

# Selenium: A Vital Trace Element For Your Tests

Phil Darnowsky

Boston.rb, January 12, 2010

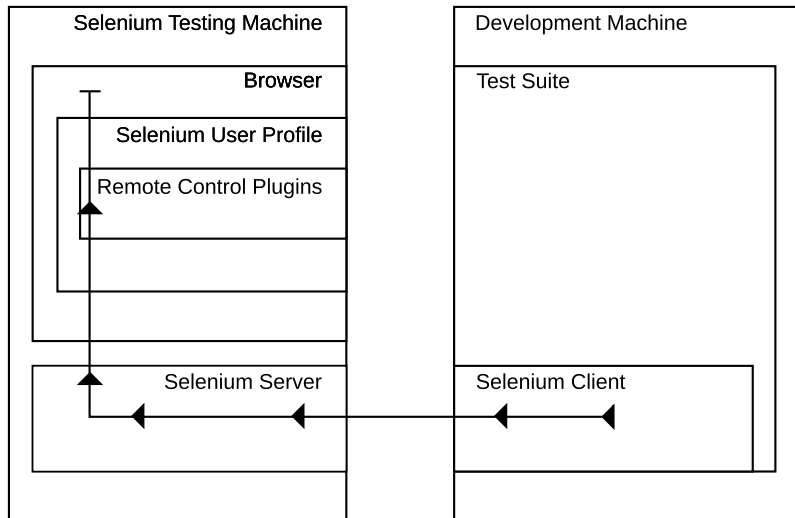
Copyright ©2010 Phil Darnowsky.  
Licensed under the Creative Commons Attribution-Noncommercial-Share Alike 3.0 Unported License.

# What Is Selenium? (1)

Selenium is a system to record, script, and play back browser interactions. There are several related Selenium packages. We'll be looking specifically at Selenium Remote Control.

## What Is Selenium? (2)

Selenium RC creates a browser profile with special plugins that allow a local process to control the browser. It then starts a server that you can connect to remotely to control the browser.



## What Is Selenium? (3)

You can send the browser commands like

- ▶ click this button,
- ▶ type “foo” in that text field,
- ▶ go to `http://www.whatever.com`,

and watch in slack-jawed amazement as the browser acts as though you are actually typing and clicking.

# What Is Selenium? (4)

You can also send queries like

- ▶ What is the text content of this element?
- ▶ Is such-and-such an element visible?
- ▶ What's the value of some arbitrary JavaScript expression?

## At This Point, You Are Saying To Yourself:

“OK, I guess that’s kind of cool. Why bother?”

# Why Indeed

Automating client-side code testing is why.

# When Not To Use Selenium

Don't use Selenium to test functionality that you can test without a real client.

- ▶ Selenium tests take a long time to write and a long time to run.
- ▶ They're sensitive to UI changes, so they can be brittle.

## Setting Up: Server Side

- ▶ Download Selenium RC from <http://seleniumhq.org> and unzip.
- ▶ `java -jar selenium-server.jar  
-firefoxProfileTemplate selenium_test_profile`

## Setting Up: Client Side

- ▶ Install the `selenium-client` gem
- ▶ Create a Rake task to run Selenium tests separately—you don't want them in the regular test suite.
- ▶ If you've got a CI setup, that's also good place to run Selenium tests.

## Writing Tests: Set Up The Driver

You control the Selenium server with an instance of `Selenium::Client::Driver`.

```
def setup
  @selenium = Selenium::Client::Driver.new(
    :browser => '*chrome',

    # host and port on which Selenium RC is running
    :host    => 'seleniumbox.example.com',
    :port    => 4444,

    # URL where application under test is running
    :url     => 'http://webapplication-test.example.com'
  )

  @selenium.start_new_browser_session
end

def teardown
  @selenium.close_new_browser_session
end
```

Self-explanatory—except that “\*chrome” doesn’t mean Google Chrome. It’s the name of a special mode that Selenium runs Firefox in.

