py-component-controller Documentation

Release 0.2.6

John Nolette

Jun 29, 2018

Contents

1 Introduction 1
- 1.1 About . . . 1
- 1.2 Why . . . 1

2 Installation 3

3 Getting Started 5
- 3.1 Sample Usage . . . 5
- 3.2 Example Explained . . . 6
- 3.3 Writing Tests . . . 7

4 Components 9
- 4.1 About . . . 9
- 4.2 Best Practices . . . 9
- 4.3 Attributes . . . 9
- 4.4 Descriptions . . . 9
- 4.5 Element (wrapper) . . . 10
- 4.6 Elements (wrapper) . . . 15
- 4.7 Component Groups . . . 18

5 Controller 21
- 5.1 About . . . 21
- 5.2 Best Practices . . . 21
- 5.3 Polyfills . . . 21
- 5.4 Logging . . . 22
- 5.5 Attributes . . . 22
- 5.6 Adding Services . . . 22
- 5.7 Getting Current Location . . . 23
- 5.8 Getting Current Page Title . . . 23
- 5.9 Navigation . . . 23
- 5.10 Checking Location . . . 23
- 5.11 Switching to Window by Title . . . 24
- 5.12 Switching to Window by Location . . . 24
- 5.13 Conditional Waits . . . 25
- 5.14 Take a Screenshot . . . 25
- 5.15 Get Browser Console Logs . . . 25
- 5.16 Terminate Webdriver Session . . . 26

6 Controller Service **27**
6.1 About 27
6.2 Best Practices 27
6.3 Attributes 27
6.4 Example 27

CHAPTER 1

Introduction

1.1 About

What this project strives to do is deter from redundant tasks, and help provide an interface to tackeling larger web applications. This project is a wrapper for the official selenium bindings and pyselenium-js, offering a more object orientated approach. py-component-controller also includes polyfills for conforming webdriver behavior – such as the safari webdriver's handling of multiple element queries.

1.2 Why

The official selenium bindings for Python feel rather dated, and a single interaction such as checking an element's visibility after clicking it can take a myriad of api calls. Additionally, the official Selenium bindings operate in the most natural way a user would operate against a given web page; simple actions such as clicking on an element can be disasterous for modern web applications using a z-index on panels and more commonly modals, because the selenium bindings attempt to get the element's coordinate position before actually executing the click. Using pyselenium-js under the hood, this framework can alleviate the many burdens regularly encountered while using selenium.

CHAPTER 2

Installation

The py-component-controller project depends on both the selenium, and pyselenium-js packages. Upon installation of this project, these two packages will automatically be installed. Under the hood, py-component-controller currently relies on selenium version *3.6.0* and pyselenium-js *1.3.6*. py-component-controller can be used on both python 2.7 and python 3.6, as well as the required packages listed above.

You can download the framework using pip like this:

> pip install pyscc

You may consider using virtualenv to create isolated Python environments.

CHAPTER 3

Getting Started

3.1 Sample Usage

The following code is an example of a controller for Google's search engine:

```
from pyscc import Controller, Component, component_element, \
    component_elements, component_group


class Home(Component):

    _ = 'body > app'  # optional root selector to be applied to all component items

    @component_element
    def search_bar(self):
        # we only need to return the selector
        # pyscc will automatically determine whether it's xpath or css
        return 'input#lst-ib'

    @component_element
    def search_button(self):
        # the element wrapper allows us to format our selectors
        # ref.search_button.fmt(value='Google Search')
        return '//input[@type="submit"][@value="{value}"]'


class Results(Component):

    @component_elements
    def results(self):
        return 'div.g'


class Google(Controller):
```

(continues on next page)

(continued from previous page)

```
    def __init__(self, browser, base_url):
        super(Product, self).__init__(browser, base_url, {
            'home': Home,
            'results': Results
        })

    def search(self, query, redirect=False):
        # navigate to base url
        self.navigate('/')
        # create reference to our home component
        home = self.components.home
        # wait 5 seconds for search bar to be visible
        # write query in the search bar
        home.search_bar\
            .wait_visible(5, error=True)\
            .send_input(query)

        # target the search button with the text "Google Search"
        # click on the search button
        home.search_button\
            .fmt('Google Search')\
            .click()

    def number_of_results():
        return self.components.results.results.count()
```

