pyselenium-js Documentation

Release 1.3.8

John Nolette

Jun 29, 2018

Contents

CHAPTER 1

Introduction

1.1 About

pyselenium-js is a very simple, lightweight module that helps relieve some of the burden of e2e testing with the official Selenium bindings. The official Selenium bindings operate in the most natural way a user would operate against a given web page. The problem with this, is with more advanced and modern websites, these bindings may not always work as expected on custom elements and components.

An example of this being a div tag taking keyboard input, where div tags do not support the onfocus event listener by design – and the selenium bindings invoke this before attempting to send input to the target DOM.

CHAPTER 2

Installation

The pyselenium-js project depends on both the selenium package. Upon installation of this project, selenium will automatically be installed. Under the hood, pyselenium-js currently relies on selenium version *3.12.0*. pyselenium-js can be used on both python 2.7 and python 3.4+, as well as the required selenium package.

You can download the module using pip like this:

```
pip install pyseleniumjs
```

You may consider using virtualenv to create isolated Python environments.

CHAPTER 3

Getting Started

3.1 Sample Usage

The following code example features a simple implementation using page objects:

```
from pyseleniumjs import E2EJS
from selenium import webdriver

class Page(object):

    def __init__(self, browser):
        self.browser = browser  # specify reference to browser instance
        self.js = E2EJS(browser=browser)  # instantiate instance of js driver

    def exit(self):
        self.browser.quit()


class MyPage(Page):

    @property
    def div_with_text(self):
        return self.browser.find_element_by_css_selector('div.withText')


page = MyPage(browser=webdriver.Firefox())

# selenium bindings cannot pull text from divs
# selenium bindings try to pull the text element property
print page.div_with_text.text
>> ''

# js driver will try to extract the innerText property
print page.js.get_text(element=page.div_with_text)
```

(continues on next page)

(continued from previous page)

```
>> 'foobar'

# alternatively can pull innerHTML
print page.js.get_raw_text(element=page.div_with_text)
>> '<span>foobar</span>'

page.exit()
```

3.2 Example Explained

The pyselenium-js driver can be used in just about any context with a selenium webdriver. The driver only requires a selenium webdriver instance to consume, to execute javascript under the hood. The example above shows how easy, and seamlessly pyselenium-js can be implemented into existing projects and test suites.

CHAPTER 4

Driver

4.1 About

The pyselenium-js driver is **just** a utility that exercises the existing selenium webdriver api *execute_script*. This project was originally created for convenience sake, but has been shaped into a reusable module to help alleviate some of the terrible burdens experienced while relying on the official selenium bindings for more modern websites and web applications.

4.2 Best Practices

pyselenium-js was designed to be a lightweight, and easily accessible all-purpose javascript driver to cover any functionality the official selenium bindings either don't support or don't cover well. To truly exercise the javascript driver, we suggest leveraging page objects and or page factories to simplify integration into your project or test suite.

4.3 Asynchronous Javascript Wait

One of the more unique and powerful features of pyselenium-js, is the ability to create an asynchronous wait using javascript. This wait request is farmed out to your target web browser, and it's condition can be checked by a globally defined handle.

```
# create wait on a 250ms interval for button to be enabled and class to include
↪'danger'
# this example is passing the the element as a WebElement instance
# the element can be referenced in the condition by the alias $el
page.js.wait(
    '$el.disabled == false && \
     $el.getAttribute("class").includes("danger")',
    250, page.button)
>> string
```

(continues on next page)

(continued from previous page)

```
# the wait method also consumes selectors
# if a selector is passed, the browser will query for the element
# this is especially helpful for creating waits for elements that may not yet be␣
↪available
page.js.wait(
    '$el.disabled == false && \
     $el.getAttribute("class").includes("danger")',
    250, 'button[ng-click="login()"]')
>> string

# multiple elements can be used with the javascript wait
# the elements can be referenced in the condition with the alias $el[x]
page.js.wait(
    '$el[0].disabled == false && \
     $el[1].getAttribute("class").includes("active")',
    250, page.button, 'input[ng-bind="username"]')
>> string
```

To check the wait status of your dispatched asynchronous wait, refer to the api method *wait_status*:

```
handle = page.js.wait(
    '$el.disabled == false && \
     $el.getAttribute("class").includes("danger")',
    250, 'button[ng-click="login()"]')
...
page.js.wait_status(handle)
>> True, False
```

4.4 Browser Console Logging

Another very helpful feature of the pyselenium-js driver, is the ability to store and retrieve console browser logs. This functionality is **not** supported by the official selenium bindings.

