



Selenium WebDriver Recipes in Ruby

The Problem Solving Guide to Selenium WebDriver



Zhimin Zhan

Selenium WebDriver Recipes in Ruby

The problem solving guide to Selenium WebDriver in Ruby

Zhimin Zhan

This book is for sale at <http://leanpub.com/selenium-recipes-in-ruby>

This version was published on 2015-10-08



This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

©2013 - 2015 Zhimin Zhan

To Dominic and Courtney!

Contents

1. Preface	1
Who should read this book	2
How to read this book	2
Recipe test scripts	2
Send me feedback	2
2. Introduction	3
Selenium	3
Selenium language bindings: Java, C#, Python and Ruby	3
Cross browser testing	6
RSpec	9
Run recipe scripts	13
3. Locating web elements	18
Start browser	18
Find element by ID	19
Find element by Name	19
Find element by Link Text	19
Find element by Partial Link Text	19
Find element by XPath	20
Find element by Tag Name	21
Find element by Class	22
Find element by CSS Selector	22
Chain find_element to find child elements	22
Find multiple elements	23
4. Hyperlink	24
Click a link by text	24
Click a link by ID	24

CONTENTS

Click a link by partial text	24
Click a link by XPath	25
Click Nth link with exact same label	26
Verify a link present or not?	26
Getting link data attributes	27
Test links open a new browser window	27

1. Preface

After observing many failed test automation attempts by using expensive commercial test automation tools, I am delighted to see that the value of open-source testing frameworks has finally been recognized. I still remember the day (a rainy day at a Gold Coast hotel in 2011) when I found out that the Selenium WebDriver was the most wanted testing skill in terms of the number of job ads on the Australia's top job-seeking site.

Now Selenium WebDriver is big in the testing world. We all know software giants such as Facebook and LinkedIn use it, immensely-comprehensive automated UI testing enables them [pushing out releases several times a day](#)¹. However, from my observation, many software projects, while using Selenium, are not getting much value from test automation, and certainly nowhere near its potential. A clear sign of this is that the regression testing is not conducted on a daily basis (if test automation is done well, it will happen naturally).

Among the factors contributing to test automation failures, a key one is that automation testers lack sufficient knowledge in the test framework. It is quite common to see some testers or developers get excited when they first create a few simple test cases and see them run in a browser. However, it doesn't take long for them to encounter some obstacles: such as being unable to automate certain operations. If one step cannot be automated, the whole test case does not work, which is the nature of test automation. Searching solutions online is not always successful, and posting questions on forums and waiting can be frustrating (usually, very few people seek professional help from test automation coaches). Not surprisingly, many projects eventually gave up test automation or just used it for testing a handful of scenarios.

The motivation of this book is to help motivated testers work better with Selenium. The book contains over 100 recipes for web application tests with Selenium. If you have read one of my other books: '[Practical Web Test Automation](#)'², you probably know my style: practical. I will let the test scripts do most of the talking. These recipe test scripts are 'live', as I have created the target test site and included offline test web pages. With both, you can:

1. **Identify** your issue
2. **Find** the recipe
3. **Run** the test case
4. **See** test execution in your browser

¹<http://www.wired.com/business/2013/04/linkedin-software-revolution/>

²<https://leanpub.com/practical-web-test-automation>

Who should read this book

This book is for testers or programmers who are writing (or want to learn) automated tests with Selenium WebDriver. In order to get the most of this book, basic Ruby coding skill is required.

How to read this book

Usually, a ‘recipe’ book is a reference book. Readers can go directly to the part that interests them. For example, if you are testing a multiple select list and don’t know how, you can look up in the Table of Contents, then go to the chapter. This book supports this style of reading. Since the recipes are arranged according to their levels of complexity, readers will also be able to work through the book from the front to back if they are looking to learn test automation with Selenium.

Recipe test scripts

To help readers to learn more effectively, this book has a [dedicated site](http://zhimin.com/books/selenium-recipes)³ which contains the sample test scripts and related resources.

As an old saying goes, “There’s more than one way to skin a cat.” You can achieve the same testing outcome with test scripts implemented in different ways. The recipe test scripts in this book are written for simplicity, there is always room for improvement. But for many, to understand the solution quickly and get the job done are probably more important.

If you have a better and simpler way, please let me know.

All recipe test scripts are Selenium 2 (aka Selenium WebDriver) compliant, and can be run on Firefox, Chrome and Internet Explorer on multiple platforms. I plan to keep the test scripts updated with the latest stable Selenium version.

Send me feedback

I would appreciate your comments, suggestions, reports on errors in the book and the recipe test scripts. You may submit your feedback on the book’s site.

Zhimin Zhan

March 2014

³<http://zhimin.com/books/selenium-recipes>

2. Introduction

Selenium is a free and open source library for automated testing web applications. I assume that you have had some knowledge of Selenium, based on the fact that you picked up this book (or opened it in your eBook reader).

Selenium

Selenium was originally created in 2004 by Jason Huggins, who was later joined by his other ThoughtWorks colleagues. Selenium supports all major browsers and tests can be written in many programming languages and run on Windows, Linux and Macintosh platforms.

Selenium 2 is merged with another test framework WebDriver (that's why you see 'selenium-webdriver') led by Simon Stewart at Google (update: Simon now works at FaceBook), Selenium 2.0 was released in July 2011.