The sample above can be utilized like so:

```
from selenium import webdriver

google = Google(webdriver.Chrome(), 'https://google.com')
google.search('py-component-controller')
# ensure at least one result is available within 5 seconds
assert google.components.results.results\
    .wait_for(5, length=1)

# terminate our controller's webdriver
google.exit()
```

3.2 Example Explained

As seen in the example above, components can be defined relatively quickly without any hassle. Simply import *Component* from pyscc and define your class. When specifying your component properties, the following decorators can be used to construct Element and Elements wrappers.

- **@component_element**: Expects a single css or xpath selector, will return an *Element* object when referenced.
- **@component_elements**: Expects a single css or xpath selector, will return an *Elements* object when referenced.
- **@component_group**: Expects a dictionary of element name and selector pairs, will return a resource with attributes relevant to your provided pairs returning *Element* objects when referenced.

Using the intended design pattern, Component instances should never be instantiated outside of the scope of the controller. When the controller is intantiated, it will take the provided component name pairs and automatically instantiate them in a *components* attribute.

3.3 Writing Tests

The pyscc framework works very well for creating scrapers and other automation tools, but it was designed with end to end testing in mind. Controllers were also designed to allow developers to easily export their work into client packages for larger suites. The following is an example as to how one may structure tests:

```python
from project import Google
from selenium import webdriver
from unittest import TestCase


class GoogleBaseTest(TestCase):

    def setUp(self):
        self.google = Google(webdriver.Chrome(), 'https://google.com')

    def tearDown(self):
        self.google.exit()


...


class TestGoogleHome(GoogleBaseTest):

    def test_search(self):
        self.google.search('py-component-controller')
        # ensure at least one result is available within 5 seconds
        self.assertNotNone(self.google.components.results.results\
            .wait_for(5, length=1))

    def test_search_autocomplete(self):
        home = self.google.components.home
        home.search_bar\
            .wait_visible(5, error='Google search bar was not visible')\
            .click()\
            .send_input("python")
        # ensure autocomplete popup appears
        self.assertEqual(home.get_attribute('aria-haspopup'))
```

As can be seen in the example above, our product logic is actually loosely coupled with the test. Our controller allows us to define shorthand functionality ie; search, but we can still directly access each individual component and their respective elements. The controller and component have also been designed to work on any webdriver across any platform (excluding mobile forks) using polyfills, so you can write your code once and provision it to run in any environment you please!

CHAPTER 4

Components

4.1 About

A component represents an area in the web application user interface that your test is interacting with. Component objects were created to represent every possible element a controller and/or test would need to reference. Component objects allow us to define an element once, and reference it however many times we need by it's defined property. If any major changes are made to our target interface, we can change the definition of a component's property once and it will work across all of our given controllers and tests. Reference: Page Object

4.2 Best Practices

Following this architectural pattern (controller, component) your component should **only** consist of element properties. Any accessory functionality should be provided by your controller.

If you haven't already check out Getting Started for examples.

4.3 Attributes

When a component is instantiated, the constructor automatically binds the following attributes:

- **browser**: Reference to parent controller's webdriver.
- **env**: Reference to parent controller's env resource.

4.4 Descriptions

Components are constructed with a property __*describe*__ which will return a dictionary of Element, Element, and Component Group instances. This can be helpful for dynamically mapping and operating against arbitrary components.

```
print(controller.components.page.__describe__)
>> {
>>   'element': [],
>>   'elements': [],
>>   'group': [],
>> {
```

4.5 Element (wrapper)

The Element object, is a wrapper for managing individual web elements. This object provides a targeted api for particular elements, rather than having to depend solely on the selenium webdriver. It features many useful short-hand properties and methods, and uses the pyselenium-js driver under the hood for stability.

This entity is not to be confused with the official selenium api's WebElement entity. Given the decorator **@component_element**, whenever referenced the component property will return a *new* instance of the Element wrapper catered to the specific element using the provided selector.

