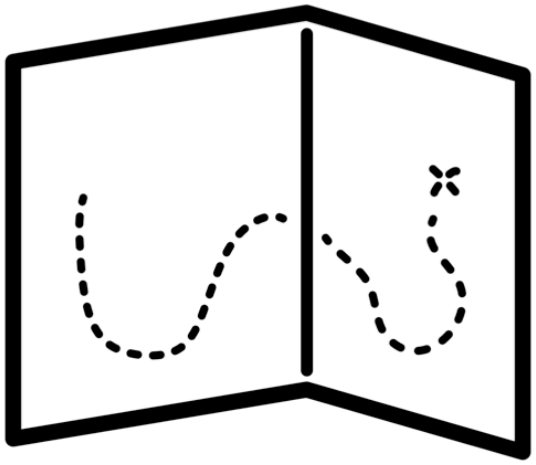
How to use Selenium, successfully



The Selenium Guidebook

Ruby Edition

by Dave Haeffner

Version 4.0.0

Preface

This book is not a full and comprehensive treatise that outlines every possible permutation of Selenium (the open-source software test automation tool for web applications). There are other books that already do this. My goal, instead, is to teach you the necessary pieces to use Selenium successfully for your circumstance.

What you have before you is a distilled and actionable guide culled from my consulting practice and full time positions held doing Quality Assurance over the past ten years.

My goal in writing this is to provide you with the materials I wish existed when I was starting out with automated acceptance testing. I hope it serves you well.

What This Book Will Cover

This book focuses on the latest stable version of Selenium 4 (a.k.a. Selenium WebDriver) and its use to test desktop browsers.

Record and Playback tools like Selenium IDE are a great option nowadays (no, really). But they will not be covered in this book. Instead, an approach of writing well factored tests, in code, is the focus of this book.

Who This Book Is For

This book is for anyone who wants to take automated acceptance testing seriously and isn't afraid to get their hands a little dirty.

That is to say, this book is for anyone who wants to use computers for what they're good at, and free you up (and potentially the people on your team) to do what they are inherently good at (which does not include repetitive, mundane testing tasks). And don't worry if you're new to programming. I'll cover the essentials so you'll have a good place to start from.

About The Examples In This Book

The examples in this book are written in Ruby, but the strategies and patterns used are applicable regardless of your technical stack.

The tests in this book are written to exercise functionality from an open-source project I created and maintain called the-internet -- available here on GitHub and viewable here on Heroku.

The tests are written for version 3.8.x of RSpec (a popular open source Behavior Driven Development Ruby testing framework).

All third-party libraries (a.k.a. "gems") are specified in a `Gemfile` and installed using Bundler with `bundle install`.

All of the code examples from the book are available in an accompanying zip file. It contains folders for each chapter where code was written or altered. Chapters with multiple parts will have multiple sub-folders (e.g., code examples referenced Chapter 9, Part 2 can be found in `09/02/` in the zip file).

How To Read This Book

Chapters 1 through 5 focus on the things you need to consider when it comes to test strategy, programming language selection, and good test design. Chapter 6 is where we first start to code. From there, the examples build upon each other through chapter 16.

Chapter 17 paints a picture of the Selenium landscape so you're better able to find information on your own.

Feedback

If you find an error in the book (e.g., grammar issue, code issue, etc.) or have questions/feedback -- please feel free to e-mail me at hello@seleniumguidebook.com.

If you submit something and I end up using it in a future version of the book I'll give you a shout-out in the Acknowledgements.

Table of Contents

Chapter 1
Selenium In A Nutshell

Selenium is a software robot sent from the future to help us test web applications. But keep in mind that it's not one of those fancy shape-shifting robots than can run really fast. It's more like one of those really strong robots that's not very fast and is best suited for accomplishing a certain objective.

That is to say -- Selenium is really good at a specific set of things. If you know what those are and stick to them then you can easily write reliable, scalable, and maintainable tests that you and your team can trust.

What Selenium Is and Is Not Good At

Selenium is built to automate browsers, specifically human interaction with them. Things like navigating to pages, clicking on elements, typing text into input fields, etc.

It's less ideal for checking lower-level functionality like HTTP status codes or HTTP headers. While you can use Selenium this way, it requires additional setup of a third-party tool (e.g., a proxy server like BrowserMob Proxy), and it is a slippery slope since there are numerous edge cases to consider at this level.