Selenium language bindings: Java, C#, Python and Ruby

Selenium tests can be written in multiple programming languages such as Java, C#, Python and Ruby (the core ones). Quite commonly, I heard the saying such as *"This is a Java project, so we shall write tests in Java as well"*. I disagree. Software testing is to verify whether programmer's work meets customer's needs. In a sense, testers are representing customers. Testers should have more weight on deciding the test syntax than programmers. Plus, why would you mandate that your testers should have the same programming language skills as the programmers. In my subjective view, scripting languages such as Ruby and Python are more suitable for test scripts than compiled languages such as C# and Java (Confession: I have been programming in Java for over 10 years). By the way, we call them test scripts, for a reason.

All examples in this book are written in Selenium with Ruby binding. This does not mean this book is limited to testers/developers who know Ruby. As you will see the examples below, the use of Selenium in different bindings are very similar. Once you master one, you can apply it to others quite easily. Take a look at a simple Selenium test script in four different language bindings: Java, C#, Python and Ruby.

Java:


```
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.firefox.FirefoxDriver;

public class GoogleSearch {
    public static void main(String[] args) {
        // Create a new instance of the html unit driver
        // Notice that the remainder of the code relies on the interface,
        // not the implementation.
        WebDriver driver = new FirefoxDriver();

        // And now use this to visit Google
        driver.get("http://www.google.com");

        // Find the text input element by its name
        WebElement element = driver.findElement(By.name("q"));

        // Enter something to search for
        element.sendKeys("Hello Selenium WebDriver!");

        // Submit the form based on an element in the form
        element.submit();

        // Check the title of the page
        System.out.println("Page title is: " + driver.getTitle());
    }
}
```

C#:

```
using System;
using OpenQA.Selenium;
using OpenQA.Selenium.Firefox;
using OpenQA.Selenium.Support.UI;

class GoogleSearch
{
    static void Main()
    {
        IWebDriver driver = new FirefoxDriver();
        driver.Navigate().GoToUrl("http://www.google.com");
        IWebElement query = driver.FindElement(By.Name("q"));
        query.SendKeys("Hello Selenium WebDriver!");
        query.Submit();
        Console.WriteLine(driver.Title);
    }
}
```

Python:

```
from selenium import webdriver

driver = webdriver.Firefox()
driver.get("http://www.google.com")

elem = driver.find_element_by_name("q")
elem.send_keys("Hello WebDriver!")
elem.submit()

print(driver.title)
```

Ruby:

```
require "selenium-webdriver"

driver = Selenium::WebDriver.for :firefox
driver.navigate.to "http://www.google.com"

element = driver.find_element(:name, 'q')
element.send_keys "Hello Selenium WebDriver!"
element.submit

puts driver.title
```

Cross browser testing

The biggest advantage of Selenium over other web test frameworks, in my opinion, is that it supports all major web browsers: Firefox, Chrome and Internet Explorer. The browser market nowadays is more diversified (based on the [StatsCounter](http://www.statscounter.com)¹, the usage share in November 2014 for Chrome, IE and Firefox are 51.8%, 21.7% and 18.5% respectively). It is logical that all external facing web sites require serious cross-browser testing. Selenium is a natural choice for this purpose, as it far exceeds other commercial tools and free test frameworks.

Firefox

Selenium supports Firefox out of the box, i.e., as long as you have Firefox (and a not too outdated version) installed, you are ready to go. The test script below (in a file named: *ch01_open_firefox.rb*) will open a web site in a new Firefox window.

```
require 'selenium-webdriver'
driver = Selenium::WebDriver.for(:firefox)
driver.navigate.to("http://testwisely.com/demo")
```

For readers who can't wait to see the test running, below is the command you need to use to execute a test, which you can download from the book's site (Ruby and *selenium-webdriver* gem need to be installed first. See instructions towards the end of this chapter).

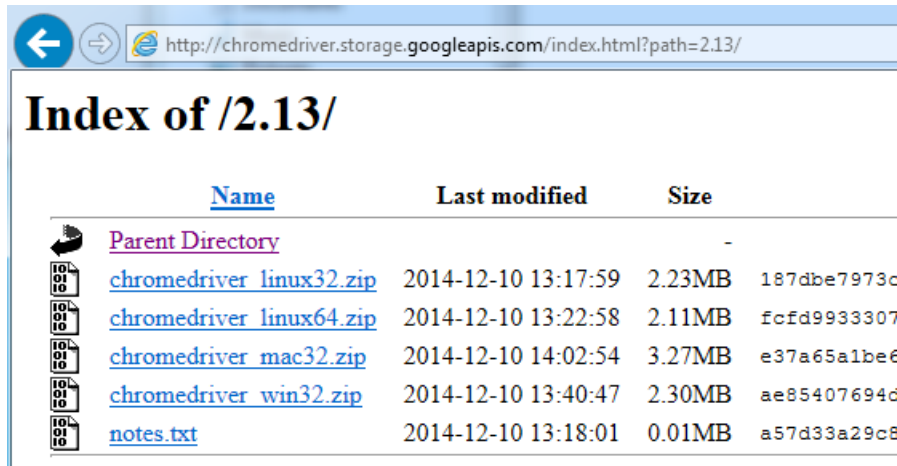
¹http://en.wikipedia.org/wiki/Usage_share_of_web_browsers







```
> ruby ch01_open_firefox.rb
```

Chrome

To run Selenium tests in Google Chrome, besides the Chrome browser itself, *ChromeDriver* needs to be installed.