4.5.1 Formatting Selectors

As shown in Getting Started, elements may be defined with template selectors. Take for example the following component element provider:

```
@component_element
def button(self):
    return 'button[ng-click="${method}"]'
```

You can format the element's selector prior to executing any operations.

```
component.button.fmt(method="addUser()")\
    .wait_for(5)\
    .click()
```

4.5.2 Fetching a Selenium WebElement

The element wrapper will still allow you to fetch selenium WebElement objects and access the standard selenium bindings.

```
component.button.get()
>> WebElement
```

4.5.3 Getting Element Text

To scrape text from an element, the element wrapper provides an api method *text*:

```
component.button.text()

# scrape raw text (inner html)
component.button.text(raw=True)
```

4.5.4 Getting Element Value

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

```
component.username_field.value()
```

4.5.5 Getting and Setting Element Attribute

Using the element wrapper, an element's attribute can be fetched like so:

```
component.username_field.get_attribute('aria-toggled')
```

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

```
component.username_field\
    .set_attribute('hidden', False)\
    .wait_visible(3, error=True)
```

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

4.5.6 Getting and Setting Element Property

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

Using the element wrapper, an element's property can be fetched like so:

```
component.remind_me.get_property('checked')
```

Additionally, an element's attribute can be set using the *set_property* api method (chainable):

```
component.username_field\
    .set_property('checked', True)\
    .fmt(class='checked')\
    .wait_for(3, error=True)
```

As explained in the attribute section, pyselenium-js under the hood will automatically convert javascript types into pythonic types and inverse.

4.5.7 Clicking and Double Clicking an Element

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

These two methods are also chainable:

```
component.button\
    .click()\
    .dbl_click()
```

If you require the traditional clicking behavior, simplify fetch a selenium WebElement like so:

```
component.button.get().click()
```

Additionally, for elements that do not listen on the click event but rather mouseup or mousedown, you may refer to the api methods *mouseup* and *mousedown* (chainable):

```
component.button\
    .mouseup()\
    .mousedown()
```

You may also leverage the *select* api method for option child elements of select elements for proper selection.

HTML

```
<select>
    <option value="py">Python</option>
    <option value="js">Javascript</option>
</select>
```

Example

```
component.language_options.python.select()
```

4.5.8 Scrolling To an Element

Scrolling to an element can be done using the *scroll_to* api method (chainable).

```
component.button\
    .scroll_to()\
    .click()
```

4.5.9 Dispatching an Event

Flexing the capabilities of pyselenium-js, we can construct and dispatch events to a given element like so:

```
component.button\
    .trigger_event('click', 'MouseEvent', {'bubbles': True})\
    .wait_invisible(timeout=5, error=True)
```

This method is chainable as the example details.

4.5.10 Sending Input

To send input to an element, refer to the *send_input* api method (chainable):

```
# send_input will always clear the provided field or element before sending input
component.username_field\
    .send_input('py-component-controller')\
    .get_attribute('class')

# you may disable the field or element clearing by using the clear flag
component.username_field\
    .send_input('rocks', clear=False)

# in the event the component element is a custom element that accepts input
# and it does not support the focus event, the selenium bindings will raise a␣
↪WebDriverException
# you may use the force flag to overwrite the innerhtml of the element rather than
# traditionally sending input
component.email_field.send_input('pyscc', force=True)
```

4.5.11 Waiting For an Element

One of the more helpful features of the element wrapper is it's suite of built in waits. To simply wait for an element to be available, you may use the *wait_for* api method (chainable) like so:

```
# wait 5 seconds for element to be available
component.button\
    .wait_for(5)\
    .click()

# alternatively you can also have the wait automatically error out if the condition␣
↪is not met
component.button\
    .wait_for(5, error=True)\
    .click()

# custom error messages can also be specified as the error flag
component.button\
    .wait_for(5, error='Component button was not avaialable as expected')\
    .click()

# when waiting for an element to be unavailable, simply use the available flag
component.button\
    .click()\
    .wait_for(5, available=False, error=True)
```

