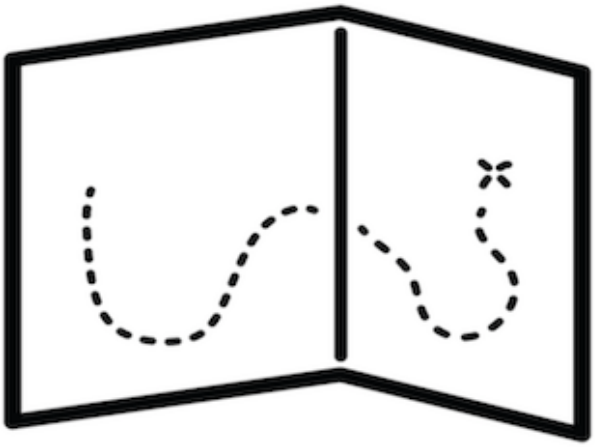
How to use Selenium, successfully



The Selenium Guidebook

by Dave Haeffner

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 bits 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 last five 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 version of Selenium (a.k.a. Selenium WebDriver) and its use to test desktop browsers. It's predecessor, Selenium RC, will be covered only insofar as to help you upgrade from it. Mobile testing with Appium will also briefly be covered.

Record and Playback tools like Selenium IDE and Selenium Builder are great for getting started, but abysmal for growing past that point. To that end, they will not be covered in this book. An approach on writing well factored tests, in code, will be the focus instead.

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 are good at, and free up you and 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 super new to programming, I'll cover the basics so you'll have a good platform to build from.

About The Examples In This Book

The examples in this book are written in Ruby. This is partly because that is the language in which I am strongest. But really, it has more to do with how readable and approachable the language is. By sticking with a scripting language that reads a lot like English, this book can be picked up by almost anyone regardless of their technical background and be immediately useful. And even if Ruby isn't your preferred language, the strategies and patterns used are applicable regardless of your tech stack.

The web functionality used to write tests against is from an open source project called the-internet -- available on GitHub and viewable through a Heroku site.

And the test examples are written to run against the RSpec test runner, with Bundler managing the third party libraries (a.k.a. gems).

How To Read This Book

Chapters 1 through 5 are focused on priming you with knowledge and questions to consider when it comes to test strategy, language selection, and test design. Chapter 6 is where we first start to code. From there, the examples build upon each other through chapter 16.

Chapter 17 is an accumulation of 13 sub-chapters which outline things you will likely see and how to address them. They do not build upon each other or other parts of the book, and are each meant to stand alone.

All of the code used is available in the `code_examples` folder provided within your zip file. Chapters with multiple parts will have multiple folders. Each part (as noted in the chapter) will correspond to a numbered folder (e.g., code examples referenced in Part 2 of Chapter 9 can be found in `code_examples/09_base_page_object/02/`).

Feedback

If you find a glitch in the book (e.g., grammar issue, code issue, etc.) or have questions/feedback -- please feel free to e-mail me at dhaeffner@gmail.com.

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

Acknowledgements

I'd like to take a moment to thank the following people (sorted alphabetically):

- Katy Burns
- Brian Goad
- Jonathan Lipps
- Mary Ann May-Pumphrey
- Ken Pier

They provided tremendous feedback that helped influence my thinking and ultimately transformed the book into something far greater than what it was before.

Katy, I somehow convinced you that doing one-on-one sessions with me over Google Hangout an hour a week would be a great way to learn more about Selenium. Over the course of that time an outline for the book emerged, and with your feedback and insights I was able to shape it into something useful and compelling. Without your help, I'm not sure how far along the book would be or if it would be as useful.

Brian, Ken, Jonathan, and Mary Ann -- your feedback was thorough and challenging. You asked tough questions and provided great suggestions for improvements. A good deal of the book has been rewritten based on your feedback. Examples have been refined. Concepts have been made more concise. And generally, there is a better build-up and flow throughout the book.

THANK YOU!

Table of Contents

1. Selenium In A Nutshell
2. Defining A Test Strategy
3. Picking A Language
4. A Programming Primer
5. Anatomy Of A Good Acceptance Test
6. Writing Your First Test
7. Verifying Your Locators
8. Writing Re-usable Test Code
9. Writing Really Re-usable Test Code
10. Writing Resilient Test Code
11. Prepping For Use
12. Running A Different Browser Locally
13. Running Browsers In The Cloud
14. Speeding Up Your Test Runs
15. Organizing Your Test Runs
16. Automating Your Test Runs
17. Things You will See And How To Work With Them
    1. A/B Testing
    2. Basic Auth
    3. Dropdown Lists
    4. File Download (Browser Specific)
    5. File Download (browser agnostic)
    6. File Upload
    7. Frames
    8. HTTP Status Codes
    9. Mobile
    10. Multiple Windows
    11. Secure File Downloads
    12. Tables
    13. Upgrading From Selenium RC
18. Finding Information On Your Own
19. Now You Are Ready

Chapter 1

Selenium In A Nutshell

What Selenium Is and Is Not Good At

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. Instead 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 will be able to easily write reliable, scalable, and maintainable tests that you and your team can trust.

Selenium is built to automate browsers, specifically human interaction with them -- things like navigating to pages, clicking, typing, dragging, and dropping.

It's less ideal for checking lower-level functionality like HTTP status codes or HTTP headers. While you can use Selenium this way, it necessitates additional setup of a third-party tool (e.g., BrowserMob Proxy), and it can be a bit of a rabbit hole to fall down since there are numerous test cases to consider at this level.

Selenium Highlights

Selenium works on every major browser, with a number of major programming languages, 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!).

Selenium can be run on your local computer, on a server (with Selenium Remote), on your own 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.

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 picture, it's not entirely accurate. There are plenty of gotchas to watch out for. We'll cover these in detail throughout the book.

Chapter 2
Defining A Test Strategy

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

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

Note: for entities 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.

After answering these questions, you should have an understanding of the functionality and browsers that matter to the application you are testing. This will help you narrow down your efforts to 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 you should end up with a prioritized 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. the 'funnel'. Your answers to this question will help you determine what your funnel is.

This will be your highest priority for automation. Start an automation backlog to keep track.

Question 2: Usage Data

Odds are your application offers a robust set of functionality well beyond the funnel. Your answers to this question will help highlight what it is. And if you're basing these answers on usage data, then it will be broken down from highly used to lightly used. Tack these items onto your automation backlog 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 around 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.

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

If the issue has happened numerous times and has the potential to occur again, move that backlog item up in priority. And if issues keep cropping up that are related to a specific browser, compare this browser to your short list 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 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 should be able to make sure you are on the right track -- focusing on things that matter for the 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 acceptance tests in. Conventional wisdom will tell you to choose the same language as what 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 development), then your progress will be slow and you'll likely end up asking for more developer 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 gaining experience with, rather than what the application is built in. Your Selenium tests DO NOT have 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.

For example, there's not much advantage in writing your test automation in Java if your team has no previous development experience and will be the ones owning it. Instead, try to choose a language that is more approachable -- e.g., a scripting language like Ruby or Python.

But 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 growing a framework (a.k.a. a test harness).

As you are considering which language to go with, consider what open source frameworks already exist for the languages you are eyeing. Going with one will save you a lot of time and give you a host of functionality out of the box that you would otherwise have to build and maintain yourself -- and it's FREE.

Here's a short list of open source Selenium WebDriver frameworks and their respective languages (sorted alphabetically):

- ChemistryKit (Ruby)
- Geb (Groovy)
- Robot Framework
- Saunter (Python and PHP)
- wd (JavaScript)

Choosing a programming language for automated testing is not a decision that should be taken lightly. If you're just starting out (or looking to port your tests) then considering and discussing these things will help position you for greater success.

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

Chapter 4
A Programming Primer

This section will prime you with just enough programming concepts (and how they pertain to testing) to get you going so that you have some working knowledge and a vocabulary that will enable you to more effectively comprehend what you will see throughout the remainder of 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.

- OSX
- Windows
- Linux

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 gems. 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.

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 extremely simple to quickly build complex things. You can find out more information on gems on the official Ruby site's libraries page.

To install a gem, you just have to type `gem install` and the gem name (from your command-line) -- e.g., `gem install selenium-webdriver`.

A helpful library called Bundler is a great way to easily 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.

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, and Sublime Text.

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 rabbit hole if you're new to it. But don't worry. When it comes to testing there is only a small subset of Ruby that we really need to work with 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 first:

- Objects 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 referenced with names that:

- 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 place things into them by using an equals sign (`=`). And 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 storing 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 and 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.

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

example_method
```

Additionally, we can specify arguments we want to pass in when calling the method. When setting an argument, we can also set a default value to use if no argument is provided.

```ruby
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 setting up our explicit wait action (more on that in Chapter 10).

```ruby
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 appear 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 this instance, you can access the methods within it to trigger an action.

```ruby
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 store the page's elements and behavior.

```ruby
class Login do
  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 braces (`{}`) are called hashes, and they are using something called symbols. More on all that soon.

Scope

Now that we have our different object structures it's worth briefly noting how they can and cannot access one another -- a.k.a. Scope. 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, 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 -- both inside and outside

of methods. They are annotated the same way as regular variables, except that they are prepended with an `@`.

A common example you will see throughout this book is the usage of a `@driver` variable. This is an instance of Selenium. This object is what enables us to control the browser. Storing it as an instance variable enables our test to more easily use it.

Constants

Constants are for storing information that will not dynamically 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 do

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

Global Variables

Global variables are a way to make information available to an entire program. They start with a dollar-sign (`$`) and are created the same way as regular variables.

```
$example_global_variable = 'value you want to make widely available'
```

They are powerful since they are so widely accessible and can be of any type. But they should be used sparingly because there are more intuitive and effective ways to construct your programs.

You will see them used in your tests for rare one-off circumstances when you need to access something that is not available within your current scope and an environment variable has not been set for it either.

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 value to store in the variable with single quotes (`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 your application.

```
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' ruby example_script.rb
```

Object Types

Strings

Strings are alpha-numeric characters strung 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 -- which 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

You can do a lot of math in Ruby. 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/etc them, then this will come in handy. Although, you may need to

convert the values from a String to an Fixnum first -- which is trivial to do in Ruby since it is a dynamically typed language.

```
count = @driver.find_elements(css: '#count').text.to_i
# do your comparison or maths with count here
```

The conversion from a String to an Fixnum was 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 listed, each with an index number (which starts at 0), and are surrounded by square brackets (`[]`). Hashes store values in the order they were added -- but use a key/value pair to store and retrieve data. Hashes are surrounded by braces (`{}`).

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]
puts an_array[1]
puts an_array[2]
puts an_array[3]

# outputs:
# one
# 2
# 3.0
# four
```

Note that in the array we are storing two strings, a fixnum, and a float. And we are accessing each of the values by their index.

You'll end up working with Arrays (or some type of object that is of a similar construct) 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]
puts a_hash[:two]
puts a_hash[:three].class

# outputs:
one
2
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 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 do

  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')

# outputs: 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 this book we will be using RSpec.

RSpec enables easy to read assertions through it's built-in matchers. With them, our assertions will start with things like `should`, `should_not`, `should be`, `should include`.

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

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

# or

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

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

Conditionals

Conditionals work off of booleans as well and enable you to accomplish different things based on context.

The most common conditionals in Ruby are `if` and `case` statements. They both accomplish the same thing. They are just stylistically different approaches (with some minute performance differences). 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 and 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
```

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

You may end up working with Conditionals in your Page Objects. This is more likely if your application has tiered user accounts or if you support older browsers and run into issues with the locators you are using.

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 is pretty straightforward.

Each collection comes with a set of methods that enables various iterative functionality. The most common one is `.each`. When using it you will need to specify a block of code and a variable.

The variable will be enclosed in pipes (`||`). It represents each piece of the collection and is accessible only within the code block. The code block 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 = [1,2,3,4,5,6,7,8,9,10]
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

```ruby
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:

- Ruby Monk
- Codecademy
- 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 acceptance tests that perform well and are both maintainable and resilient there are some simple guidelines to follow:

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

Grouping Tests

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

Being Descriptive

Each test file should be named appropriately, and each test within it should have an informative name (even if it may be a bit verbose). Also, each test (or grouping of tests) should be tagged with some helpful information to provide context and enable more flexible test execution (more on Tagging in Chapter 16).

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

This approach is useful for both humans and robots alike. It enables developers to run a subset of tests to exercise functionality they just modified (as part of their pre-check-in testing) while also enabling you to intelligently wire your suite up to a Continuous Integration server (more on them in Chapter 16) for fast and dynamic feedback.

Test Runners

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

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

Version Control

In order to effectively collaborate with other testers and developers, your test code must live in a version control system of some sort. 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 testing will be with Selenium) 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 main page of a site
2. Find the login button and click it
3. Find the login form's username field and input text
4. Find the login form's password field and input text
5. Find the login form 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, popular 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>
```

Note the unique elements on the form. The username input field has a unique `id`, as does the password input field. The submit button doesn't, but the parent element (`form`) does. So instead of clicking the submit button, we will have to submit the form instead.

Let's put these elements to use in our first test (or 'spec' as it's called in RSpec).

NOTE: Elements written in a test are called 'locators' and will be referenced as such.