Installing ChromeDriver is easy: go to [ChromeDriver site](http://chromedriver.storage.googleapis.com/index.html?path=2.13/)².



	<u>Name</u>	<u>Last modified</u>	<u>Size</u>	
	Parent Directory		-	
	chromedriver linux32.zip	2014-12-10 13:17:59	2.23MB	187dbe7973c
	chromedriver linux64.zip	2014-12-10 13:22:58	2.11MB	fcfd9933307
	chromedriver mac32.zip	2014-12-10 14:02:54	3.27MB	e37a65a1be6
	chromedriver win32.zip	2014-12-10 13:40:47	2.30MB	ae85407694d
	notes.txt	2014-12-10 13:18:01	0.01MB	a57d33a29c8

Download the one for your target platform, unzip it and put **chromedriver** executable in your PATH. To verify the installation, open a command window (terminal for Unix/Mac), execute command *chromedriver*, You shall see:

```
C:\>chromedriver
Starting ChromeDriver 2.13.307647 (5a7d0541ebc58e69994a6fb2ed930f45261f3c29) on port 9515
Only local connections are allowed.
```

The test script below opens a site in a new Chrome browser window and closes it one second later.

```
require 'selenium-webdriver'
driver = Selenium::WebDriver.for(:chrome)
driver.navigate.to("http://testwisely.com/demo")
sleep 1 # wait 1 second
driver.quit
```

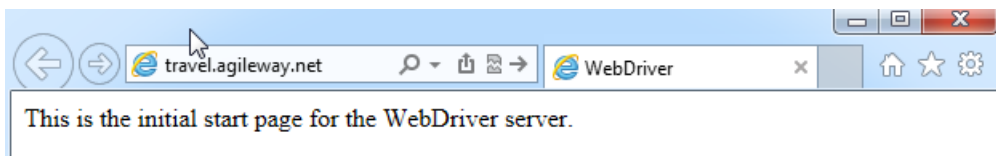
²<https://sites.google.com/a/chromium.org/chromedriver/downloads>

Internet Explorer

Selenium requires IEDriverServer to drive IE browser. Its installation process is very similar to *chromedriver*. IEDriverServer is available at <http://www.seleniumhq.org/download/>³. Choose the right one based on your windows version (32 or 64 bit).

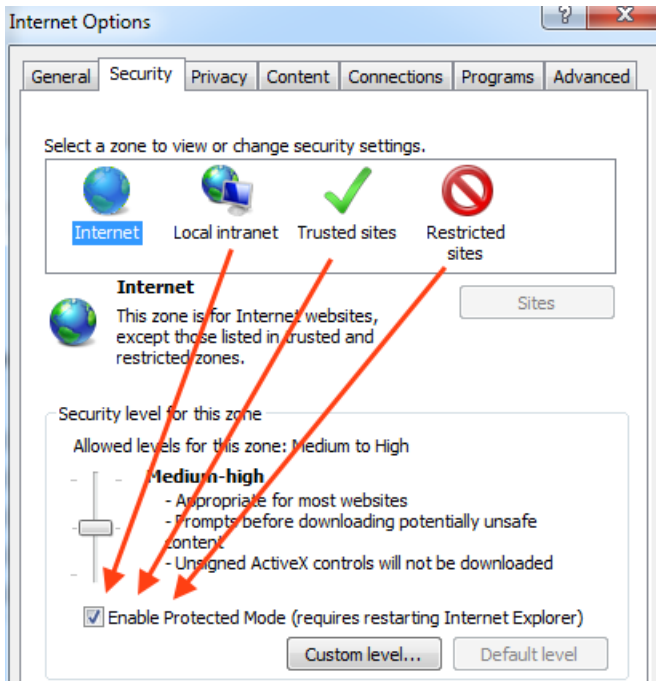
Download version 2.44.0 for (recommended) [32 bit Windows IE](#) or [64 bit Windows IE](#)
[CHANGELOG](#)

When a tests starts to execute in IE, before navigating the target test site, you will see this first:



If you get this on IE9: “Unexpected error launching Internet Explorer. Protected Mode must be set to the same value (enabled or disabled) for all zones.” Go to ‘Internet Options’, select each zone (as illustrated below) and make sure they are all set to the same mode (protected or not).

³<http://www.seleniumhq.org/download/>



Further configuration is required for IE10 and IE11, see [IE and IEDriverServer Runtime Configuration](#)⁴ for details.

```
require 'selenium-webdriver'
driver = Selenium::WebDriver.for(:ie)
driver.navigate.to("http://testwisely.com/demo")
```

RSpec

Selenium drives browsers. However, to make the effective use of Selenium scripts for testing, we need to put them in a test framework that defines test structures and provides assertions (performing checks in test scripts). Typical choices are:

- xUnit Unit Test Frameworks such as JUnit (for Java), NUnit (for C#) and minitest (for Ruby).
- Behaviour Driven Frameworks such as RSpec and Cucumber (for Ruby).

