in the database
```

1.3.2 Run

The run command can run your test remotely or locally.

Local

To run a test locally use the *-l* flag:

```
$ ./bro run -l 'browser-id' -s 'test-name'
```

So if you want to run the test named */path/to/project/tests/test_login.py* on firefox then use this command:

```
$ ./bro run -l 'firefox' -s 'login'
```

Remote

If you want to run your test remotely then use the *-r* flag:

```
$ ./bro run -r 'firefox_virtualbox' -s 'login'
```

Brome config

You can overwrite a brome config for one particular run with the *–brome-config* flag. Let say you want to disable the sound on a test crash and on an assertion failure:

```
$ ./bro run -l 'firefox' -s 'login' --brome-config "runner:play_sound_on_test_crash=False,runner:play
```

Test config

You can pass a config value to a test scenario also using *–test-config*:

```
$ ./bro run -l 'firefox' -s 'login' --test-config "register=True,username='test'"

#/path/to/project/tests/test_login.py
from model.basetest import BaseTest

class Test(BaseTest):

    name = 'Login'

    def run(self, **kwargs):
```

```
        if self._test_config.get('register'):
            self.app.register(username = self._test_config.get('username'))
```

Test discovery

You have 3 ways of telling brome which test scenario to run.

Search

The first one is with the *-s* flag. The *-s* stand for search. Brome will search for a test scenario under your *tests* folder that start with the prefix *test_* and end with *.py*. If you want to run the scenario named *test_login.py* then search for *login*. You can also use a python list index here. Let say you want to run the test scenario index *2* then use *-s [2]*. To find out your test scenario index use the *list* command (see *List*). Python slice are also supported e.g.: *-s [3:7]* will run the test scenario index from 3 to 7.

Name

The second way is with the *-n* flag. The *-n* flag stand for name. If your test scenario doesn't start with the prefix *test_* then brome won't consider it when you use the search flag. The use case for this is when you have some code that you don't want to run automatically (e.g.: data creation, administrative stuff, etc):

```
$ ls /path/to/project/tests
> register_user.py
> test_login.py
> test_register.py
$ ./bro run -l 'firefox' -n 'register_user' #Work
$ ./bro run -l 'firefox' -s 'register_user' #Won't work
```

This separation is pretty useful when you use the webserver to launch a test batch.

Test file

The last way is by using a yaml file that contains a list of all the test scenario that you want to run (work only with the *-s* flag):

```
$ cat test_file.yml
> - wait_until_present
> - is_present
> - assert_present

$ ./bro run -l 'firefox' --test-file test_file.yml
```

1.3.3 Admin

Reset

This command will reset the database, delete all the test files, update the test table and delete all the test states:

```
$ ./bro admin --reset
```

Generate config

This command will generate a brome default config:

```
$ ./bro admin --generate-config
```

It will overwrite your actual brome.yml config with the default value for each config.

Create database

This command is not useful unless you use a server database like MySQL. It is not necessary to use this command with SQLite:

```
$ ./bro admin --create-database
```

Reset database

This will delete the database and then recreate it:

```
$ ./bro admin --reset-database
```

Delete database

This will delete the database:

```
$ ./bro admin --delete-database
```

Delete test states

This will delete all the pickle file found in */path/to/project/tests/states/* (see *State*):

```
$ ./bro admin --delete-test-states
```

Delete test results

This will delete all the test batch data files found under your brome config *project:test_batch_result_path*:

```
$ ./bro admin --delete-test-results
```

Update test

This command is not that useful since the test table is updated automatically but if you find that the test table has not been updated automatically the use this command to force it:

```
$ ./bro admin --update-test
```

1.3.4 Webserver

To start the webserver use the *webserver* command:

```
$ ./bro webserver
```

This use the build in Flask webserver.

Tornado

If you want to start a tornado webserver instead use the *-t* flag:

```
$ ./bro webserver -t
```

If you are over ssh and you want to start the webserver in the background and detach it from the current ssh session then use this bash command:

```
$ nohup ./bro webserver -t &
```

1.3.5 List

To find out the index of your test scenario use the *list* command:

```
$ ./bro list
> [0]   login
> [1]   register
> [2]   logout
```

1.3.6 Find

The *find* command is use to find either a *test_id* or a *selector_variable*:

```
$ ./bro find --test-id '#1'