```ruby
# filename: login_spec.rb

require 'selenium-webdriver'

describe 'Login' do

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

  after(:each) do
    @driver.quit
  end

  it '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(id: 'login').submit
  end

end
```

There are some new keywords in this code example that may not look familiar. Things like `describe`, `before(:each)`, `after(:each)`, and `it`. They are part of RSpec's syntax. Let's step through this piece by piece. After that, it should all make sense.

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. A simple and helpful name is chosen to note the suite's intent (e.g., 'Login'). And a closing `end` is required, which is placed at the end of the file.

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

Last up is the `it` block, which is the test. Similar to the `describe` block, 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 key (e.g., `id:`) and the locator value (e.g.,

`username`, `password`, and `login`). Once found, we take an action against each of them, which in this case is inputting text (with `.send_keys`) and submitting the form (with `.submit`).

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

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 key off of 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 an element with multiple locators like this we will need to combine them together into a single locator string that is written in either CSS or XPath. Either approach works well, but the examples throughout this book will focus on how to use CSS.

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, sending keys, 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

```
require 'selenium-webdriver'

describe 'Login' do

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

  after(:each) do
    @driver.quit
  end

  it 'successful' do
    @driver.get 'http://the-internet.herokuapp.com/login'
    @driver.find_element(id: 'username').send_keys('username')
    @driver.find_element(id: 'password').send_keys('password')
    @driver.find_element(id: 'login').submit
    @driver.find_element(css: '.flash.success').displayed?.should be_true
  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 (`.should`) and an RSpec matcher (`be_true`).

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.

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

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

Chapter 7
Verifying Your Locators

If you're fortunate enough to be working with unique IDs and Classes, then you're usually all set. But when you have to handle more complex actions like traversing a page -- or you need to run down odd test behavior -- it can be a real challenge to verify that you have the right locators to accomplish what you want.

Instead of the painful and tedious process of trying out various locators in your tests until you get what you're looking for, give Firefinder a try.

Firefinder is an add-on to the popular web-development Firefox tool Firebug.

You first need to install Firebug, then Firefinder. Once you have that, verifying locators is a trivial task. And it works for both CSS and XPath locators.

NOTE: An alternative to FireFinder is FirePath. If you have problems with FireFinder, or if you just want to see which one works better for you, then give it a try as well.

An Example

Let's try to identify the locators necessary to traverse a few levels into a large set of nested divs.

```
# a snippet from http://the-internet.herokuapp.com/large

<div id='siblings'>
  <div id='sibling-1.1'>1.1
  <div id='sibling-1.2'>1.2</div>
  <div id='sibling-1.3'>1.3</div>
    <div id='sibling-2.1'>2.1
    <div id='sibling-2.2'>2.2</div>
    <div id='sibling-2.2'>2.3</div>
      <div id='sibling-3.1'>3.1
      <div id='sibling-3.2'>3.2</div>
      <div id='sibling-3.2'>3.3</div>
        <div id='sibling-3.1'>4.1
        <div id='sibling-3.2'>4.2</div>
        <div id='sibling-3.2'>4.3</div>
        ...
```

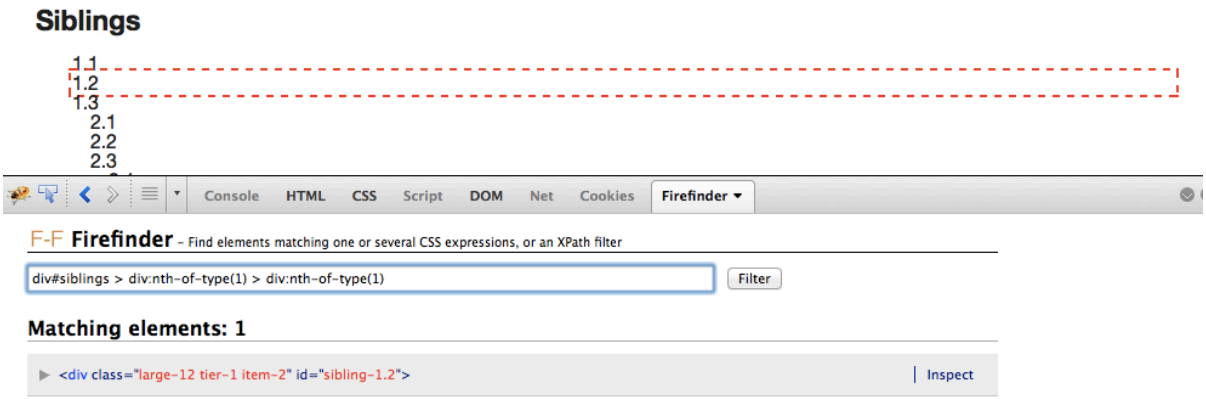
If we were to perform a `find_element` action using the following locator, it will work.

```
@driver.find_element(css: "div#siblings > div:nth-of-type(1) > div:nth-of-type(1)")
```

But if we try to go one level deeper, it won't work.

```
@driver.find_element(css: "div#siblings > div:nth-of-type(1) > div:nth-of-type(1) >
div:nth-of-type(1)")
```

Fortunately with Firefinder we can actually see what our locators are doing. Here's what it shows us for the locators that "worked".

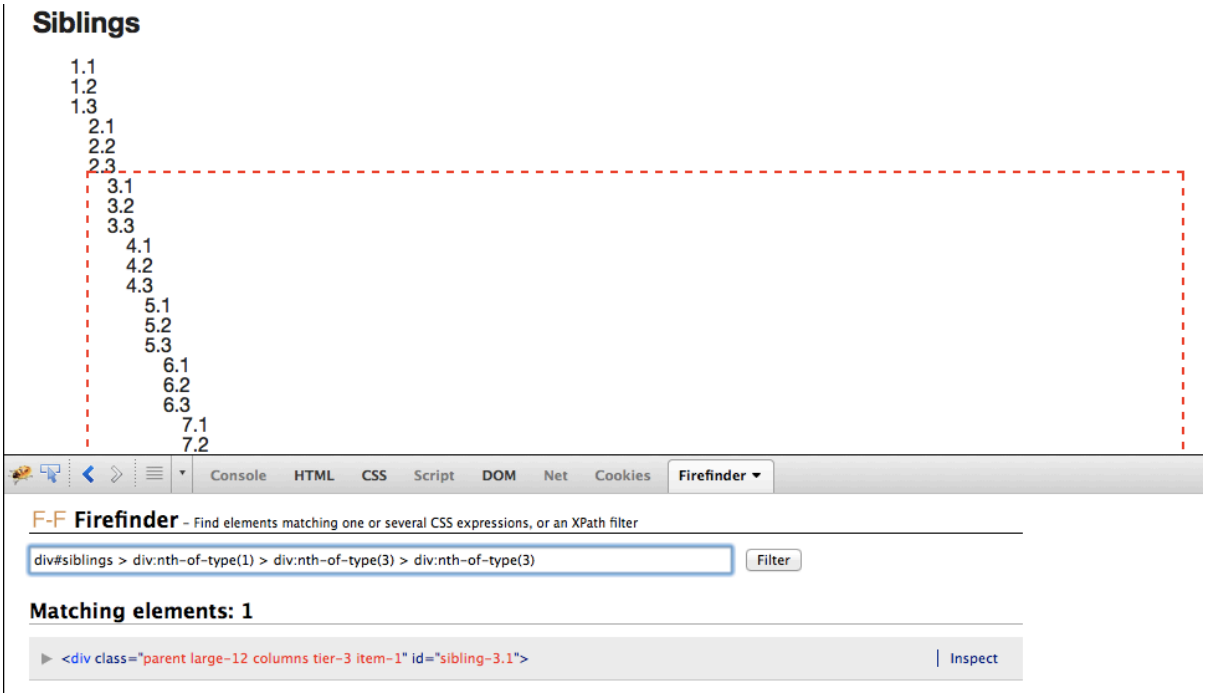


It looks like our locators are scoping to the wrong part of the first level (1.2). We need to reference the third part of each level (e.g., 1.3, 2.3, 3.3) in order to traverse deeper since the nested divs live under the third part of each level.

So if we try this locator instead, it should work.

```
@driver.find_element(css: "div#siblings > div:nth-of-type(1) > div:nth-of-type(3) >
div:nth-of-type(3)")
```

And we can confirm this before changing any test code by looking in Firefinder.



This should help save you time and frustration when running down tricky locators in your tests. It definitely has for me.

Chapter 8
Writing Re-usable Test Code

One of the biggest challenges with Selenium tests is that they can be brittle and challenging to maintain over time. This is largely due to the fact that things in the app you're testing change, breaking your tests.

But the reality of a software project is that change is a constant. So we need to account for this reality somehow in our test code in order to be successful.

Enter Page Objects.

Rather than write your test code directly against your app, you can model the behavior of your application into simple objects -- and write your tests against them instead. That way when your app changes and your tests break, you only have to update your test code in one place to fix it.

And with this approach, we not only get the benefit of controlled chaos, we also get the benefit of reusable functionality across our tests.

An Example

Part 1: Create A Page Object And Update Test

Let's take our login example from the previous chapter and pull it out into a page object and update our test accordingly.

```ruby
# filename: login.rb

class Login

  LOGIN_FORM  = { id: 'login' }
    USERNAME_INPUT = { id: 'username' }
    PASSWORD_INPUT = { id: 'password' }
  SUCCESS_MESSAGE = { css: '.flash.success' }

  def initialize(driver)
    @driver = driver
    @driver.get ENV['base_url'] + '/login'
  end

  def with(username, password)
    @driver.find_element(USERNAME_INPUT).send_keys(username)
    @driver.find_element(PASSWORD_INPUT).send_keys(password)
    @driver.find_element(LOGIN_FORM).submit
  end

  def success_message_present?
    @driver.find_element(SUCCESS_MESSAGE).displayed?
  end

end
```

We start by creating our own class, naming it accordingly (`Login`), and storing our locators in hashes within constants along the top.

We then use an `initialize` method. This is a built in method for classes in Ruby that enable you to automatically execute code upon instantiating the class. With it we are taking an argument to receive the Selenium driver object -- and storing it in an instance variable. This will enable the class to drive the browser. We're also visiting the page with the help of an environment variable (instead of a hard-coded value).

In our `with` method we are capturing the core functionality of the login page by accepting the username and password input values as well as housing the input and submit actions.

Since our behavior now lives in a page object, we want a clean way to make an assertion in our test. This is where `success_message_present?` comes in. Notice that it ends with a question mark. Methods that end like this are known as mutator methods. And when they end with a question mark, they imply that they will return a boolean. So in it we want to ask a question of the page so that we can return a boolean. And in our test we will put this boolean to work by making an assertion against it.

Now let's update our test to use this page object.

```ruby
# filename: login_spec.rb

require 'selenium-webdriver'
require_relative 'login'

describe 'Login' do

  before(:each) do
    @driver = Selenium::WebDriver.for :firefox
    ENV['base_url'] = 'http://the-internet.herokuapp.com'
    @login = Login.new(@driver)
  end

  after(:each) do
    @driver.quit
  end

  it 'succeeded' do
    @login.with('tomsmith', 'SuperSecretPassword!')
    @login.success_message_present?.should be_true
  end

end
```

At the top of the file we include the page object by using `require_relative`. This enables us to reference another file based on the current file's path. We then create an environment variable for the base_url (`ENV['base_url']`). Since we only have one test file, it will live here and be hard-coded for now. But don't worry, we'll address this later.

Next we instantiate our login page object in `before(:each)`, passing in `@driver` as an argument, and storing it all in an instance variable (`@login`). We then wire up our test to use the actions available in `@login`.

We finish things up by making an assertion against our helper method (`.success_message_present?.should be_true`).

Part 2: Write Another Test

This may feel like more work than what we had when we first started. But we're in a much sturdier position and able to write follow-on tests more easily. Let's add another test to demonstrate a failed login.

If we provide incorrect credentials, the following markup gets rendered on the page.

```
<div id="flash-messages" class="large-12 columns">
  <div data-alert="" id="flash" class="flash error">
    Your username is invalid!
    <a href="#" class="close">x</a>
  </div>
</div>
```

This is similar to the markup from our successful flash message, so let's mimic the behavior we used in our page object to create another mutator method to help in our assertion.

First we'll add a new locator for the failure message in our list of constants at the top of our class (just below our success message locator).

```
# filename: login.rb

class Login

  LOGIN_FORM  = { id: 'login' }
    USERNAME_INPUT = { id: 'username' }
    PASSWORD_INPUT = { id: 'password' }
  SUCCESS_MESSAGE = { css: '.flash.success' }
  FAILURE_MESSAGE = { css: '.flash.error' }
  ...
```

Further down the file (next to the existing mutator method) we'll add a new method to check for the existence of this message and return a boolean response.

```
def success_message_present?
  driver.find_element(:css, SUCCESS_MESSAGE).displayed?
end

def failure_message_present?
  driver.find_element(:css, FAILURE_MESSAGE).displayed?
end
```

Lastly, we add a new test in our spec file just below our existing one, specifying invalid credentials to force a failure.

```
it 'succeeded' do
  @login.with('tomsmith', 'SuperSecretPassword!')
  @login.success_message_present?.should be_true
end

it 'failed' do
  @login.with('asdf', 'asdf')
  @login.failure_message_present?.should be_true
end
```

Now if we run our spec file (`rspec login_spec.rb`) we will see two browser windows open (one after the other) testing both the successful and failure login conditions.

Why Asserting False Won't Work (yet)

You may be wondering why we didn't check to see if the success message wasn't present.

```
it 'failed' do
  @login.with('tomsmith', 'SuperSecretPassword!')
  @login.success_message_present?.should be_false
end
```

There are two problems with this approach. First, our test will fail. This is because it errors out when looking looking for an element that's not present -- which looks like this:

```
.F

Failures:

  1) Login failed
     Failure/Error: @login.success_message_present?.should be_false
     Selenium::WebDriver::Error::NoSuchElementError:
       Unable to locate element: {"method":"css selector","selector":".flash.success"}
     # [remote server]
file:///tmp/webdriver-profile20140125-21003-3brprw/extensions/fxdriver@googlecode.com/comp
`FirefoxDriver.prototype.findElementInternal_'
     # [remote server]
file:///tmp/webdriver-profile20140125-21003-3brprw/extensions/fxdriver@googlecode.com/comp
`FirefoxDriver.prototype.findElement'
     # [remote server]
file:///tmp/webdriver-profile20140125-21003-3brprw/extensions/fxdriver@googlecode.com/comp
`DelayedCommand.prototype.executeInternal_/h'
     # [remote server]
file:///tmp/webdriver-profile20140125-21003-3brprw/extensions/fxdriver@googlecode.com/comp
`DelayedCommand.prototype.executeInternal_'
     # [remote server]
file:///tmp/webdriver-profile20140125-21003-3brprw/extensions/fxdriver@googlecode.com/comp
`DelayedCommand.prototype.execute/<'
     # ./code_examples/07/02/login.rb:21:in `success_message_present?'
     # ./code_examples/07/02/login_spec.rb:23:in `block (2 levels) in <top (required)>'

Finished in 12.08 seconds
2 examples, 1 failure
```

But don't worry, we'll address this limitation in the next chapter.

