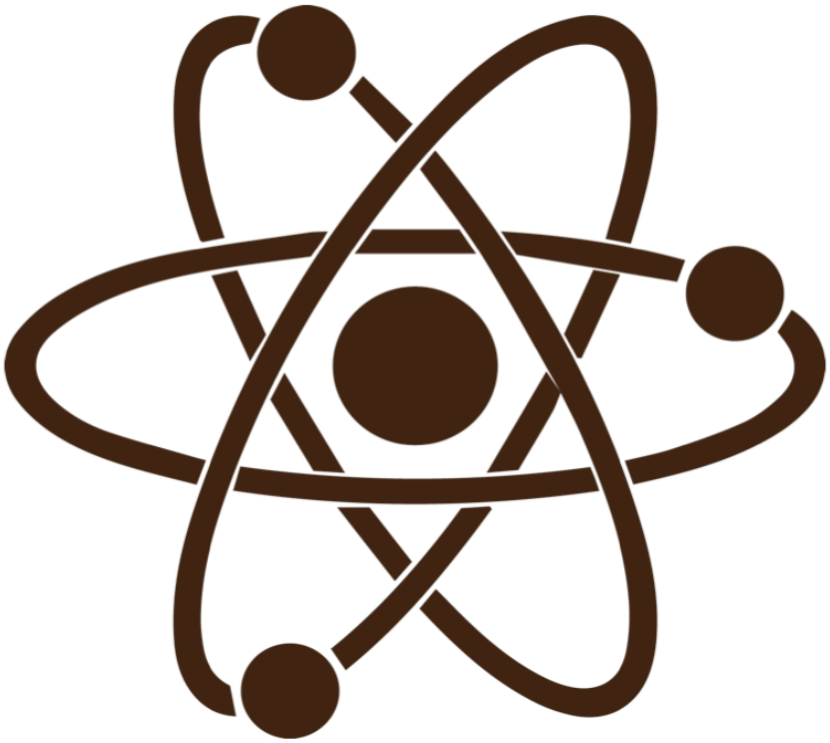
Elemental Selenium



Ruby Tips

By Dave Haeffner

4.0.0

Preface

This book is a compendium of tips from my weekly Selenium tip newsletter (Elemental Selenium). Its aim is to give you a glimpse into the things you'll see in the wild, and a reference for how to deal with them.

The tips differ from The Selenium Guidebook in that they do not build upon previous examples. Instead, they serve as standalone works that can be consumed individually. So feel free to read them in order, or jump around.

Enjoy!

Table of Contents

Chapter 1

How To Upload a File

The Problem

Uploading a file is a common piece of functionality found on the web. But when trying to automate it you get prompted with a with a dialog box that is just out of reach for Selenium.

In these cases people often look to a third-party tool to manipulate this window (e.g., AutoIt). While this can help solve your short-term need, it sets you up for failure later by chaining you to a specific platform (e.g., AutoIt only works on Windows), effectively limiting your ability to test this functionality on different browser & operating system combinations.

A Solution

A work-around for this problem is to side-step the system dialog box entirely. We can do this by using Selenium to insert the full path of the file we want to upload (as text) into the form and then submit the form.

Let's step through an example.

An Example

NOTE: We are using a file upload example found on the-internet.

First let's include our requisite libraries (e.g., `selenium-webdriver` to drive the browser, and `rspec/expectations` & `RSpec::Matchers` for our assertion) and wire-up some simple `setup`, `teardown`, and `run` methods.

```
# filename: upload.rb

require 'selenium-webdriver'
require 'rspec/expectations'
include RSpec::Matchers

def setup
  @driver = Selenium::WebDriver.for :firefox
end

def teardown
  @driver.quit
end

def run
  setup
  yield
  teardown
end
```

Now we're ready to write our file upload test.

```
run do
  filename = 'some-file.txt'
  file = File.join(Dir.pwd, filename)

  @driver.get 'http://the-internet.herokuapp.com/upload'
  @driver.find_element(id: 'file-upload').send_keys file
  @driver.find_element(id: 'file-submit').click

  uploaded_file = @driver.find_element(id: 'uploaded-files').text
  expect(uploaded_file).to eql filename
end
```

After specifying the `filename` we get the full path to the file (which we're doing with `File.join(Dir.pwd, filename)`). This approach assumes that the file is living in the same directory as the test script.

Next we visit the page with the upload form, input the `file` text (e.g., the full path to the file plus the filename with it's extension), and submit the form. After the file is uploaded the page will display the filename that it just processed. We use this text to perform our assertion (making sure the uploaded file is what we expect).

Expected Behavior

When we save this file and run it (e.g., `ruby upload.rb` from the command-line) here is what will happen:

- Open the browser
- Visit the upload example page
- Inject the file path into the form and submit the form
- Grab the filename text that the page displays after processing the upload
- Assert that the filename text matches the filename provided in the test

Outro

This approach will work across all browsers. But if you want to use it with a remote instance (e.g., a Selenium Grid or Sauce Labs), then you'll want to have a look at the `file_detector` driver extension.

Happy Testing!

Chapter 2

How To Download a File

The Problem

Just like with uploading files we hit the same issue with downloading them. A dialog box just out of Selenium's reach.

A Solution

With some additional configuration when loading Selenium we can easily side-step the dialog box. This is done by instructing the browser to download files to a specific location without being prompted.

After the file is downloaded we can then perform some simple checks to make sure the file is what we expect.

Let's dig in with an example.

An Example

Let's start off by pulling in our requisite libraries (e.g., `selenium-webdriver` to drive the browser, `rspec/expectations` and `RSpec::Matchers` for assertions, `uuid` to help create a uniquely named temporary download directory, and `fileutils` to create & destroy the temp directory) and wiring up our `setup` method.

```ruby
# filename: download_file.rb

require 'selenium-webdriver'
require 'rspec/expectations'
include RSpec::Matchers
require 'uuid'
require 'fileutils'

def setup
  @download_dir = File.join(Dir.pwd, UUID.new.generate)
  FileUtils.mkdir_p @download_dir

  profile = Selenium::WebDriver::Firefox::Profile.new
  profile['browser.download.dir'] = @download_dir
  profile['browser.download.folderList'] = 2 # the last folder specified for download
  profile['browser.helperApps.neverAsk.saveToDisk'] = 'image/jpeg, application/pdf,
application/octet-stream'
  profile['pdfjs.disabled'] = true # need to set with PDFs or else it will view them in
the browser
  options = Selenium::WebDriver::Firefox::Options.new
  options.profile = profile
  @driver = Selenium::WebDriver.for(:firefox, options: options)
end
```

Our `setup` method is where the magic is happening in this example. In it we're creating a uniquely named temp directory and storing the absolute path of it in an instance variable that we'll use throughout this file.

We're also configuring a browser profile object (for Firefox in this case) and plying it with the necessary configuration parameters to make it automatically download the file where we want. Here's a breakdown of each of them:

- `browser.download.dir` accepts a string. This is how we set the custom download path. It needs to be an absolute path.
- `browser.download.folderList` takes a number. It tells Firefox which download directory to use. `2` tells it to use a custom download path, wheras `1` would use the browser's default path, and `0` would place them on the Desktop.
- `browser.helperApps.neverAsk.saveToDisk` tells Firefox when not to prompt for a file download. It accepts a string of the file's MIME type. If you want to specify more than one, you do it with a comma-separated string.
- `pdfjs.disabled` is for when downloading PDFs. This overrides the sensible default in Firefox that previews PDFs in the browser. It accepts a boolean.

This profile object is then stored in a options object and passed into our instance of Selenium.

Now let's add some `teardown` and `run` methods.

```
def teardown
  @driver.quit
  FileUtils.rm_rf @download_dir
end

def run
  setup
  yield
  teardown
end
```

In `teardown` we make sure to clean up the temp directory after closing the browser. Other than that, it's business as usual.

Now onto the test itself.

```
run do
  @driver.get 'http://the-internet.herokuapp.com/download'
  download_link = @driver.find_element(css: '.example a')
  download_link.click

  files = Dir.glob("#{@download_dir}/*")
  expect(files.empty?).to eql false
  expect(File.size(files.first)).to be > 0
end
```

After loading the page we find the first download link and click it. The click triggers an automatic download to the temp directory created in `setup`. After that, we perform some rudimentary checks to make sure the directory isn't empty and that the file isn't empty either.