Selenium Highlights

Selenium works on every major browser, in every major programming language, and on every major operating system. Each language binding and browser are actively being developed to stay current. Yes, even Internet Explorer (thanks to Jim Evans!). And WebDriver (the thing which drivers Selenium) has become a W3C specification.

Selenium can be run on your local computer, on a remote server, on a set of servers (with Selenium Grid), or on a third-party cloud provider (like Sauce Labs). As your test suite grows, your test runs will take longer to complete. To speed them up you will want to run them in parallel, which is where the benefit of having your own servers or using a cloud provider comes in -- that, and the ability to have numerous browser and operating system combinations to run your tests on.

One of the guiding philosophies of Selenium is to be able to write your tests once and run them across multiple browsers. While this is a rosy proposition, it's not entirely accurate. There are some gotchas to watch out for when you get into it. But don't worry, We'll step through these in detail throughout the book.

Chapter 2
Defining A Test Strategy

A great way to increase your chances of success with automated web testing is to first map out a testing strategy. The best way to do it is to answer these four questions:

1. How does your business make money?
2. What features in your application are being used?
3. What browsers are your users using?
4. What things have broken in the application before?

NOTE: For organizations that don't deal directly in dollars and cents (e.g., non-profits, federal agencies, etc.) you should focus on how the application generates value for the end user and the organization.

After answering these questions you will have an understanding of the functionality and browsers that matter for the application you're testing. This will help you focus your efforts on the things that matter most.

This strategy works best for applications with existing functionality and does not speak directly to testing new functionality that's being actively developed. That's not to say that the two couldn't co-exist. It largely depends on your available resources and pace of development. But in order to reach high quality at speed, you first have to go slow.

What To Do With The Answers

After answering these questions you should end up with a prioritized punch list (a.k.a., backlog) of test automation tasks that you can work off of and track progress against.

Question 1: Money/Value

Every company's application makes money (or generates value) through core functionality that is defined by a series of increasingly-valuable user interactions -- a.k.a. a "funnel". Your answers to this question will help you determine what your funnel is.

These items will be your highest priority for automation. Start an automation backlog to keep track of them.

Question 2: Usage Data

Odds are your application offers a robust set of functionality well beyond your funnel. Your answers to this question will help highlight what it is. And if you're basing these answers on usage data (e.g., Google Analytics, etc.), then it will be broken down from highly used to lightly used.

Tack these items onto your automation backlog (below the items from question #1) based on their frequency of use.

Question 3: Browsers

Now that you know what functionality is business critical and widely adopted by your users, you need to determine what browsers to focus your automated web testing efforts on. Your usage data will tell you this as well. It will help you determine which browsers you can reasonably avoid testing in (e.g., based on non-existent or low usage numbers).

Note the top 2 (or 3 depending on your numbers), but focus on the top 1 for now. This is the browser you will start using for automation.

Question 4: Risky Bits

To round out the strategy it is also best to think about what things have broken in the application before. To answer this question it's best to check your defect/bug tracker (if you have one) and to ask your team. What you end up with may read like a laundry list of browser specific issues or functionality that has been flaky or forgotten about in the past. But it's all useful information.

Be sure to check this list against your automation backlog. If something's not there, add it to the bottom of the backlog. If it is there, make a note in the backlog item that it has been an issue in the past.

If the issue has happened numerous times and has the potential to occur again, move the item up in the backlog. And if issues keep cropping up that are related to a specific browser, compare this browser to your short list of browsers from question #3. If it's a browser that's not in your list but it's still important (e.g., a small pocket of influential users), track it on the backlog, but put it at the bottom.

Now You Are Ready

Having answered these questions, you should now have a prioritized backlog of critical business functionality, a short list of browsers to focus on, and an understanding of the risky parts of your application to watch out for. With it, you're on the right track -- focusing on things that matter for your business and its users.

Chapter 3
Picking A Language

In order to work well with Selenium you need to choose a programming language to write your automated acceptance tests in. Conventional wisdom will tell you to choose the same language that the application is written in. That way if you get stuck you can ask the developers on your team for help. But if you're not proficient in this language (or new to programming), then your progress will be slow and you'll likely end up asking for more of a developer's help than they have time for -- hindering your automation efforts and setting you up for failure.

A great way to determine which language to go with is to answer one simple question: Who will own the automated tests?

The answer to this, and the discussion that unfolds from it, will help you more effectively choose a programming language.