# Writing Tests: Exercising The System

This is a typical helper function that goes to the root URL, clicks a login link, enters a username and password into a form, and clicks submit.

```
def login(email, password)
  @selenium.open '/'

  @selenium.click 'link=Log In', :wait_for => :page

  @selenium.type 'email', email
  @selenium.type 'password', password
  @selenium.click 'Log In', :wait_for => :page
end
```

## Writing Tests: Locators

Selenium lets you use many different types of locators to specify elements. They normally take the form “strategy=identifier”  
Some of the most commonly used strategies are:

- ▶ “identifier=foo”: Matches the first element with `id` attribute `foo`. If none, matches the first element with `name` attribute `foo`.
- ▶ “xpath=//foo”: Interprets the identifier as an XPath expression and matches against it.
- ▶ “css=div.foo span#bar”: Interprets the identifier as a CSS selector and matches against it.
- ▶ “link=Listing of foos”: Matches the first link that contains text matching the identifier (e.g. `<a href="/foos">Listing of foos</a>`).

## Writing Tests: Default Locators

You can omit the strategy part of the locator. If the identifier begins with “//”, Selenium will use the `xpath` strategy. Otherwise, Selenium will use the `identifier` strategy.

## Writing Tests: Waiting

click can (and usually should) wait for certain conditions to hold before proceeding. Here's an excerpt of a list from the documentation:

```
# wait for a new page to load
click "a_locator", :wait_for => :page

# wait for all javascript effects to be rendered,
# using semantics of default javascript framework
click "a_locator", :wait_for => :effects

# wait for an element to be present/appear, or to disappear
click "a_locator", :wait_for => :element,
                  :element => "new_element_id"
click "a_locator", :wait_for => :no_element,
                  :element => "new_element_id"

# wait for some text to be present/appear
click "a_locator", :wait_for => :text,
                  :text => "some text" # or /Some Regexp/

# wait for the field value of "a_locator" to be "some value"
click "a_locator", :wait_for => :value,
                  :element => "a_locator",
                  :value => "some value"

# wait for the javascript expression to be true
click "a_locator", :wait_for => :condition,
                  :javascript => "some expression"
```

There are a number of other variations and options not shown here.

## Writing Tests: Verification

The client API has no assertions built in, but there are accessors that are handy to use in conjunction with regular `Test::Unit` or `RSpec` assertions. Here's another sampling:

```
# Assert something present somewhere in the body text of the page...
assert_match "Text In An Unspecified Location", @selenium.body_text

# ...or in a certain element.
assert_match "Text In A Specific Location",
  @selenium.text("css=div#dynamic_status_thingy")

# Assert some cookie set to a certain value, as in an Ajax-based login widget.
assert_equal @expected_session_id, @selenium.cookie("session_id")

# Assert some element is not visible.
assert !(@selenium.visible? "css=#trouble_indicator")

# Assert something about some arbitrary JavaScript expression.
#
# Note that the default context for this is the "selenium" object,
# not "window" as usual.
assert_equal 5, @selenium.js_eval('MyApp.times_certain_thingy_clicked')
```

## Alternatives: Just Skip Testing

I feel weird telling people not to test code. But if your JavaScript is simple, you're not using it for anything critical, or you have a strong graceful degradation strategy, you may be able to justify this.

## Alternatives: relevance-blueridge

blueridge is a bundle that combines several useful testing tools:

- ▶ Rhino: A JS interpreter written in Java
- ▶ Screw.Unit: A BDD library for JS
- ▶ Smoke: A JS mocking/stubbing library
- ▶ env.js: A pure-JS DOM implementation

This package lets you do both command-line and in-browser JavaScript unit tests. I chose Selenium because I was under the mistaken impression that blueridge didn't do in-browser testing.

## Alternatives: Watir

Watir is very similar to Selenium in both intent and approach: launch a specially hacked browser and send it commands. My main reason for choosing Selenium instead is that Watir seemed more Windows-centric.

# Conclusion

Any automated test is a simulation of how the application will behave in production.

The more accurate the test, the higher your confidence in the code can be.

The browser has turned out to be difficult to simulate accurately.

Thus, the approach of turning an enduser application like a browser into an automated client is not as insane as it might seem.

Questions?