To enable logging, use the api method *console_logger*:

```
page.js.console_logger()
```

To retrieve your browser's logs, refer to *console_dump*:

```
page.js.console_dump()
>> string
```

The string returned will be in JSON format.

4.5 Checking Element Visibility

The javascript driver allows you to check for the visibility of a WebElement instance like so:

```
page.js.is_visible(page.element)
>> True, False
```

This visibility check consists of an element's offset coordinates, and computed style visibility and opacity.

4.6 Clicking and Double Clicking Elements

The official selenium bindings attempt to click on an element based on it's coordinate position, to emulate a natural click event on a given element. The problem with this, is more modern websites rely on z-index styling rules for pop ups and raised panels; making it impossible to locate the correct coordinates otherwise raising a WebDriverException exception. This behavior has also shown to be especially problematic in nested iframes.

The javascript driver's click method will dispatch a click event directly to the target element. Additionally, the driver provides an api method *dbl_click* to double click on a given element – this feature is **not** supported by the official selenium bindings.

```
page.js.click(page.button)

# double click on an element
page.js.dbl_click(page.button)
```

4.7 Selecting Options From Select Elements

The official selenium bindings provide a very round about method of selecting select element options. This method also does not work for the Safari webdriver.

The pyselenium-js driver offers an api method *select* that will work across any webdriver on any platform without the use of action chains.

```
page.country_selection.click()
page.country_option('United States').click()
# trigger event "select" to notify the browser this element value has been modified
page.js.select(page.country_selection)
```

4.8 Getting and Setting Element Attributes

Using the pyselenium-js driver, an element's attribute can be fetched like so:

```
page.js.get_attribute(page.checkbox, 'aria-toggled')
```

Additionally, an element's attribute can be set using the *set_attribute* api method:

```
page.js.set_attribute(page.checkbox, 'aria-toggled', True)
```

Under the hood, pyselenium-js will automatically convert javascript types into pythonic types and inverse.

4.9 Getting and Setting Element Properties

This feature is not supported by the official selenium bindings (or remote api).

Using the pyselenium-js driver, an element's property can be fetched like so:

```
page.js.get_property(page.checkbox, 'disabled')
```

Additionally, an element's property can be set using the *set_property* api method:

```
page.js.set_property(page.checkbox, 'disabled', True)
```

Under the hood, pyselenium-js will automatically convert javascript types into pythonic types and inverse.

4.10 Getting Element Text

To scrape text from an element, refer to the api method *get_text*:

```
# pulls the innerText property value from a given element
page.js.get_text(page.element)
>> 'foobar'
```

You may alternatively use the api method *get_raw_text* for elements that do not support the *innerText* property.

```
# pulls the innerHTML property value from a given element
page.js.get_raw_text(page.element)
>> '<span>foobar</span>'
```

4.11 Getting Element Value

Input elements provide a property, value, which selenium does **not** provide explicit bindings for. Using the api method *get_value* you may pull the value from any input element (including select, button, radiobutton).

```
page.js.get_value(page.username_field)
>> string
```

4.12 Dispatching Events

The pyselenium-js driver allows developers the ability to dispatch configuragle events to a given element. Refer to the api method *trigger_event*, which can be used like so:

```
# dispatch a naked event 'click'
page.js.trigger_event(page.button, event='click')

# dispatch an event 'click' of type MouseEvent
# pass the event options 'bubbles' and 'cancelable'
page.js.trigger_event(page.button, event='click', event_type='MouseEvent', options={
    'bubbles': True,
    'cancelable': False
})
```

4.13 Scrolling an Element Into View

To scroll an element into view, use the api method *scroll_into_view*:

```
page.js.scroll_into_view(page.button)
```

4.14 Get Page Scrolling Offsets

The driver provides a property *get_scrolling_offsets* to pull the webdriver's current scrolling coordinates. This can be especially helpful when testing fragment identifiers and continuously scrolling content.

```
coords = page.js.get_scrolling_offsets
page.scroll_to_bottom.click()
assert coords['y'] < page.js.get_scrolling_offsets['y']
```

4.15 Angular.js

These methods shouldn't be a go-to for many test cases, but they can certainly helpful for more advanced web applications.