What To Do With The Answer

If you're a tester (or a team of testers) and you will be building and maintaining the test automation, then it's best to consider what languages you (and your team) already have experience with, or are interested in working with. Since your Selenium tests don't need to be written in the same language as the application you are testing, have a discussion with your team to see what interests them and go from there.

If you're a developer who is working on the application and just looking to add automated acceptance testing to the mix, then it makes sense to continue with the same language.

Some Additional Food For Thought

As your suite of tests grows you will find the need to add functionality to make things easier to use, run faster, offer better reporting, etc. When this happens you will effectively be doing two jobs; no longer just writing automated tests but also building and maintaining a framework (a.k.a. a test harness).

As you're considering which language to go with consider what open source frameworks already exist for the languages you are considering. Going with one can save you a lot of time and give you a host of functionality out of the box that you would otherwise have to create yourself -- and they're FREE.

You can find a list of open source Selenium WebDriver frameworks and their respective languages here.

Outro

Choosing a programming language for automated testing is not a decision that should be taken lightly. If you're just starting out then considering and discussing these things will help position you for long term success.

With all that being said, the rest of this book will show you how to build your own test harness (in addition to writing well factored tests) in Ruby with RSpec.

Chapter 4
A Programming Primer

This section will ply you with just enough programming concepts (and how they pertain to automated web testing) to get you going so that you have some working knowledge and a vocabulary that will enable you follow along with what you will see throughout this book and in your work after you put this book down.

Don't get too hung up on the details yet. If something doesn't make sense, it should once we dig into the full examples in the following chapters. Consider this more of a glossary than a code school.

Installation

Ruby is a quickly evolving ecosystem. In order to find the latest on how to install Ruby, I encourage you to check out the official Ruby site's download page.

That being said, here are some install instructions to help you get started quickly.

- Linux
- OSX
- Windows

If you plan on doing serious development with Ruby, then I would consider using a version manager instead of a direct Ruby installation (e.g., RVM or rbenv). This will enable you to run different versions of Ruby at the same time on your machine as well as have different sets of dependencies. It's crucial if you plan to work with multiple projects in Ruby that have different dependencies. But it's not required for this book.

The examples in this book have been tested in Ruby 2.2.2p95. So as long as you have a fairly recent version (at least 2.2.x) you should be good to go.

Installing Third-Party Libraries

One of the main benefits of Ruby is that it has a vibrant open source community with copious libraries (a.k.a. "gems") immediately available, making it simple to build complex things quickly. You can find out more about gems on the official Ruby site's libraries page.

To install a gem directly, you just have to type `gem install` and the gem name (from your command-line) -- e.g., `gem install selenium-webdriver`. If you get a permission error then you will need to start the command with `sudo` in order to temporary elevate your terminal session to the correct level (e.g., `sudo gem install selenium-webdriver`). When this happens, you will be

prompted for your password. Once provided, the gem will install and you'll be able to move forward.

A helpful library called Bundler is the recommended way to manage gems within a project. This is what I use and what was used in building out the examples for this book. All you need to know is that if you see a `Gemfile` in a directory, just run `bundle install` from your terminal window (assuming you have already installed the bundler gem -- `gem install bundler`). When you do, all the necessary gems (and their dependencies) will be installed for you.

If after installing gems with Bundler you have trouble executing your test code, try prepending your execution with `bundle exec` (e.g., `bundle exec rspec`).

Interactive Prompt

One of the immediate advantages to using a scripting language like Ruby is that you get access to an interactive prompt. Just type `irb` (which stands for "interactive ruby shell") from the command-line. It will load a prompt that looks like this:

```
irb(main):001:0>
```

In this prompt you can type out Ruby code. It will evaluate it and return the result. As you step through this chapter it will serve as a great place to practice the commands and concepts you will see.

When you're done, just type `quit`.

Choosing A Text Editor

In order to write Ruby code, you will need to use a text editor. Some popular ones in the Ruby community are Vim, Emacs, Sublime Text, and RubyMine.

It's important to pick an editor that works for you and makes you productive. So if you're new to programming and text editors then it's probably best to go with something more intuitive like Sublime Text.

Programming Concepts In A Nutshell

Programming can be a deep and intimidating thing if you're new to it. But don't worry. When it comes to testing there is only a handful of concepts that we really need to get started. Granted, the more you know, the better off you'll be. But you don't need to know a whole lot to be an effective test automator right now.