4.5.12 Waiting For Visibility

If you require the visibility of an element, the element wrapper allows you to wait for the visibility or invisibility of an element with the api methods *wait_visible* and *wait_invisible* (chainable).

```
# wait for an element to become visible
component.button\
    .wait_visible(5, error=True)\
    .click()

# wait for an element to become invisible
component.button\
```

(continues on next page)

(continued from previous page)

```
    .click()\
    .wait_invisible(5)
```

4.5.13 Waiting For an Element To Be Enabled

To wait for an element to be enabled or disabled, refer to the api methods *wait_enabled* and *wait_disabled* (chainable)

```
# wait for an element to become enabled
component.button\
    .wait_enabled(5, error=True)\
    .click()

# wait for an element to become disabled
component.button\
    .click()\
    .wait_disabled(5)
```

4.5.14 Javascript Conditional Wait

Pulling from the pyselenium-js api, you may alternatively asynchronously farm a wait to your target browser. This can also be especially useful when waiting for conditions that occur in timespans < 1 second.

Syntax is as follows:

```
# wait on an interval of every 150 ms for element to include as class 'btn-danger'
component.button.wait_js('$el.getAttribute("class").includes("btn-danger")', 150)
```

The element can be accessed within the condition by the alias $el. To validate the javascript wait status, refer to Checking Wait Status (javascript).

4.5.15 Checking Availability

The element wrapper provides two callable, explicit check for element availablity. Refer to the *available* and *not_available* api methods.

```
component.button.check.available()
>> True, False

component.button.check.not_available()
>> True, False
```

4.5.16 Checking Visibility

The element wrapper provides two callable, explicit check for element visibility. Refer to the *visible* and *invisible* api methods.

```
component.button.check.visible()
>> True, False

component.button.check.invisible()
>> True, False
```

4.5.17 Checking Enabled

Refer to the api methods *enabled* and *disabled* to check whether a DOM node is enabled or disabled.

```
component.button.check.enabled()
>> True, False

component.button.check.disabled()
>> True, False
```

4.5.18 Checking Wait Status (javascript)

The api method *wait_status* can be used to validate the wait status of a previously dispatched wait request on an element instance.

```
button = component.button
# wait on an interval of every 150 ms for element to include as class 'btn-danger'
button.wait_js('$el.getAttribute("class").includes("btn-danger")', 150)
...
button.check.wait_status()
>> True, False
```

4.6 Elements (wrapper)

The Elements object, is a wrapper for managing groups of web elements. This object provides a targeted api for particular elements, rather than having to depend solely on the selenium webdriver. It features many useful short-hand properties and methods, and uses the pyselenium-js driver under the hood for stability.

This entity is not to be confused with the official selenium api's WebElement entity. Given the decorator **@component_elements**, whenever referenced the component property will return a *new* instance of the Elements wrapper catered to the specific elements using the provided selector.

4.6.1 Formatting Selectors

As shown in Getting Started, elements may be defined with template selectors. Take for example the following component elements provider:

```
@component_elements
def users(self):
    return 'a.users.${class}'
```

You can format the element's selector prior to executing any operations.

```
component.users\
    .fmt(class="active")\
    .wait_for(5, length=1)
```

4.6.2 Fetching List of Selenium WebElements

The elements wrapper will still allow you to fetch selenium WebElements and access the standard selenium bindings.

```
component.users.get()
>> [WebElement, ...]
```

```
for user in component.users.get():
    user.click()
```

4.6.3 Counting Existing Matches

The elements wrapper provides an api method *count* to allow you to fetch the number of given elements available.

```
component.users.count()
>> int
```

4.6.4 Getting List of Element Text

To pull the text from every available element into a list, you may use the *text* api method like so:

```
component.users.text()
>> [string, ...]

# scrape raw text (inner html)
component.users.text(raw=True)
>> [string, ...]
```

4.6.5 Getting List of Element Values

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

```
component.users.value()
>> [string, ...]
```