Second, the absence of success message does not necessarily indicate a failed login. The assertion we ended up with is more concise.

Part 3: Confirm We're In The Right Place

Before we can call our page object finished, there's one more addition we'll want to make. We'll want to add an assertion to make sure that Selenium is in the right place before proceeding. This will help add some initial resiliency to our test.

As a rule, you want to keep assertions in your tests and out of your page objects. But this is the exception to the rule.

```
class Login

  LOGIN_FORM  = { id: 'login' }
    USERNAME_INPUT = { id: 'username' }
    PASSWORD_INPUT = { id: 'password' }
  SUCCESS_MESSAGE = { css: '.flash.success' }
  FAILURE_MESSAGE = { css: '.flash.error' }

  def initialize(driver)
    @driver = driver
    @driver.get ENV['base_url'] + '/login'
    @driver.find_element(LOGIN_FORM).displayed?.should == true
  end
  ...
```

We simply add a new line to the end of our `initialize` method. In it we are checking to see if the login form is displayed, and making an assertion against the boolean response returned from Selenium.

The only unfortunate part of doing an assertion in the page object is that we don't have access to RSpec's matchers (e.g., `be_true`). Instead we use a Ruby comparison operator to see if the boolean equals true (`== true`).

Now if we run our tests again, they should pass just like before. But now we can rest assured that the test will only proceed if the login form is present.

Chapter 9
Writing Really Re-usable Test Code

In the previous chapter we stepped through creating a simple page object to capture the behavior of the page we were interacting with. While this was a good start, it leaves a lot of room for improvement.

As our test suite grows and we add more page objects, we will start to see common behavior that we will want to use over and over again. If we leave this unchecked we will end up with duplicative code which will slowly make our test suite harder to maintain. And right now we are using Selenium actions directly in our page object. While on the face of it this may seem fine, it has some long term impacts, like:

- the inability to drive your tests with a different driver (e.g., mobile)
- test maintenance issues when Selenium actions change (e.g., major upgrades ala RC to WebDriver)
- slower test writing due to the lack of a simple Domain Specific Language (DSL)

With a Base Page Object (a.k.a. a facade layer) we can easily side step these concerns by abstracting all of our common actions into a central location and leverage them in our page objects.

Let's step through an example with our login page object.

An Example

Part 1: Create The Base Page Object

First we will need to create the base page object.

```ruby
# filename: base_page.rb

require 'selenium-webdriver'

class BasePage

  def initialize(driver)
    @driver = driver
  end

  def visit(url_path)
    @driver.get ENV['base_url'] + url_path
  end

  def find(locator)
    @driver.find_element locator
  end

  def type(text, locator)
    find(locator).send_keys text
  end

  def submit(locator)
    find(locator).submit
  end

  def is_displayed?(locator)
    find(locator).displayed?
  end

end
```

We first `require` our Selenium library (`selenium-webdriver`). By placing it in this file, we will no longer need to include it anywhere else in our test suite (assuming all of our tests leverage page objects). And it makes sense to have it live here since this is the only file where we will be referencing Selenium commands directly.

Next we declare our class and name it appropriately with camel-casing (`BasePage`).

In our `initialize` method we are accepting a driver object as an argument and setting it as an instance variable to make it available throughout this class. We then add common methods that we will reference in our page objects -- `visit` , `find` , `type` , `submit` , and `is_displayed?` .

Now let's update our login page object to leverage this.

```ruby
# filename: login.rb

require_relative 'base_page'

class Login < BasePage

  LOGIN_FORM  = { id: 'login' }
    USERNAME_INPUT = { id: 'username' }
    PASSWORD_INPUT = { id: 'password' }
  SUCCESS_MESSAGE = { css: '.flash.success' }
  FAILURE_MESSAGE = { css: '.flash.error' }

  def initialize(driver)
    super
    visit '/login'
    is_displayed?(LOGIN_FORM).should == true
  end

  def with(username, password)
    type username, USERNAME_INPUT
    type password, PASSWORD_INPUT
    submit LOGIN_FORM
  end

  def success_message_present?
    is_displayed? SUCCESS_MESSAGE
  end

  def failure_message_present?
    is_displayed? FAILURE_MESSAGE
  end

end
```

In order to use the base page object we need to do two things. First we need to include the file, which is handled using the `require_relative` at the top of the file. And second, we need to connect them using inheritance (which is handled in the class declaration with the `<` operator). This is effectively saying `Login` is a child of `BasePage`. Or, `BasePage` is the parent of `Login`. This enables us to freely reference the methods we created in the base page object.

As a result of this refactoring, our `initialize` method changes a bit. Rather than setting the driver object as an instance variable, we are calling `super`. This is a built-in keyword in Ruby for class inheritance. It triggers a method of the same name to be run from the parent class. So in this case, `initialize` from `BasePage` gets run, which is responsible for handling the driver object retrieval and instance variable creation. And since the `BasePage` is now the only place where we reference Selenium actions directly, it stands that it should be the only place where we need to

reference a `@driver` object. After that, we load the page and perform an assertion. And since our base URL is being leveraged in the base page object, we don't have to reference it here in our page object.

To finish things off, we replace all of the remaining driver incantations with calls to the methods we created in the base page object. Our tests remain unchanged, but our page objects are more flexible and readable.

Running our tests (`rspec login_spec.rb` from the command-line) should still yield a passing result.

Part 2: Add Some Error Handling

Remember in the previous chapter when we ran into an error with Selenium when we looked for an element that wasn't on the page? Let's address that now.

To recap -- here's the error message we saw:

```
.F

Failures:

  1) Login failed
     Failure/Error: @login.success_message_present?.should be_false
     Selenium::WebDriver::Error::NoSuchElementError:
       Unable to locate element: {"method":"css selector","selector":".flash.success"}
     # [remote server]
file:///tmp/webdriver-profile20140125-21003-3brprw/extensions/fxdriver@googlecode.com/comp
`FirefoxDriver.prototype.findElementInternal_'
     # [remote server]
file:///tmp/webdriver-profile20140125-21003-3brprw/extensions/fxdriver@googlecode.com/comp
`FirefoxDriver.prototype.findElement'
     # [remote server]
file:///tmp/webdriver-profile20140125-21003-3brprw/extensions/fxdriver@googlecode.com/comp
`DelayedCommand.prototype.executeInternal_/h'
     # [remote server]
file:///tmp/webdriver-profile20140125-21003-3brprw/extensions/fxdriver@googlecode.com/comp
`DelayedCommand.prototype.executeInternal_'
     # [remote server]
file:///tmp/webdriver-profile20140125-21003-3brprw/extensions/fxdriver@googlecode.com/comp
`DelayedCommand.prototype.execute/<'
     # ./code_examples/07/02/login.rb:21:in `success_message_present?'
     # ./code_examples/07/02/login_spec.rb:23:in `block (2 levels) in <top (required)>'

Finished in 12.08 seconds
2 examples, 1 failure
```

The important thing to note is the specific error message Selenium offered up -- the part that comes just before the line that says 'Unable to locate element'. We're interested in `Selenium::WebDriver::Error::NoSuchElementError`. With it we can make our code return a `false` boolean instead of failing.

Now that we have it, let's update the `is_displayed?` method in our base page object to use it.

```
# filename: base_page.rb

require 'selenium-webdriver'

class BasePage

...

def is_displayed?(locator)
  begin
    find(locator).displayed?
  rescue Selenium::WebDriver::Error::NoSuchElementError
    false
  end

end
```

In Ruby, we can make our code more resilient by wrapping an action in a `begin` / `rescue` block. In this case, we're wrapping our check to see if an element is displayed. If the element is not on the page, Selenium will raise an error. When that happens, our code will now rescue for that specific error condition, and return false instead of failing the test.

Now if we wanted to see if an element was not on the page, we can, without fear of our test blowing up.

Just to verify, let's revisit our failed login test and update it to assert that the success message isn't present.

```ruby
# filename: login_spec.rb

require_relative 'login'

describe 'Login' do
...

  it 'failed' do
    @login.with('asdf', 'asdf')
    @login.success_message_present?.should be_false
    #@login.failure_message_present?.should be_true
  end

end
```

Now when we run this (`rspec login_spec.rb` from the command-line) it will run without error, and pass. Feel free to change the assertion back when you're done.

Chapter 10

Writing Resilient Test Code

Ideally, you should be able to write your tests once and run them across all supported browsers. While this is a rosy proposition, there is some work to make this a reliable success. And sometimes there may be a hack or two involved. But the lengths you must go really depends on the browsers you care about and the functionality you're dealing with.

By using high quality locators we are already well ahead of the pack, but there are still some persnickity issues to deal with. Most notably -- timing. This is especially true when working with dynamic, JavaScript heavy pages (which is more the rule than the exception in a majority of applications you'll deal with).

But there is a simple approach that makes up the bedrock of reliable and resilient Selenium tests -- and that's how you wait and interact with elements. Gone are the days of waiting for the page to finish loading before proceeding, or hard-coding sleeps, or doing blanket wait times (a.k.a. implicit waits). Nay. Now are the wonder years of waiting for a specific element to appear for a specific amount of time, and taking an action against it.

We accomplish this through the use of explicit waits.

An Example

Part 1: Create A New Page Object And Update The Base Page Object

Let's step through an example that demonstrates this against a dynamic page on the-internet. The functionality is pretty simple -- there is a button. When you click it a loading bar appears for 5 seconds, then disappears, and gets replaced with the text 'Hello World!'.

Let's start by looking at the markup on the page.

```
<div class="example">
  <h3>Dynamically Loaded Page Elements</h3>
  <h4>Example 1: Element on page that is hidden</h4>

  <br>

  <div id="start">
    <button>Start</button>
  </div>

  <div id="finish" style="display:none">
    <h4>Hello World!</h4>
  </div>

</div>
```

At a glance it's simple enough to tell that there are unique `id` attributes that we can use to reference the start button and finish text.

Let's add a page object for Dynamic Loading.

```
# filename: dynamic_loading.rb

require_relative 'base_page'

class DynamicLoading < BasePage

  START_BUTTON = { css: '#start button' }
  FINISH_TEXT = { id: 'finish' }

  def initialize(driver)
    super
    visit '/dynamic_loading/1'
  end

  def start
    click START_BUTTON
  end

  def finish_text_present?
    wait_for(10) { is_displayed? FINISH_TEXT }
  end

end
```

At the top of the file we require our base page object and set up inheritance when declaring our class so we get our common Selenium actions. After that we wire up our locators in constants, add a visit action to our `initialize` method, and drop in some methods to start the loading process and to see if the finish text is present.

In `start` and `finish_text_present?` we are using two methods which have not been added to our base page object. One to click an element, and another to wait for an action to complete. So let's hop into our base page object and add them to the bottom of the class.

```
# filename: base_page.rb

require 'selenium-webdriver'

class BasePage
  ...

  def click(locator)
    find(locator).click
  end

  def wait_for(seconds = 15)
    Selenium::WebDriver::Wait.new(timeout: seconds).until { yield }
  end

end
```

`click` is simply finding an element on the page and clicking it. `wait_for` is where we are defining our explicit wait.

With `wait_for` we are accepting an integer for the number of seconds that we would like to wait. If nothing is provided, 15 seconds will be used. We then use this value to tell the explicit wait mechanism that Selenium offers how long to wait, and what to wait for. By using a `yield` we are able to easily pass in a code block to run (e.g., a Selenium command surrounded by brackets (`{}`)).

More On Explicit Waits

It's important to set a reasonably sized default timeout for the explicit wait. But you want to be careful not to make it too high. Otherwise you run into a lot of the same timing issues you get from implicit waits. But set it too low and your tests will be brittle, forcing you to run down trivial and transient issues.

In our page object when we're using `wait_for(10) { is_displayed? FINISH_TEXT }` we are telling Selenium to to see if the finish text is displayed on the page. It will keep trying until it either returns `true` or reaches ten seconds -- whichever comes first.

If the behavior on the page takes longer than we expect (e.g., due to slow load times, or a feature change), we can simply adjust this one wait time to fix the test -- rather than increase a blanket wait time (which impacts every test). And since it's dynamic, it won't always take the full amount of time to complete.

Part 2: Write A Test To Use The New Page Object

Now that we have our page object and requisite base page methods we can wire this up in a new test file.

```
require_relative 'dynamic_loading'

describe 'Dynamic Loading' do

  before(:each) do
    @driver = Selenium::WebDriver.for :firefox
    ENV['base_url'] = 'http://the-internet.herokuapp.com'
    @dynamic_loading = DynamicLoading.new(@driver)
  end

  after(:each) do
    @driver.quit
  end

  it 'Example 1: Hidden Element' do
    @dynamic_loading.start
    @dynamic_loading.finish_text_present?.should be_true
  end

end
```

When we run it (`rspec dynamic_loading_page.rb` from the command-line) it should pass.

Part 3: Update Page Object And Add A New Test

Let's step through one more dynamic page example to see if our explicit wait approach holds up.