Of all the programming concepts out there, here are the ones you'll need to pay attention to now:

- Object Structures (Variables, Methods, and Classes)

- Scope
- Object Types (Strings, Numbers, Collections, Booleans)
- Actions (Assertions, Conditionals, Iteration)
- Inheritance

Let's step through each and how they pertain to testing with Selenium.

Object Structures

Variables

Variables are places to store and retrieve values of various types (e.g., Strings, Integers, Booleans, Collections, etc). Variables are created and then referenced by their name.

A variable name:

- can be one or more words in length
- use an underbar (`_`) to separate the words (e.g., `example_variable`)
- start with a lowercase letter
- are often entirely lowercase

You can store things in them by using an equals sign (`=`) after their name. In Ruby, a variable takes on the type of the value you store in it (more on object types later).

```
example_variable = "42"
puts example_variable.class
# outputs: String

example_variable = 42
puts example_variable.class
# outputs: Fixnum
```

In the above example `puts` is used to output a message. This is a common command that is useful for generating output to the terminal.

In Selenium, a common example of a variable is when we need to store an element (or the value from an element) in the beginning of a test to be referenced later. For example, getting a page's title.

```
page_title = @driver.title
```

`@driver` is the variable we will use to interact with Selenium. More on why it uses `@` soon (in Scope).

Methods

One way to group common actions (a.k.a. behavior) for easy reuse is to place them into methods. We define a method with the opening keyword `def`, a name (in the same fashion as a variable), and close it with the keyword `end`. Referencing a method is done the same way as a variable -- by it's name.

```
def example_method
  # your code
  # goes here
end

example_method
```

Additionally, we can specify arguments we want to pass into the method when calling it.

```
def say(message)
  puts message
end

say 'Hello World!'

# outputs:
# Hello World!
```

When setting an argument, we can also set a default value to use if no argument is provided.

```
def say(message = 'Hello World!')
  puts message
end

say
say 'something else'

# outputs:
# Hello World!
# something else
```

We'll see this tactic used in Selenium when we are telling Selenium how to wait with explicit waits (more on that in Chapter 10).

```
def wait_for(seconds=8)
  Selenium::WebDriver::Wait.new(:timeout => seconds).until { yield }
end
```

Classes

Classes are a useful way to represent concepts that will be reused numerous times in multiple places. They can contain variables and methods and are defined with the word `class` followed by the name you wish to give it.

Class names:

- must start with a capital letter
- should be CamelCase for multiple words (e.g., `class ExampleClass`)
- should be descriptive

You first have to define a class, and then create an instance of it (a.k.a. instantiation) in order to use it. Once you have an instance you can access the methods within it to trigger an action.

```
class Message
  def say(message = 'Hello World!')
    puts message
  end
end

message_instance = Message.new
message_instance.say 'This is an instance of a class'

# outputs: This is an instance of a class
```

An example of this in Selenium is the representation of a web page -- also known as a 'Page Object'. In it you will store the page's elements and behavior.

```
class Login
  LOGIN_FORM     = { id: 'login' }
  USERNAME_INPUT = { id: 'username' }
  PASSWORD_INPUT = { id: 'password' }

  def with(username, password)
...
```

The variables that are fully capitalized are called constants, the values in curly brackets (`{}`) are called hashes, and they are using something called symbols. More on all of that soon.

Scope

Now that we have our different object structures it's worth briefly covering how they can and cannot access one another. The easiest way to do this is by discussing the different types of variables you are likely to run into.

Local Variables

Local variables enable you to store and retrieve values from within a confined area (this is no different than what was described in the variables section earlier). A variable that was declared within a method is a great example of this. It is useful within the method it was declared, but inaccessible outside of it.

In your Selenium tests, a local variable will only be available from within the test that it was created.

Instance Variables

Instance variables enable you to store and retrieve values more broadly (e.g., both inside and outside of methods). They are annotated the same way as regular variables, except that they start with `@`.

A common example you will see throughout this book is the usage of `@driver`. This is an instance of Selenium stored in an instance variable. This object is what enables us to control the browser and by storing it as an instance variable our tests can easily use it.

Constants

Constants are for storing information that will not change. They are easy to spot since they start with a capital letter, and are often all uppercase. They share similarities to instance variables since they can be accessed more broadly.

