Migrating From Selenium RC to Selenium WebDriver — Selenium Documentation

Migrating From Selenium RC to Selenium WebDriver

How to Migrate to Selenium WebDriver

A common question when adopting Selenium 2 is what's the correct thing to do when adding new tests to an existing set of tests? Users who are new to the framework can begin by using the new WebDriver APIs for writing their tests. But what of users who already have suites of existing tests? This guide is designed to demonstrate how to migrate your existing tests to the new APIs, allowing all new tests to be written using the new features offered by WebDriver.

The method presented here describes a piecemeal migration to the WebDriver APIs without needing to rework everything in one massive push. This means that you can allow more time for migrating your existing tests, which may make it easier for you to decide where to spend your effort.

This guide is written using Java, because this has the best support for making the migration. As we provide better tools for other languages, this guide shall be expanded to include those languages.

Why Migrate to WebDriver

Moving a suite of tests from one API to another API requires an enormous amount of effort. Why would you and your team consider making this move? Here are some reasons why you should consider migrating your Selenium Tests to use WebDriver.

- Smaller, compact API. WebDriver's API is more Object Oriented than the original Selenium RC API. This can make it easier to work with.
- Better emulation of user interactions. Where possible, WebDriver makes use of native events in order to interact with a web page. This more closely mimics the way that your users work with your site and apps. In addition, WebDriver offers the advanced user interactions APIs which allow you to model complex interactions with your site.
- Support by browser vendors. Opera, Mozilla and Google are all active participants in WebDriver's development, and each have engineers working to improve the framework. Often, this means that support for WebDriver is baked into the browser itself: your tests run as fast and as stably as possible.

Before Starting

In order to make the process of migrating as painless as possible, make sure that all your tests run properly with the latest Selenium release. This may sound obvious, but it's best to have it said!

Getting Started

The first step when starting the migration is to change how you obtain your instance of Selenium. When using Selenium RC, this is done like so:

```
Selenium selenium = new DefaultSelenium(
    "localhost", 4444, "*firefox", "http://www.yoursite.com");
selenium.start();
```

This should be replaced like so:

```
WebDriver driver = new FirefoxDriver();
Selenium selenium = new WebDriverBackedSelenium(driver, "http://www.yoursite.com");
```

Once you've done this, run your existing tests. This will give you a fair idea of how much work needs to be done. The Selenium emulation is good, but it's not completely perfect, so it's completely normal for there to be some bumps and hiccups.

Next Steps

Once your tests execute without errors, the next stage is to migrate the actual test code to use the WebDriver APIs. Depending on how well abstracted your code is, this might be a short process or a long one. In either case, the approach is the same and can be summed up simply: modify code to use the new API when you come to edit it.

If you need to extract the underlying WebDriver implementation from the Selenium instance, you can simply cast it to WrapsDriver:

```
WebDriver driver = ((WrapsDriver) selenium).getWrappedDriver();
```

This allows you to continue passing the Selenium instance around as normal, but to unwrap the WebDriver instance as required.

At some point, you're codebase will mostly be using the newer APIs. At this point, you can flip the relationship, using WebDriver throughout and instantiating a Selenium instance on demand:

```
Selenium selenium = new WebDriverBackedSelenium(driver, baseUrl);
```

Common Problems

Fortunately, you're not the first person to go through this migration, so here are some common problems that others have seen, and how to solve them.

Clicking and Typing is More Complete

A common pattern in a Selenium RC test is to see something like:

```
selenium.type("name", "exciting tex");
selenium.keyDown("name", "t");
selenium.keyPress("name", "t");
selenium.keyUp("name", "t");
```

This relies on the fact that “type” simply replaces the content of the identified element without also firing all the events that would normally be fired if a user interacts with the page. The final direct invocations of “key*” cause the JS handlers to fire as expected.

When using the WebDriverBackedSelenium, the result of filling in the form field would be “exciting texttt”: not what you’d expect! The reason for this is that WebDriver more accurately emulates user behavior, and so will have been firing events all along.