Expected Behavior

When you save this file and run it (e.g., `ruby download_file.rb` from the command-line) here is will happen.

- Create a uniquely named temp directory
- Load the browser
- Visit the page
- Find and click the first download link on the page
- Automatically download the file to the temp directory without prompting
- Check that the temp directory is not empty
- Check that the downloaded file is not empty
- Close the browser

- Delete the temp directory and it's contents

Outro

A similar approach can be applied to some other browsers with varying configurations. But downloading files this way is not sustainable or recommended. Mark Collin articulates this point well in his prominent write-up about it here.

In a future tip I'll cover a more reliable, faster, and scalable browser agnostic approach to downloading files.

Until then, Happy Testing!

Chapter 3

How To Download a File Without a Browser

The Problem

In a previous tip we stepped through how to download files with Selenium by configuring the browser to download them locally and verifying their file size when done.

While this works it requires a custom configuration that is inconsistent from browser to browser.

A Solution

Ultimately we shouldn't care if a file was downloaded or not. Instead, we should care that a file can be downloaded. And we can do that by using an HTTP alongside Selenium in our test.

With an HTTP library we can perform a header (or `HEAD`) request for the file. Instead of downloading the file we'll receive header information for the file which contains information like the content type and content length (amongst other things). With this information we can easily confirm the file is what we expect without onerous configuration, local disk usage, or lengthy download times (depending on the file size).

Let's dig with an example.

An Example

To start things off let's pull in our requisite libraries (e.g., `selenium-webdriver` to drive the browser, `rspec/expectations` and `RSpec::Matchers` for our assertions, and `rest-client` for our HTTP request) and wire up some simple `setup`, `teardown`, and `run` methods.

```ruby
# filename: download_revisited.rb

require 'selenium-webdriver'
require 'rspec/expectations'
require 'rest-client'
include RSpec::Matchers

def setup
  @driver = Selenium::WebDriver.for :firefox
end

def teardown
  @driver.quit
end

def run
  setup
  yield
  teardown
end
```

Now we're ready to wire up our test.

It's just a simple matter of visiting the page with download links, grabbing a URL from one of them, and performing a `HEAD` request with it.

```ruby
run do
  @driver.get 'http://the-internet.herokuapp.com/download'
  link = @driver.find_element(css: '.example a').attribute('href')
  response = RestClient.head link
  expect(response.headers[:content_type]).to eql('application/octet-stream')
  expect(response.headers[:content_length].to_i).to be > 0
end
```

Once we receive the response we can check it's header for the `content_type` and `content_length` to make sure the file is the correct type and not empty.

Expected Behavior

If you save this and run it (e.g., `ruby download_revisited.rb` from the command-line) here is what will will happen:

- Open the browser
- Load the page

- Grab the URL of the first download link
- Perform a `HEAD` request against it with an HTTP library
- Store the response
- Check the response headers to see that the file type is correct
- Check the response headers to see that the file is not empty

Outro

Compared to the browser specific configuration with Selenium this is hands-down a leaner, faster, and more maintainable approach.

Happy Testing!

Chapter 4

How To Take A Screenshot on Failure

The Problem

With browser tests it can often be challenging to track down the issue that caused a failure. By itself a failure message along with a stack trace is hardly enough to go on. Especially when you run the test again and it passes.

A Solution

A simple way to gain insight into your test failures is to capture screenshots at the moment of failure. And it's a quick and easy thing to add to your tests.

Let's dig in with an example.

An Example

Let's start by including our requisite libraries (`selenium-webdriver` to drive the browser and `rspec/expectations` & `RSpec::Matchers` for our assertion) and wire up some simple `setup` and `teardown` methods.

```
# filename: screenshot.rb

require 'selenium-webdriver'
require 'rspec/expectations'
include RSpec::Matchers

def setup
  @driver = Selenium::WebDriver.for :firefox
end

def teardown
  @driver.quit
end
```

Next we'll want to create a method to execute our tests. This is where we'll capture when a failure occurs and take a screenshot.

```
def run
  setup
  begin
    yield
  rescue RSpec::Expectations::ExpectationNotMetError => error
    puts error.message
    @driver.save_screenshot "./#{Time.now.strftime("failshot__%d_%m_%Y__%H_%M_%S")}
.png"
  end
  teardown
end
```

After calling `setup` and before calling `teardown` we wrap our test code execution (e.g., `yield`) in a `rescue` block. This handles the exception that a test failure will return. When a failure occurs the error message will get outputted to the terminal (just like it normally would) but now we are also capturing a screenshot through the help of Selenium's `.save_screenshot` method.

`.save_screenshot` accepts a filename as a string (e.g., `'failshot.png'`). When this command executes it will save an image file to your local system in the current working directory.

Note the use of `Time.now.strftime` in the screenshot filename. This is adding a timestamp (down to the second) to the filename. This provides a (reasonably) unique file name and has the added benefit of listing the files in the order taken.

Now let's wire up our test.

```
run do
  @driver.get 'http://the-internet.herokuapp.com'
  expect(@driver.find_element(css: 'h1').text).to eql 'blah blah blah'
end
```

Expected Behavior

If we save this file and run it (`ruby screenshot.rb` from the command-line) here is what would happen:

- Open the browser
- Load the homepage of the-internet
- Check the text of the page header and fail
- Output a failure message in the terminal
- Capture a timestamped screenshot in the current working directory
- Close the browser

Outro

For more info on `strftime` (a.k.a. String Formatted Time) go here.

But if you want truly unique filenames, then you should use a unique ID in the filename instead of a timestamp (e.g., something like `uuid`). This will prevent screenshots from getting overwritten when you have two (or more) tests taking screenshots at the same time.

Happy Testing!

Chapter 5

How To Use Selenium Grid

The Problem

If you're looking to run your tests on different browser and operating system combinations but you're unable to justify using a third-party solution like Sauce Labs or Browser Stack then what do you do?

A Solution

With Selenium Grid you can stand up a simple infrastructure of various browsers on different operating systems to not only distribute test load, but also give you a diversity of browsers to work with.

A brief Selenium Grid primer

Selenium Grid is part of the Selenium project. It lets you distribute test execution across several machines. You can connect to it with Selenium Remote by specifying the browser, browser version, and operating system you want. You specify these values through Selenium Remote's `Capabilities`.

There are two main elements to Selenium Grid -- a hub, and nodes. First you need to stand up a hub. Then you can connect (or "register") nodes to that hub. Nodes are where your tests will run, and the hub is responsible for making sure your tests end up on the right one (e.g., the machine with the operating system and browser you specified in your test).

Let's step through an example.

An Example

Part 1: Grid Setup

Selenium Grid comes built into the Selenium Standalone Server. So to get started we'll need to download the latest version of it from here.

Then we need to start the hub.

```
> java -jar selenium-server-standalone.jar -role hub
19:05:12.718 INFO - Launching Selenium Grid hub
...
```

After that we can register nodes to it.

```
> java -jar selenium-server-standalone.jar -role node -hub
http://localhost:4444/grid/register
19:05:57.880 INFO - Launching a Selenium Grid node
...
```

NOTE: This example only demonstrates a single node on the same machine as the hub. To span nodes across multiple machines you will need to place the standalone server on each machine and launch it with the same registration command (replacing `http://localhost` with the location of your hub, and specifying additional parameters as needed).

Now that the grid is running we can view the available browsers by visiting our Grid's console at `http://localhost:4444/grid/console`.

To refine the list of available browsers, we can specify an additional `-browser` parameter when registering the node. For instance, if we wanted to only offer Safari on a node, we could specify it with `-browser browserName=safari`, which would look like this:

```
java -jar selenium-server-standalone.jar -role node -browser browserName=safari -hub
http://localhost:4444/grid/register
```

We could also repeat this parameter again if we wanted to explicitly specify more than one browser.

```
java -jar selenium-server-standalone.jar -role node -browser browserName=safari
-browser browserName=chrome -browser browserName=firefox -hub
http://localhost:4444/grid/register
```

There are numerous parameters that we can use at run time. You can see a full list here.

Part 2: Test Setup

Now let's wire up a simple test script to use our new Grid.