They are commonly used to store element locator information at the top of Page Objects.

```
class Login

  LOGIN_FORM     = { id: 'login' }
  USERNAME_INPUT = { id: 'username' }
  PASSWORD_INPUT = { id: 'password' }
  ...
```

Environment Variables

Environment variables are a way to pass information into Ruby from the command-line. They are also a way to make a value globally accessible (e.g., across an entire program). They can be set and retrieved from within your code by:

- starting with the keyword `ENV`
- specifying the name of the variable in brackets (`ENV[]`)
- surrounding the variable name with single-quotes (`ENV['']`)
- setting a value for the variable using an equals sign (`ENV['example_variable'] =`)

- specifying a string value to store in the variable (`ENV['example_variable'] = 'example value'`)

Environment variables are often used to store configuration values that could change. A great example of this is the base URL for the application you're testing.

```
ENV['base_url'] = 'http://the-internet.herokuapp.com'
```

To change the value when running your Ruby application, you just have to specify a new value before the application.

```
base_url='http://localhost:4567' rspec login_spec.rb
```

Object Types

Strings

Strings are alpha-numeric characters packed together (e.g., letters, numbers, and most special characters) surrounded by either single (`'`) or double (`"`) quotes. Single quotes are encouraged unless you intend to manipulate the string value through a tactic called interpolation. Interpolation enables you to inject code into your string to create dynamic output.

```
motd = 'Hello World!'
puts "The message of the day is: #{motd}"

# outputs:
# The message of the day is: Hello World!
```

You'll also want to use double-quotes if you need to have a single quote in your string.

```
puts "How's this?"
```

You'll run into Strings when working with copy on a page (e.g., pulling a page's URL, title, or h1 tag to determine if your test is in the right place before proceeding).

Numbers

The two common types of numbers you will run into with testing are Fixnum (whole numbers or integers) and Float (decimals).

If you have a test that needs to pull some values from a page and add/subtract/multiply/divide them, then this will come in handy. Although you may need to convert the values from a String to an Fixnum first. But don't sweat it, this is a trivial thing to do in Ruby.

```
count = @driver.find_elements(id: '#count').text.to_i
```

The conversion from a String to an Fixnum is done with `.to_i` (the 'i' stands for Integer). If you're working with decimals, you can use `.to_f` to convert it to a Float instead.

Collections

Collections enable you to gather a set of data for later use. In Ruby there are two types of built-in collections -- Arrays and Hashes.

Array values are stored in an ordered list, each with an index number (which starts at 0), and are surrounded by square brackets (`[]`). Hashes store values in the order they were added and use a key/value pair to store and retrieve them. Hashes are surrounded by curly brackets (`{}`).

Both Arrays and Hashes are able to receive values of any type.

```
# Array Example

an_array = ["one", 2, 3.0, "four"]
puts an_array[0].class
puts an_array[1].class
puts an_array[2].class
puts an_array[3].class

# outputs:
# String
# Fixnum
# Float
# String
```

The array has four elements and the count starts at `0`. So when we access each of the values in the array we start at `0` and go till `3`.

You'll end up working with Arrays (or something similar) if you need to test things like HTML data tables, drop-down lists, or if you need to take an action against a specific element within a large list but there are no specific locators for it.

```ruby
# Hash Example

a_hash = {one: "one", two: 2, three: [3]}
puts a_hash[:one].class
puts a_hash[:two].class
puts a_hash[:three].class

# outputs:
String
Fixnum
Array
```

Note that in the hash we are storing a string, a fixnum, and an array. And we are accessing each of the values by their key. Which in this case starts with a colon (`:`). This type of object is known as a Symbol, and it is often used as an identifier for objects in Ruby.

You'll end up working with Hashes and Symbols in your Page Objects to store and retrieve your page's locators.

```ruby
class Login

  LOGIN_FORM     = { id: 'login' }
  USERNAME_INPUT = { id: 'username' }
  PASSWORD_INPUT = { id: 'password' }
  ...
```

Booleans

Booleans are binary values that are returned when asking a question of your code. They are what enable us to complete assertions.

There are numerous ways to ask questions. Some involve various comparison operators (e.g., `==` , `!=` , `<` , `>` , `<=>`), and others end in a question mark (e.g., `include?`). The response is either `true` or `false` .

```ruby
@driver.get 'http://the-internet.herokuapp.com'
@driver.title.include?('The Internet')

# returns: true
```

Actions

Assertions

Assertions are made against booleans and result in either a passing or failing test. In order to leverage assertions we will need to use a testing framework (e.g., RSpec, minitest, or test-unit). For the examples in this book we will be using RSpec (version 3.4.0).