4.6.6 Waiting For Number of Elements

A helpful feature of the elements wrapper, is the ability to wait for a number of elements to become available. This can be done using the *wait_for* api method (chainable).

```
component.users.wait_for(5, length=3)

# alternatively you can also have the wait automatically error out if the condition␣
↪is not met
component.users.wait_for(5, length=3, error=True)

# custom error messages can also be specified as the error flag
component.users.wait_for(5, length=3,
    error="Expected at least 3 users to be available within 5 seconds")

# formatting error messages
component.users.wait_for(5, length=3,
    error="Expected at least ${expected} users to be available found ${found}")
```

(continues on next page)

(continued from previous page)

```
# by default, wait_for will wait for the number of specified elements to be present
# to execute a strict wait on length, you may use the strict flag
component.users.wait_for(5, length=5, strict=True,
    error="Expected 5 users to be available within 5 seconds")
```

4.6.7 Waiting For Visibility of Elements

Another powerful feature of the elements wrapper, is the ability to wait for both a number of elements to be available **and** visible. Refer to the *wait_visible* api method (chainable):

```
component.users.wait_visible(5, length=5, error=True)
```

You may also use the *wait_invisible* api method (chainable) to wait for your target elements to be invisible.

```
component.users.wait_invisible(5, length=5, error=True)
```

4.6.8 Waiting For Elements To Be Enabled

To wait for a collection of elements to be available and enabled, you may leverage the *wait_enabled* api method (chainable):

```
component.users.wait_enabled(5, length=5, error=True)
```

You may also use the *wait_disabled* api method (chainable) to wait for your target elements to be disabled.

```
component.users.wait_disabled(5, length=5)
```

4.6.9 Check Visibility

The api method *visible* is a callable check to ensure the currently available elements are all visible.

```
component.users.check.visible()
>> True, False
```

4.6.10 Check Enabled

Refer to the api methods *enabled* and *disabled* to check whether the currently available elements are enabled or disabled.

```
component.users.check.enabled()
>> True, False

component.users.check.disabled()
>> True, False
```

4.6.11 Getting and Setting Elements' Attribute

Using the elements wrapper, an attribute of a given list of elements can be fetched like so:

```
component.user_list.get_attribute('aria-toggled')
```

Additionally, a list of elements' attribute can be set using the *set_attribute* api method (chainable):

```
component.user_list\
    .set_attribute('hidden', False)\
    .wait_visible(3, error=True)
```

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

4.6.12 Getting and Setting Elements' Property

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

Using the elements wrapper, a property of a given list of elements can be fetched like so:

```
component.user_list.get_property('disabled')
```

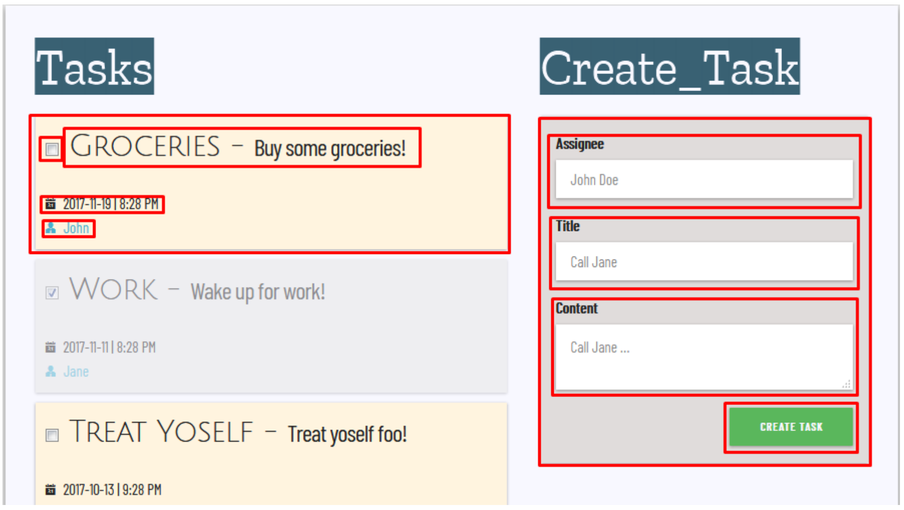
Additionally, a list of elements' properties can be set using the *set_property* api method (chainable):

```
component.user_list\
    .set_property('disabled', True)\
    .fmt(class='disabled')\
    .wait_for(3, error=True)
```

As explained in the attribute section, pyselenium-js under the hood will automatically convert javascript types into pythonic types and inverse.