First, we'll need to require our necessary libraries (e.g., `selenium-webdriver` to connect to the Grid/control the browser and `rspec/expectations` & `RSpec::Matchers` for an assertion), wire up some simple `setup`, `teardown`, and `run` methods, and add a simple test.

```ruby
# filename: grid.rb

require 'selenium-webdriver'
require 'rspec/expectations'
include RSpec::Matchers

def setup
  @driver = Selenium::WebDriver.for(
    :remote,
    url: 'http://localhost:4444/wd/hub',
    desired_capabilities: :firefox) # you can also use :chrome, :safari, etc.
end

def teardown
  @driver.quit
end

def run
  setup
  yield
  teardown
end

run do
  @driver.get 'http://the-internet.herokuapp.com'
  expect(@driver.title).to eq('The Internet')
end
```

Notice in this configuration we're using Selenium Remote (e.g., `Selenium::WebDriver.for(:remote,`) to connect to the grid. And we are telling the grid which browser we want to use with `desired_capabilities` (e.g., `desired_capabilities: :firefox`).

An alternative way to configure this would be to create a Selenium Remote Capabilities object for the browser we want, modify it as needed, and then pass it to `desired_capabilities`.

```ruby
caps = Selenium::WebDriver::Remote::Capabilities.firefox
caps[:platform] = :mac # you can also use :any, :win, or :x

@driver = Selenium::WebDriver.for(
  :remote,
  url: 'http://localhost:4444/wd/hub',
  desired_capabilities: caps)
```

You can see a full list of the available Selenium `Capabilities` options here.

Expected Behavior

When we save this file and run it (e.g., `ruby grid.rb` from the command-line) here is what will happen:

- connect to the grid
- hub determines which node has the necessary browser/platform combination and connects the test to it
- hub opens an instance of the browser on the node
- test runs
- browser closes on the node

Outro

If you're looking to set up Selenium Grid to work with Internet Explorer or Chrome, be sure to read up on how to set them up since there is additional configuration required for each. And if you run into issues, be sure to check out the browser driver documentation for the browser you're working with:

- ChromeDriver
- EdgeDriver
- geckodriver (Firefox)
- InternetExplorerDriver
- SafariDriver

Also, it's worth noting that while Selenium Grid is a great option for scaling your test infrastructure, it by itself will NOT give you parallelization. That is to say, it can handle as many connections as you throw at it (within reason), but you will still need to find a way to execute your tests in parallel. You can see some previous write-ups on how to accomplish that here.

Happy Testing!

Chapter 6

How To Test Checkboxes

The Problem

Checkboxes are an often used element in web applications. But how do you work with them in your Selenium tests? Intuitively you may reach for a method that has the word 'checked' in it -- like `.checked?` or `.isChecked`. But this doesn't exist in Selenium. So how do you do it?

A Solution

There are two ways to approach this -- by seeing if an element has a `checked` attribute (a.k.a. performing an attribute lookup), or by asking an element if it has been selected.

Let's step through each approach to see their pros and cons.

An Example

For reference, here is the markup from the page we will be testing against (an example from the-internet).

```
<form>
  <input type="checkbox"> unchecked<br>
  <input type="checkbox" checked=""> checked
</form>
```

We kick things off by requiring our dependent libraries (e.g., `selenium-webdriver` to drive the browser, and `rspec/expectations` & `RSpec::Matchers` to handle our assertions) and wire up some simple `setup`, `teardown`, and `run` methods to abstract our test configuration.

```ruby
# filename: checkboxes.rb

require 'selenium-webdriver'
require 'rspec/expectations'
include RSpec::Matchers

def setup
  @driver = Selenium::WebDriver.for :firefox
end

def teardown
  @driver.quit
end

def run
  setup
  yield
  teardown
end
```

Before we write any tests, let's walk through both checkbox approaches to see what Selenium gives us.

To do that we'll want to grab all of the checkboxes on the page, and iterate through them. Once using an attribute lookup, and again asking if the element is selected -- each time outputting the return value we get from Selenium.

```ruby
run do
  @driver.get 'http://the-internet.herokuapp.com/checkboxes'
  checkboxes = @driver.find_elements(css: 'input[type="checkbox"]')

  puts "With .attribute('checked')"
  checkboxes.each { |checkbox| puts checkbox.attribute('checked').inspect }

  puts "\nWith .selected?"
  checkboxes.each { |checkbox| puts checkbox.selected?.inspect }
end
```

When we save our file and run it (e.g., `ruby checkboxes.rb` from the command-line), here is the output we'll see.

```
With .attribute('checked')
nil
"true"

With .selected?
false
true
```

With the attribute lookup, depending on the state of the checkbox, we receive either a `nil` or a string with the value `"true"`. Whereas with `.selected?` we get a boolean (`true` or `false`) response.

Let's see what these approaches look like when put to use in a test.

```
run do
  @driver.get 'http://the-internet.herokuapp.com/checkboxes'
  checkboxes = @driver.find_elements(css: 'input[type="checkbox"]')
  expect(checkboxes.last.attribute('checked')).not_to be_nil
  # alternatively
  expect(checkboxes.last.attribute('checked')).to eql("true")
end
```

With an attribute lookup, the simplest thing to do is to assert that the return value is not `nil`. Alternatively we could have checked for the value `"true"`. Let's see what the other approach looks like.

```
run do
  @driver.get 'http://the-internet.herokuapp.com/checkboxes'
  checkboxes = @driver.find_elements(css: 'input[type="checkbox"]')
  expect(checkboxes.last.selected?).to eql true
end
```

When checking to see if a checkbox has been selected, it's a simple matter of checking for a boolean value.

Expected Behavior

When you save and run the file (e.g., `ruby checkboxes.rb` from the command-line) here is what will happen:

- Open the browser
- Visit the page
- Find all of the checkboxes on the page

- Assert that the last checkbox (the one that is supposed to be checked on initial page load) is checked
- Close the browser

Outro

Attribute lookups are meant for pulling information out of the page for review. While they work in this circumstance you are better off using a selected lookup. But the approach you choose will depend on how the checkbox you're testing is constructed.

Happy Testing!

Chapter 7

How To Test For Disabled Elements

The Problem

On occasion you may have the need to check if an element on a page is disabled or enabled. Sounds simple enough, but how do you do it? It's not a well documented function of Selenium. So doing a trivial action like this can quickly become a pain.

A Solution

If we look at the API documentation for Selenium's Element class we can see there is an available method called `enabled?` that can help us accomplish what we want.

Let's take a look at how to use it.

An Example

For this example we will use a dropdown list from the-internet. In this list there a few options to select, one which should be disabled. Let's find this element and assert that it is disabled.

First let's require our dependent libraries (e.g., `selenium-webdriver` to control the browser and `rspec/expectations` and `RSpec::Matchers` for our assertion) and wire up some simple `setup`, `teardown`, and `run` methods.

```ruby
# filename: dropdown_disabled.rb

require 'selenium-webdriver'
require 'rspec/expectations'

include RSpec::Matchers

def setup
  @driver = Selenium::WebDriver.for :firefox
end

def teardown
  @driver.quit
end

def run
  setup
  yield
  teardown
end
```

Now let's wire up our test.

```ruby
run do
  @driver.get 'http://the-internet.herokuapp.com/dropdown'
  dropdowns = @driver.find_elements(tag_name: 'option')
  item_of_interest = dropdowns.find { |dropdown| dropdown.text == 'Please select an
option' }
  expect(item_of_interest.enabled?).to eql false
end
```

After loading the page, we find all of the elements that have an option tag (which are all of the items in the dropdown list). This will return an array, so we iterate over the collection and find the item we want based on a text comparison.

Once we have the element we want we see if it's enabled (with `.enabled?`) and assert based on the response.

Expected Behavior

If you save this file and run it (e.g., `ruby dropdown_disabled.rb` from the command-line) here is what will happen:

- Open a browser

- Visit the page
- Grab all dropdown list elements and find the one we want by it's text
- Assert that the element is not enabled
- Close the browser

Outro

Hopefully this tip has helped make the simple task of seeing if an element is enabled or disabled more approachable.

Happy Testing!

Chapter 8

How To Select From a Dropdown List

The Problem

Selecting from a dropdown list seems like one of those simple things. Just grab the list by it's element and select an item within it based on the text you want.

While it sounds pretty straightforward, there is a bit more finesse to it.

Let's take a look at a couple of different approaches.