⁴https://code.google.com/p/selenium/wiki/InternetExplorerDriver#Required_Configuration

In this book, I use RSpec, the de facto Behaviour Driven Development (BDD) framework for Ruby. Here is an example.

```
require 'selenium-webdriver'

describe "Selenium Recipes - Start different browsers" do

  it "Start Chrome" do
    driver = Selenium::WebDriver.for(:chrome)
    driver.navigate.to("http://travel.agileway.net")
    sleep 1
    driver.quit
  end

  it "Start FireFox" do
    driver = Selenium::WebDriver.for(:firefox)
    driver.navigate.to("http://travel.agileway.net")
    expect(driver.title).to eq("Agile Travel")
    sleep 1
    driver.quit
  end

  it "Start IE" do
    driver = Selenium::WebDriver.for(:ie)
    driver.get("http://travel.agileway.net")
    sleep 1
    expect(driver.page_source).to include("User Name")
    driver.quit
  end
end
```

The keywords describe and it define the structure of a test script.

- **describe "..."** do

Description of a collection of related test cases

- `it "..."` `do`

Individual test case.

`expect().to` statements are called `rspec-expectations`, which are used to perform checks (also known as assertions).

RSpec's old "should-based" expectation syntax

There is also an older `should-based` syntax, which is still supported in RSpec 3 but deprecated. Here is the `should-syntax` version of the above example:

```
driver.title.should == "Selenium Recipes"
driver.title.include?("Selenium").should be_truthy
driver.title.include?("Selenium").should_not be_falsey
driver.title.should_not include("Watir")
```

Typical RSpec test script

The above example test script shows the basic structure and assertions in RSpec. However, we typically don't write a test case explicitly for multiple browsers, instead, we focus on test scenarios like below.

```
load File.dirname(__FILE__) + '/../test_helper.rb'

describe "Selenium Recipes - Start different browsers" do
  include TestHelper

  before(:all) do
    @driver = Selenium::WebDriver.for(:chrome)
  end

  before(:each) do
    @driver.navigate.to("http://travel.agileway.net")
  end
```



```
after(:all) do
  @driver.quit
end

it "Invalid Login" do
  @driver.find_element(:id, "username").send_keys("agileway")
  @driver.find_element(:id, "password").send_keys("changeme")
  @driver.find_element(:xpath, "///input[@value='Sign in']").click
  expect(@driver.page_source).to include("Invalid email or password")
end

it "Login successfully" do
  @driver.find_element(:id, "username").send_keys("agileway")
  @driver.find_element(:id, "password").send_keys("testwise")
  @driver.find_element(:xpath, "///input[@value='Sign in']").click
  expect(@driver.page_source).to include("Signed in!")
end

end
```

In this test script, we see another RSpec feature.

- **before()** and **after()** hooks.

Optional test statements run before and after each or all test cases. Use these hooks effectively can help you writing the test scripts that are more concise and easier to maintain.

You will find more about RSpec from [its home page](http://rspec.info)⁵. However, I honestly don't think it is necessary. The part used for test scripts is not much and quite intuitive. After studying and trying out some examples, you will be quite comfortable with RSpec.

As a general good practice, all test scripts include a common test helper (include TestHelper) which loads required libraries and defines a set of reusable functions that are available to all test cases. For more on designing maintainable test scripts, refer to my other book: *Practical Web Test Automation*⁶.

⁵<http://rspec.info>

⁶<https://leanpub.com/practical-web-test-automation>

Run recipe scripts

Test scripts for all recipes can be downloaded from the book site. They are all in ready-to-run state. I include the target web pages/sites as well as Selenium test scripts. There are two kinds of target web pages: local HTML files and web pages on a live site. Running tests written for a live site requires Internet connection.

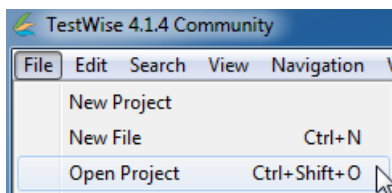
Run tests in TestWise

TestWise is a functional testing Integration Development Environment (IDE) that supports Selenium and Watir (another popular web test library, used for testing Internet Explorer). TestWise community edition is free of charge.

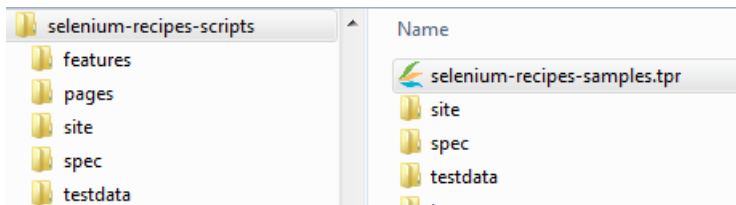
In this book, I refer to TestWise when editing or executing test scripts. If you have a preferred testing tools or IDE, such as Apatana Studio and NetBeans 6, go for it. It shall not affect your learning this book or running recipe test scripts.

Installing TestWise is easy. It only takes a couple of minutes (unless your Internet speed is very slow) to download and install. And TestWise is the only software you need to use while learning with this book (or developing Selenium test scripts for your work). Using it is likely to make it easier for you to focus on learning Selenium.

