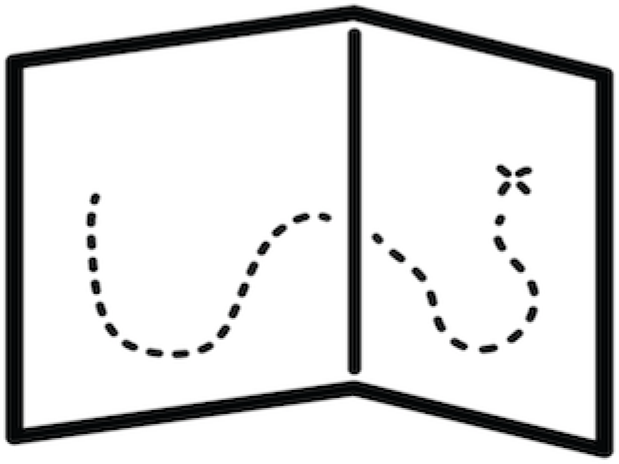
How to use Selenium, successfully



The Selenium Guidebook

by Dave Haeffner

Table of Contents

Anatomy Of A Good Acceptance Test

In order to write effective acceptance tests that are performant, maintainable, and resilient, there are some simple guidelines to follow:

- Write atomic and autonomous tests
- Group like tests together in small batches
- Be descriptive
- Use a Test Runner
- Store it in a Version Control System

Atomic & Autonomous Tests

Each test needs to be concise (e.g. testing a single feature rather than multiple features) and be able to be run independently (e.g. sets up its own data rather than relying on a previous test to do it). Doing this may require a mental shift, discipline, and more up front effort. But it will make a dramatic impact on the quality, effectiveness, and maintainability of your tests.

Grouping Tests

As your test suite grows you should have multiple test files, each containing a small grouping of tests broken out by functionality that they're exercising. This will go a long way towards organization and maintenance as your test suite grows -- as well as faster execution times (depending on your approach to parallelization).

Being Descriptive

Each test file should be named appropriately, and each test within it should have an informative name (even if it may be a bit verbose). Also, each test (or grouping of tests) should be tagged for flexible execution later (e.g. on a Continuous Integration server).

This way you can run parts of your test suite as needed, and the results will be informative thanks to helpful naming.

Test Runners

At the heart of every test suite is some kind of a test runner that does a lot of the heavy lifting (e.g. test group execution, easy global configuration for setup and teardown, reporting, etc.). Rather than reinvent the wheel, you can use one of the many that already exists (there's more than one for every language). And with it you can bolt on third party libraries to extend its functionality if there's something missing -- like parallelization.

The examples in this book use RSpec as the test runner.

Version Control

In order to effectively collaborate with other Testers and Developers, your test code must live in a version control system of some sort (e.g. git, subversion, etc). Look to see what your Development Team uses and get a repository stood up there.

Keep in mind that your test code can live separate from the code of the application you're testing. There are cases and advantages where combining them may be advantageous. But if all your doing is writing and running tests against web endpoints (which is a majority of what your testing will be) then leaving your test code in a separate repository is a solid way to go.

Writing Your First Test

Fundamentally, Selenium works with two pieces of information -- the element on a page you want to use and what you want to do with it. This one-two punch will be repeated over and over until you achieve the outcome you want in your application -- at which point you will perform an assertion to confirm that the result is what you intended.

Let's take logging in to a website as an example. With Selenium you would:

1. Visit the main page of a site
2. Find the login button, and click it
3. Find the login form's userame field and input text
4. Find the login form's password field and input text
5. Find the login form and submit it

Selenium is able to find and interact with elements on a page by way of various locator strategies. The list includes Identifier, Id, Name, Link, DOM, XPath, and CSS.

While each serves a purpose there is one approach that is best of breed. One that is cross-browser performant, simpler to maintain, and leverages code on the page -- and that's CSS Selectors.

A Quick Primer on CSS Selectors

In web design, CSS (Cascading Style Sheets) is used to apply styles to the markup (HTML) on a page. CSS is able to do this by declaring which bits of the markup it wants to interact with through the use of selectors. Selenium operates in a similar manner but instead of changing the style of elements, it interacts with them by clicking, getting values, typing, sending keys, etc.

How To Find CSS Selectors

The simplest way to find CSS Selectors is to inspect the elements on a page. The best way to do this is from within your web browser. Fortunately popular browsers of today come pre-loaded with development tools that make this simple to accomplish.

When viewing the page, right-click on the element you want to interact with and click Inspect Element. This will bring up a small window with all of the HTML for the page but zoomed into your highlighted selection. From here you can see if there are unique attributes you can work with (e.g. `id`, `class`, etc). And at the bottom of the window a set of CSS selectors will be listed showing you the heirarchy of the elements on the page that lead up to your selection. In some cases this info may be useful, in others, not so much.

From here you should able to get started with constructing a CSS Selector to use in your test.

How To Find Quality Elements

Your focus with picking an effective element should be on finding something that is unique and unlikely to change. Ripe candidates for this are `id` and `class` attributes (in CSS Selector parlance `id`'s are prepended with a `#`, and `class`'s are prepended with a `.`). Whereas copy (e.g. the text of a link) is less ideal since it is more apt to change. This may not hold true for when you make assertions, but it's a good goal to strive for.

