TAG ARCHIVES: WEBDRIVERWAIT

WebDriver's implicit wait and deleting elements

Posted on July 12, 2012 | 6 Comments

WebDriver has introduced implicit waits on functions like `find_element`. That means that when WebDriver cannot find an element, it will automatically wait for a defined amount of time for it to appear. 10 seconds is the default duration.

This is very useful behaviour and makes dealing with ambiguous page load events and dynamic elements far more tolerable; where it seems obvious to a human that the test should wait briefly then WebDriver does exactly that. Tests are now more patient and durable throughout tests on dynamic webpages.

However with Selenium RC and Selenium IDE we became quite used to using methods like `is_element_present` and `wait_for_element_present` to deal with Ajax and page loading events. With WebDriver we have to write them ourselves.

Checking an element is not present
When dealing with Ajax or javascript we might want to wait until an element is not present, when an element is being deleted, for example. Trying to find an element after it has been deleted will cause WebDriver to implicitly wait for the element to appear. This is not WebDriver's fault. It's doing the right thing, but we need to tell it not to implicitly wait just for this moment otherwise we will waste time waiting when we don't need to. This time can really add up if you check a few times in each test.

The Web QA team has written its own is_element_present method for WebDriver:

```
def is_element_present(self, *locator):
    self.selenium.implicitly_wait(0)
    try:
        self.selenium.find_element(*locator)
        return True
    except NoSuchElementException:
        return False
    finally:
        # set back to where you once belonged
        self.selenium.implicitly_wait(default_implicit_wait)
```

There are 4 important things going on here. In order:

1. Setting implicity_wait to 0 so that WebDriver does not implicitly wait.
2. Returning True when the element is found.
3. Catching the NoSuchElementException and returning False when we discover that the element is not present instead of stopping the test with an exception.
4. Setting implicitly_wait back to 10 after the action is complete so that WebDriver will implicitly wait in future.

(Note that we have previously stored the default implicit wait value in the default_implicit_wait variable)

You may use this in logic but we mostly use this in WebDriverWait. It is important for bypassing WebDriverWait's catching of the ElementNotFoundException:
`WebDriverWait(self.selenium, 10).until(lambda s: not self.is_element_present((By.ID, 'delete-me')))`

This method works well and most importantly the implicit wait is not triggered meaning your test does not needlessly wait!

Posted in WebDriver → **6 Comments**
Tagged Implicit Wait, NoSuchElementException, Python, Selenium, WebDriver, WebDriverWait

How to WebDriverWait

Posted on July 12, 2012 | 18 Comments

As WebDriver moves more towards being more of an API and less of a testing tool, functions that contained the logic to wait for pages such as `wait_for_page_to_load()` are being removed. The reason for this is that it is difficult to maintain consistent behaviour across all of the browsers that WebDriver supports on modern, dynamic webpages.

That leaves the onus on the people writing the framework and tests (that's you and me!) to write the logic. This is both good and bad. The bad side is that it adds a lot of extra work for us to do and a lot of extra things for us to think about. Your tests might be frail if you don't get your head around how to wait properly. But the good side is that we can control WebDriver and make our tests more stable so let's get on with learning about it!

The first issue to understand is that detecting when a click is just a click and when a click loads a page is difficult for WebDriver. There are just too many things going on on modern webpages with Ajax, Javascript, CSS animations and so forth. So let's forget all about that and just think about what we need on the page to be ready before the test can proceed.

What we are looking for is a good signal. The signal can be an element appearing, disappearing, being created, being deleted or something else altogether! However what is important is that it's relevant to the action you are performing. For example if you are scrolling through pages of search results and waiting for the page of results to change then you should instruct WebDriver to wait for something in the new set of search results. Waiting for something outside of that area can be an unreliable signal.

At this point it's a good start to step through the test manually or if you're debugging, watch the test run on your computer. Watch for elements appearing, javascript, ajax, css animations. Narrow your target on the page down to the area that is changing dynamically or even better the specific element that you want to interact with in the next step of the test. Firebug and Firediff are very useful for this task.

WebDriver's aim is to replicate the user's action and as such if an element is not displayed then you can't click it. This is where a lot of tests come unstuck. By stepping through manually or watching the test run we are looking from the user's perspective. WebDriver can't see elements changing so we need to see them with our own eye before we can tell WebDriver to check on them.