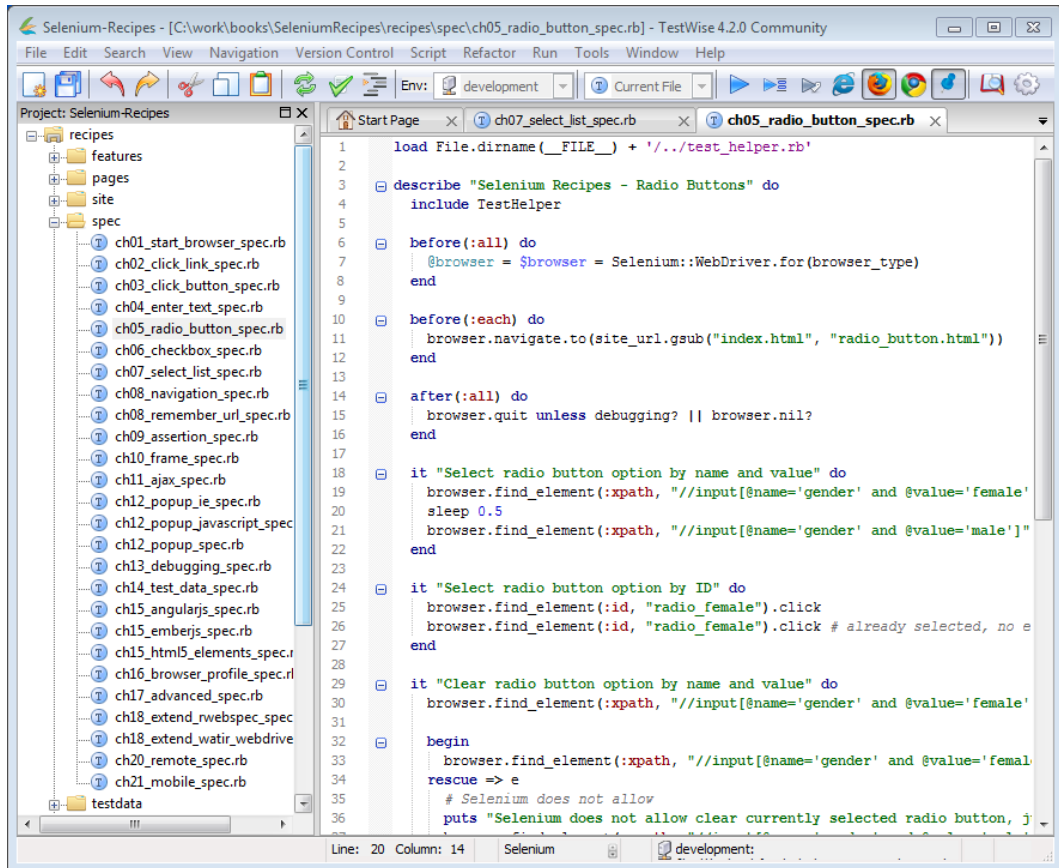
To open recipe test scripts (after you download the recipe tests (a zip file) from the book site and unzip to a folder), close the currently opened project if there is one. Select menu File → Open Project,



select the project file *selenium-recipes-scripts\selenium-recipes-samples.tpr*



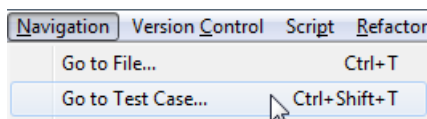
The TestWise window (after loading a test project) should look like the following:



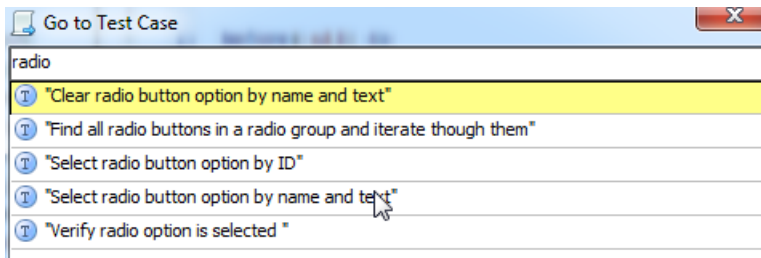
Find a test case

You can locate the recipe either by following the chapter or searching by name. There are over 100 test cases in one test project. Here is the quickest way to find the one you want in TestWise.

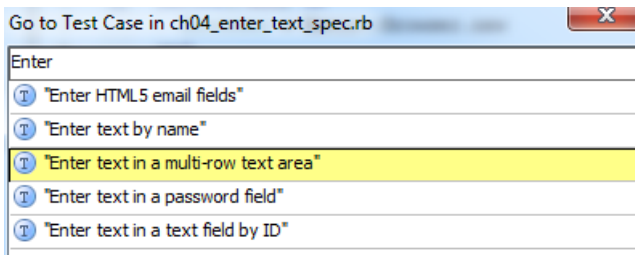
Select menu 'Navigation' → 'Go to Test Case..'



A pop up window lists all test cases in the project for your selection. The finding starts as soon as you type.

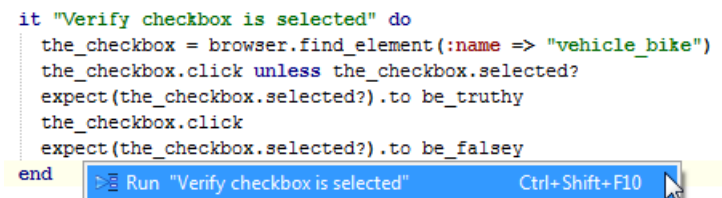


Within a test script file (opened in the editor), press Ctrl+F12 to show and select test cases inside it.