An Example

First let's pull in our requisite libraries and wire up some simple `setup`, `teardown`, and `run` methods.

```
# filename: dropdown.rb

require 'selenium-webdriver'
require 'rspec/expectations'
include RSpec::Matchers

def setup
  @driver = Selenium::WebDriver.for :firefox
end

def teardown
  @driver.quit
end

def run
  setup
  yield
  teardown
end
```

Now lets' wire up our test.

```ruby
# filename: dropdown.rb
...
run do
  @driver.get 'http://the-internet.herokuapp.com/dropdown'

  dropdown_list = @driver.find_element(id: 'dropdown')
  options = dropdown_list.find_elements(tag_name: 'option')
  options.each { |option| option.click if option.text == 'Option 1' }

  selected_option = options.map { |option| option.text if option.selected? }.join
  expect(selected_option).to eql 'Option 1'
end
```

After visiting the example application we find the dropdown list by its ID and store it in a variable. We then find each clickable element in the dropdown list (e.g., each `option`) with `find_elements`.

Grabbing all of the options returns a collection that we iterate over and when the text matches what we want it will click on it.

We finish the test by performing a check to see that our selection was made correctly. This is done by reiterating over the dropdown options collection one more time. This time we're getting the text of the item that was selected, storing it in a variable, and making an assertion against it.

While this works, there is a simpler, built-in way to do this with Selenium. Let's give that a go.

Another Example

```ruby
# filename: dropdown.rb
...
run do
  @driver.get 'http://the-internet.herokuapp.com/dropdown'

  dropdown = @driver.find_element(id: 'dropdown')
  select_list = Selenium::WebDriver::Support::Select.new(dropdown)
  select_list.select_by(:text, 'Option 1')

  selected_option = select_list.selected_options[0].text
  expect(selected_option).to eql 'Option 1'
end
```

Similar to the first example, we are finding the dropdown list by its ID. But instead of iterating over its option elements and clicking based on a conditional check, we are leveraging a built-in helper function of Selenium, `Select`, and it's `select_by` method to choose the item we want.

We then ask the `select_list` what option was selected by using the `selected_options` method. This returns an array of Selenium Elements (which in this case is an array of just one element). So we need to reference the first element by it's index (e.g., `[0]`), ask for it's text, and store it in a variable (e.g., `selected_option`).

Then we perform our assertion against this variable (just like in the previous example).

NOTE: In addition to selecting by text, you can also select by value.

```
select_list.select_by(:value, '1')
```

Expected Behavior

If you save this file with either of these examples and run it (e.g., `ruby dropdown.rb` from the command-line) here is what will happen:

- Open the browser
- Visit the example application
- Find the dropdown list
- Select the requested item from the dropdown list
- Assert that the selected option is the one you expect
- Close the browser

Outro

Hopefully this tip will help you breeze through selecting items from a dropdown list.

Happy Testing!

Chapter 9

How To Work With Hovers

The Problem

If you need to work with mouse hovers in your tests it may not be obvious how to do this with Selenium. And a quick search through the documentation will likely leave you befuddled forcing you to go spelunking through StackOverflow for the solution.

A Solution

By leveraging Selenium's Action Builder we can handle more complex user interactions like hovers. This is done by telling Selenium which element we want to move the mouse to, and then performing what we need to after.

Let's dig in with an example.

An Example

Our example application is available here on the-internet. It has a few avatars displayed in a grid layout. When you hover over each of them, they display additional user information and a link to view a full profile.

Let's write a test that will hover over the first avatar and make sure that this additional information appears.

First, we'll want to include our requisite libraries (e.g., `selenium-webdriver` to control the browser, and `rspec/expectations` and `RSpec::Matchers` for our assertion) and wire up some `setup`, `teardown`, and `run` methods.

```ruby
# filename: hover.rb

require 'selenium-webdriver'
require 'rspec/expectations'
include RSpec::Matchers

def setup
  @driver = Selenium::WebDriver.for :firefox
end

def teardown
  @driver.quit
end

def run
  setup
  yield
  teardown
end
```

Now let's write our test.

```ruby
run do
  @driver.get 'http://the-internet.herokuapp.com/hovers'
  an_avatar = @driver.find_element(class: 'figure')
  @driver.action.move_to(an_avatar).perform
  expect(@driver.find_element(class: 'figcaption').displayed?).to eql true
end
```

After loading the page we find the first avatar and store it in a variable (`an_avatar`). We then use Selenium's `action.move_to` method and feed the avatar variable to it (which triggers the hover).

We then check to see if the additional user information is displayed with `.displayed?` and wrap that in an assertion.

Expected Behavior

If we save this file and run it (e.g., `ruby hover.rb` from the command-line) here is what will happen:

- Open the browser
- Visit the page
- Hover over the first avatar
- Assert that the caption appeared on the page

- Close the browser

Outro

Happy Testing!

Chapter 10

How To Work With JavaScript Alerts

The Problem

If your application triggers any JavaScript pop-ups (a.k.a. alerts, dialogs, etc.) then you need to know how to handle them in your Selenium tests.

A Solution

Built into Selenium is the ability to switch to an alert window and either accept or dismiss it. This way your tests can continue unencumbered by dialog boxes that may feel just out of reach.

Let's dig in with an example.

An Example

Our example application is available here on the-internet. It has various JavaScript Alerts available (e.g., an alert, a confirmation, and a prompt). Let's demonstrate testing a confirmation dialog (e.g., a prompt which asks the user to click `Ok` or `Cancel`).

First, we'll include our requisite libraries (e.g., `selenium-webdriver` to control the browser and `rspec/expectations` and `RSpec::Matchers` for our assertion) and wire up some simple `setup`, `teardown`, and `run` methods.

```ruby
# filename: javascript_alerts.rb

require 'selenium-webdriver'
require 'rspec/expectations'
include RSpec::Matchers

def setup
  @driver = Selenium::WebDriver.for :firefox
end

def teardown
  @driver.quit
end

def run
  setup
  yield
  teardown
end
```

Now let's write our test.

```ruby
run do
  @driver.get 'http://the-internet.herokuapp.com/javascript_alerts'
  @driver.find_elements(css: 'button')[1].click

  popup = @driver.switch_to.alert
  popup.accept

  result = @driver.find_element(id: 'result').text
  expect(result).to eql('You clicked: Ok')
end
```

A quick glance at the page's markup shows that there are no unique IDs on the buttons. So to click on the second button (to trigger the JavaScript confirmation dialog) we find all of the buttons on the page using `find_elements` and click on the second one. Since `find_elements` returns an Array of all found elements, we can assume that the first item can be selected using `[0]` (since Arrays in Ruby start counting at `0`). So the second item would be `[1]`.

After click the button to trigger the JavaScript Alert we use Selenium's `switch_to.alert` method to focus on the JavaScript pop-up and use `.accept` to click `Ok`. If we wanted to click `Cancel` we would have used `.dismiss`.

After accepting the alert, our main browser window will automatically regain focus and the page will display the result that we chose. This text is what we use for our assertion, making sure that

the words `You clicked: Ok` are displayed on the page.

Expected Behavior

If you save this file and run it (e.g., `ruby javascript_alerts.rb` from the command-line) here is what will happen:

- Open the browser
- Load the page
- Click the second button on the page
- JavaScript Confirmation Alert appears
- Accept the JavaScript Confirmation Alert
- Assert that the result on the page is what we expect
- Close the browser

Outro

Happy Testing!

Chapter 11

How To Work With HTML Data Tables

The Problem

Odds are at some point you've come across the use of tables in a web application to display data or information to a user, giving them the option to sort and manipulate it. Depending on your application it can be quite common and something you will want to write an automated test for.

But when the table has no helpful, semantic markup (e.g. easy to use `id` or `class` attributes) it quickly becomes more difficult to work with and write tests against it. And if you're able to pull something together, it will likely not work against older browsers.

A Solution

You can easily traverse a table through the use of CSS Pseudo-classes.

But keep in mind that if you care about older browsers (e.g., Internet Explorer 8, et al), then this approach won't work on them. In those cases your best bet is to find a workable solution for the short term and get a front-end developer to update the table with helpful attributes.

A quick primer on Tables and CSS Pseudo-classes

Understanding the broad strokes of an HTML table's structure goes a long way in writing effective automation against it. So here's a quick primer.