This example is laid out similarly to the last one, the main difference is that it will render the final result after the progress bar completes. Here's the markup for it.

```
<div class="example">
  <h3>Dynamically Loaded Page Elements</h3>
  <h4>Example 2: Element rendered after the fact</h4>

  <br>

  <div id="start">
    <button>Start</button>
  </div>

  <br>
</div>
```

In order to find the selector for the finish text element we need to inspect the page after the loading bar sequence finishes. Here's what it looks like.

```
<div id="finish" style=""><h4>Hello World!</h4></div>
```

Before we add our test, we need to modify our page object to accommodate visiting the different example URLs.

```
# filename: dynamic_loading.rb

require_relative 'base_page'

class DynamicLoading < BasePage

  START_BUTTON = { css: '#start button' }
  FINISH_TEXT = { id: 'finish' }

  def example(example_number)
    visit "/dynamic_loading/#{example_number}"
  end

  def start
    click START_BUTTON
  end

  def finish_text_present?
    wait_for(10) { is_displayed? FINISH_TEXT }
  end

end
```

We're able to remove the `initialize` method entirely since we no longer need the `visit` action as part of loading the page object. That functionality now lives in the `example` method. Thanks to inheritance our parent class is still able to do what it needs, even without the use of an `initialize` in the child class.

Now that we have that sorted, let's add a new test to reference the markup shown above (and update our existing test to use the new `.example` method).

```
# filename: dynamic_loading_spec.rb

require_relative 'dynamic_loading'

describe 'Dynamic Loading' do

  ...

  it 'Example 1: Hidden Element' do
    @dynamic_loading.example 1
    @dynamic_loading.start
    @dynamic_loading.finish_text_present?.should be_true
  end

  it 'Example 2: Rendered after the fact' do
    @dynamic_loading.example 2
    @dynamic_loading.start
    @dynamic_loading.finish_text_present?.should be_true
  end

end
```

If we run these tests (`rspec dynamic_loading_spec.rb` from the command-line) then the same approach should work for both cases.

Browser Timing

Using explicit waits gets you pretty far. But there are a few things you'll want to think about when it comes to writing your tests to work against various browsers.

It's simple enough to write your tests locally against Firefox and assume you're all set. Once you start to run things against other browsers, you may be in for a rude awakening. The first thing you're likely to run into is the speed of execution. A lot of your tests will start to fail when you point them at either Chrome or Internet Explorer, and likely for different reasons.

In my experience, Chrome execution is very fast, so you will see some odd timeout failures. This is an indicator that you need to add explicit waits to parts of your page objects that don't already have them. And the inverse is true when running things against older version of Internet Explorer

(e.g., IE 8). This is an indicator that your explicit wait times are not long enough since the browser is taking longer to respond -- so your tests timeout.

The best approach to solve this is an iterative one. Run your tests and find the failures. Take each failed test, adjust your code as needed, and run it against the browsers you care about. Repeat until you make a pass all the way through each of the failed tests. Then run a batch of all your tests to see where they fall down. Repeat until everything's green.

Once you're on the other side of these issues, the amount of effort you need to put into it should diminish dramatically.

Closing Thoughts

By explicitly waiting to complete an action, our tests are in a much more resilient position because Selenium will keep trying for a reasonable amount of time rather than trying just once. And each action can be tuned to meet the needs of each circumstance. Couple that with the dynamic nature of explicit waits, and you have something that will work in a multitude of circumstances -- helping you endure even the toughest of browsers.

This is one of the most important concepts in testing with Selenium. Use explicits waits often.

Chapter 11

Prepping For Use

Now that we have tests, page objects, and a base page object, let's package things into a more useful structure.

Global Setup And Teardown

First we'll need to pull the test setup and teardown actions out of our tests and into a central place. In RSpec this is straight-forward through the use of a 'spec_helper' file.

```
# filename: spec_helper.rb

require 'selenium-webdriver'

RSpec.configure do |config|

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

  config.after(:each) do
    @driver.quit
  end

end
```

We need to include the Selenium library here, and by doing so, can remove it from our base page object. And by having our test configuration here, we can remove it from the `before(:each)` and `after(:each)` in our tests -- replacing them with a simple require statement (`require_relative 'spec_helper'`).

Base URL

We'll want to put our base URL someplace central, and we'll want it to be configurable. So let's create a 'config.rb' file in the parent directory and place it there.

```
# filename: config.rb

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

By using a conditional when setting the environment variable (`||=`) we are making it so we can override this value when launching our test suite. It essentially means if the environment variable already exists and contains a value, use it. Otherwise, set it to 'http://the-internet.herokuapp.com'. This will come in handy later, and is something we'll use frequently.

Folder Organization

It's about time we create some folders for our specs and page objects. To err on the side of simplicity, let's call the folders 'spec' (for our tests) and 'pages' (for our page objects). We are using 'spec' since it is a default folder that RSpec will use.

Here's everything we should have after creating folders and moving files around:

```
.
|-- config.rb
|-- Gemfile
|-- pages
|   |-- base_page.rb
|   |-- dynamic_loading.rb
|   `-- login.rb
`-- spec
    |-- dynamic_loading_spec.rb
    |-- login_spec.rb
    `-- spec_helper.rb
```

Updating Require Statements

As a result of doing this, we will need to update the require statements in our tests.

```
# filename: spec/login_spec.rb

require_relative 'spec_helper'
require_relative '../pages/login'

describe 'Login' do
...
```

```
# filename: spec/dynamic_loading_spec.rb

require_relative 'spec_helper'
require_relative '../pages/dynamic_loading'

describe 'Dynamic Loading' do
...
```

Note the use of double-dots (`..`) in the page object require statement. This is how we tell Ruby to traverse up a directory (from our spec directory) before trying to access the page objects folder. The `spec_helper` require remains unchanged since this file lives in the same directory as our tests.

Running Everything

Now that things are cleaned up, we can run everything. To do that we'll want to make sure to include our new config file. We can do that by specifying it at run time with `rspec --require ./config.rb`, or, `rspec -r ./config.rb` (for short).

Note the `./` before `config.rb`. This tells RSpec that the config file is in the current directory.

Give it a shot. All of the tests should run and pass.

Chapter 12

Running A Different Browser Locally

It's straightforward to get your tests running locally against Firefox (that's what we've been doing up until now). But when you want to run them against a different browser like Chrome you quickly run into configuration overhead that can seem overly complex and lacking in good documentation or examples.

A Brief Primer On Browser Drivers

With the introduction of WebDriver (circa Selenium 2) a lot of benefits were realized (e.g., more effective and faster browser execution, no more single host origin issues, etc). But with it came some architectural and configuration differences that may not be widely known. Namely -- browser drivers.

WebDriver works with each of the major browsers through a browser driver which is (ideally) maintained by the browser manufacturer. It is an executable file (consider it a thin layer or a shim) that acts as a bridge between Selenium and the browser.

Let's step through an example using ChromeDriver.

An Example

Before starting, we'll need to grab the latest ChromeDriver binary executable from Google and store the unzipped contents of it in our test code. The simplest thing to do is create a new folder for it (and other things like it).

Let's create a 'vendor' directory and place it there.

```
.
|-- config.rb
|-- Gemfile
|-- pages
|   |-- base_page.rb
|   |-- dynamic_loading.rb
|   `-- login.rb
|-- spec
|   |-- dynamic_loading_spec.rb
|   |-- login_spec.rb
|   `-- spec_helper.rb
`-- vendor
    `-- chromedriver
```

In order for Selenium to use this binary, we have to make sure it knows where it is. There are two ways to do that. We can add the `chromedriver` file to the path of our system, or pass in the file's path when launching Selenium. Let's do the latter option.

But before we go too far, we want to make it so our test suite can run either Firefox or Chrome. To do that, we need to add a browser environment variable to our 'config.rb' and some conditional logic to our 'spec_helper' file.

```
# filename: config.rb

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

```
# filename: spec/spec_helper.rb

require 'selenium-webdriver'

RSpec.configure do |config|

  config.before(:each) do
    case ENV['browser']
    when 'firefox'
      @driver = Selenium::WebDriver.for :firefox
    when 'chrome'
      Selenium::WebDriver::Chrome::Service.executable_path = File.join(Dir.pwd,
'vendor/chromedriver')
      @driver = Selenium::WebDriver.for :chrome
    end
  end

  config.after(:each) do
    @driver.quit
  end

end
```

In the `config.before(:each)` block of our 'spec_helper' we're keying off of the browser environment variable we set in 'config.rb'; using a conditional to determine which browser driver to load. For Chrome we are first telling Selenium where the chromedriver binary executable is located on disk, and then instantiating it.

Assuming Chrome is already installed on the system (in a standard location) when we run our tests (specifying Chrome as the browser) -- `browser='chrome' rspec -r ./config.rb` -- then we

should see them execute in Chrome. If that's not the case for you, then check out the ChromeDriver documentation site to see if there are additional requirements for your specific system setup.

It's worth noting that this will only be reasonably performant since it is launching and terminating the chromedriver binary executable before and after every test. There are alternative approaches we can take, but this is good enough to see where our tests fall down in Chrome (and it will not be the primary way we will run our tests a majority of the time).

Additional Browsers

A similar approach can be applied to other browser drivers, with the only real limitation being the operating system you're running. But remember -- no two browser drivers are alike, so be sure to check out the documentation for the browser you care about to find out the specific requirements:

- ChromeDriver
- FirefoxDriver
- InternetExplorer Driver
- OperaDriver
- SafariDriver

Chapter 13

Running Browsers In The Cloud

If you've ever needed to test features in an older browser like Internet Explorer 8 then odds are you ran a virtual machine (VM) on your computer with a "legit" version of Windows XP.

Handy, but what happens when you need to check things on multiple versions of IE? Now you're looking at multiple VMs. And what about when you need to scale and cover other browser and Operating System (OS) combinations? Now you're looking at provisioning, running, and maintaining your own farm of machines and standing up something like Selenium Grid to coordinate tests across them.

And all you wanted to do was run your tests on the browsers you cared about...

Rather than take on the overhead of a test infrastructure you can easily outsource things to a third-party cloud provider. There are a handful of players in this space, but there's one that stands out -- Sauce Labs.

Why Sauce Labs?

They don't stand out because they enable you to run your tests on pretty much every browser/OS/platform combination. Or because they have great test reporting (like screenshots AND video recordings of your tests). Or for any of the other killer features they have.

They stand out because they're not just building something to sell. They work hard to help make the Selenium Community (and open source software) better. And it's through their contributions that they are able to discern what to build -- so it's something that is attuned to the needs of the community.

A Selenium Remote, Selenium Grid, And Sauce Labs Primer

At the heart of Selenium at scale is the use of Selenium Grid and Selenium Remote. Selenium Grid lets you distribute test execution across several machines and you connect to it with Selenium Remote -- specifying the browser type and version through the use of Selenium Remote's `Capabilities`.

This is fundamentally how Sauce Labs works. Behind the curtain they are ultimately running Selenium Grid, and they receive and execute your tests through Selenium Remote -- knowing which browser and operating system to use because of the `Capabilities` it's users specify.

Let's dig in with an example.

An Example

Part 1: Initial Setup

NOTE: You'll need an account to use Sauce Labs. Their free one offers enough to get you started. And if you're signing up because you want to test an open source project, then be sure to give their 'Open Sauce' account a look (tl;dr -- it's completely free).

With Sauce Labs we need to provide specifics about what we want in our test environment, our credentials, and configure Selenium a little bit differently than we have been. Let's start by creating a new config file for cloud execution ('config_cloud.rb').

```
# filename: config_cloud.rb

ENV['base_url']         ||= 'http://the-internet.herokuapp.com'
ENV['host']             = 'saucelabs'
ENV['operating_system'] ||= 'Windows XP'
ENV['browser']          ||= 'internet_explorer'
ENV['browser_version']  ||= '8'
ENV['SAUCE_USERNAME']   = 'your-sauce-username'
ENV['SAUCE_ACCESS_KEY'] = 'your-sauce-access-key'
```

Notice the use of a host environment variable. This is what we'll use in our `spec_helper` file to determine whether to run things locally or in the cloud. And we'll use the other environment variables to populate the `Capabilities`.

```
# filename: spec/spec_helper.rb

require 'selenium-webdriver'

RSpec.configure do |config|

  config.before(:each) do
    case ENV['host']
    when 'saucelabs'
      caps = Selenium::WebDriver::Remote::Capabilities.send(ENV['browser'])
      caps.version = ENV['browser_version']
      caps.platform = ENV['operating_system']
      caps[:name] = example.metadata[:full_description]

      @driver = Selenium::WebDriver.for(
        :remote,
        url: "http://#{ENV['SAUCE_USERNAME']}:#{ENV['SAUCE_ACCESS_KEY']}
@ondemand.saucelabs.com:80/wd/hub",
        desired_capabilities: caps)
    else
      case ENV['browser']
      when 'firefox'
        @driver = Selenium::WebDriver.for :firefox
      when 'chrome'
        Selenium::WebDriver::Chrome::Service.executable_path = File.join(Dir.pwd,
'vendor/chromedriver')
        @driver = Selenium::WebDriver.for :chrome
      end
    end
  end

  config.after(:each) do
    @driver.quit
  end

end
```

We've taken our original conditional and made it nested underneath a check for the host environment variable. If the host is set to 'saucelabs', then we configure the capabilities for Selenium Remote, passing in the requisite information that we will need for our Sauce Labs session. Otherwise, it will run our tests locally.

The actual username and API key have been omitted from this example. To make this work, simply add yours in to the 'config_cloud.rb' file.

There are a few things in this example that may be worth elaborating on.