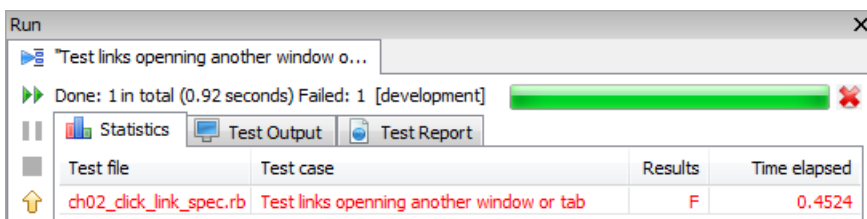


Run individual test case

Move caret to a line within a test case (between it "... do and end). Right mouse click and select "Run '...'".

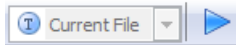


The below is a screenshot of execution panel when one test case failed,

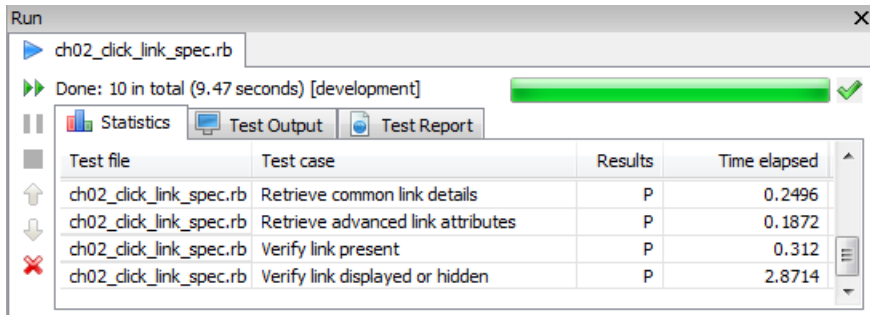


Run all test cases in a test script file

You can also run all test cases in the currently opened test script file by clicking the blue triangle button on the tool bar.



The below is a screenshot of the execution panel when all test cases in a test script file passed,



Run tests from command line

One advantage of open-source test frameworks, such as Selenium, is FREEDOM. You can edit the test scripts in any text editors and run them from a command line.

You need to install Ruby first, then install RSpec and your preferred web test driver and library (known as Gem in Ruby). Basic steps are:

- install Ruby interpreter
Window installer: <http://rubyinstaller.org>, Linux/Mac: included or compile from source
- install RSpec
> *gem install rspec*
- install test framework gem(s)
> *gem install selenium-webdriver*

For windows users, you may simply download and install the free pre-packaged RubyShell (based on Ruby Windows Installer) at testwisely.com⁷.

⁷<http://testwisely.com/testwise/downloads>

Once the installation (takes about 1 minute) is complete, we can run an RSpec test from the command line. You need to have some knowledge of typing commands in a console (Unix) or command window.

To run test cases in a test script file (named *google_spec.rb*), enter command

```
> rspec google_spec.rb
```

Run multiple test script files in one go:

```
> rspec first_spec.rb second_test.rb
```

Run an individual test case in a test script file, supply a line number in a chosen test case range.

```
> rspec google_spec.rb:30
```

To generate a test report (HTML) after test execution:

```
> rspec -fh google_spec.rb > test_report.html
```

The command syntax is identical for Mac OS X and Linux platforms.

3. Locating web elements

As you might have already figured out, to drive an element in a page, we need to find it first. Selenium uses what is called locators to find and match the elements on web page. There are 8 locators in Selenium:

Locator	Example
ID	<code>find_element(:id, "user")</code>
Name	<code>find_element(:name, "username")</code>
Link Text	<code>find_element(:link_text, "Login")</code> <code>find_element(:link, "Login")</code>
Partial Link Text	<code>find_element(:partial_link_text, "Next")</code>
XPath	<code>find_element(:xpath, "//div[@id='login']/input")</code>
Tag Name	<code>find_element(:tag_name, "body")</code>
Class Name	<code>find_element(:class_name, "table")</code> <code>find_element(:class, "body")</code>
CSS	<code>find_element(:css, "#login > input[type='text']")</code>

You may use any one of them to narrow down the element you are looking for.

Start browser

Testing websites starts with a browser.

```
require 'selenium-webdriver'
driver = Selenium::WebDriver.for(:firefox)
driver.navigate.to("http://testwisely.com/demo")
```

Use `:chrome` and `:ie` for testing in Chrome and IE respectively.