A table has...

- a header (e.g. `<thead>`)
- a body (e.g. `<tbody>`).
- rows (e.g. `<tr>`) -- horizontal slats of data
- columns -- vertical slats of data

Columns are made up of cells which are...

- a header (e.g., `<th>`)
- one or more standard cells (e.g., `<td>` -- which is short for table data)

CSS Pseudo-classes work by walking through the structure of an object and targeting a specific part of it based on a relative number (e.g. the third `<td>` cell from a row in the table body). This

works well with tables since we can grab all instances of a target (e.g. the third `<td>` cell from each `<tr>` in the table body) and use it in our test -- which would give us all of the data for the third column.

Let's step through some examples for a common set of table functionality like sorting columns in ascending and descending order.

An Example

NOTE: You can see the application under test here. It's an example from the-internet. In the example there are 2 tables. We will start with the first table and then work with the second.

We kick things off by requiring our dependent libraries (`selenium-webriver` to drive the browser and `rspec/expectations` and its matchers to handle our assertions) and wiring up some `setup`, `teardown`, and `run` methods to handle our test configuration.

```
# filename: table_sort.rb

require 'selenium-webdriver'
require 'rspec/expectations'
include RSpec::Matchers

def setup
  @driver = Selenium::WebDriver.for :firefox
end

def teardown
  @driver.quit
end

def run
  setup
  yield
  teardown
end
```

Here is the markup from the first table example we're working with. Note that it does not have any `id` or `class` attributes.

```
<table id="table1" class="tablesorter">
    <thead>
      <tr>
        <th><span>Last Name</span></th>
        <th><span>First Name</span></th>
        <th><span>Email</span></th>
        <th><span>Due</span></th>
        <th><span>Web Site</span></th>
        <th><span>Action</span></th>
      </tr>
    </thead>
    <tbody>
      <tr>
        <td>Smith</td>
        <td>John</td>
        <td>jsmith@gmail.com</td>
        <td>$50.00</td>
        <td>http://www.jsmith.com</td>
        <td>
          <a href='#edit'>edit</a>
          <a href='#delete'>delete</a>
        </td>
      </tr>
```

There are 6 columns (`Last Name` , `First Name` , `Email` , `Due` , `Web Site` , and `Action`). Each one is sortable by clicking on the column header. The first click will sort them in ascending order, the second click in descending order.

There is a small sampling of data in the table to work with (4 rows worth), so we should be able to sort the data, grab it, and confirm that it sorted correctly. So let's do that in our first test with the `Due` column using a CSS Pseudo Class.

```
# ...

run do
  @driver.get 'http://the-internet.herokuapp.com/tables'

  @driver.find_element(css: '#table1 thead tr th:nth-of-type(4)').click

  dues = @driver.find_elements(css: '#table1 tbody tr td:nth-of-type(4)')
  due_values = dues.map { |due| due.text.delete('$').to_f }

  expect(due_values).to eql due_values.sort
end
```

After loading the page we find and click the column heading that we want with a CSS Pseudo-class (e.g. `#table1 thead tr th:nth-of-type(4)`). This locator targets the 4th `<th>` element in the table heading section (e.g., `<thead>`) (which is the `Due` column heading).

We then use another pseudo-class to find all `<td>` elements within the `Due` column by looking for the 4th `<td>` of each row in the table body. Once we have them we grab each of their text values, clean them up (`.delete($)`), convert them to a number (`.to_f`), and store them all in a collection called `due_values`. We then compare this collection to a sorted version of itself to see if they match. If they do, then the `Due` column was sorted in ascending order and the test will pass.

If we wanted to test for descending order, we would need to click the `Due` heading twice after loading the page. Other than that the code is identical except for the assertion which is checking the same thing but reversing the sort order.

```
# ...

run do
  @driver.get 'http://the-internet.herokuapp.com/tables'

  @driver.find_element(css: '#table1 thead tr th:nth-of-type(4)').click
  @driver.find_element(css: '#table1 thead tr th:nth-of-type(4)').click

  dues = @driver.find_elements(css: '#table1 tbody tr td:nth-of-type(4)')
  due_values = dues.map { |due| due.text.delete('$').to_f }

  expect(due_values).to eql due_values.sort.reverse
end
```

We can easily use this approach to test a different column (e.g., one that doesn't deal with numbers) and see that it gets sorted correctly too. Here's a test that exercises the `Email` column.

```
# ...

run do
  @driver.get 'http://the-internet.herokuapp.com/tables'

  @driver.find_element(css: '#table1 thead tr th:nth-of-type(3)').click

  emails = @driver.find_elements(css: '#table1 tbody tr td:nth-of-type(3)')
  email_values = emails.map { |email| email.text }

  expect(email_values).to eql email_values.sort
end
```

The mechanism for this is the same except that we don't need to clean the text up or convert it

before performing our assertion.

But What About Older Browsers?

Now we have some working tests that will load the page and check sorting for a couple of columns in both ascending and descending order. Great! But if we run these again an older browser (e.g., Internet Explorer 8, etc.) it will throw an exception stating `Unable to find element`. This is because older browsers don't support CSS Pseudo-classes.

You've come a long way, so it's best to get value out of what you've just written. To do that you can run these tests on current browsers and submit a request to your front-end developers to update the table markup with some semantic `class` attributes. Later, when these new locators have been implemented on the page, you can revisit these tests and update them accordingly.

Here is markup of what our original table would look like with some helpful attributes added in. It's also the markup from the second example of our application under test.

```
<table id="table2" class="tablesorter">
    <thead>
      <tr>
        <th><span class='last-name'>Last Name</span></th>
        <th><span class='first-name'>First Name</span></th>
        <th><span class='email'>Email</span></th>
        <th><span class='dues'>Due</span></th>
        <th><span class='web-site'>Web Site</span></th>
        <th><span class='action'>Action</span></th>
      </tr>
    </thead>
    <tbody>
      <tr>
        <td class='last-name'>Smith</td>
        <td class='first-name'>John</td>
        <td class='email'>jsmith@gmail.com</td>
        <td class='dues'>$50.00</td>
        <td class='web-site'>http://www.jsmith.com</td>
        <td class='action'>
          <a href='#edit'>edit</a>
          <a href='#delete'>delete</a>
        </td>
      </tr>
```

With these selectors our sorting tests become a lot simpler and more expressive. Let's take our previous `Due` ascending test and update it to demonstrate.

```ruby
# ...
run do
  @driver.get 'http://the-internet.herokuapp.com/tables'

  @driver.find_element(css: '#table2 thead .dues').click

  dues = @driver.find_elements(css: '#table2 tbody .dues')
  due_values = dues.map { |due| due.text.delete('$').to_f }

  expect(due_values).to eql due_values.sort
end
```

Not only will these selectors work in current and older browsers, but they are also more resilient to changes in the table layout since they are not using hard-coded numbers that rely on the column order.

Expected Behavior

If you save this file and run it (e.g., `ruby table_sort.rb` from the command-line) here is what will happen:

- Open the browser
- Load the page
- Click the column heading
- Grab the values for that column
- Assert that the columns are sorted in the correct order (ascending or descending)
- Close the browser

Outro

CSS Pseudo-classes are a great resource and unlock a lot of potential for your tests; enabling a bit of CSS gymnastics assuming you've come up with a test strategy that rules out older browsers. If you don't have a test strategy or are curious to see how yours compares, check out tip 18.

For more info on CSS Pseudo-classes see this write-up by Sauce Labs, and maybe the W3C spec CSS3 if you're feeling adventurous. And for a more in-depth walk-through on HTML Table design check out Treehouse's write-up here.

Happy Testing!

Chapter 12

How To Work with Frames

The Problem

On occasion you'll run into a relic of the front-end world -- frames. And when writing a test against them, you can easily get tripped if you're not paying attention.

A Solution

Rather than gnash your teeth when authoring your tests, you can easily work with the elements in a frame by telling Selenium to switch to that frame first. Then the rest of your test should be business as usual.

Let's dig in with some examples.