RSpec enables easy to read assertions through it's built-in matchers. With them our assertions will start with the word `expect` and end with `.to` followed by things like `equal` (or `eql`), `include`, or `be`.

We use these matchers by calling them on the variable we want to ask a question of.

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

# or

@driver.get 'http://the-internet.herokuapp.com'
title_present? = @driver.title.include?('The Internet')
expect(title_present?).to eql true
```

Both approaches will work, resulting in a passing assertion. If this is the only assertion in your test then this will result in a passing test. More on good test writing practices in Chapter 5.

Conditionals

Conditionals work with booleans as well. They enable you execute different code paths based on their values.

The most common conditionals in Ruby are `if` and `case` statements. They both accomplish the same thing. They are just stylistically different approaches. Which approach you end up going with initially is really just a matter of preference.

```
number = 10
if number > 10
  puts 'The number is greater than 10'
elsif number < 10
  puts 'The number is less than 10'
elsif number == 10
  puts 'The number is 10'
else
  puts "I don't know what the number is."
end

# outputs: The number is 10
```

Note that in order to do an else/if statement it is `elsif`, not `elseif`.

```ruby
number = 10
case number
when 11..100
  puts 'The number is greater than 10'
when 0..9
  puts 'The number is less than 10'
when 10
  puts 'The number is 10'
else
  puts "I don't know what the number is."
end

# outputs: The number is 10
```

You can do greater-than (`>`) and less-than (`<`) comparisons in a `case` statement as well, but it ends up looking a lot like an `if` block which takes away from the simplicity of the `case` statement.

You'll end up using conditionals in your test setup code to determine which browser to load based on an environment variable. Or whether or not to run your tests locally or somewhere else.

```ruby
config.before(:each) do
    case ENV['browser']
    when 'firefox'
      @driver = Selenium::WebDriver.for :firefox
    when 'chrome'
      @driver = Selenium::WebDriver.for :chrome
    end
  end
```

Iteration

Collections wouldn't be nearly as valuable without the ability to iterate over them one at a time. And in Ruby, it's simple to do. The syntax may initially feel awkward, but once you get the hang of it it's pretty straightforward.

Each collection comes enabled with methods for iteration. The most common one is `.each`. When using it you will need to specify a variable and a block of code.

The variable will be enclosed in pipes (`||`) and represent the item of the collection that is being referenced one at a time. The variable will only be accessible within the code block. The block of code will open and close with the keywords `do` and `end`. In between the keywords is where you will put the code you want to execute (which will interact with the variable).

```ruby
numbers = [1,2,3,4,5]
numbers.each do |number|
  puts number
end

# outputs:
# 1
# 2
# 3
# 4
# 5
# 6
# 7
# 8
# 9
# 10
```

After setting numbers to an array we are able to iterate over it and display each of its values with `puts`.

We can take this approach and couple it with a conditional to influence the output.

```ruby
numbers = [1,2,3,4,5,6,7,8,9,10]
numbers.each do |number|
  puts number if number > 5
end

# outputs:
# 6
# 7
# 8
# 9
# 10
```

Iteration will come in handy in your Selenium tests if you have to loop over a collection of page elements to interact with them (e.g., HTML data tables, drop-down lists, etc.).

Inheritance

Classes have the ability to connect to one-another through parent/child inheritance. By having a single parent class we can store common actions in methods that can be readily available to all child classes.

Inheritance is done when declaring a child class by:

- providing the class name
- using a less-than symbol (`<`)
- providing the name of the parent class