First, we are using something called metaprogramming (a.k.a. code that writes code) when we are calling `Selenium::WebDriver::Remote::Capabilities`. We are using the `.send` method to pass in the environment variable. The value of which, in this case, is the same name as the method to configure Selenium Remote to use Internet Explorer. So, we are in effect, creating `Selenium::WebDriver::Remote::Capabilitites.internet_explorer`. And if we were to specify 'chrome' for `ENV['browser']`, then it would give us `Selenium::WebDriver::Remote::Capabilities.chrome`.

Second, for `caps[:name]` we are using a piece of functionality built into RSpec which gives us the name of each test as it is being run. This will make it so each individual job in Sauce Labs will have the name of the test that was run.

Third, for the `url:` line in our `@driver` instantiation, we are injecting our environment variable through the use of string interpolation. This is why we are using double-quoted strings. If they were single-quotes then we wouldn't be able to do it.

Now if we run our test suite (`rspec -r ./config_cloud.rb`) and navigate to our Sauce Labs Account page then we should see each of the tests running in their own job, with proper names, against Internet Explorer 8.

Part 2: Test Status

The only thing missing now is the pass/fail status of the job. In our local terminal window everything should be coming up green. But in the list of our Sauce jobs, the 'Results' panel for everything will just say 'Finished'. This will make our results less useful in the long run, so let's fix it.

Thanks to Sauce Labs' 'sauce_whisk' gem, it's really simple to do.

After we install it we will need to require it somewhere, and our 'config_cloud.rb' file seems like a logical place, since we will only need it when running our tests in Sauce. So let's add it to the top of the file.

```
# filename: config_cloud.rb

require 'sauce_whisk'
...
```

All that's left is to add an action to our `after(:each)` block in our 'spec_helper' file.

Before we issue `@driver.quit` we will want to grab the job ID from our `@driver` object and set the job status based on the test result. Also, we'll want to make sure that it only executes when running tests against Sauce Labs -- so we'll want to wrap it in a conditional check against the host environment variable.

```ruby
# filename: spec/spec_helper.rb
require 'selenium-webdriver'

RSpec.configure do |config|
  ...

  config.after(:each) do
    if ENV['host'] == 'saucelabs'
      if example.exception.nil?
        SauceWhisk::Jobs.pass_job @driver.session_id
      else
        SauceWhisk::Jobs.fail_job @driver.session_id
      end
    end

    @driver.quit
  end

end
```

Now when we run our tests (`rspec -r ./config_cloud.rb`) and navigate to our Sauce Labs Account page, we should see our tests running like before -- but now when they finish there should be a proper test status (e.g., 'Pass' or 'Fail').

Part 3: Accessing Private Apps

There are various ways that companies make their pre-production application available for testing. Some use an obscure public URL and protect it with some form of authentication (e.g., Basic Auth, or cert based authentication). Others keep it behind their firewall.

For those that stay behind a firewall, Sauce Labs has got you covered. They have a program called Sauce Connect that creates a secure tunnel between your machine and their cloud. With it, you can run tests in Sauce Labs and still have them reach the apps that are only available on your network.

In Ruby there are two ways to leverage Sauce Connect; a separate download, or through another one of Sauce's Ruby libraries -- 'sauce-connect'.

Let's step through an example using the 'sauce-connect' gem and running a copy of 'the-internet' locally.

An Example

First we'll need to install the 'sauce-connect' library. Once we have it, we'll want to add it to our 'config_cloud.rb' file. But since the Sauce Connect tunnel will automatically load when we require

the library, we will want to wrap the require statement in a conditional. And to require it we will need to refer to it as `sauce`.

```
# filename: config_cloud.rb

require 'sauce_whisk'

ENV['base_url']         ||= 'http://the-internet.herokuapp.com'
ENV['host']             = 'saucelabs'
ENV['operating_system'] ||= 'Windows XP'
ENV['browser']          ||= 'internet_explorer'
ENV['browser_version']  ||= '8'
ENV['SAUCE_USERNAME']   = 'your-sauce-username'
ENV['SAUCE_ACCESS_KEY'] = 'your-sauce-access-key'
ENV['tunnel']           ||= ''

unless ENV['tunnel'].empty?
  require 'sauce'
  ENV['base_url']       = 'http://the-internet-local:4567'
end
```

At the bottom of the file we've added a new environment variable (`ENV['tunnel']`), set it to an empty string, and made it so we can override it at runtime.

We then check to see if it's empty. If it is empty, then we do nothing. If it isn't empty, then we load the sauce-connect library (`require 'sauce'`) and change the base URL to run our tests against a different end-point (`'http://the-internet-local:4567'`).

This new URL is how we will interact with our local copy of 'the-internet'. To use it, we'll first need to download and unzip a copy of it from it's GitHub repo and place it in 'vendor/the-internet'.

When we're done, our directory tree should look like this:

```
.
|-- config_cloud.rb
|-- config.rb
|-- pages
|   |-- base_page.rb
|   |-- dynamic_loading.rb
|   `-- login.rb
|-- sauce_connect.log
|-- spec
|   |-- dynamic_loading_spec.rb
|   |-- login_spec.rb
|   `-- spec_helper.rb
`-- vendor
    |-- chromedriver
    `-- the-internet
```

Next we need to update our local hosts file to map our local IP address (127.0.0.1) to a DNS helper name (e.g., 'the-internet-local'). This is so our tests running on Sauce Labs will be able to find our local server.

If we just provide 'localhost' or 127.0.0.1, it won't work (as noted in this Sauce Support post. Updating the hosts file varies depending on your operating system. So if you're not sure how to do it, or what that means, then give this How-To Geek article a read.

Now we can open a new terminal window, change into the 'vendor/the-internet' directory, and start the app (with `ruby server.rb`) -- which will make it available on port 4567. If you receive an error, be sure to install the necessary gems (e.g., try running `bundle install` before trying the server again).

Now that our server is running locally, let's switch back over to our original terminal window and run our tests. In order to trigger our tunnel, we will need to override our tunnel environment variable. So let's run the following: `tunnel='on' rspec -r ./config_cloud.rb`.

We should then see the following appear in our terminal.

```
[Connecting to Sauce Labs...]
```

If we navigate to our Sauce Labs Tunnel Status page then we should see a tunnel listed as active. And if we navigate to our Sauce Labs Account page then we should see each of the tests running in their own job -- each executing against our local server.

Once the tests finish, the tunnel should close. We can quickly confirm this by revisiting the Sauce Labs Tunnel Status page.

Part 4: A Small Bit Of Clean-up

One by-product of this setup is that a 'sauce_connect.log' file will get outputted into our root directory. We won't really end up using it, nor is it easily configurable to suppress or move it to a different location. Therefore we can choose to ignore it by adding it to a version control system ignore file (which is '.gitignore' in my case) so that it doesn't get pushed up into the test repository.

```
# filename: .gitignore

*.log
```

Now that we have two config files, let's move them into their own directory and pare down the names for simplicity. Let's create a `config` directory and rename the files to `local.rb` and `cloud.rb`.

This leaves our directory tree structure looking like this:

```
.
|-- config
|   |-- cloud.rb
|   `-- local.rb
|-- pages
|   |-- base_page.rb
|   |-- dynamic_loading.rb
|   `-- login.rb
|-- spec
|   |-- dynamic_loading_spec.rb
|   |-- login_spec.rb
|   `-- spec_helper.rb
`-- vendor
    |-- chromedriver
    `-- the-internet
```

Chapter 14

Speeding Up Your Test Runs

It's great that we can easily run our tests in Sauce Labs. But it's a real bummer that all of our tests are executing in series. As our suite grows, things will quickly start to add up and really hamper our ability to get feedback quickly.

With parallelization we can quickly remedy this -- and in Ruby there is a library that makes this simple to accomplish.

Enter parallel_tests.

Setup

After installing the library we just have to change out the command we use to execute our test runs. Instead of `rspec` we will use one that parallel_tests provides -- `parallel_rspec`.

`parallel_rspec` is effectively a wrapper around RSpec. It's responsible for breaking our spec files into groups and launching each of them in separate system processes along with any arguments we pass in to configure RSpec. So we can still provide our `--r`, we just have to specify `--test-options` before-hand.

So if we want to run our tests in Sauce Labs in parallel (without the secure tunnel), then we would use `parallel_rspec --test-options '-r ./config_cloud.rb' spec`. Note the additional `spec` at the end of the command. This tells `parallel_rspec` where the test files live (which is in the 'spec' directory).

These extra arguments can seem a little verbose, but it's a necessary evil with this library -- and it's a small price to pay to get parallelization.

Number Of Processes

By default `parallel_rspec` will use two processes. We can increase this number by passing an additional argument of `-n` and the number of processes that we want -- e.g., `parallel_rspec -n 5`. But keep in mind that the actual number of parallel processes that get spawned depends on how many spec files there are (since this library works at the file level). So for now, two processes is all we'll get (since we only have two test files). But as our suite grows, increasing the number of processes will yield a huge performance improvement.

It's also worth noting that Sauce Labs is built to handle test execution in parallel. You just need to be cognizant of how many parallel sessions your account is set up to handle (e.g., three for Sauce Open accounts).

Randomizing

A great way to make sure your tests don't have any cross-dependencies (and to ferret out possible anomalies in your application under test) is to run your tests in a random order. Within RSpec this is an easy thing to accomplish. It's just another command-line argument to pass in on test execution `--order random`.

```
parralel_rspec --test-options '-r ./config_cloud.rb --order random'
```

This coupled with parallelization will help make your tests really sing.

Chapter 15

Organizing Your Test Runs

> "Am I the only Ruby dev who writes rake tasks for everything? Don't want to worry about the syntax of 50 different cmds or which dir I'm in." [source]
>
> -- Russ Olsen (author of "Eloquent Ruby" and "Design Patterns in Ruby")

There's a lot to remember every time we want to run our tests now. So let's clean things up with some Rake tasks.

Rake is a library in Ruby that enables us to easily capture and run repetitive tasks. To set it up, we just need to install the 'rake' gem, add a 'Rakefile' in our root directory, and add our tasks to the 'Rakefile'. Then all we have to do is run our tasks using `rake` from the command-line.

Let's step through an example.

An Example

Part 1: Create A Rakefile And Add Initial Tasks

First we'll want to create a 'Rakefile' with some initial tasks.

```ruby
# filename: Rakefile

def launch_in_parallel(config_file)
  system("parallel_rspec #{'-n ' + ENV['processes'] if ENV['processes']} --test-options
'-r ./#{config_file} --order random' spec")
end

desc 'Run tests locally, by browser'
task :local, :browser do |t, args|
  ENV['browser'] = args[:browser]
  launch_in_parallel('config.rb')
end

desc "Run tests in Sauce"
task :cloud do
  launch_in_parallel('config/cloud.rb')
end

desc "Run tests in Sauce Connect"
task :cloud_tunnel do
  ENV['tunnel'] = 'on'
  launch_in_parallel('config/cloud.rb')
end
```

Note that this looks similar to a standard Ruby file. We declare a method, reference that method, do some string interpolation, and set some environment variable values. It's the task declaration that's a little bit different (and specific to Rake).

To declare a task, we first want to give it a description. Descriptions are useful since they show up in our task list (next to the command to launch the task). If one is not set, then the task will not appear in the task list. To set one, we use the keyword `desc` and provide a string value (e.g., `'Run tests locally, by browser'`).

Once we have that, we can create the task itself. This is done with the keyword `task` followed by the name of the task (written as a symbol -- `:cloud`). The first line of a task ends with the word `do` -- this opens up the code block and terminates with the word `end`. Between `do` and `end` is where we place the code for our task.

Our `:local` task is a bit different than the others since we are taking an argument. To accomplish this we specify an additional parameter on the `task` line (e.g., `task :local, :browser`) and some variables in pipes after `do` (e.g., `do |t, args|`). With the `args` variable we are able to receive an argument at run time and pass it into the browser environment variable (e.g., `ENV['browser'] = args[:browser]`).

NOTE: For more information on Rake, check out this tutorial.

In our `launch_in_parallel` method we pull out the lengthy command used to run our tests and wrap it in a `system` command to launch it. We also make this method accept an argument for the config file and inject it into the execution string through interpolation. This enables us to use it for each of our rake tasks. It also keeps our tasks clean -- removing redundancy -- and enables us to easily change our execution string (if we need to add additional arguments).

If we save this file, close it, and run `rake -T` from the command-line, we should see a list of our available Rake actions.

```
rake cloud           # Run tests in Sauce
rake cloud_tunnel    # Run tests in Sauce Connect
rake local[browser]  # Run tests locally, by browser
```

To run a task, just type `rake` followed by the task name. For tasks that take an argument, wrap the argument in single-quotes (e.g., `rake local['firefox']`). And to specify the number of parallel processes in our test runs, we do that with an environment variable before our `rake` command (e.g., `processes='5' rake cloud`).

If we run either the `cloud` or `local` tasks, they run our tests in parallel with 2 processes, in a random order, and should pass. But the `cloud_tunnel` task will result in failing tests since our local copy of 'the-internet' is probably not running. Let's remedy that by updating the `cloud_tunnel` task to start and stop it.

Part 2: Automate Starting And Stopping Our Local Application

In our `:cloud_tunnel` task, we want to wrap our environment variable setting and test execution command with the necessary starting and stopping actions for our server. We already know how to start the server (`ruby server.rb` from within the 'vendor/the-internet' directory). But since there's no clean way to stop the server, we need to muck about with the process ID and send a kill command to it.

```
# filename: Rakefile

...

desc "Run tests in Sauce Connect"
task :cloud_tunnel do
  Dir.chdir('vendor/the-internet')
  process_id = Process.spawn('ruby server.rb')
  Dir.chdir('../..')
    ENV['tunnel'] = 'on'
    launch_in_parallel('config/cloud.rb')
  Process.kill("KILL", process_id)
end
```

First, we change directories into `'vendor/the-internet'` to get access to the application and its dependent libraries. We then launch the server by spawning it in its own process (so it runs in the background); storing the return result (which is the process ID of the server, and a Fixnum value) in a variable.