4.15.1 Enable Debugging

To enable angular debugging for access to angular element scropes and controllers, refer to the api method *ng_enable_debugging*. This method *will* reload the driver's current location.

```
page.js.ng_enable_debugging()
```

To verify angular debugging is enabled, a well regarded trick is to search for any existing elements with the class *ng-binding*.

4.15.2 Get and Set Element Text

To pull the inner text of a given angular element, the javascript driver provides an api method *ng_get_text*

```
# angular.element('#someSelector').text()
page.js.ng_get_text(page.username_field)
```

Additionally, the driver provides another api method *ng_set_text* to modify the text of a given angular element.

```
# angular.element('#someSelector').text('john_doe')
page.js.ng_set_text(page.username_field, 'john_doe')
```

4.15.3 Toggle Element Class

Toggling the class of an angular element can be done using the api method *ng_toggle_class*:

```
# angular.element('#someSelector').toggleSelector('active')
page.js.ng_toggle_class(page.button, 'active')
```

4.15.4 Trigger Event Handler

Angular.js provides a relatively simple interface for triggering angular element event handlers. You may trigger an angular.js element event handler like so:

```
# angular.element('#someSelector').triggerHandler('click')
page.js.ng_trigger_event_handler(page.button, 'click')
```

4.15.5 Get and Set Scope Property

The pyselenium-js driver enables angular element scope manipulation, and allows for the extraction of scope property values. Refer to the api methods *ng_get_scope_property* and *ng_set_scope_property*:

```
# angular.element('#someSelector').scope().data.username = 'foobar'
page.js.ng_set_scope_property(page.user_tile, 'data.username', 'foobar')

assert page.js.ng_get_scope_property(
    page.user_tile, 'data.username') == 'foobar'
```

Though this shouldn't be a go-to for many test cases, it's certainly helpful for more advanced web applications.

4.15.6 Call Scope Function

A more advanced feature of the angular.js utilities for pyselenium-js, is the ability to directly invoke scope functions. Take for example the following angular.js controller,

```
angular.controller('homeCtrl', ['$scope', ($scope) => {
    $scope.addUser(username, email, age) {
        ...
    }
}])
```

Using the api method *ng_call_scope_function* you may call the scope method directly like so:

```
# angular.element('#someSelector').scope().addUser('john', 'john@neetgroup.net', 22)
page.js.ng_call_scope_function(
    page.username_field, 'addUser', ['john', 'john@neetgroup.net', 22])
```

4.15.7 Get and Set Controller Property

The pyselenium-js driver enables angular element controller manipulation, and allows for the extraction of controller property values. Refer to the api methods *ng_get_ctrl_property* and *ng_set_ctrl_property*:

```
# angular.element('#someSelector').controller().UserService.username = 'foobar'
page.js.ng_ctrl_scope_property(page.user_tile, 'UserService.username', 'foobar')

assert page.js.ng_ctrl_scope_property(
    page.user_tile, 'UserService.username') == 'foobar'
```

4.15.8 Call Controller Function

A more advanced feature of the angular.js utilities for pyselenium-js, is the ability to directly invoke controller functions. Take for example the following angular.js controller,

```
angular.controller('homeCtrl', () => {
    this.deleteUser(userId) {
        ...
    }
})
```

Using the api method *ng_call_ctrl_function* you may call the controller method directly like so:

```
# angular.element('#someSelector').controller().deleteUser(100100)
page.js.ng_call_ctrl_function(page.username_field, 'deleteUser', [100100])
```

4.16 Angular (2-5)

These methods shouldn't be a go-to for many test cases, but they can certainly helpful for more advanced web applications.

4.16.1 Get and Set Component Property

The pyselenium-js driver provides a simple and easy to use interface for Angular applications to get and set component properties. Refer to the api methods *ng2_get_component_property* and *ng2_set_component_property*.

```
# ng.probe('#someSelector').componentInstance.username = 'jack'
page.js.ng2_set_component_property(page.profile_username, 'username', 'jack')

assert page.js.ng2_get_component_property(
    page.profile_username, 'username') == 'jack'
```

4.16.2 Call Component Function

To invoke an Angular application component function, refer to the *ng2_call_component_function* api method.

```
# ng.probe('#someSelector').componentInstance.logout()
page.js.ng2_call_component_function(page.profile_username, 'logout', [])

# ng.probe('#someSelector').componentInstance.login('username', 'password')
page.js.ng2_call_component_function(page.username_field, 'login', ['username',
↪'password'])
```