4.7 Component Groups

Several elements that can be attributed to a particular root element, can be denoted as a component group. Component groups help keep our component objects clean, readable, and maintanable. The following is an example of when you would use a component group,



As can be seen from the image above, a *task* can be interpreted as a component group because each task is composed of several elements that are a child of the task's container. This can be converted into a component group like so:

```
@component_group
def task(self):
    return {
        '_': 'todo-task#${id}',  # optional root selector
        'checkbox': 'input[type="checkbox"]', # becomes: 'todo-task#${id} input[type=
↪"checkbox"]'
        'title': 'span#title',
        'created': 'span#created',
        'assignee': 'a#assignee'
    }

...

task = component.task.fmt(id=1)
task.check.available()
task.checkbox\
    .wait_visible(5, error=True)\
    .click()
```

Component groups also have a very simplistic api to exercise basic checks on their child elements.

4.7.1 Check For Availability of Elements

You may check if a component group's child elements are available with a single call to the *available* api method:

```
component.group.check.available()
>> True, False
```

Alternatively, you may also check if all child elements are unavailable by using *not_available*.

```
component.group.check.not_available()
>> True, False
```

4.7.2 Check For Visibility of Elements

To check the visibility of a component group's child elements, you may refer to the api methods *visible* and *invisible*.

```
component.group.check.visible()
>> True, False

component.group.check.invisible()
>> True, False
```

4.7.3 Checking Elements Enabled

Refer to the api methods *enabled* and *disabled* to check whether a component group's child elements are enabled or disabled.

```
component.group.check.enabled()
>> True, False

component.group.check.disabled()
>> True, False
```

4.7.4 Finding Child Elements

Component group's offer a the *find* api method to dynamically search for child elements by their name.

```
component.social.find('twitter')
>> Element
```

CHAPTER 5

Controller

5.1 About

Controllers were created to utilize our defined components and to farm out tedious rudimentary tasks such as navigating and managing our context. Controllers allow us to define our logic in a behavioral manner, outside of our test cases, keeping everything clean, simple, and manageable. Controllers also allow us to reference a browser once and pass it to all of our defined components. Reference: Page Object Model

There are a vast other quirks to using the controller, component architecture – such as:

- Simple exportable client packages for larger suites (multi-application end to end tests).
- Clean tests; no need for instance-hacks.
- Maintainability, easily documentable.
- Application/website state management.

5.2 Best Practices

Following this architectural pattern (controller, component) your controller **not** contain any accessory methods to search for or provide elements. This class of functionality should be handled exclusively by your defined components.

If you haven't already check out Getting Started for examples.

5.3 Polyfills

Believe it or not, each webdriver for each major browser is maintained by their respective company. Selenium offers an api and a spec that these companies simply adhere to, which is what allows the selenium bindings to conformally speak to any of your desired browsers (for the most part). When a controller instance is instantiated, it will consume the webdriver passed and monkey patch operations and or functions for equality.

The most prevelant polyfill featured is uniformity for *find_elements_by_x*. Every webdriver, but Safari (of course), will return an empty list as where Safari's will raise an *ElementNotFoundException*.

Any irregularities between webdrivers should be reported via the py-component-controller github with steps to reproduce and a related issue or task if available for the corresponding webdriver.

5.4 Logging

The controller supports logging out of the box. Three settings may be toggled to better exercise the logger:

- _FILTER_SELENIUM_LOGS_: Filters selenium logs all together.
- _FILTER_SELENIUM_LOG_STREAM_: Filters selenium logs from stdout stream.
- _LOG_TO_FILE_: Drop logs to hard disk. Will drop individual logs for the controller and selenium.

These settings are all disabled by default, but can be toggled like so:

. . . code-block:: python

from pyscc import Controller

class App(Controller):

    _LOG_TO_FILE_ = True . . .

To filter logs, refer to the api method *add_filter* which can be accessed via the controller's logger instance.