Waiting for element visibility
In WebDriver an element can be present but not visible – be wary of this! If an element is not visible we can't click, type or interact with it so the test is not ready to proceed. It's hard to judge whether you will be checking for element's presence or visibility; every case might be different. But generally when dealing with CSS animation or ajax transitions we will check visibility. In this example we've just clicked on a button that changes the loginbox to be displayed:
```
WebDriverWait(self.selenium, 10).until(lambda s: s.find_element(By.ID, loginbox).is_displayed())
```

Waiting for elements to be deleted
When dealing with elements being deleted from the page we check that there are 0 on the page (WebDriverWait will suppress the ElementNotFoundException). This example is checking that all items in a list have been deleted:
```
WebDriverWait(self.selenium, 10).until(lambda s: len(s.find_elements(By.CSS_SELECTOR, 'list-item')) == 0)
```

You may have noticed in the example of waiting for elements to have been deleted that I used `find_elements` instead of `find_element`. This is because WebDriverWait's `until` is written to wait for elements to *appear* and as such suppresses the ElementNotFoundException.
If you try and use this code WebDriverWait will timeout and finish your test even if the element is not present:
```
WebDriverWait(self.selenium, 10).until(lambda s: not s.find_element(By.ID, 'delete-me'))
```

Waiting for attributes: avoid this!
Waiting for attributes (class, text, etc) of an element can be unreliable as it relies on the element being stable inside WebDriver's element cache. In you-and-me terms that means that waiting for a new node to be present is safer than waiting for an existing one to have changed.
Unreliable:
```
WebDriverWait(self.selenium, 10).until(lambda s: s.find_element(By.ID, 'label').text == "Finished")
```

When performing an action that requires a wait you can always log a value before (for example page number of the search results), perform the action and wait for that value to have changed:
```
page = page_object.page_number
self.selenium.find_element(By.ID, 'next-page').click()
WebDriverWait(self.selenium, 10).until(lambda s: page_object.page_number == page+1)
```

Reporting failures upon timeout
Reporting to the user a clear reason for a timeout failure is very valuable. In cases where the user has no knowledge of the steps of the test or the workflow of the AUT it saves time in having to re-run and debug the test investigating a failure.
As much as we try to make locators and variable names readable, sometimes a complex explicit wait is not clear. Treat it like an inline code comment where you want to communicate to the user, but keep the message brief.
To add a failure message simply add the message to the 'until' method:
```
WebDriverWait(self.selenium, 10).until(lambda s: s.find_elements(By.CSS_SELECTOR, 'list-item') == 0, "The list
items were not deleted before the timeout")
```

Tracking DOM attributes
Occasionally if a javascript package like jQuery is used to manipulate the contents of the page. Can you look at the DOM attributes to see when ajax actions are occurring? Use firebug's DOM panel to inspect values or set a breakpoint and then replicate the action and watch the value change. This is a very stable option because it bypasses WebDriver's element cache. jQuery has an attribute called 'active' that is easily watchable using this code:
```
WebDriverWait(self.selenium, 10).until(lambda s: s.execute_script("return jQuery.active == 0"))
```

Dealing with loading spinners and animations
Catching spinners or loading animations that come and go can be tricky! If you detect the spinner not being present then this might resolve to true before the spinner exists! Occasionally it's more reliable to ignore the spinner altogether and just focus on waiting for an element on the page that the user will be waiting for. If you're really struggling you can use a combination of the spinner and the dynamic element. Here is an example of both catching the spinner being deleted and a new element arriving:

```
WebDriverWait(self.selenium, 10).until(lambda s: s.find_element(By.ID, 'new-element') and
s.find_elements(By.ID, 'spinner') == 0)
```

The order of WebDriverWait's polling
While dealing with Ajax and WebDriverWait it is helpful to know a bit about exactly how the internals of WebDriverWait work. In simplified terms it will check the until equation, sleep, then check the equation again until the timeout is reached. The default setting for polling frequency (that means how much sleep between each the until equation) is 0.5 seconds.
The tricky part, however, is that WebDriverWait will check the until equation before it performs the first sleep. Thus if your Ajax has a slight delay, the very first poll of WebDriverWait might resolve true before the ajax has started. In effect, the the wait will not really have occurred at all because the first sleep was never reached.
There is no workaround for this and the only way to avoid it is to change the way or which element you are waiting for.

The StaleElementReferenceException during Waits
A StaleElementReferenceException may occur if javascript or Ajax is reloading the page during your explicit wait. The exception is thrown because, while WebDriver can find the locator before and after the page reload, it can also see that the element is different and it deems it untrustworthy (or stale). This relates to the previous section about WebDriverWait's polling order.
If the developers are changing the classes of an element before and after then one effective way to wait is to use two locators to locate a single element in each of its states. This is slightly more verbose but the trade-off is a reliable test.

Before login: `(By.CSS_SELECTOR, 'div#user.not_authenticated')`
After login: `(By.CSS_SELECTOR, 'div#user.authenticated')`

```
WebDriverWait(self.selenium, 10).until(lambda s: s.find_element(By.ID,
'div#user.authenticated').is_displayed())
```

In this case even though the HTML

is the same, WebDriver will consider the elements to be different and hence one will only be found after the page refresh and the `authenticated` class is set.

Posted in WebDriver → **18 Comments**
Tagged Ajax, CSS, javascript, Python, Selenium, WebDriver, WebDriverWait