I recommend, for beginners, to close the browser window at the end of a test case.

```
driver.quit # or driver.close
```

Find element by ID

Using IDs is the easiest and the safest way to locate an element in HTML. If the page is [W3C HTML conformed](http://www.w3.org/TR/WCAG20-TECHS/H93.html)¹, the IDs should be unique and identified in web controls. In comparison to texts, test scripts that use IDs are less prone to application changes (e.g. developers may decide to change the label, but are less likely to change the ID).

```
driver.find_element(:id, "submit_btn").click    # Button
driver.find_element(:id, "cancel_link").click   # Link
driver.find_element(:id, "username").send_keys("agileway") # Textfield
driver.find_element(:id, "alert_div").text      # HTML Div element
```

Find element by Name

The name attributes are used in form controls such as text fields and radio buttons. The values of the name attributes are passed to the server when a form is submitted. In terms of least likelihood of a change, the name attribute is probably only second to ID.

```
driver.find_element(:name, "comment").send_keys("Selenium Cool")
```

Find element by Link Text

For Hyperlinks only. Using a link's text is probably the most direct way to click a link, as it is what we see on the page.

```
driver.find_element(:link_text, "Cancel").click
```

Find element by Partial Link Text

Selenium allows you to identify a hyperlink control with a partial text. This can be quite useful when the text is dynamically generated. In other words, the text on one web page might be different on your next visit. We might be able to use the common text shared by these dynamically generated link texts to identify them.

¹<http://www.w3.org/TR/WCAG20-TECHS/H93.html>


```
# will click the "Cancel" link  
driver.find_element(:partial_link_text, "ance").click
```

Find element by XPath

XPath, the XML Path Language, is a query language for selecting nodes from an XML document. When a browser renders a web page, it parses it into a DOM tree or similar. XPath can be used to refer a certain node in the DOM tree. If this sounds a little too much technical for you, don't worry, just remember XPath is the most powerful way to find a specific web control.

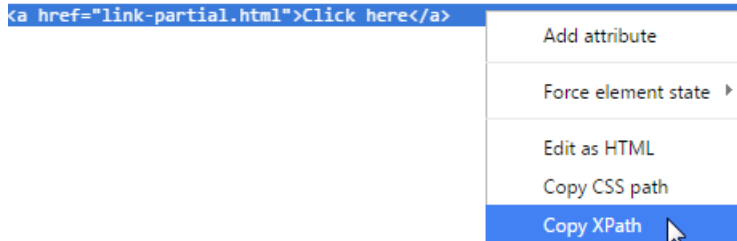
```
# clicking the checkbox under 'div2' container  
driver.find_element(:xpath, "//*[@id='div2']/input[@type='checkbox']").click
```

Some testers feel intimidated by the complexity of XPath. However, in practice, there is only limited scope of XPath to master for testers.



Avoid using copied XPath from Browser's Developer Tool

Browser's Developer Tool (right click to select 'Inspect element' to show) is very useful for identifying a web element in web page. You may get the XPath of a web element there, as shown below (in Chrome):



The copied XPath for the second "Click here" link in the example is:

```
//*[@id="container"]/div[3]/div[2]/a
```

It works. However, I do not recommend this approach as the test script is fragile. If developer adds another div under `<div id='container'>`, the copied XPath is no longer correct for the element while `//div[contains(text(), "Second")]/a[text()='Click here']` still works.

In summary, XPath is a very powerful way to locate web elements when `:id`, `:name` or `:link_text` are not applicable. Try to use a XPath expression that is less vulnerable to structure changes around the web element.

Find element by Tag Name

There are a limited set of tag names in HTML. In other words, many elements share the same tag names on a web page. We normally don't use the `tag_name` locator by itself to locate an element. We often use it with others in a chained locators (see the section below). However, there is an exception.

```
driver.find_element(:tag, "body").text
```

The above test statement returns the text view of a web page. This is a very useful one as Selenium WebDriver does not have built-in method to return the text of a web page.

Find element by Class

The `class` attribute of a HTML element is used for styling. It can also be used for identifying elements. Commonly, a HTML element's class attribute has multiple values, like below.

```
<a href="back.html" class="btn btn-default">Cancel</a>
<input type="submit" class="btn btn-default btn-primary">Submit</input>
```

You may use any one of them.

```
driver.find_element(:class, "btn-primary").click # Submit button
driver.find_element(:class, "btn").click         # Cancel link
```

```
# the below will return error "Compound class names not permitted"
# driver.find_element(:class, "btn btn-default btn-primary").click
```

The `class` locator is convenient for testing JavaScript/CSS libraries (such as TinyMCE) which typically use a set of defined class names.

```
# inline editing
driver.find_element(:id, "client_notes").click
sleep
driver.find_element(:class, "editable-textarea").send_keys("inline notes")
sleep 0.5
driver.find_element(:class, "editable-submit").click
```

Find element by CSS Selector

You may also use CSS Path to locate a web element.

```
driver.find_element(:css, "#div2 > input[type='checkbox']").click
```

However, the use of CSS selector is generally more prone to structure changes of a web page.

Chain `find_element` to find child elements

For a page containing more than one elements with the same attributes, like the one below, we could use XPath locator.

```
<div id="div1">
  <input type="checkbox" name="same" value="on"> Same checkbox in Div 1
</div>
<div id="div2">
  <input type="checkbox" name="same" value="on"> Same checkbox in Div 2
</div>
```

There is another way: chain `find_element` to find a child element.

```
driver.find_element(:id, "div2").find_element(:name, "same").click
```

Find multiple elements

As its name suggests, `find_elements` return a list of matched elements. Its syntax is exactly the same as `find_element`, i.e. can use any of 8 locators.