We then traverse back up to the root directory, run our tests, and then stop the server when they're done -- issuing a kill command with the process ID that we stored.

Now if we run this task (`rake cloud_tunnel`) then our tests should run in Sauce Labs, through the secure tunnel, and exercise our locally served app.

Part 3: Add Tasks For The Browsers We Care About

Now that we have Rake tasks we are in a better position to run tests quickly. But in order to make this approach worthwhile, we want to add more tasks to support different browser combinations. And while we're at it, we might as well make our local browser options explicit (rather than just argument based).

Let's start with the local tasks, and then add the cloud ones.

```
# filename: Rakefile

def launch_in_parallel(config_file)
  system("parallel_rspec #{'-n ' + ENV['processes'] if ENV['processes']} --test-options
'-r ./#{config_file} --order random' spec")
end

namespace :local do
  desc 'Run tests in Firefox'
  task :firefox do
    ENV['browser'] = 'firefox'
    launch_in_parallel('config/local.rb')
  end

  desc 'Run tests in Chrome'
  task :chrome do
    ENV['browser'] = 'chrome'
    launch_in_parallel('config/local.rb')
  end
end
...
```

Our `launch_in_parallel` method remains unchanged. We instead focus on reworking our local task -- turning it into two simple tasks which live underneath a namespace.

In Rake we can group common tasks together with a `namespace`, which follows the same naming

conventions as a task. This helps organize our task list, making things more intuitive to run. You'll see what I mean soon.

Now let's add in our cloud tasks.

```
# filename: Rakefile

...

namespace :cloud do
  desc "Run tests in IE, by version"
  task :ie, :version do |t, args|
    ENV['browser']  = 'internet_explorer'
    ENV['browser_version'] = args[:version]
    launch_in_parallel('config/cloud.rb')
  end

  desc "Run tests in Firefox, by version"
  task :firefox, :version do |t, args|
    ENV['browser']  = 'firefox'
    ENV['browser_version'] = args[:version]
    launch_in_parallel('config/cloud.rb')
  end

  desc "Run tests in Chrome, by version"
  task :chrome, :version do |t, args|
    ENV['browser']  = 'chrome'
    ENV['browser_version'] = args[:version]
    launch_in_parallel('config/cloud.rb')
  end

  desc "Run tests against local app in Sauce Connect"
  task :tunnel do
    Dir.chdir('vendor/the-internet')
    process_id = Process.spawn('ruby server.rb')
    Dir.chdir('../..')
      ENV['tunnel'] = 'on'
      launch_in_parallel('config/cloud.rb')
    Process.kill("KILL", process_id)
  end
end
```

By using a `:cloud` namespace we are able to group our cloud tasks together.

We've renamed our `:cloud_tunnel` task to `:tunnel` and removed our `:cloud` task -- replacing it with tasks for various browsers which take an argument for the browser version.

If we run `rake -T` we should see the following.

```
rake cloud:chrome[version]   # Run tests in Chrome, by version
rake cloud:firefox[version]  # Run tests in Firefox, by version
rake cloud:ie[version]       # Run tests in IE, by version
rake cloud:tunnel            # Run tests against local app in Sauce Connect
rake local:chrome            # Run tests in Chrome
rake local:firefox           # Run tests in Firefox
```

If we want to run our tests against Internet Explorer 8, all we have to do now is run `rake cloud:ie['8']`.

Keep in mind that our default operating system environment variable is set to 'Windows XP'. So if we try for a browser version that's too high (e.g., `rake cloud:ie['11']`), then we will receive an error message from Sauce Labs stating 'Unsupported OS/browser/version combo'.

To handle this, we need to alter the operating system environment variable (`ENV['operating_system']`) in our Rake tasks.

Part 4: Update Tasks to Make The Operating System Configurable

We can pretty easily remedy this by adding an additional argument to our Rake tasks. Let's see what it would look like in our `:ie` task.

```
# filename: Rakefile

namespace :cloud do

  desc "Run tests in IE, by OS and version"
  task :ie, :os, :version do |t, args|
    ENV['browser']  = 'internet_explorer'
    ENV['operating_system'] = args[:os]
    ENV['browser_version'] = args[:version]
    launch_in_parallel('config/cloud.rb')
  end

...
```

When we run `rake -T` we'll see our updated task list:

```
rake cloud:chrome[version]      # Run tests in Chrome, by version
rake cloud:firefox[version]     # Run tests in Firefox, by version
rake cloud:ie[os,version]       # Run tests in IE, by OS and version
rake cloud:tunnel               # Run tests against local app in Sauce Connect
rake local:chrome               # Run tests in Chrome
rake local:firefox              # Run tests in Firefox
```

Now if we wanted to run our tests against different versions of Internet Explorer, we can freely do it by specifying both the operating system and the version of IE. For example:

```
# note that there are no spaces around the comma (,)
rake cloud:ie['Windows XP','6']
rake cloud:ie['Windows XP','7']
rake cloud:ie['Windows XP','8']
rake cloud:ie['Windows 7','8']
rake cloud:ie['Windows 7','9']
rake cloud:ie['Windows 7','10']
rake cloud:ie['Windows 8','10']
```

Let's apply the same configuration to the Firefox and Chrome cloud tasks. And while we're at it, let's add the same (plus a browser argument) for the tunnel task.

```
# filename: Rakefile

...

namespace :cloud do

  ...

  desc "Run tests in Firefox"
  task :firefox, :os, :version do |t, args|
    ENV['browser']  = 'firefox'
    ENV['operating_system'] = args[:os]
    ENV['browser_version'] = args[:version]
    launch_in_parallel('config/cloud.rb')
  end

  desc "Run tests in Chrome"
  task :chrome, :os, :version do |t, args|
    ENV['browser']  = 'chrome'
    ENV['operating_system'] = args[:os]
    ENV['browser_version'] = args[:version]
    launch_in_parallel('config/cloud.rb')
  end

  desc "Run tests against local app in Sauce Connect"
  task :tunnel, :os, :browser, :version do |t, args|
    ENV['browser']  = args[:browser]
    ENV['operating_system'] = args[:os]
    ENV['browser_version'] = args[:version]
    Dir.chdir('vendor/the-internet')
    process_id = Process.spawn('ruby server.rb')
    Dir.chdir('../..')
      ENV['tunnel'] = 'on'
      launch_in_parallel('config/cloud.rb')
    Process.kill("KILL", process_id)
  end
end
```

Now our task list should look like this.

```
rake cloud:chrome[os,version]          # Run tests in Chrome
rake cloud:firefox[os,version]         # Run tests in Firefox
rake cloud:ie[os,version]              # Run tests in IE
rake cloud:tunnel[os,browser,version]  # Run tests against local app in Sauce Connect
rake local:chrome                      # Run tests in Chrome
rake local:firefox                     # Run tests in Firefox
```

There are over 250 browser/OS combinations to choose from in Sauce Labs (you can see a full list here), and there are numerous ways to organize your Rake tasks. So if you aren't looking forward to the prospect of remembering and typing out the operating system, browser, or browser version for your test runs, simply create tasks for each of the browser/OS combinations you care about.

That's one of the biggest advantages to having a Rakefile -- you can be as explicit as you want, have a central list to see what is available, and can easily run each item on that list.

Chapter 16

Automating Your Test Runs

You'll probably get a lot of mileage out of your test suite in its current form if you just run things from your computer, look at the results, and inform people when there are issues. But that only helps you solve part of the problem.

The real goal in all of this is to find issues reliably, quickly, and continuously. We've built things to be reliable and quick. Now we need to make them run continuously -- and ideally in sync with the development workflow you are a part of. In order to do that, we want to use a Continuous Integration (CI) server.

A Continuous Integration Primer

Continous Integration (a.k.a. CI) is the practice of merging code that is actively being worked on into a shared mainline (e.g., trunk or master) as often as possible (e.g., several times a day). This is with the hopes of finding issues early and avoiding merging and integration issues that are not only considered a special kind of hell, but can dramatically slow the time it takes to release software.

The use of a CI server (a.k.a. build server) enables this practice to be automated, and to have tests run as part of the work flow. The lion's share of tests that are typically run on a CI Server are unit (and potentially integration) tests. But we can very easily add in our automated acceptance tests.

There are numerous CI Servers available for use today, most notably:

- Bamboo
- Jenkins
- Tddium
- TravisCI

Part 1: Tagging & Workflow

In order to get the most out of our test runs in a CI environment, we want to break up our test suite into small, relevant chunks and have separate jobs for each. This helps keep test runs fast (so people on your team will care about them) and informative. Gojko Adzic refers to these as 'Test Packs'.

The workflow is pretty straightforward. The CI Server pulls in the latest code, merges it, and runs unit tests. We then have the CI Server kick off a new job to deploy to a test server and run a

subset of our acceptance tests -- our critical ones (e.g., smoke or sanity tests). Assuming those pass, we can have another job run the remaining tests after that -- our longer running tests. Adam Goucher refers to this strategy as a 'shallow' and 'deep' tagging model.

To demonstrate this, let's tag our tests and update our rake tasks to use them.

An Example

RSpec comes built in with tagging support. It's a simple matter of adding a key/value pair to denote what you want. You can place it on individual tests, or a group of tests -- using as many tags as you want (separating them with commas).

Let's add some to our specs -- following Adam Goucher's shallow and deep approach.

```
# filename: spec/dynamic_loading_spec.rb

require_relative 'spec_helper'
require_relative '../pages/dynamic_loading'

describe 'Dynamic Loading', depth: 'deep' do
...
```

```
# filename: spec/login_spec.rb

require_relative 'spec_helper'
require_relative '../pages/login'

describe 'Login', depth: 'shallow' do
...
```

If we wanted to apply this tag directly to a test, then it would look like this:

```
it 'succeeded', depth: 'shallow' do
```

To run tests based on a specific tag, we will need to pass in an additional argument to RSpec. It starts with `--tag` and accepts a tag's key/value pair (e.g., `--tag depth:shallow` or `--tag depth:deep`).

Let's update our 'Rakefile' tasks to handle this.

```
# filename: Rakefile

def launch_in_parallel(config_file)
  system("parallel_rspec #{'-n ' + ENV['processes'] if ENV['processes']} --test-options
'-r ./#{config_file} --order random #{'--tag ' + ENV['tag'] if ENV['tag']}' spec")
end
...
```

By updating our `launch_in_parallel` method to use a conditional based on the existence of the tag environment variable (`ENV['tag']`), we are able to dynamically alter our execution string at run time without having to change any of our tasks.

We just have to specify the tags at run time before our rake task.

```
tag='depth:shallow' rake local:firefox
```

Part 2: Reporting

In order to make our test output useful for a CI Server we need to generate it in a standard way. One format that works across most CI Servers is JUnit XML.

An Example

This functionality doesn't come built into RSpec, but it's simple enough to add through the use of a third-party library. There are plenty to choose from, but we'll go with 'rspec_junit_formatter'.

After we install it we need to specify the formatter type and an output file for RSpec to consume. This is done through the use of two new arguments; `--format RspecJunitFormatter` and `--out results.xml` . But in order to use them we need to create a uniquely named XML output file for each test process. To accomplish that with `parallel_tests` we will need to take advantage of another feature in RSpec -- the `.rspec` file.

RSpec comes with the ability to place command arguments in a `.rspec` file within the root directory of our project, which will automatically be consumed at runtime. So we'll place our new arguments there, and use a kind of interpolation to inject an environment variable from `parallel_tests` which will be unique for each test process.

We're doing this instead of adding it to the `launch_in_parallel` method in our 'Rakefile' because of how parallel_tests handles execution. In order to get dynamically named XML output, we have to use the `.rspec` file. And we don't want our XML output to happen all the time. So let's make it conditional based on a new environment variable.

```
# filename: .rspec

<% if ENV['ci'] == 'on' %>
--format RspecJunitFormatter
--out tmp/result<%= ENV['TEST_ENV_NUMBER'] %>.xml
<% end %>
```

Now when we run our tests with the new environment variable set to 'on' (e.g., `ci='on' rake local:firefox`), each test process creates it's own result.xml by number (e.g., result.xml, result2.xml, etc.), and ends up in a `tmp` directory (`tmp` being short for temporary).

After a test run our directory structure should look like this:

```
.
|-- config
|   |-- cloud.rb
|   `-- local.rb
|-- .gitignore
|-- pages
|   |-- base_page.rb
|   |-- dynamic_loading.rb
|   `-- login.rb
|-- Rakefile
|-- .rspec
|-- spec
|   |-- dynamic_loading_spec.rb
|   |-- login_spec.rb
|   `-- spec_helper.rb
|-- tmp
|   |-- result2.xml
|   `-- result.xml
`-- vendor
    |-- chromedriver
    `-- the-internet
```

If we open one of the XML files from the `tmp` directory, it will contain a bunch of info from the test run. This is what our CI server will use to track test results (e.g., passes, failures, time to complete, etc.) and trend them over time.

Here's what one from a passing run looks like:

```
<?xml version="1.0" encoding="UTF-8"?>
<testsuite tests="2" failures="0" errors="0" time="20.104438" timestamp=
"2014-02-01T19:22:05+00:00">
  <properties/>
  <testcase classname="spec.dynamic_loading_spec" name="Dynamic Loading Example 1:
Hidden Element" time="10.926016"/>
  <testcase classname="spec.dynamic_loading_spec" name="Dynamic Loading Example 2:
Rendered after the fact" time="9.177722"/>
</testsuite>
```

We don't want to commit the `tmp` directory or the XML files our repository, so let's make sure to add them to our ignore file.

```
# filename: .gitignore

*.log
tmp/
```

We're almost ready to wire things up to our CI Server, but before we do, we want to customize the failure messages in our tests.