$ ./bro find --selector 'sv:login_username'
```

1.4 Locating Elements

1.4.1 Find methods

Use the *proxy_driver* (*pdriver*) to locate element on the web page. Three methods exist:

```
pdriver.find("sv:selector_variable")

pdriver.find_last("nm:form")

pdriver.find_all("tn:div")
```

The find method accept three kwargs:

```
pdriver.find("nm:button", raise_exception = False)

pdriver.find_all("id:1", wait_until_visible = False)

pdriver.find_last("cs:div > span", wait_until_present = False, raise_exception = True)

pdriver.find_all(
    "sv:selector_variable",
    wait_until_visible = True,
    wait_until_present = False,
    raise_exception = True
)
```

The defaults for the kwargs wait_until_present, wait_until_visible and raise_exception can be set respectively with: * proxy_driver:wait_until_present_before_find * proxy_driver:wait_until_visible_before_find * proxy_driver:raise_exception

The *find* and *find_last* method return a *proxy_element*.

The *find_all* method return a *proxy_element_list*.

By id

```
pdriver.find("id:button-1")
```

By css selector

```
pdriver.find("cs:div > span")
```

By name

```
pdriver.find("nm:button-1")
```

By class name

```
pdriver.find("cn:button")
```

By tag name

```
pdriver.find_all("tn:div")
```

By link text

```
pdriver.find("lt:register now!")
```

By partial link text

```
pdriver.find("pl:register")
```

By selector variable

```
pdriver.find("sv:button_1")
```

1.4.2 List of selectors

If you want to create a selector from multiple selector you can pass a list of selector to the *find_** method:

```
pdriver.find(["sv:selector_v1", "sv:selector_v2", "xp://*[contains(@class, 'button')]"])
```

1.4.3 Selectors validation

The xpath and css selector will be validated if the config *proxy_driver:validate_xpath_selector* and *proxy_driver:validate_css_selector* are set to true.

1.4.4 Selector variable dictionary

A selector dictionary can be provided to brome:

```
selector_dict = {}
selector_dict['example_find_by_tag_name'] = "tn:a"
selector_dict['example_find_by_id'] = "id:1"
selector_dict['example_find_by_xpath'] = "xp://*[@class = 'xpath']"
selector_dict['example_find_by_partial_link_text'] = "pl:partial link text"
selector_dict['example_find_by_link_text'] = "lt:link text"
selector_dict['example_find_by_css_selector'] = "cs:.classname"
selector_dict['example_find_by_classname'] = "cn:classname"
selector_dict['example_find_by_name'] = "nm:name"
selector_dict['example_multiple_selector'] = {
    "default" : "xp://*[contains(@class, 'default')]",
    "chrome|iphone|android" : "xp://*[contains(@class, 'special')]"
}

brome = Brome(
    config_path = os.path.join(HERE, "config", "brome.yml"),
    selector_dict = selector_dict, #<--- this dict
    test_dict = test_dict,
    browsers_config_path = os.path.join(HERE, "config", "browsers_config.yml"),
    absolute_path = HERE
)
```

So later on in your code you can use the selector variable to find elements:

```
pdriver.find("sv:example_find_by_name")
```

Also a selector variable can vary from browser to browser:

```
selector_dict['example_multiple_selector'] = {
    "default" : "xp://*[contains(@class, 'default')]",
    "chrome|iphone|android" : "xp://*[contains(@class, 'special')]"
}
```

It support the browserName, version and platform returned by the pdriver._driver.capabilities

1.4.5 Plain selenium methods

If you want to use the selenium location methods just use:

```
pdriver._driver.find_element_by_id
pdriver._driver.find_element_by_name
pdriver._driver.find_element_by_xpath
pdriver._driver.find_element_by_link_text
pdriver._driver.find_element_by_partial_link_text
pdriver._driver.find_element_by_tag_name
pdriver._driver.find_element_by_class_name
pdriver._driver.find_element_by_css_selector
pdriver._driver.find_elements_by_name
pdriver._driver.find_elements_by_xpath
pdriver._driver.find_elements_by_link_text
pdriver._driver.find_elements_by_partial_link_text
pdriver._driver.find_elements_by_tag_name
pdriver._driver.find_elements_by_class_name
pdriver._driver.find_elements_by_css_selector
```

Note that this will return a selenium webelement and not a *proxy_element* or *proxy_element_list*

1.5 Waiting For Elements

1.5.1 Waiting methods

Use the *proxy_driver* (*pdriver*) to wait for element to be clickable, present, not present, visible or not visible:

```
pdriver.wait_until_clickable("xp://*[contains(@class, 'button')]")

pdriver.wait_until_present("cn:test")

pdriver.wait_until_not_present("sv:submit_button")

pdriver.wait_until_visible("tn:div")

pdriver.wait_until_not_visible("cs:div > span")
```

Kwargs and config

All the waiting methods accept the two following kwargs:

- *timeout* (int) second before a timeout exception is raise
- *raise_exception* (bool) raise an exception or return a bool

The config default for these kwargs are respectively:

- *proxy_driver:default_timeout*
- *proxy_driver:raise_exception*

Examples

```
pdriver.wait_until_clickable("xp://*[contains(@class, 'button')]", timeout = 30)

pdriver.wait_until_present("cn:test", raise_exception = True)
```

1.5.2 Not waiting methods

Use the *proxy_driver* (*pdriver*) to find out if an element is present or visible:

```
pdriver.is_visible("xp://*[contains(@class, 'button')])

pdriver.is_present("cn:test")
```

Examples

```
if pdriver.is_visible("sv:login_btn'):
    pdriver.find("sv:login_btn").click()

if pdriver.is_present("sv:hidden_field"):
    pdriver.find("sv:username_field").clear()
```

1.6 Interactions

1.6.1 Proxy driver

Get

Same as the selenium *get* method except that if the config *proxy_driver:intercept_javascript_error* is set to True then each time you call *get* it will inject the javascript code to intercept javascript error:

```
pdriver.get("http://www.example.com")
```

Inject javascript script

This method will inject the provided javascript script into the current page:

```
pdriver.inject_js_script("https://code.jquery.com/jquery-2.1.4.min.js")

js_path = os.path.join(
    pdriver.get_config_value("project:absolute_path"),
    "resources",
    "special_module.js"
)
pdriver.inject_js_script(js_path)
```

Init javascript error interception

This will inject the javascript code responsible of intercepting the javascript error:

```
self._driver.execute_script("""
    window.jsErrors = [];
    window.onerror = function (errorMessage, url, lineNumber) {
        var message = 'Error: ' + errorMessage;
        window.jsErrors.push(message);
        return false;
    };
""")
```

Print javascript error

This will print the gathered javascript error using the logger:

```
pdriver.print_javascript_error()
```

Note that each time you access the javascript error, the list holding them is reset.

Get javascript error

This will return a string or a list of all the gathered javascript error:

```
pdriver.get_javascript_error(return_type = 'string')

pdriver.get_javascript_error(return_type = 'list')
```

Note that each time you access the javascript error, the list holding them is reset.

Pdb

This will start a python debugger:

```
pdriver.pdb()
```

Note that calling *pdb* won't work in a multithread context, so whenever you use the -r switch with the *bro* executable the *pdb* call won't work.

Embed

This will start a ipython embed:

```
pdriver.embed()

pdriver.embed(title = 'Ipython embed')
```

Note that calling *embed* won't work in a multithread context, so whenever you use the -r switch with the *bro* executable the *embed* call won't work.

Drag and drop

The drag and drop have two implementation: one use javascript; the other use the selenium ActionChains:

```
pdriver.drag_and_drop("sv:source", "sv:destination", use_javascript_dnd = True)

pdriver.drag_and_drop("sv:source", "sv:destination", use_javascript_dnd = False)

pdriver.drag_and_drop("sv:source", "sv:destination") #use_javascript_dnd will be initialize from the
```

Take screenshot

Take a screenshot:

```
pdriver.get_config_value('runner:cache_screenshot')
>>>True
pdriver.take_screenshot('login screen')
pdriver.take_screenshot('login screen') #won't save any thing to disc

pdriver.get_config_value('runner:cache_screenshot')
>>>False
pdriver.take_screenshot('login screen')
pdriver.take_screenshot('login screen') #first screenshot will be overridden
```

Note: If the *runner:cache_screenshot* config is set to True then screenshot sharing all the same name will be saved only once

1.6.2 Proxy element

Click

Click on the element:

```
pdriver.find("sv:login_button").click()
pdriver.find("sv:login_button").click(highlight = False)
pdriver.find("sv:login_button").click(wait_until_clickable = False)
```

Note: if the first *click* raise an exception then another *click* will be attempt; if the second *click* also fail then a exception will be raised. If you don't want this kind of behaviour then you can use the *click* selenium method instead:

```
pdriver.find("sv:login_button")._element.click() #plain selenium method
```

Send keys

Send keys to the element:

```
pdriver.find("sv:username_input"").send_keys('username')
pdriver.find("sv:username_input"").send_keys('new_username', clear = True)
pdriver.find("sv:username_input"").send_keys('new_username', highlight = False)
pdriver.find("sv:username_input"").send_keys('new_username', wait_until_clickable = False)
```

Note: if the first *send_keys* raise an exception then another click will be attempt; if the second *send_keys* also fail then a exception will be raised. If you don't want this kind of behaviour then you can use the *send_keys* selenium method instead:

```
pdriver.find("sv:login_button")._element.send_keys('username') #plain selenium method
```

Clear

Clear the element:

```
pdriver.find("sv:username_input").clear()
```

Note: if the first *clear* raise an exception then another *clear* will be attempt; if the second *clear* also fail then a exception will be raised. If you don't want this kind of behaviour then you can use the *clear* selenium method instead:

```
pdriver.find("sv:login_button")._element.clear() #plain selenium method
```

Highlight

Highlight an element:

```
style = 'background: red; border: 2px solid black;'
pdriver.find("sv:login_button").highlight(style = style, highlight_time = .3)
```

Scroll into view

Scroll into view where the element is located:

```
pdriver.find("sv:last_element").scroll_into_view()
```

Select all

Select all text found in the element:

```
pdriver.find("sv:title_input").select_all()
```

1.7 Assertion

All assert function return a boolean: True if the assertion succeed; False otherwise.

If you gave a test dictionary to the Brome object in the bro executable then you can give a test id to the assert function so it can save the test result for you:

```
test_dict = {}
test_dict['#1'] = "The login button is visible"
test_dict['#2'] = {
    'name': 'Test',
    'embed': False
}

brome = Brome(
    config_path = os.path.join(HERE, "config", "brome.yml"),
    selector_dict = selector_dict,
    test_dict = test_dict, # <-- this dict
    browsers_config_path = os.path.join(HERE, "config", "browsers_config.yml"),
    absolute_path = HERE
)
```

So later in your code you can do this:

```
pdriver.assert_visible("sv:login_button", '#1')
>>>True
```

Or if you prefer you can write the test description inline instead:

```
pdriver.assert_visible("sv:login_button", 'The login button is visible')
>>>True
```

If you don't want this assertion to be saved then use the assert function like this:

```
pdriver.assert_visible("sv:login_button")
>>>True
```

Assertion function never raise an exception.

The following config affect the fonctionality of assert functions:

- *runner:embed_on_assertion_success*
- *runner:embed_on_assertion_failure*
- *runner:play_sound_on_assertion_success*
- *runner:play_sound_on_assertion_failure*
- *proxy_driver:take_screenshot_on_assertion_failure*
- *proxy_driver:take_screenshot_on_assertion_success*

If you have a bug that won't be fixed anytime soon and your config *runner:embed_on_assertion_failure* is set to True then you can change your test_dict like so to stop embed on this particular test:

```
test_dict['#2'] = {
    'name': 'Test',
    'embed': False
}
```

1.7.1 Assertion functions

Here is the list of all assert function found in the *pdriver*:

Assert present

This will assert that the element is present in the dom:

```
assert_present(selector, "#2")

assert_present(selector, "#2", wait_until_present = False)

assert_present(selector, "#2", wait_until_present = True)
```

- The default for the *wait_until_present* kwargs is *proxy_driver:wait_until_present_before_assert_present*.

Assert not present

This will assert that the element is not present in the dom:

```
assert_not_present(selector, "#2")

assert_not_present(selector, "#2", wait_until_not_present = True)

assert_not_present(selector, "#2", wait_until_not_present = False)
```

- The default for the *wait_until_not_present* kwargs is *proxy_driver:wait_until_not_present_before_assert_not_present*.

Assert visible

This will assert that the element is visible in the dom:

```
assert_visible(selector, "#2")

assert_visible(selector, "#2", highlight = False)

assert_visible(selector, "#2", wait_until_visible = False)
```

- The default for the *wait_until_visible* kwargs is *proxy_driver:wait_until_visible_before_assert_visible*.
- The default for the *highlight* kwargs is *proxy_driver:highlight_on_assertion_success*.
- The highlight style is configurable with the config *highlight:style_on_assertion_success*.

Assert not visible

This will assert that the element is not visible in the dom:

```
assert_not_visible(selector, "#2")

assert_not_visible(selector, "#2", highlight = False)

assert_not_visible(selector, "#2", wait_until_not_visible = False)
```

- The default for the *wait_until_not_visible* kwargs is *proxy_driver:wait_until_not_visible_before_assert_not_visible*.
- The default for the *highlight* kwargs is *proxy_driver:highlight_on_assertion_failure*.
- The highlight style is configurable with the config *highlight:style_on_assertion_failure*.

Assert text equal

This will assert that the element's test is equal to the given value:

```
assert_text_equal("sv:username_input", "user", "#2")

pdriver.assert_text_equal("sv:username_input", "error", '#2', highlight = False)

pdriver.assert_text_equal("sv:username_input", "error", '#2', wait_until_visible = False)
```

- The default for the *wait_until_visible* kwargs is *proxy_driver:wait_until_visible_before_assert_visible*.
- The default for the *highlight* kwargs is *proxy_driver:highlight_on_assertion_success*.
- The highlight style is configurable with the config *highlight:style_on_assertion_success*.
- The highlight style is configurable with the config *highlight:style_on_assertion_failure*.

Assert text not equal

This will assert that the element's test is not equal to the given value:

```
pdriver.assert_text_not_equal("sv:username_input", "error", '#2')

pdriver.assert_text_not_equal("sv:username_input", "error", '#2', highlight = False)

pdriver.assert_text_not_equal("sv:username_input", "error", '#2', wait_until_visible = False)
```

- The default for the *wait_until_visible* kwargs is *proxy_driver:wait_until_visible_before_assert_visible*.
- The default for the *highlight* kwargs is *proxy_driver:highlight_on_assertion_success*.
- The highlight style is configurable with the config *highlight:style_on_assertion_success*.
- The highlight style is configurable with the config *highlight:style_on_assertion_failure*.

1.8 Browsers

1.8.1 Browser config

The brome object need a browser config (yaml). You provide it in the bro executable:

```
brome = Brome(
    config_path = os.path.join(HERE, "config", "brome.yml"),
    selector_dict = selector_dict,
    test_dict = test_dict,
    browsers_config_path = os.path.join(HERE, "config", "browsers_config.yml"), # <-- this file
    absolute_path = HERE
)
```

The browser config look something like this:

```
firefox:
  browserName: 'Firefox'

chrome:
  browserName: 'Chrome'
```

So when you want to run a test using firefox you specify it to the bro executable:

```
$./bro run -l 'firefox'

$./bro run -l 'c'
```

You can add brome config in the browser config also:

```
firefox:
  browserName: 'Firefox'
  "highlight:use_highlight": false
  maximize_window: true
  "runner:embed_on_test_crash": true

chrome:
  browserName: 'Chrome'
  window_height: 950
  window_width: 1550
  "runner:embed_on_test_crash": false
```

You can override a brome config for a specific browser, for example if the config *runner:embed_on_test_crash* is set to True in the brome.yml and you wish to not embed_on_test_crash in chrome then you can set *"runner:embed_on_test_crash"* to false in the browser_config under the chrome section.

1.8.2 Init driver

If you want to change the way the browser is initiliazed then you can do the following:

```
#/path/to/project/model/basetest.py
from selenium import webdriver

from brome.core.model.basetest import BaseTest as BromeBaseTest
from brome.core.model.proxy_driver import ProxyDriver

class BaseTest(BromeBaseTest):

    def init_driver(self, *args, **kwargs):
        #DO WHATEVER YOU WANT
        driver = Firefox()

        #Make sure that you wrap the selenium driver in the ProxyDriver tho
        return ProxyDriver(
            driver = driver,
            test_instance = self,
            runner = self._runner
        )

#/path/to/project/tests/test_scenario.py
#Make sure that your test inherit from your BaseTest
from model.basetest import BaseTest

class Test(BaseTest):
    pass
```

You can look at how the brome basetest implement the init_driver (https://github.com/brome-hq/brome/search?utf8=%E2%9C%93&q=init_driver)

1.8.3 Examples

Localhost

Chrome