If the elements you are attempting to work with don't have unique `id` or `class` attributes directly on them, look at the element that houses them (a.k.a. the parent element). Oftentimes the parent element has a unique locator that you can use to start with and drill down into the element you want to use.

And if you can't find any unique elements, have a conversation with your Development team letting them know what you are trying to accomplish. It's not hard for them to add helpful, symantic markup to make test automation easier, especially when they know the use case you are trying to automate. The alternative can be a lengthy, painful process which will probably yield working test code -- but it will be brittle and hard to maintain.

An Example

Let's take our login example from before and put it to test code.

Here's the markup for a standard login form (pulled from the login example on the-internet).

```html
<form name="login" id="login" action="/authenticate" method="post">
   <div class="row">
    <div class="large-6 small-12 columns">
      <label for="username">Username</label>
      <input type="text" name="username" id="username">
    </div>
  </div>
  <div class="row">
    <div class="large-6 small-12 columns">
      <label for="password">Password</label>
      <input type="password" name="password" id="password">
    </div>
  </div>
    <button class="radius" type="submit"><i class="icon-2x icon-signin"> Login
</i></button>
</form>
```

Note the unique elements on the form. The username field has a unique `id`, as does the password field. The submit button doesn't, but the parent element does. Now let's put them to use in our first test (or 'spec' in RSpec parlance).

```ruby
# filename: login_spec.rb

require 'selenium-webdriver'

describe 'Login' do

  before(:each) do
    @driver = Selenium::WebDriver.for :firefox
  end

  after(:each) do
    @driver.quit
  end

  it 'successful' do
    @driver.get 'http://the-internet.herokuapp.com/login'
    @driver.find_element(css: '#username').send_keys('username')
    @driver.find_element(css: '#password').send_keys('password')
    @driver.find_element(css: '#login').submit
  end

end
```

At the top of the file we are pulling in `selenium-webdriver` which is the third-party library (or 'gem' in Ruby parlance) that gives us the bindings necessary to use Selenium to drive the browser.

Next is the `describe`, which is effectively the title of the test. A simple and helpful name is chosen to note the intent of the file (e.g. 'Login').

In order to use Selenium, we need to instantiate it (with Firefox in this case). And when we're done, we don't want the browser we opened to hang around, so we close it. This is what we're doing in the `before(:each)` and `after(:each)` blocks. Note that we are storing our instance of Selenium in an instance variable (`@driver`) so it is available throughout our test.

And lastly is the `it` block, which is the test. And similar to the `describe` block, we can give it a simple and helpful name (e.g. 'successful').

In the test we are interacting with the unique CSS Selectors we identified from the markup by first finding them (e.g. `#username` , `#password` , and `#login`) and then taking an action against them. Which in this case is inputting text (with `send_keys`) and submitting the form (with `submit`).

If we run this (e.g. `rspec login_spec.rb` from the command-line), it will work and pass. But there's one thing missing -- an assertion. But in order to find an element to make an assertion against, we need to see what the markup is after submitting the login form.

```
<div class="row">
  <div id="flash-messages" class="large-12 columns">
    <div data-alert="" id="flash" class="flash success">
      You logged into a secure area!
      <a href="#" class="close">x</a>
    </div>
  </div>
</div>

<div id="content" class="large-12 columns">
  <div class="example">
    <h2><i class="icon-lock"></i> Secure Area</h2>
    <h4 class="subheader">Welcome to the Secure Area. When you are done click logout
below.</h4>
    <a class="button secondary radius" href="/logout"><i class="icon-2x icon-signout">
Logout</i></a>
  </div>
</div>
```

After logging it, there looks to be a couple of things we can key off of for our assertion. There's the flash message that appears at the top of the page (most appealing), the logout button (appealing), or the copy from the h2 (least appealing). Let's use the flash message.

Now, with the flash message, there are two options. We can either leverage the unique class name, or the copy that it displays. Since the class name is descriptive and denotes success (which is the use case we are concerned with) let's go with that.

```ruby
require 'selenium-webdriver'

describe 'Login' do

  before(:each) do
    @driver = Selenium::WebDriver.for :firefox
  end

  after(:each) do
    @driver.quit
  end

  it 'successful' do
    @driver.get 'http://the-internet.herokuapp.com/login'
    @driver.find_element(css: '#username').send_keys('username')
    @driver.find_element(css: '#password').send_keys('password')
    @driver.find_element(css: '#login').submit
    @driver.find_element(css: '.flash.success').displayed?.should be_true
  end

end
```

Just To Make Sure

Now when we run this test it will pass just like before, but now there is an assertion which should catch a failure if something is amiss. Just to make certain that this test is doing what we think it should, let's change the assertion to force a failure. A simple fudging of the CSS Selector will suffice.

```ruby
@driver.find_element(css: '.flash.successasdf').displayed?.should be_true
```

If it fails, then we can feel confident that it's doing what we expect and can change the assertion back to normal. This trick will save you more trouble that you know. Practice it often.