An Example

We'll first need to pull in our requisite libraries (`selenium-webdriver` to drive the browser, and `rspec/expecations` & `RSpec::Matchers` to perform our assertions) and wire up some simple `setup`, `teardown`, and `run` methods.

```
# filename: frames.rb

require 'selenium-webdriver'
require 'rspec/expectations'
include RSpec::Matchers

def setup
  @driver = Selenium::WebDriver.for :firefox
end

def teardown
  @driver.quit
end

def run
  setup
  yield
  teardown
end
```

Now onto our test. In it we'll step through an example of nested frames which can be found on the-internet.

```
run do
  @driver.get 'http://the-internet.herokuapp.com/nested_frames'
  frame_top = @driver.find_element(name: 'frame-top')
  @driver.switch_to.frame(frame_top)
  frame_middle = @driver.find_element(name: 'frame-middle')
  @driver.switch_to.frame(frame_middle)
  expect(@driver.find_element(id: 'content').text).to eql 'MIDDLE'
end
```

With Selenium's `.switch_to.frame` method we can easily switch to the frame we want. It accepts a found element. But in order to get the text of the middle frame (e.g., a frame nested within another frame), we need to switch to the parent frame (e.g., the top frame) and then switch to the child frame (e.g., the middle frame).

Once we've done that we're able to find the element we need, grab it's text, and assert that it's what we expect.

While this example helps illustrate the point of frame switching, it's not very practical.

A More Practical Example

Here is a more likely example you'll run into -- working with a WYSIWYG Editor like TinyMCE. You can see the page we're testing here.

```
# filename: frames.rb

run do
  @driver.get 'http://the-internet.herokuapp.com/tinymce'
  @driver.switch_to.frame(@driver.find_element(id: 'mce_0_ifr'))
  editor = @driver.find_element(id: 'tinymce')
  before_text = editor.text
  editor.clear
  editor.send_keys 'Hello World!'
  after_text = editor.text
  expect(after_text).not_to eql before_text
end
```

Once the page loads we switch into the frame that contains TinyMCE and...

- grab the original text and store it
- clear and input new text
- grab the new text value

- assert that the original and new texts are not the same

Keep in mind that if we need to access a part of the page outside of the frame we are currently in we'll need to switch to it. Thankfully Selenium has method that enables us to quickly jump back to the top level of the page -- `switch_to.default_content`.

Here is what that looks like in practice.

```
@driver.switch_to.default_content
  expect(@driver.find_element(css: 'h3').text).not_to be_empty
```

Expected Behavior

If we save the file and run it (e.g., `ruby frames.rb` from the command-line) here is what will happen:

Example 1

- Open the browser
- Visit the page
- Switch to the nested frame
- Grab the text from the frame and assert that Selenium is in the correct place
- Close the browser

Example 2

- Open the browser
- Visit the page
- Switch to the frame that contains the TinyMCE editor
- Grab and clear the text in the editor
- Input and grab new text in the edtitor
- Assert that the original and new text entries don't match
- Switch to the top level of the page
- Grab the text from the top of the page and assert that it's not empty
- Close the browser

Outro

Now you're ready to handily defeat frames when they cross your path.

Happy Testing!

Chapter 13

How To Work with Multiple Windows

The Problem

Occasionally you'll run into a link or action in the application you're testing that will open a new window. In order to work with both the new and originating windows you'll need to switch between them.

On the face of it, this is a pretty straightforward concept. But lurking within it is a small gotcha to watch out for that will bite you in some browsers and not others.

Let's step through a couple of examples to demonstrate.

An Example

First, let's pull in our requisite libraries (e.g., `selenium-webdriver` to control the browser and `rspec/expectations` & `RSpec::Matchers` for our assertions) and add some simple `setup`, `teardown`, and `run` methods.

```
# filename: new_window.rb

require 'selenium-webdriver'
require 'rspec/expectations'
include RSpec::Matchers

def setup
  @driver = Selenium::WebDriver.for :firefox
end

def teardown
  @driver.quit
end

def run
  setup
  yield
  teardown
end
```

Now let's write a test that exercises new window functionality from an application. In this case, we'll be using the new window example found on the-internet.

```ruby
# filename: new_window.rb

# ...

run do
  @driver.get 'http://the-internet.herokuapp.com/windows'
  @driver.find_element(css: '.example a').click
  @driver.switch_to.window(@driver.window_handles.first)
  expect(@driver.title).not_to eql 'New Window'
  @driver.switch_to.window(@driver.window_handles.last)
  expect(@driver.title).to eql 'New Window'
end
```

After loading the page we click the link which spawns a new window. We then grab the window handles (a.k.a. unique identifier strings which represent each open browser window) and switch between them based on their order (assuming that the first one is the originating window, and that the last one is the new window). We round this test out by performing a simple check against the title of the page to make sure Selenium is focused on the correct window.

While this may seem like a good approach, it can present problems later. That's because the order of the window handles is not consistent across all browsers. Some return in the order opened, others alphabetically.

Here's a more resilient approach. One that will work across all browsers.

A Better Example

```ruby
# filename: new_window.rb

run do
  @driver.get 'http://the-internet.herokuapp.com/windows'

  first_window = @driver.window_handle
  @driver.find_element(css: '.example a').click
  sleep 2 # to account for new window loading
  all_windows = @driver.window_handles
  new_window = all_windows.find { |this_window| this_window != first_window }

  @driver.switch_to.window(first_window)
  expect(@driver.title).not_to eql 'New Window'

  @driver.switch_to.window(new_window)
  expect(@driver.title).to eql 'New Window'
end
```

After loading the page we store the window handle in a variable (e.g., `first_window`) and then proceed with clicking the new window link.

Now that we have two windows open we grab all of the window handles and search through them to find the new window handle (e.g., the handle that doesn't match the first one we've already stored). We store the result in another variable (e.g., `second_window`) and then switch between the windows. Each time checking the page title to make sure the correct window is in focus.

Expected Behavior

- Open the browser
- Visit the page
- Find the window handle for the current window
- Click a link that opens a new window
- Find the window handle out of all available window handles
- Switch between the windows
- Assert that the correct window is in focus
- Close the browser

Outro

Hat tip to Jim Evans for providing the info for this tip.

Happy Testing!

Chapter 14

How To Press Keyboard Keys

The Problem

On occasion you'll come across functionality that requires the use of keyboard key presses in your tests.

Perhaps you'll need to tab to traverse from one portion of the page to another, back out of some kind of menu or overlay with the escape key, or even submit a form with Enter.

But how do you do it and where do you start?

A Solution

You can easily issue a key press by using the `send_keys` command.

This can be done to a specific element, or generically with Selenium's Action Builder (which has been documented on the Selenium project's Wiki page for Advanced User Interactions). Either approach will send a key press. The latter will send it to the element that's currently in focus in the browser (so you don't have to specify a locator along with it), whereas the prior approach will send the key press directly to the element found.

When sending keys be sure to specify the key-press as a symbol (e.g. space is `:space`, tab is `:tab`, etc.). You can see a full list of keyboard key symbols here.

Let's step through a couple of examples.

An Example

First we'll set up our requisite libraries to drive the browser (e.g., `selenium-webdriver`) and perform an assertion (e.g., `rspec/expectations` and `RSpec::Matchers`). After that, we'll create some simple `setup`, `teardown`, and `run` methods.

```ruby
# filename: key_presses.rb

require 'selenium-webdriver'
require 'rspec/expectations'
include RSpec::Matchers

def setup
  @driver = Selenium::WebDriver.for :firefox
end

def teardown
  @driver.quit
end

def run
  setup
  yield
  teardown
end
```

Now we can wire up our test.

Let's use an example from the-internet that will display what key has been pressed (link). We'll use the result text that gets displayed to perform our assertion.

```ruby
run do
  @driver.get 'http://the-internet.herokuapp.com/key_presses'
  @driver.find_element(id: 'target').send_keys :space
  expect(@driver.find_element(id: 'result').text).to eql('You entered: SPACE')
end
```

After visiting the page we find an element that's visible (e.g., the one that contains the example on the page) and send the space key to it (e.g., `.send_keys :space`). Then we grab the resulting text (e.g., `@driver.find_element(id: 'result').text`) and assert that it says what we expect (e.g., `'You entered: SPACE'`).