```
chrome:
  browserName: 'Chrome'
```

IE

```
ie:
  browserName: 'internet explorer'
```

Firefox

```
firefox:
  browserName: 'Firefox'
```

Safari

```
safari:
  browserName: 'Safari'
```

PhantomJS

```
phantomjs:
  browserName: 'PhantomJS'
```

IOS Simulator

```
iphone:
  appium: true
  deviceName: 'iPhone 5'
  platformName: 'iOS'
  platformVersion: '9.0'
  browserName: 'Safari'
  nativeWebTap: true
  "proxy_element:use_touch_instead_of_click": true
  udid: ''
```

Android

```
android:
  appium: true
  "proxy_element:use_touch_instead_of_click": true
  deviceName: 'Android'
  platformName: 'Android'
  version: '4.2.2'
  browserName: 'chrome'
```

Remote

EC2

```
chrome_ec2:
  amiid: ''
  browserName: 'chrome'
  available_in_webserver: True
  hub_ip: '127.0.0.1'
  platform: 'LINUX'
  launch: True
  ssh_key_path: '/path/to/identity.pem'
```

```
  terminate: True
  nb_browser_by_instance: 1
  max_number_of_instance: 30
  username: 'ubuntu'
  window_height: 950
  window_width: 1550
  region: 'us-east-1'
  security_group_ids: ['sg-xxxxxxx']
  instance_type: 't2.micro'
  selenium_command: "DISPLAY=:0 nohup java -jar selenium-server.jar -role node -hub http://{hub_ip}:4
```

Virtual Box

```
firefox_vbox:
  browserName: 'firefox'
  available_in_webserver: true
  hub_ip: 'localhost'
  password: ''
  platform: 'LINUX'
  launch: true
  terminate: true
  username: ''
  vbname: 'ubuntu-firefox'
  vbox_type: 'gui' #'headless'
  version: '31.0'
```

Sauce Labs

```
chrome_saucelabs:
  saucelabs: True
  platform: "Mac OS X 10.9"
  browserName: "chrome"
  version: "31"
```

Browserstack

```
ie_browserstack:
  browserstack: True
  os: 'Windows'
  os_version: 'xp'
  browser: 'IE'
  browser_version: '7.0'
```

1.9 State

The brome's state system allow you to save a test state in order to speed up and ease your test flow.

You need to set the config *project:url* in order to use the state since the state name is created with the host name and the test name:

```
self.pdriver.get_config_value("project:url")
>>>'http://example.com'

self._name
>>>Test1

self.get_state_pickle_path()
>>>/path/to/project/tests/states/Test1_example.com.pkl
```

This is mainly to support switching the host name in your test. Maybe sometime the code to be tested is on another server, so the state won't exist on this server.

1.9.1 Stateful mixin

The brome's state system will save the following build-in python type:

- str
- unicode
- int
- float
- dict
- list

However not build-in python class won't be saved into the state unless they inherit the Stateful mixin. Here is an example:

```
#/path/to/project/tests/test_1.py
from brome.core.model.stateful import Stateful

class User(Stateful):

    def __init__(self, pdriver, username):
        self.pdriver = pdriver
        self.username = username

class UnStateful(object):
    pass

class Test(BaseTest):

    name = 'State'

    def create_state(self):
        self.unstateful = UnStateful()
        self.stateful = User(self.pdriver, 'test')
        self.int_ = 1
        self.float_ = 0.1
        self.unicode_ = u'test'
        self.str_ = 'str'
        self.list_ = [1,2]
        self.dict_ = {'key' : 1}

    def run(self, **kwargs):

        self.info_log("Running...")
```

```
        #TEST
        assert not hasattr(self, 'unstateful')

        assert hasattr(self, 'stateful')

        assert hasattr(self, 'int_')

        assert hasattr(self, 'float_')

        assert hasattr(self, 'unicode_')

        assert hasattr(self, 'str_')

        assert hasattr(self, 'list_')

        assert hasattr(self, 'dict_')
```

Note: Your stateful class must accept the pdriver in his __init__ function:

```
from brome.core.model.stateful import Stateful

class User(Stateful):

    def __init__(self, pdriver, username):
        self.pdriver = pdriver #<-- this
        self.username = username
```

The state will only be cleaned when it is loaded; so unstateful object will be in the locals() of the test object on the first run but not on the subsequent run. The cleaning function of the state is recursive, so unstateful object found in dict and list will be clearned up also. The stateful cleanup is mainly to satisfy the pickle python module...

1.9.2 Create state

You can either use the automatic state creation (recommended) or create it manually.

Automatic state creation

```
class Test(BaseTest):

    name = 'Test'

    def create_state(self):
        self.dict_ = {'key' : 1}

    def run(self, **kwargs):

        self.info_log("Running...")