```
class Parent
  def hair_color
    puts 'Brown'
  end
end

class Child < Parent
end

child = Child.new
puts child.hair_color

# outputs: Brown
```

You'll see this in your tests when writing all of the common Selenium actions you intend to use into methods in a parent class. Inheriting this class will allow you to call these methods in your child classes (more on this in Chapter 9).

Additional Resources

If you want to dive deeper into Ruby then I encourage you to check out some of the following resources:

- Learn To Program
- The Pick-axe book
- The Pragmatic Studio's Online Course

Chapter 5

Anatomy Of A Good Acceptance Test

In order to write automated web tests that are easy to maintain, perform well, and are ultimately resilient there are some simple guidelines to follow:

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

Atomic & Autonomous Tests

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

Grouping Tests

As your test suite grows you will have numerous test files. Each one containing a grouping of tests that exercise similar functionality. These test files should be grouped together in a simple directory structure as the groupings become obvious. If you want to create a test run of disparate tests, this is something that is easy to handle when using a Test Runner (covered briefly below, and in-depth in Chapter 16).

Being Descriptive

A test file should have a high level name that denotes what the tests within it are exercising. Each test should have an informative name (even if it is a bit verbose). Also, each test (or grouping of tests) should include some helpful metadata (e.g., tags, or categories) which can provide additional information about the test as well as enable flexible test execution (more on that in Chapter 16). This way all or part of your test suite can be run, and the results will be informative thanks to helpful naming.

This enables developers to run a subset of tests to exercise functionality they just modified while also enabling you to intelligently wire your test suite up to a Continuous Integration (CI) server for fast and dynamic feedback (more on CI servers in Chapter 17).

Test Runners

At the heart of every test harness is some kind of a test runner that does a lot of the heavy lifting (e.g., test execution, centralized configuration, test output, etc.). Rather than reinvent the wheel you can use one of the many test runners that exist today. With it you can bolt on third party libraries to extend its functionality if there's something missing.

Version Control

In order to effectively collaborate with other testers and developers on your team, your test code must live in a version control system. Look to see what your development team uses and add your code to it. Otherwise, set up one of the following:

- Git
- Mercurial
- Subversion

Keep in mind that your test code can live in a separate repository from the code of the application you're testing. Combining them may be advantageous but if all you're doing is writing and running tests against web endpoints (which is a majority of what your Selenium tests will be doing) then leaving your test code in a separate repository is a fine way to go.

Chapter 6
Writing Your First Test

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

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

1. Visit the page with the login form
2. Find the login form's username field and input text
3. Find the login form's password field and input text
4. Find the login form submit button and click it

Selenium is able to find and interact with elements on a page by way of various "locator strategies". The list includes `Class`, `CSS`, `ID`, `Link Text`, `Name`, `Partial Link Text`, `Tag Name`, and `XPath`.

While each serves a purpose, you only need to know a few to start writing effective tests.

How To Find Locators

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

When viewing the page, right-click on the element you want to interact with and click Inspect Element. This will bring up a small window with all of the HTML for the page but zoomed into your highlighted selection. From here you can see if there are unique or descriptive attributes you can work with.

How To Find Quality Elements

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

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

element you want to use.

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

An Example

Part 1: Find The Elements And Write The Test

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

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

Notice the unique elements on the form. The username input field has a unique `id` (e.g., `<input type="text" name="username" id="username">`), as does the password input field (e.g., `<input type="password" name="password" id="password">`). The submit button doesn't have an `id`, but it's the only button on the page.

Let's put these elements to use in our first test (or 'spec' as it's called in RSpec). First, let's make sure we have the correct versions of `rspec` and `selenium-webdriver` installed.

```
# filename: Gemfile
source 'https://rubygems.org'

gem 'rspec', '~>3.8.0'
gem 'selenium-webdriver', '~>4.0.0.alpha.2'
```

After updating our Gemfile we can install the specified libraries with `bundle install` in the command-prompt.

Now let's create a folder for our tests called `spec` a new file called `login_spec.rb` in it. We'll also need to create a `vendor` directory for third-party files and download `geckodriver` into it. Grab the latest release for your operating system from here and unpack its contents into the `vendor` directory. This is a required file (known as a browser driver) in order to make Selenium work with Firefox. See Chapter 12 for more detail about browser drivers.

When we're done our directory structure should look like this.

```
Gemfile
spec
   login_spec.rb
vendor
    geckodriver
```

Here is the code we will add to the test file for our Selenium commands, locators, etc.

```ruby
# filename: spec/login_spec.rb

require 'selenium-webdriver'

describe 'Login' do

  before(:each) do
    driver_path = File.join(Dir.pwd, 'vendor', 'geckodriver')
    if File.file? driver_path
      service = Selenium::WebDriver::Service.firefox(path: driver_path)
      @driver = Selenium::WebDriver.for :firefox, service: service
    else
      @driver = Selenium::WebDriver.for :firefox
    end
  end

  after(:each) do
    @driver.quit
  end

  it 'succeeded' do
    @driver.get 'http://the-internet.herokuapp.com/login'
    @driver.find_element(id: 'username').send_keys('tomsmith')
    @driver.find_element(id: 'password').send_keys('SuperSecretPassword!')
    @driver.find_element(css: 'button').submit
  end