Part 3: Custom Failure Messaging

If we leave things as they are, we will receive messages that say 'expected true but got false' with no easy way to correlate the CI failure to the Sauce Labs job that it ran in -- which makes it hard to discern if the test failure was legitimate and painful to run down the real issue. Let's remedy that now.

An Example

```
# filename: spec/spec_helper.rb

require 'selenium-webdriver'
require_relative 'support/matchers'

RSpec.configure do |config|

  config.before(:each) do
    ...

    if ENV['host'] == 'saucelabs'
      $job_message = "Watch a video of the test at https://saucelabs.com/tests/#{
@driver.session_id}"
    else
      $job_message = ""
    end
  end
  ...
```

First, we require a new file (which we need to create). It will contain our custom failure message (a.k.a. a custom matcher).

Next, in the `config.before(:each)` section of our `spec_helper`, just after the `@driver` has been instantiated, we add a conditional to set a global variable based on where the tests are being run. If Sauce Labs, then we create a message that includes the URL for the job being run. Otherwise, we set the message to be blank. Having the logic here means we only have to define it once, and can keep it out of our custom matchers.

In RSpec, we can create our own matchers (just like `be_true`). In them, we can specify custom failure messages. This is where we will take advantage of the `$job_message` variable.

```ruby
# filename: spec/support/matchers.rb

RSpec::Matchers.define :be_displayed do |expected|

  match do |actual|
    actual == true
  end

  failure_message_for_should do |actual|
    "Expected element to display, but it didn't. #{$job_message}"
  end

  failure_message_for_should_not do |actual|
    "Expected element to NOT display, but it did. #{$job_message}"
  end

end
```

We create a new `support` directory within our `spec` directory and add a `matchers.rb` file to hold our new `be_displayed` matcher. In our custom matcher we are doing 3 things:

1. Performing a comparison against the value being provided by our tests
2. Setting the custom failure message for the `.should` case
3. Setting the custom failure message for the `.should_not` case

In the custom failure messages, we are consuming the global variable that we created in our `spec_helper` file. For more info on custom matchers, check out the write-up for them on the RSpec wiki.

Now we need to update our tests to use these matchers (replacing `be_true` with `be_displayed`).

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

...

  it 'Example 1: Hidden Element' do
    @dynamic_loading.example 1
    @dynamic_loading.start
    @dynamic_loading.finish_text?.should be_displayed
  end

  it 'Example 2: Rendered after the fact' do
    @dynamic_loading.example 2
    @dynamic_loading.start
    @dynamic_loading.finish_text?.should be_displayed
  end

end
```

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

...

  it 'succeeded' do
    @login.with('tomsmith', 'SuperSecretPassword!')
    @login.success_message?.should be_displayed
  end

  it 'failed' do
    @login.with('asdf', 'asdf')
    @login.failure_message?.should be_displayed
  end

  it 'forced failure' do
    @login.with('asdf', 'asdf')
    @login.failure_message?.should_not be_displayed
  end

end
```

Notice that the assertion helper methods used in our assertions have been shortened. This was done so that the tests read more intelligibly. We still want to keep the question mark though since it signifies that a boolean result is being returned.

Also, we added a new test in `login_spec.rb` titled `'forced failured'`. This helps us see what the failure output will look like.

All that's left to do now is update the page objects so our methods match our tests.

```
# filename: pages/dynamic_loading.rb

...

  def finish_text?
    wait_for(10) { is_displayed? FINISH_TEXT }
  end

end
```

```
# filename: pages/login.rb

  def success_message?
    is_displayed? SUCCESS_MESSAGE
  end

  def failure_message?
    is_displayed? FAILURE_MESSAGE
  end

end
```

Now when we run our test suite and a test fails, it will read with a helpful failure message, and a link to the Sauce Labs job.

> Expected element to NOT display, but it did. Watch a video of the test at https://saucelabs.com/tests/8b075a49cb20477f9aa820de4d196ac5

And if we need to add a different matcher, it's simple enough to do.

Now we're ready to wire things up to our CI server.

Part 4: CI Server Job Configuration

The first thing we need to do (assuming you already have a CI Server setup) is create a job, name it appropriately (e.g., 'Shallow Tests IE8'), and configure it to meet the needs of you and your team.

If your team already uses CI and has a job that runs unit or integration tests, you can configure your job to key off of that one (e.g., if the unit tests pass, then run your shallow tests). This is ideal since their job is part of their workflow (e.g., it triggers every time code is pushed to the mainline), or there is some kind of time delayed polling (e.g., see if new code was pushed every 5 minutes). This means code is compiled and tests are run often.

Depending on the application you're testing, there may need to be an intermediate action that involves deploying code to a test environment which has test data. But by keying your job off of theirs, you are plugging your test suite directly into the development workflow. This is why running shallow tests to start is important -- we want only fast and critical tests to run to make sure they are providing valuable results.

If this is not how things are set up, then look to set up a job that polls the development codebase so that your tests can be triggered when there are code changes pushed by developers (e.g., with post-receive hooks or on every pull request).

Alternatively, you can set up scheduled runs for different times of the day or week. They're less ideal, and better suited for long running jobs during off-hours.

Let's step through an example showing you how to set up a job with Jenkins.

An Example

Jenkins can be set up on your own server or through a hosted offering like CloudBees. Once you have it set up with Ruby on the system (which is the default with CloudBees), you just need to install some plugins and set up a job.

In your job you can configure where to pull your tests from and when to run them.

Let's set up a job for our tests (pulling them in from a GitHub repository) and make them run after another job (e.g., a development build) completes successfully.

Installing Plugins

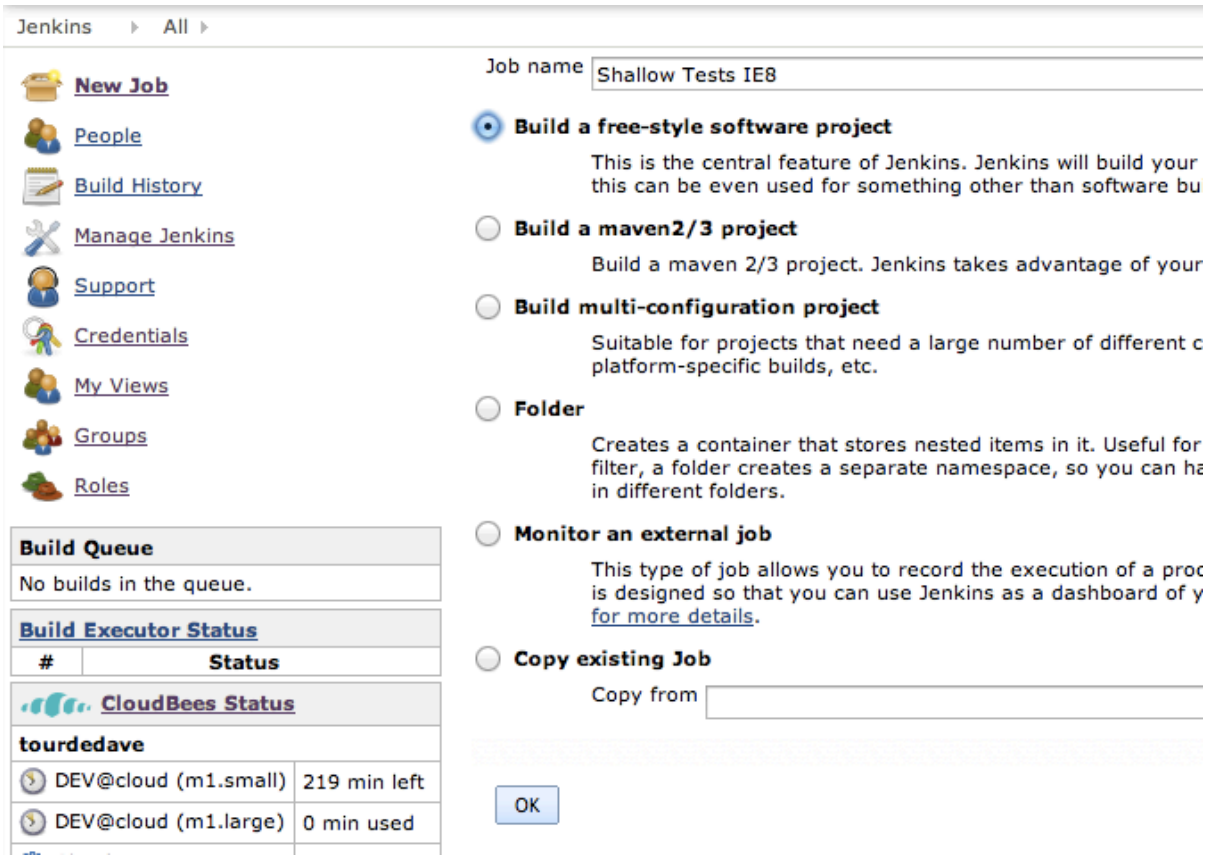
1. From the Dashboard, click on `Manage Jenkins`
2. Then click on `Manage Plugins`
3. Once the Manage Plugins menu loads, click on the `Available` tab
4. In the top-right is a search field (labeled `filter`) -- input the name of what you want to install
5. In the list of plugins, find the one you want, and check the box to the left of its name
6. Click `Install without restart` at the bottom of the screen
7. On the next screen, check the box that says `Restart Jenkins when installation is complete and no jobs are running`

You should see a series of screens saying `Please wait while Jenkins is restarting` and `Please wait while Jenkins is getting ready to work`. When it's done, you'll be dropped back to the dashboard.

Setting Up Your Test Job

Step 1: Create A Job

1. Click `New Job` from the top-left of the Dashboard
2. Give it a name (e.g., 'Shallow Tests IE8')
3. Select the `Build a free-style software project` option
4. Click `OK`



This will take you to the configuration menu for the job you just created.

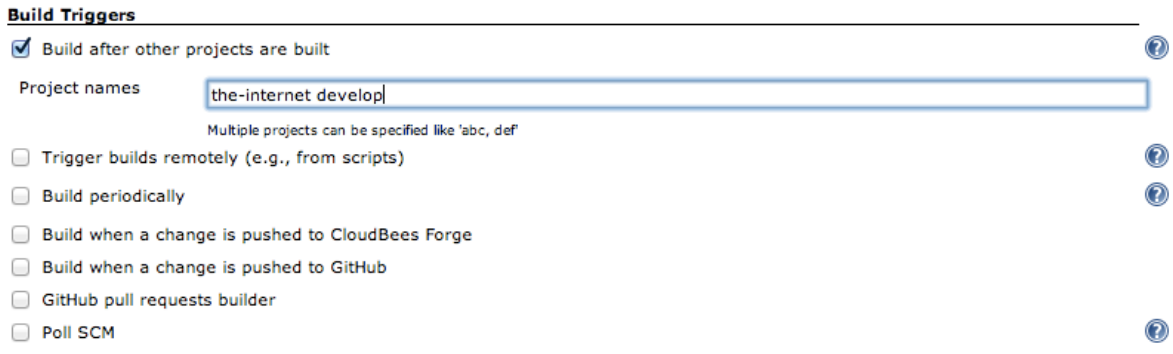
Step 2: Pull In Your Test Code

1. Scroll down to the `Source Code Management` section
2. Select the 'Git' option
3. Input the Repository URL (e.g., `git@github.com:tourdedave/selenium-guidebook-example.git`)
4. Specify the branch (e.g, `origin/master`)

In order to use a private repository (like the example above) you need to set up deploy keys in GitHub. You can read more about that here.

Step 3: Set up Build Triggers

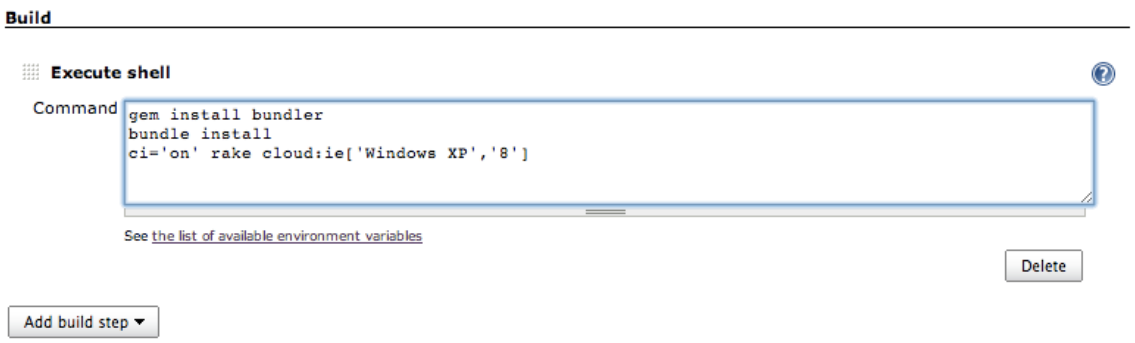
1. Scroll down to the `Build Triggers` section
2. Check `Build after other projects are built`
3. Input the name of the development project you want to key off of (e.g. 'the-internet develop')



Step 4: Build

1. Scroll down to the `Build` section
2. Click on `Add Build Step` and select `Execute Shell`
3. In the `Command` input box, add the following commands:

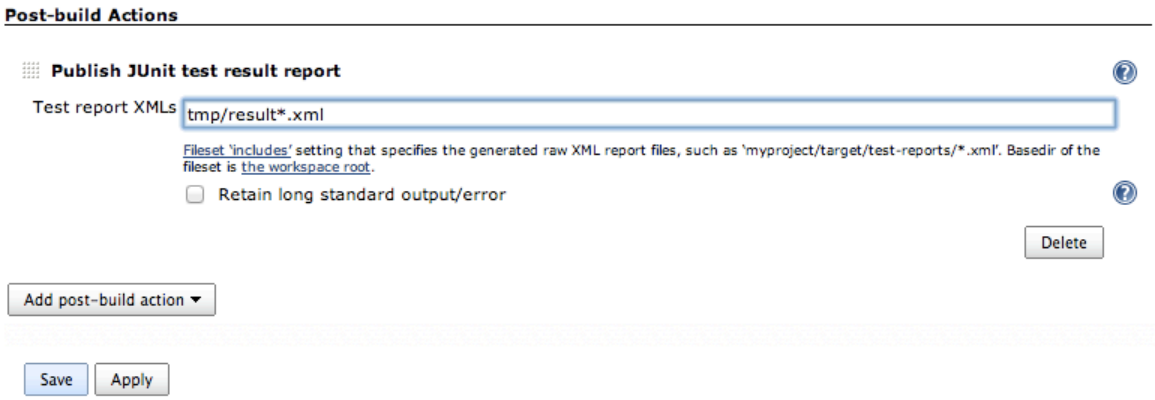
```
gem install bundler
bundle install
ci='on' rake cloud:ie['Windows XP','8']
```



Since our tests have never run on this server, it's best to include the installation of the bundler gem (`gem install bundler`), running bundler to pull in our test suite's gems (`bundle install`), and then launching the test suite.