        #TEST
        self.dict_['key']
```

Manual state creation

```
#/path/to/project/tests/test_1.py
class Test(BaseTest):

    name = 'Test 1'

    #...

    def run(self, **kwargs):

        #...

        state_loaded = self.load_state()
        if not state_loaded:
            self.string_1 = 'test'

            self.save_state()

        self.info_log(self.string_1)
```

1.9.3 Loading state

If you use the automatic state management them the state will be loaded automatically if one exist. The test logger will tell you if a state was found or not.

If you created the state manually them you also need to load it manually:

```
#/path/to/project/tests/test_1.py
class Test(BaseTest):

    name = 'Test 1'

    #...

    def run(self, **kwargs):

        #...

        state_loaded = self.load_state()

        if state_loaded:
            #Now you have access to the object that were saved in the state
            self.info_log(self.string_1)
```

1.9.4 Deleting state

Deleting a particular state

If you want to delete a particular test's state you can tell the bro executable to delete it before running the test:

```
$ ./bro run -l firefox -s "test_1" --test-config "delete_state=True"
```

or delete it manually:

```
$ rm /path/to/project/tests/states/teststate.pkl
```

Deleting all the states

If you want to delete all the states, the bro executable have a command for that:

```
$ ./bro admin --delete-test-states
```

Or delete them manually:

```
$ rm /path/to/project/tests/states/*.pkl
```

1.10 Network Capture

Brome use the mitm proxy to gather the network data. It support chrome and firefox on localhost.

Below are the step to follow in order to capture the network data.

1.10.1 Install Mitmproxy

Follow the steps found here: https://mitmproxy.org/doc/install.html

1.10.2 Browser config

Add the *enable_proxy* config to your browser config:

```
#/path/to/project/config/browsers_config.yml
firefox:
  browserName: 'Firefox'
  enable_proxy: True
chrome:
  browserName: 'Chrome'
  enable_proxy: True
```

1.10.3 Brome config

Add the following config to your brome yaml:

```
#/path/to/project/config/brome.yml
mitmproxy:
  path: '/path/to/proxy/mitmdump'
  filter: "~m post" #optional this filter will only gather post requests
webserver:
  SHOW_NETWORK_CAPTURE: true
  analyse_network_capture_func: 'model.network_analysis'
```

1.10.4 Network capture data

The network capture data will be stored in the test results folder under the network_capture folder of the specific test batch. For example: */path/to/project/test_results/tb_1/network_data/test.data*

You can analyse the data manually with this code:

```
#!/usr/bin/env python
import pprint
import sys

from libmproxy import flow

with open(sys.argv[1], "rb") as logfile:
    freader = flow.FlowReader(logfile)
    pp = pprint.PrettyPrinter(indent=4)
    try:
        for f in freader.stream():
            print(f)
            print(f.request.host)
            pp.pprint(f.get_state())
            print("")
    except flow.FlowReadError as v:
        print "Flow file corrupted. Stopped loading."
```

1.10.5 Analysis in the webserver

The webserver can automatically analyse the network captured data for you if you provided a method. Here is an example:

```
#/path/to/project/model/network_analysis.py
import json

from libmproxy import flow

def analyse(network_capture_path):
    with open(network_capture_path, "rb") as logfile:
        freader = flow.FlowReader(logfile)

        nb_success = 0
        nb_failure = 0
        try:
            for f in freader.stream():
                try:
                    result = json.loads(f.response.get_decoded_content())
                    if result['success']:
                        nb_success += 1
                    else:
                        nb_failure += 1
                except:
                    pass

            return "<p>Nb success: %s</p><p>Nb failure: %s</p>"%(nb_success, nb_failure)
        except flow.FlowReadError as v:
            return "<p>Flow file corrupted. Stopped loading.</p>"
```

1.11 Session Recording

Brome use the package CastroRedux to record the session. CastroRedux record the session using vnc.

1.11.1 Configuration

Hub configuration

The only thing you need to configure on the hub machine is the vnc password file:

```
$ echo "$VNCPASSWD" >> ~/.vnc/passwd
```

Node configuration

You need to start vnc on your node machine. See this post and this post for some inspiration.

1.11.2 Watch

First you need to enable the config *SHOW_VIDEO_CAPTURE* in your brome.yml:

```
#/path/to/project/config/brome.yml
[...]
webserver:
  SHOW_VIDEO_CAPTURE: true
[...]
```

Launch the webserver (*./bro webserver*), navigate to a specific test batch and click on the *Video recordings* button.

1.12 Resources

1.12.1 Groups

- **Software Quality Assurance & Testing (Stack Exchange):** http://sqa.stackexchange.com/
- **webdriver (reddit):** https://www.reddit.com/r/webdriver
- **selenium (reddit):** https://www.reddit.com/r/selenium
- **Quality Assurance (reddit):** https://www.reddit.com/r/QualityAssurance
- **Selenium-users (google groups):** https://groups.google.com/forum/#!forum/selenium-users
- **Selenium-developers (google groups):** https://groups.google.com/forum/#!forum/selenium-developers
- **webdriver (google groups):** https://groups.google.com/forum/#!forum/webdriver
- **Selenium and Python (LinkedIn):** https://www.linkedin.com/grp/home?gid=2305920&trk=my_groups-tile-grp
- **Selenium Experts (LinkedIn):** https://www.linkedin.com/grp/home?gid=2074523&trk=my_groups-tile-flipgrp
- **Selenium Testing (LinkedIn):** https://www.linkedin.com/grp/home?gid=2102114&trk=my_groups-tile-grp
- **Selenium Grid (LinkedIn):** https://www.linkedin.com/grp/home?gid=4368621&trk=my_groups-tile-grp

- **Selenium 2.0 and WebDriver (LinkedIn):** https://www.linkedin.com/grp/home?gid=3985798&trk=my_groups-tile-flipgrp
- **Selenium WebDriver (LinkedIn):** https://www.linkedin.com/grp/home?gid=4067187&trk=my_groups-tile-flipgrp
- **Selenium WebDriver & Appium (LinkedIn):** https://www.linkedin.com/grp/home?gid=6669152&trk=my_groups-tile-grp
- **Selenium irc:** http://elementalselenium.com/tips/20-irc-chat

1.12.2 Conf / Presentation

- **Python for Blackbox Testers by Sajnikanth Suriyanarayanan | PyCon SG 2013:** https://www.youtube.com/watch?v=2ggWbGLkBPk&feature=youtu.be
- **Selenium Conf:** https://www.youtube.com/user/seleniumconf/videos
- **Agile India:** https://www.youtube.com/user/AgileIndia2012/videos

1.12.3 Tools / Frameworks / Library

- **Awesome Python Test Automation:** https://github.com/atinfo/awesome-test-automation/blob/master/python-test-automation.md
- **AutoItDriverServer:** https://github.com/daluu/AutoItDriverServer
- **RedGlass:** https://github.com/bimech/domreactor-redglass
- **SinonJS:** http://sinonjs.org/
- **Holmium:** https://holmiumcore.readthedocs.org/en/latest/
- **Faker:** https://github.com/joke2k/faker
- **Splinter:** http://splinter.readthedocs.org/en/latest/
- **Saunter:** https://github.com/Element-34/py.saunter/tree/master/saunter
- **Gremlin:** https://github.com/marmelab/gremlins.js

1.12.4 Grid

- **Selenium-Grid-Extras:** https://github.com/groupon/Selenium-Grid-Extras
- **SeLion:** https://github.com/paypal/SeLion

1.12.5 Chrome

- **Chrome Driver:** https://sites.google.com/a/chromium.org/chromedriver/home
- **Web Performance Testing with WebDriver:** https://gist.github.com/klepikov/5457750
- **Chrome capabilities:** https://sites.google.com/a/chromium.org/chromedriver/capabilities

CHAPTER 2

Features

- Simple API
- Focused on test stability and uniformity
- Highly configurable
- Runner for Amazon EC2, Saucelabs, Browserstack, Virtualbox and Appium
- Javascript implementation of select, drag and drop, scroll into view
- IPython embed on assertion for debugging
- Session Video recording
- Network capture with mitmproxy (firefox & chrome)
- Persistent test report
- Webserver
- Test state persistence system
- Support mobile easier (e.g.: click use Touch)