Alternatively, we can also issue a key press without finding the element first.

```ruby
run do
  # ...
  @driver.action.send_keys(:tab).perform
  expect(@driver.find_element(id: 'result').text).to eql('You entered: TAB')
end
```

Expected Behavior

If we save this and run it (e.g. `ruby key_presses.rb` from the command-line) here is what will happen:

- Open the browser
- Visit the page
- Find the element and send the space key to it
- Find the result text on the page and check to that it's what we expect
- Send the tab key to the element that's currently in focus
- Find the result text on the page and check to that it's what we expect
- Close the browser

Outro

If you have a specific element that you want to issue key presses to, then finding the element first is the way to go. But if you don't have a receiving element, or you need to string together multiple key presses, then the action builder is what you should reach for.

Happy Testing!

Chapter 15

How To Right-click

The Problem

Sometimes you'll run into an app that has functionality hidden behind a right-click menu (a.k.a. a context menu). These menus tend to be system level menus that are untouchable by Selenium. So how do you test this functionality?

A Solution

By leveraging Selenium's Action Builder we can issue a right-click command (a.k.a. a `context_click`).

We can then select an option from the menu by traversing it with keyboard arrow keys (which we can issue with the Action Builder's `send_keys` command). For a full write-up on working with keyboard keys in Selenium, see tip 61.

Let's dig in with an example.

An Example

Let's start by pulling in the necessary libraries (`selenium-webdriver` to control the browser and `rspec/expectations` & `RSpec::Matchers` to perform an assertion) and wiring up some simple `setup`, `teardown`, and `run` methods.

```ruby
# filename: right_click.rb

require 'selenium-webdriver'
require 'rspec-expectations'
include RSpec::Matchers

def setup
  @driver = Selenium::WebDriver.for :firefox
end

def teardown
  @driver.quit
end

def run
  setup
  yield
  teardown
end
```

Now we're ready to write our test.

Let's use an example from the-internet that will render a custom context menu when we right-click on a specific area of the page (link). Clicking the context menu will trigger a JavaScript alert which will say `You selected a context menu`. We'll grab this text and use it to assert that the menu was actually triggered.

```ruby
run do
  @driver.get 'http://the-internet.herokuapp.com/context_menu'
  menu_area = @driver.find_element id: 'hot-spot'
  @driver.action.context_click(menu_area).perform
  alert = @driver.switch_to.alert
  expect(alert.text).to eq('You selected a context menu')
end
```

Expected Behavior

If we save this file and run it (e.g., `ruby right_click.rb`) from the command-line) here is what will happen:

- Open the browser and visit the page
- Find and right-click the area which will render a custom context menu
- Select the context menu option with keyboard keys
- JavaScript alert appears

- Grab the text of the JavaScript alert
- Assert that the text from the alert is what we expect

Outro

To learn more about context menus, you can read this write-up from the Tree House blog. And for more thorough examples on working with keyboard keys and JavaScript alerts in your Selenium tests, check out tips 61 and 51.

Happy Testing!

Chapter 16

How To Opt-out of A/B Tests

The Problem

Occasionally when running tests you may see unexpected behavior due to A/B testing (a.k.a. split testing) of the application you're working with.

In order to keep your tests running without issue we need a clean way to opt-out of these split tests.

A quick primer on A/B testing

Split testing is a simple way to experiment with an application's features to see which changes lead to higher user engagement.

A simple example would be testing variations of an e-mail landing page to see if more people sign up. In such a split test there would be the control (how the application looks and behaves now) and variants (e.g., 2 or 3 changes that could include changing text or images on the page, element positioning, color of the submit button, etc).

Once the variants are configured, they are put into rotation, and the experiment starts. During this experiment each user will see a different version of the feature and their engagement with it will be tracked. Split tests live for the length of the experiment or until a winner is found (e.g., tracking indicates that a variant converted higher than the control). If no winner is found, new variants may be created and another experiment tried. If a winner is found, then the winning variant becomes the new control and the feature gets updated accordingly.

A Solution

Thankfully there are some standard opt-out mechanisms built into A/B testing platforms. They tend to come in the form of an appended URL or forging a cookie. Let's dig in with an example of each approach with a popular A/B testing platform, Optimizely.

An Example

Our example page is from the-internet and can be seen here. There are three different versions of the page that are available. On each page the heading text will vary:

- Control: `A/B Test Control`
- Variation 1: `A/B Test Variation 1`
- Opt-out: `No A/B Test`

Let's kick things off by loading our requisite libraries (e.g., `selenium-webdriver` to control the browser and `rspec/expectations` and it's matchers for our assertions) and adding some simple `setup`, `teardown`, and `run` methods to handle our test configuration.

```
# filename: split_test.rb

require 'selenium-webdriver'
require 'rspec/expectations'
include ::RSpec::Matchers

def setup
  @driver = Selenium::WebDriver.for :firefox
end

def teardown
  @driver.quit
end

def run
  setup
  yield
  teardown
end
```

Now let's wire up our first test to step through loading the split testing page and verifying that the text changes as we forge an opt-out cookie.

```
run do
  @driver.get 'http://the-internet.herokuapp.com/abtest'
  heading_text = @driver.find_element(css: 'h3').text
  expect(['A/B Test Variation 1', 'A/B Test Control'].include? heading_text).to eql
true
  @driver.manage.add_cookie(name: 'optimizelyOptOut', value: 'true')
  @driver.navigate.refresh
  heading_text = @driver.find_element(css: 'h3').text
  expect(heading_text).to eql('No A/B Test')
end
```

After navigating to the page we confirm that we are in one of the A/B test groups by grabbing the heading text and checking to see if it matches what we expect. After that we add the opt-out cookie, refresh the page, and then confirm that we are no longer in the A/B test group by checking the heading text again.

We could also load the opt-out cookie before navigating to this page.

```
run do
  @driver.get 'http://the-internet.herokuapp.com'
  @driver.manage.add_cookie(name: 'optimizelyOptOut', value: 'true')
  @driver.get 'http://the-internet.herokuapp.com/abtest'
  expect(@driver.find_element(css: 'h3').text).to eql('No A/B Test')
end
```

Here we are navigating to the main page of the site first and then adding the opt-out cookie. After that we navigate to the split test page and then performing our checks. Alternatively, we could opt out without forging a cookie. Instead we just need to append an opt out request to the URL.

```
run do
  @driver.get 'http://the-internet.herokuapp.com/abtest?optimizely_opt_out=true'
  @driver.switch_to.alert.dismiss
  sleep 1 # to account for the document load to start
  expect(@driver.find_element(css: 'h3').text).to eql('No A/B Test')
end
```

By appending `?optimizely_opt_out=true` we achieve the same outcome as before. Keep in mind that this approach triggers a JavaScript alert, so we have to switch to and dismiss it (e.g., `@driver.switch_to.alert.dismiss`) before performing our check.

Expected Behavior

If you save this file and run it (e.g., `ruby split_test.rb` from the command-line) here is what will happen:

- Open the browser
- Opt-out of the split tests (either by cookie or appended URL)
- Visit the split testing page
- Grab the header text
- Confirm that the session is opted out of the split tests
- Close the browser

Outro

Happy Testing!

Chapter 17

How To Access Basic Auth

The Problem

Sometimes you'll work with applications that are secured behind Basic HTTP Authentication (a.k.a. Basic Auth). In order to access them you'll need to pass credentials to the site when requesting a page. Otherwise you'll get a system level pop-up prompting you for a username and password -- rendering Selenium helpless.

Before Selenium 2 we were able to accomplish this by injecting credentials into a custom header, but now the cool kid way to do it it was something like BrowserMob Proxy. And some people are solving this with browser specific configurations too.

But all of this feels heavy. Instead, let's look at a simple approach that is browser agnostic and quick to setup.

A Solution

By specifying the username and password in the URL when visiting a page with Selenium, we can a side-step the system level dialog box and avoid setting up a proxy server. This approach will work for both HTTP or HTTPS pages.

Let's take a look at an example.

An Example

Let's start by requiring our requisite libraries (e.g., `selenium-webdriver` to driver the browser and `rspec/expectations` & `RSpec::Matchers` to handle our assertions) and implementing some helper methods (e.g., `setup`, `teardown`, and `run`) to handle our test configuration.

```ruby
# filename: basic_auth.rb

require 'selenium-webdriver'
require 'rspec/expectations'
include RSpec::Matchers

def setup
  @driver = Selenium::WebDriver.for :firefox
end

def teardown
  @driver.quit
end

def run
  setup
  yield
  teardown
end
```

Now to add our test.

```ruby
run do
  @driver.get 'http://admin:admin@the-internet.herokuapp.com/basic_auth'
  page_message = @driver.find_element(css: '.example p').text
  expect(page_message).to eql 'Congratulations! You must have the proper credentials.'
end
```

In the test we're loading the page by passing in the username and password in the front of the URL (e.g., `http://admin:admin@`). Once it loads we grab text from the page to make sure we ended up in the right place.