The test statements will find two checkboxes under `div#container` and click the second one.

```
checkboxbox_elems = driver.find_elements(:xpath, "//div[@id='container']//input\
[@type='checkbox']")
checkboxbox_elems.count # => 2
checkboxbox_elems[1].click
```

Sometimes `find_element` fails due to multiple matching elements on a page, which you were not aware of. `find_elements` will come in handy to find them out.

4. Hyperlink

Hyperlinks (or links) are fundamental elements of web pages. As a matter of fact, it is hyperlinks that makes the World Wide Web possible. A sample link is provided below, along with the HTML source.

[Recommend Selenium](#)

HTML Source

```
<a href="index.html" id="recommend_selenium_link" class="nav" data-id="123" \
style="font-size: 14px;">Recommend Selenium</a>
```

Click a link by text

```
driver.find_element(:link_text, "Recommend Selenium").click
```

Click a link by ID

```
driver.find_element(:id, "recommend_selenium_link").click
```

If you are testing a web site with multiple languages, using IDs is probably the only feasible option. You do not want to write test scripts like below:

```
if is_italian?
  driver.find_element(:link_text, "Accedi").click
elsif is_chinese? # a helper function determines the locale
  driver.find_element(:link_text, "登录").click
else
  driver.find_element(:link_text, "Sign in").click
end
```

Click a link by partial text

```
driver.find_element(:partial_link_text, "partial").click  
expect(driver.text).to include("This is partial link page")
```

Click a link by XPath

The example below is finding a link with text 'Recommend Selenium' under a <p> tag.

```
driver.find_element(:xpath, "//p/a[text()='Recommend Selenium']").click()
```

You might say the example before (find by :link_text) is simpler and more intuitive, that's correct. but let's examine another example:

First div [Click here](#)
Second div [Click here](#)

On this page, there are two 'Click here' links.

HTML Source

```
<div>  
  First div  
  <a href="link-url.html">Click here</a>  
</div>  
<div>  
  Second div  
  <a href="link-partial.html">Click here</a>  
</div>
```


If a test case requires you to click the second 'Click here' link, the simple `find_element(:link_text, 'Click here')` won't work (as it clicks the first one). Here is a way to accomplish using XPath:

```
driver.find_element(:xpath, '//div[contains(text(), "Second")]/a[text()="Click here"]').click()
```

Click Nth link with exact same label

It is not uncommon that there are more than one link with exactly the same text. By default, Selenium will choose the first one. What if you want to click the second or Nth one?

The web page below contains three ‘Show Answer’ links,

1. Do you think automated testing is important and valuable? [Show Answer](#) 
2. Why didn't you do automated testing in your projects previously? [Show Answer](#)
3. Your project now has so comprehensive automated test suite, What changed? [Show Answer](#)

To click the second one,

```
driver.find_elements(:link_text => "Show Answer")[1].click # second link
```

`find_elements` return a list (also called array) of web controls matching the criteria in appearing order. Selenium (in fact Ruby) uses 0-based indexing, i.e., the first one is 0.

Verify a link present or not?

```
assert driver.find_element(:link_text, "Recommend Selenium").displayed?
assert !driver.find_element(:id, "recommend_selenium_link").displayed?
```

Verification in RSpec (known as RSpec Expectations):

```
# RSpec's expect-based syntax
expect(driver.find_element(:link_text, "Recommend Selenium").displayed?).to \
be_truthy
expect(driver.find_element(:link_text, "Recommend Selenium").displayed?).not \
_to be_falsey
```

```
# RSpec's should-based syntax
driver.find_element(:link_text, "Recommend Selenium").displayed?.should be_t\
ruthy
driver.find_element(:id, "recommend_selenium_link").displayed?.should_not be\
_falsey
```

The *expect*, *should*, *should_not*, *be_truthy* and *be_falsey* are defined in RSpec.

For more information, check out [RSpec documentation](#)¹.

Getting link data attributes

Once a web control is identified, we can get its other attributes of the element. This is generally applicable to most of the controls.

```
expect(driver.find_element(:link_text, "Recommend Selenium")["href"]).to eq(\
site_url.gsub("link.html", "index.html"))
expect(driver.find_element(:link_text, "Recommend Selenium")["id"]).to eq("r\
ecommend_selenium_link")
expect(driver.find_element(:id, "recommend_selenium_link").text).to eq("Reco\
mmend Selenium")
expect(driver.find_element(:id, "recommend_selenium_link").tag_name).to eq("\
a")
```

Also you can get the value of custom attributes of this element and its inline CSS style.

```
expect(driver.find_element(:id, "recommend_selenium_link").attribute("data-i\
d")).to eq("123")
expect(driver.find_element(:id, "recommend_selenium_link")["style"]).to eq("\
font-size: 14px;")
```

Test links open a new browser window

Clicking the link below will open the linked URL in a new browser window or tab.

```
<a href="http://testwisely.com/demo" target="_blank">Open new window</a>
```

While we could use `switch_to` method (see chapter 10) to find the new browser window, it will be easier to perform all testing within one browser window. Here is how:

¹<https://github.com/rspec/rspec-expectations>


```
current_url = driver.current_url
new_window_url = driver.find_element(:link_text, "Open new window")["href"]
driver.navigate.to(new_window_url)
# ... testing on new site
driver.find_element(:name, "name").send_keys "sometext"
driver.navigate.to(current_url) # back
```

In this test script, we use a local variable (a programming term) 'current_url' to store the current URL.