. . . code-block:: python

    # filter messages containing the text 'selenium' ref.controller.logger.add_filter(lambda msg: 'selenium' in msg)

5.5 Attributes

When a controller is instantiated, the constructor automatically binds the following attributes:

- **browser**: Webdriver consumed in the constructor.
- **js**: Reference to instantiated pyselenium-js driver.
- **logger**: Python logger reference.
- **components** Resource for instantiated components constructed using the dictionarty provided in the constructor.
- **env**: Resource for environmental variables consumed in the form of kwargs from the constructor.

5.6 Adding Services

Services are defined in more detail here.

To add a new service to your controller, refer to the api method **add_service**:

```python
from pyscc import Service

class UserService(Service):

    def login(self, username, password):
```

(continues on next page)

(continued from previous page)

```
        # can create methods using the same context as a controller
        login_page = self.components.login
        login_page.username.send_input(username)
        ...

controller.add_service('users', UserService)
controller.services.users.login(...)
```

5.7 Getting Current Location

The controller provides a property *navigation* that can be referenced to fetch your webdriver's current location.

```
controller.location
>> http://github.com
```

5.8 Getting Current Page Title

Refer to the controller's *title* property to pull the current page title from your webdriver.

```
controller.title
>> Github
```

5.9 Navigation

When a controller is instantiated, it will automatically send the webdriver to the base url specified. You may then navigate to other routes off of the base url like so:

```
controller.navigate('/about')
```

For a *hard* navigation, you may use the selenium webdriver api method **get**.

```
controller.browser.get('http://github.com')
```

5.10 Checking Location

To check against your webdriver's current location, you can use the *is_location* method:

```
# check if the route is in your webdrivers location
controller.is_location('/neetjn/py-component-controller')

# strict check on absolute location
controller.is_location('https://github.com/neetjn/py-component-controller',␣
↪strict=True)

# timed location check, will check every second until condition met or timeout␣
↪exceeded
```

(continues on next page)

(continued from previous page)

```
controller.is_location('/neetjn/py-component-controller', timeout=5)

# error if condition is not met
controller.is_location('/neetjn/py-component-controller', timeout=5, error=True)
controller.is_location('/neetjn/py-component-controller', timeout=5,
    error='Expected to be on py-component-controller repository page')

# alternatively format provided error message
controller.is_location('/neetjn/py-component-controller', timeout=5,
    error='Expected to be on ${expected} found ${found}')

# check against a list of possible routes
controller.is_location('/neetjn/pyselenium-js', '/neetjn/py-component-controller')
```

5.11 Switching to Window by Title

For window management, the controller provides a method that allows you to switch to a window by title:

```
# partial window title check
controller.window_by_title('readthedocs')
>> True, False

# strict window title check
controller.window_by_title('readthedocs - My Docs', strict=True)
>> True, False

# polling for window by title
controller.window_by_title('readthedocs', timeout=5)
>> True, False

# error if condition is not met
controller.window_by_title('readthedocs', timeout=5, error=True)
controller.window_by_title('readthedocs', timeout=5,
    error='Could not find the expected readthedocs window')

# alternatively format error message
controller.window_by_title('readthedocs', timeout=5,
    error='Could not find the window by title ${expected} found ${found}')
```

5.12 Switching to Window by Location

The controller also provided a method that allows you to switch to a window by location:

```
# partial window location check
controller.window_by_location('readthedocs.io')
>> True, False

# strict window location check
controller.window_by_location('https://readthedocs.io/neetjn', strict=True)
>> True, False
```

(continues on next page)

(continued from previous page)

```
# poll for window by location
controller.window_by_location('readthedocs.io', timeout=5)
>> True, False

# error if condition is not met
controller.window_by_location('readthedocs.io', timeout=5, error=True)
controller.window_by_location('readthedocs.io', timeout=5,
    error='Could not find the expected readthedocs window')

# alternatively format error message
controller.window_by_location('readthedocs.io', timeout=5,
    error='Could not find the window by location ${expected} found ${found}')
```