Alternatively, we could have accessed this page as part of the test setup (after creating an instance of Selenium). This would have cached the Basic Auth session in the browser, enabling us to visit the page again without having to specify credentials. This is particularly useful if you have numerous pages behind Basic Auth.

Here's what that would look like.

```ruby
# filename: basic_auth_setup.rb

require 'selenium-webdriver'
require 'rspec/expectations'
include RSpec::Matchers

def setup
  @driver = Selenium::WebDriver.for :firefox
  @driver.get 'http://admin:admin@the-internet.herokuapp.com/basic_auth'
end

def teardown
  @driver.quit
end

def run
  setup
  yield
  teardown
end

run do
  @driver.get 'http://the-internet.herokuapp.com/basic_auth'
  page_message = @driver.find_element(css: '.example p').text
  expect(page_message).to eql 'Congratulations! You must have the proper credentials.'
end
```

NOTE: If your application serves both HTTP and HTTPS pages from behind Basic Auth then you will need to load one of each type before executing your test steps. Otherwise you will get authorization errors when switching between HTTP and HTTPS because the browser can't use Basic Auth credentials interchangeably (e.g. HTTP for HTTPS and vice versa).

Expected Behavior

When you save the first example and run it (e.g., `ruby basic_auth.rb`), here is what will happen:

- Open the browser
- Visit the page using Basic Auth
- Get the page text
- Assert that the text is what we expect
- Close the browser

And when you save the second example and run it (e.g., `ruby basic_auth_setup.rb`), here is what will happen:

- Open the browser
- Visit the page using Basic Auth in the setup
- Navigate to the Basic Auth page (without providing credentials)
- Get the page text
- Assert that the text is what we expect

Outro

Hopefully this tip will help save you from getting tripped by Basic Auth when you come across it.

Happy Testing!

Chapter 18

How To Visually Verify Your Locators

This is a pseudo guest post from Brian Goad. I've adapted a blog post of his with permission. You can see the original here. Brian is a Test Engineer at Digitalsmiths. You can follow him on Twitter at @bbbco and check out his testing blog here.

The Problem

It's likely that you'll run into odd test behavior that makes you question the locators you're using in a test. But how do you interrogate your locators to make sure they are doing what you expect?

A Solution

By leveraging some simple JavaScript and CSS styling, we can highlight the element on the page that we're targeting. This way we can visually inspect it to make sure it is the one that we want.

Let's take a look at an example.

An Example

For our initial setup let's pull in the `selenium-webdriver` gem and wire up some simple `setup`, `teardown`, and `run` methods.

```
# filename: highlight.rb

require 'selenium-webdriver'

def setup
  @driver = Selenium::WebDriver.for :firefox
end

def teardown
  @driver.quit
end

def run
  setup
  yield
  teardown
end
```

Now let's create a `highlight` helper method that will accept a Selenium WebDriver `element` and a time to wait (e.g., `duration`) as arguments.

By setting a duration, we can control how long to highlight an element on the page before reverting the styling back. And we can make this an optional argument by setting a default (e.g., 3 seconds).

```
def highlight(element, duration = 3)

  # store original style so it can be reset later
  original_style = element.attribute("style")

  # style element with yellow border
  @driver.execute_script(
    "arguments[0].setAttribute(arguments[1], arguments[2])",
    element,
    "style",
    "border: 2px solid red; border-style: dashed;")

  # keep element highlighted for a spell and then revert
  if duration > 0
    sleep duration
    @driver.execute_script(
      "arguments[0].setAttribute(arguments[1], arguments[2])",
      element,
      "style",
      original_style)
  end

end
```

There are three things going on here.

1. We store the style of the element so we can revert it back when we're done
2. We change the style of the element so it visually stands out (e.g., a red dashed border)
3. We revert the style of the element back after 3 seconds

We're accomplishing the style change through JavaScript's `setAttribute` function. And we're executing it with Selenium's `execute_script` command.

Now to use this in our test is simple, we just prepend a `find_element` action with the `highlight` command.

```ruby
run do
  @driver.get 'http://the-internet.herokuapp.com/large'
  highlight @driver.find_element(id: 'sibling-2.3')
end
```

Expected Behavior

If you were to save this file and run it (e.g., `ruby highlight.rb` from the command-line), here is what you would see.

1. Load the page
2. Find the element
3. Change the styling of the element so it has a red dashed-line border
4. Wait 3 seconds
5. Revert the styling to remove the border

Outro

If you wanted to take this a step further, you could leverage this approach along with an interactive debugger. You can read more about how to do that here in Brian's other guest post.

Alternatively, you could verify your locators by using a browser plugin like FireFinder. You can read more about how to do that here in this previous tip.

Happy Testing!

Chapter 19

How To Add Growl Notifications To Your Tests

The Problem

Good test reports are a fundamental component of successful test automation. But running down a test failure by looking at a test report can be a real pain sometimes.

Leaving you with no choice but to roll up your sleeves and get your hands dirty with debug statements, or step through things piece by piece -- all for the sake of trying to track down a transient issue.

A Solution

By leveraging something like jQuery Growl you can output non-interactive debugging statements directly to the page you're testing. This way you can see helpful information and more-likely correlate it to the test actions that are being taken. This can a boon for your test runs when coupled with screenshots and/or video recordings of your test runs

Let's step through an example of how to set this up.

An Example

First, we'll include our requisite libraries and wire up some simple `setup`, `teardown`, and `run` methods.

```ruby
# filename: growl.rb

require 'selenium-webdriver'

def setup
  @driver = Selenium::WebDriver.for :firefox
end

def teardown
  @driver.quit
end

def run
  setup
  yield
  teardown
end
```

Next we'll need to visit the page we want to display notifications on and do some work with JavaScript to load jQuery, jQuery Growl, and styles for jQuery Growl. After that, we can issue commands to jQuery Growl to make notification messages display on the page.

```
run do
  @driver.get 'http://the-internet.herokuapp.com'

  # Step 1: check for jQuery on the page, add it if needbe
  @driver.execute_script("if (!window.jQuery) {
var jquery = document.createElement('script'); jquery.type = 'text/javascript';
jquery.src = 'https://ajax.googleapis.com/ajax/libs/jquery/2.0.2/jquery.min.js';
document.getElementsByTagName('head')[0].appendChild(jquery);
}")

  # Step 2: use jQuery to add jquery-growl to the page
  @driver.execute_script(
"$.getScript('http://the-internet.herokuapp.com/js/vendor/jquery.growl.js')")

  # Step 3: use jQuery to add jquery-growl styles to the page
  @driver.execute_script("$('head').append('<link rel=\"stylesheet\" href=\"
http://the-internet.herokuapp.com/css/jquery.growl.css\" type=\"text/css\" />');")

  # add delay for resource loading
  sleep 1

  # Step 4: display a message with jquery-growl
  @driver.execute_script("$.growl({ title: 'GET', message: '/' });")

  sleep 5 # to keep the browser active long enough to see the notifications
end
```

And if we wanted to see color-coded notifications, then we could use one of the following:

```
@driver.execute_script("$.growl.error({ title: 'ERROR', message: 'your error message
goes here' });")
  @driver.execute_script("$.growl.notice({ title: 'Notice', message: 'your notice
message goes here' });")
  @driver.execute_script("$.growl.warning({ title: 'Warning!', message: 'your warning
message goes here' });")
```

Expected Behavior

- Load the page
- Display a set of notification messages in the top-right corner of the page
- Notification messages disappear

Outro

In order to use this approach, you will need to load jQuery Growl on every page you want to display output to -- which can be a bit of overhead. But if you want rich messaging like this, then that's the price you have to pay (unless you can get your team to add it to the application under test).

In a future tip I'll step through how to access Selenium logging output so we can wire it up to these notifications. I'd like to give a big thanks to Jon Austen (Twitter, GitHub, Blog) for giving me the idea to use jQuery Growl with Selenium.

Happy Testing!