end
```

At the top of the file we are pulling in `selenium-webdriver` which is the official open-source library for Selenium. It gives us the bindings necessary to drive the browser. Next is the `describe`, which is the title of this test suite (or test class). A simple and helpful name is chosen to note the suite's intent (e.g., 'Login'). An opening `do` and a closing `end` (placed at the end of the file) are required to start and finish the suite.

In order to use Selenium we need to create an instance of it. This is done with the command `Selenium::WebDriver.for :firefox`, which is made available to us through the `selenium-webdriver` library we `require` at the top of the file. In order for this to work we need to provide the path to the `geckodriver` file, which we do by finding and storing it in a local variable called `geckodriver` and passing it in as part of our Selenium instantiation with a service object (e.g., `Selenium::WebDriver.for :firefox, service: service`). Alternatively, you could have `geckodriver` listed on your system path. If that's the case, then we also want to respect that option, so we check to see if `geckodriver` is a file in the `vendor` directory. If not, then the standard instantiation for Firefox is used (without specifying a service object).

After our test we don't want the browser we opened to stick around, so we close it. This is what we're doing in `before(:each)` and `after(:each)`. Notice that we are storing our instance of Selenium in an instance variable (`@driver`) so we can access it throughout our entire test suite.

Last up is the `it` block, which is the test. Similar to `describe` we can give it a simple and helpful name (e.g., 'succeeded'). In the test we are using the locators we identified from the markup by finding them with `@driver.find_element`, passing in the locator type as a symbol (e.g., `id:`) and the locator value as a string (e.g., `username`, `password`, etc.). Once found, we take an action against each of them, which in this case is inputting text (with `.send_keys`) and click the submit button (with `.click`).

If we run this (e.g., `rspec` from the command-line), it will run and pass. But there's one thing missing -- an assertion. Without it, there's no way to tell if the end state of our page is correct. We need to see what the page looks like after logging in, look through the markup, and find an element to use in our assertion.

Part 2: Figure Out What To Assert

Here is the markup that renders on the page after submitting the login form:

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

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

After logging in, there looks to be a couple of things we can use for our assertion. There's the flash message class (most appealing), the logout button (appealing), or the copy from the h2 or the flash message (least appealing). Since the flash message class name is descriptive, denotes success, and is less likely to change than the copy, let's go with that.

But whenever we try to access a class element with two words in it (e.g., `class="flash success"`) we will need to combine them together into a single locator string that is written in either a CSS

selector or XPath. Either approach works well, but the examples throughout this book will focus on how to use CSS selectors (when not using ID or Class locators).

A Quick Primer on CSS Selectors

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

CSS selectors are a pretty straightforward and handy way to write locators, especially for hard to reach elements.

For right now, here's what you need to know. In CSS, class names start with a dot (`.`). For classes with multiple words, put a dot in front of each word, and remove the spaces (e.g., `.flash.success` for `class='flash success'`).

For a good resource on CSS Selectors, I encourage you to check out Sauce Labs' write up on them.

Part 3: Write The Assertion And Verify It

```
# filename: spec/login_spec.rb
# ...
  it 'succeeded' do
    @driver.get 'http://the-internet.herokuapp.com/login'
    @driver.find_element(id: 'username').send_keys('username')
    @driver.find_element(id: 'password').send_keys('password')
    @driver.find_element(css: 'button').click
    expect(@driver.find_element(css: '.flash.success').displayed?).to be_truthy
  end

end
```

Now our test has an assertion! But there's a lot going on in that one line, let's step through it to make sure we understand what it's doing.

First, we are finding the element we want to make an assertion against (using the locator type `css` and the locator string `.flash.success`). After that, we are asking Selenium if this element is displayed on the page (with `.displayed?`). This returns a boolean response that we can make an assertion against. To make the assertion we leverage an RSpec assertion method (`expect`) and an RSpec matcher (`to`) followed by the expected result `be_truthy` .

After logging in if the success flash message is displayed then Selenium will return `true` and the test will pass.

Just To Make Sure

Now when we run this test (`rspec login_spec.rb` from the command-line) it will pass just like before, but now there is an assertion which should catch a failure if something is amiss.

Just to make certain that this test is doing what we think it should, let's change the assertion to force a failure and run it again. A simple fudging of the locator will suffice.

```
expect(@driver.find_element(css: '.flash.successasdf').displayed?).to be_truthy
```

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

Want the rest of the book?

Buy your copy HERE