This same fact may sometimes cause a page load to fire earlier than it would do in a Selenium 1 test. You can tell that this has happened if a “StaleElementException” is thrown by WebDriver.

WaitForPageToLoad Returns Too Soon

Discovering when a page load is complete is a tricky business. Do we mean “when the load event fires”, “when all AJAX requests are complete”, “when there’s no network traffic”, “when document.readyState has changed” or something else entirely?

WebDriver attempts to simulate the original Selenium behavior, but this doesn’t always work perfectly for various reasons. The most common reason is that it’s hard to tell the difference between a page load not having started yet, and a page load having completed between method calls. This sometimes means that control is returned to your test before the page has finished (or even started!) loading.

The solution to this is to wait on something specific. Commonly, this might be for the element you want to interact with next, or for some Javascript variable to be set to a specific value. An example would be:

```
Wait<WebDriver> wait = new WebDriverWait(driver, 30);
WebElement element= wait.until(visibilityOfElementLocated(By.id("some_id")));
```

Where “visibilityOfElementLocated” is implemented as:

```
public ExpectedCondition<WebElement> visibilityOfElementLocated(final By locator) {
  return new ExpectedCondition<WebElement>() {
    public WebElement apply(WebDriver driver) {
      WebElement toReturn = driver.findElement(locator);
      if (toReturn.isDisplayed()) {
        return toReturn;
      }
      return null;
```

```
    }
  };
}
```

This may look complex, but it's almost all boiler-plate code. The only interesting bit is that the "ExpectedCondition" will be evaluated repeatedly until the "apply" method returns something that is neither "null" nor Boolean.FALSE.

Of course, adding all these "wait" calls may clutter up your code. If that's the case, and your needs are simple, consider using the implicit waits:

```
driver.manage().timeouts().implicitlyWait(30, TimeUnit.SECONDS);
```

By doing this, every time an element is located, if the element is not present, the location is retried until either it is present, or until 30 seconds have passed.

Finding By XPath or CSS Selectors Doesn't Always Work, But It Does In Selenium 1

In Selenium 1, it was common for xpath to use a bundled library rather than the capabilities of the browser itself. WebDriver will always use the native browser methods unless there's no alternative. That means that complex xpath expressions may break on some browsers.

CSS Selectors in Selenium 1 were implemented using the Sizzle library. This implements a superset of the CSS Selector spec, and it's not always clear where you've crossed the line. If you're using the WebDriverBackedSelenium and use a Sizzle locator instead of a CSS Selector for finding elements, a warning will be logged to the console. It's worth taking the time to look for these, particularly if tests are failing because of not being able to find elements.

There is No Browserbot

Selenium RC was based on Selenium Core, and therefore when you executed Javascript, you could access bits of Selenium Core to make things easier. As WebDriver is not based on Selenium Core, this is no longer possible. How can you tell if you're using Selenium Core? Simple! Just look to see if your "getEval" or similar calls are using "selenium" or "browserbot" in the evaluated Javascript.

You might be using the browserbot to obtain a handle to the current window or document of the test. Fortunately, WebDriver always evaluates JS in the context of the current window, so you can use "window" or "document" directly.

Alternatively, you might be using the browserbot to locate elements. In WebDriver, the idiom for doing this is to first locate the element, and then pass that as an argument to the Javascript. Thus:

```
String name = selenium.getEval(
    "selenium.browserbot.findElement('id=foo', browserbot.getCurrentWindow()).tagName");
```

becomes:

```
WebElement element = driver.findElement(By.id("foo"));
String name = (String) ((JavascriptExecutor) driver).executeScript(
    "return arguments[0].tagName", element);
```

Notice how the passed in “element” variable appears as the first item in the JS standard “arguments” array.

Executing Javascript Doesn’t Return Anything

WebDriver’s JavascriptExecutor will wrap all JS and evaluate it as an anonymous expression. This means that you need to use the “return” keyword:

```
String title = selenium.getEval("browserbot.getCurrentWindow().document.title");
```

becomes:

```
((JavascriptExecutor) driver).executeScript("return document.title;");
```