5.13 Conditional Waits

Unlike the official selenium bindings, the controller allows an interface for an all-purpose general conditional wait.

```
# wait 5 seconds for element to be visible
# you may pass any callable object as a condition that returns a truthy value
controller.wait(timeout=5, condition=element.check.visible)

# wait for 10 seconds for window to be available with the title "Github"
controller.wait(timeout=10,
    condition=lambda: controller.window_by_title('Github'))
>> True, False

# by design the wait will ignore any exceptions raised while checking the condition
# for debugging purposes, you may toggle the throw_error flag to raise the last error
controller.wait(timeout=5, throw_error=True, condition=lambda: 0/0)

# you may toggle the reverse flag to check for a falsy value
controller.wait(timeout=5, reverse=True, condition=element.check.invisible)
```

5.14 Take a Screenshot

To take a screenshot and drop it to your host machine, use the *screen_shot* method:

```
controller.screen_shot('logout')
```

The screenshot prefix is optional, but this method will automatically generate a unique file name to deter from any io errors and preserve your artifacts.

5.15 Get Browser Console Logs

Using pyselenium-js under the hood we can log our browser's console output. To initialize the logger, you can reference the *console_logger* method from the controller's js attribute (pysjs reference). Once you've initialized the logger, use the controller api method *browser_logs* to drop your logs to your host machine.

```
# initialize logger
controller.js.console_logger()

# dump browser console logs
controller.browser_logs()

# dump browsers logs with a log name
controller.browser_logs('error.logout.redirect')
```

5.16 Terminate Webdriver Session

Equipped with the controller is an all-webdriver termination mechanism. This can be especially helpful for provisioned environments using both local and remote webdrivers.

```
controller.exit()
```

CHAPTER 6

Controller Service

6.1 About

Controller services allow developers to modularize controllers for larger web applications. Services are a lightweight resource that simply consume a given controller, and allow developers to design functionality using the same contextual logic as they would within a controller.

6.2 Best Practices

If a controller is exceeding some 800 lines of code, it may be beneficial for both organization and productivity to compartementalize strings of functionality operating off of the same base component.

6.3 Attributes

When a service is instantiated, the constructor automatically binds the following attributes:

- **controller**: Reference to parent controller instance.
- **browser**: Reference to parent controller's webdriver
- **components**: Reference to parent controller's components.
- **env**: Reference to parent controller's env resource.

6.4 Example

The following example is an abstraction of the *Google* controller featured on Getting Started:

```
from pyscc import Controller, Component, Service, component_element, \
    component_elements, component_group


class Home(Component):

    _ = 'body > app'  # optional root selector to be applied to all component items

    @component_element
    def search_bar(self):
        # we only need to return the selector
        # pyscc will automatically determine whether it's xpath or css
        return 'input#lst-ib'

    @component_element
    def search_button(self):
        # the element wrapper allows us to format our selectors
        # ref.search_button.fmt(value='Google Search')
        return '//input[@type="submit"][@value="{value}"]'


class Results(Component):

    @component_elements
    def results(self):
        return 'div.g'


class SearchService(Service):

    def search(self, query, redirect=False):
        # navigate to base url
        self.controller.navigate('/')
        # create reference to our home component
        home = self.components.home
        # wait 5 seconds for search bar to be visible
        # write query in the search bar
        home.search_bar\
            .wait_visible(5, error=True)\
            .send_input(query)

        # target the search button with the text "Google Search"
        # click on the search button
        home.search_button\
            .fmt('Google Search')\
            .click()


class Google(Controller):

    def __init__(self, browser, base_url):
        super(Product, self).__init__(browser, base_url, {
            'home': Home,
            'results': Results
        })
        self.add_service('search', SearchService)
```

(continues on next page)

(continued from previous page)

```
    def number_of_results():
        return self.components.results.results.count()
```

The sample above can be utilized like so:

```
from selenium import webdriver

google = Google(webdriver.Chrome(), 'https://google.com')
google.services.search.search('py-component-controller')
# ensure at least one result is available within 5 seconds
assert google.components.results.results\
    .wait_for(5, length=1)

# terminate our controller's webdriver
google.exit()
```