Step 5: Configure Test Reports

1. Scroll down to the `Post-build Actions` section
2. Click on `Add post-build action`
3. Select `Publish JUnit test result report`
4. In the `Test report XMLs` input box, type `tmp/result*.xml`
5. Click `Save`

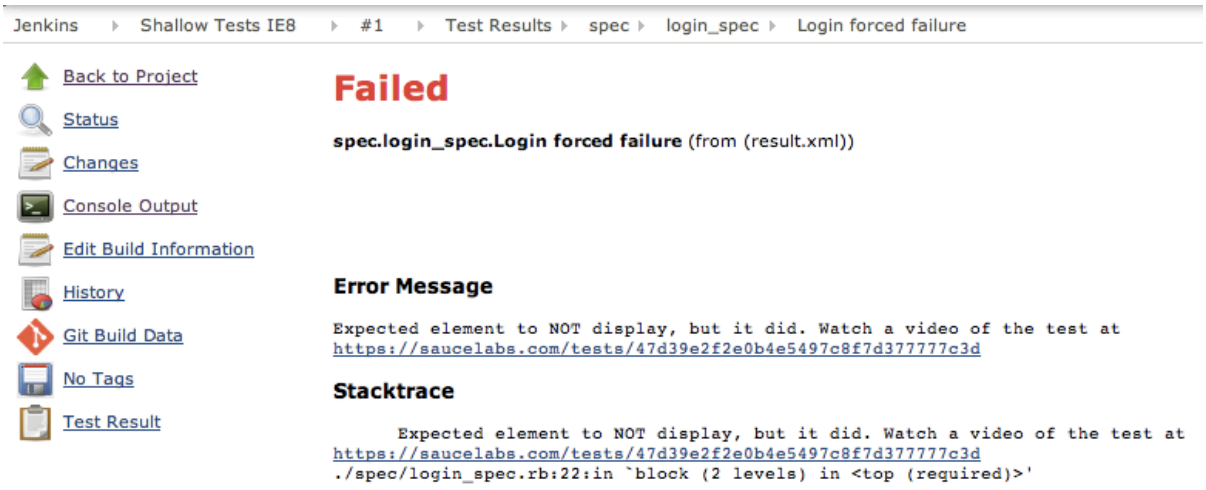


After you click `Save` you will be taken to the job dashboard.

Step 6: Run Tests & View The Results

1. Click `Build Now` from the menu on the left
2. Once the build completes, click on `Latest Test Result` (or, click on the build from `Build History` and then click `Test Result`)
3. Click on the failed test listed under `All Failed Tests`

You should see an error message that states why the test failed along with a link to the job in Sauce Labs. Click the link if you want to watch a video of the failed test.



Notifications

In order to make this job fully useful, we want to add notification to it. There are numerous ways to go about notifying team-members of failed builds (e.g., e-mail, chat, text, co-located visual cues, etc). Let's step through setting up a simple chat notification with HipChat.

Step 1: System Setup

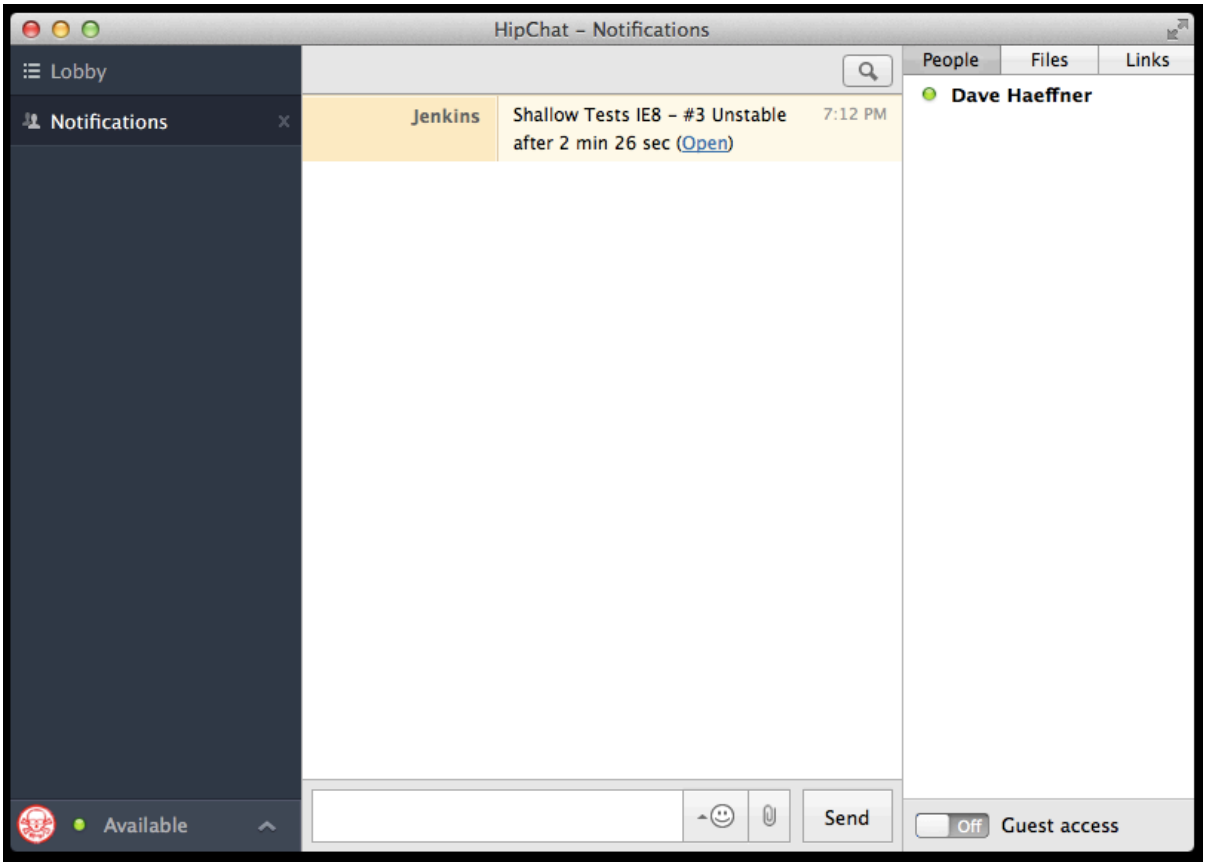
1. Install the HipChat Jenkins Plugin
2. Click on `Manage Jenkins` from the main dashboard
3. Click on `Configure System`
4. Scroll down to `Global HipChat Notifier Settings`
5. Input your HipChat API Token into the `API Token` input field (which you can get from your Admin API Tokens page)
6. Input the chat room you would like to send the notifications to in the `Room` input field (this can be the name of the room (case-sensitive) or the ID of the room)
7. Input the URL of your Jenkins server into the `Jenkins URL` input field (this will be used in the notifications)
8. Click `Save`

This will take you back to the main dashboard page.

Step 2: Configure Job Notifications

1. Click on the job you want to add chat notifications to (e.g., `Shallow Tests IE8`)
2. Click `Configure` (along the left side menu)
3. Scroll down to `Post-build Actions`
4. Click `Add post-build action`
5. Select `HipChat Notifications`
6. Click `Save`

Now when your tests run (either by manually triggering them, or from being triggered by another job) you will receive the end-result in the chat room you specified.



The only unfortunate thing about this configuration is that it will send notifications for both test passes and test failures regardless of the job's stability. So it may be a bit too noisy for your colleagues. If that's the case, then consider looking into e-mail notifications (which are also available in the `Post-build Actions` section).

Closing Thoughts

By using a CI Server you're able to put your tests to work by using computers for what they're good at -- freeing you up to focus on more important things. Keep in mind that there are numerous ways to configure your CI server. Be sure to find what works best for you and your team by testing and learning from a couple of different configurations. It's well worth the effort.

Chapter 17
Things You will See And How To Work With Them

The examples in this section aim to give you a glimpse at things you will see in the wild, and a reference for to deal with them.

They will not build upon previous examples like the earlier parts of this book. But instead, serve as standalone works that can be consumed individually -- and alphabetically.

Enjoy!

Chapter 17.1
A/B Testing

Sometimes when running tests you may hit unexpected behavior due to A/B testing being done in 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 tests.

The standard opt-out behavior we have seen with A/B testing platforms (both commercial and custom) tends to be either an appended URL that you can pass in when navigating to a page (e.g., '?optimizelyoptout=true') or forging a cookie (or some combination of the two).

An Example

For this example we are going to demonstrate how to opt-out of a page that is controlled by Optimizely (a popular A/B Testing platform) by forging a cookie.

On our example page, the heading text will change based on which test group we are place in.

You can see the example application we are testing here.

```
require 'selenium-webdriver'
require 'rspec-expectations'

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

def teardown
  @driver.quit
end

def run
  setup
  yield
  teardown
end
```

We start off the example by loading our libraries and preparing our setup, teardown, and run actions.

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

In our run action we navigate to the page we want to test. We confirm that we are in one of the A/B test cells by grabbing the heading text and seeing if it includes the proper content.

After that we add the opt-out cookie, refresh the page, and then confirm that we are no longer in the A/B test cells (again, by confirming that the heading text has changed).

```
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'
  @driver.find_element(css: 'h3').text.should == 'No A/B Test'
end
```

Alternatively we can go to the main page of the site first, add the opt-out cookie, and then navigate to the page of interest & perform our checks.

Expected Behavior

- Load a page on the site
- Add the opt-out cookie
- Access the page of interest
- Assert that the header text is correct

Chapter 17.2

Basic Auth

On some occasions you may work with apps that are secured with Basic HTTP Authentication. And in order to access them you will need to pass credentials to the server when requesting a page.

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

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

An Example

We can simply pass in the username and password in the URL that we are requesting (for both HTTP or HTTPS). Let's take a look at an example.

```
require 'selenium-webdriver'
require 'rspec-expectations'

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

def teardown
  @driver.quit
end

def run
  setup
  yield
  teardown
end
```

We start off by requiring our libraries and implementing setup, teardown, and run actions.

```
run do
  @driver.get 'http://admin:admin@the-internet.herokuapp.com/basic_auth'
  page_message = @driver.find_element(css: 'p').text
  page_message.should =~ /Congratulations!/
end
```

In the run action we are getting the page by passing in the username and password in the URL. Once it loads we grab text from the page, and make an assertion against it.

Alternatively, we could have accessed the page with credentials as part of the setup action and then accessed the page again (or any other HTTP page behind basic auth) without credentials.

Here's what that would look like.

```
require 'selenium-webdriver'
require 'rspec-expectations'

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: 'p').text
  page_message.should =~ /Congratulations!/
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. This happens because the browser can't use basic auth credentials interchangeably (e.g., HTTP for HTTPS and vice versa).

Expected Behavior

- Load the HTTP and/or HTTPS page caching basic auth credentials
- Load the page
- Get the page text
- Assert that it contains what it should

Chapter 17.3

Dropdown Lists

Selecting from a dropdown list seems like one of those simple things. Just grab the list by its element and select one of its items based on the text you want.

While it sounds pretty straightforward, there is a bit more finessing to it than you may realize.

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

An Example

NOTE: You can see the example app we are using to test here

```
require 'selenium-webdriver'
require 'rspec-expectations'

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

def teardown
  @driver.quit
end

def run
  setup
  yield
  teardown
end
```

First we load our libraries and wire up our setup, teardown, and run actions.

```
run {
  @driver.get 'http://the-internet.herokuapp.com/dropdown'
  drop_down_list = @driver.find_element(id: 'dropdown')

  options = drop_down_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
  selected_option.include?('Option 1').should == true
}
```

Next we go to our example application, find the dropdown list by its ID, and store it in a variable. From this variable we are able to tease out all of its available options to click on -- storing them in a variable as well.

Grabbing the list's options returns a collection that we then iterate over. When the text matches what we want, we click on it.

And to polish things off, we check to see if our selection was made correctly by reiterating over the dropdown options collection, getting the text of the one that was selected, returning its text, storing it in a variable, and making an assertion against it.

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

A Far Better Example

```
run {
  @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")
}
```

Similar to the first example, we are finding the dropdown list by ID. But instead of iterating over it and clicking based on a conditional, we are leveraging a built-in helper function of Selenium, Select, and its select_by action.

In addition to selecting by text, we can also select by value. Like so...

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

Expected Behavior

Under the hood both of these approaches follow the same basic behavior:

- Load the page
- Find the dropdown list
- Grab all elements in the list
- Iterate through them
- Click on the one that matches what we want

Chapter 17.4

File Download (Browser Specific)

Just like with file uploads, we hit the same issue with downloading them -- a dialog box just out of Selenium's reach.

Conventional wisdom posits that automating file downloads may be a foolish endeavor since in addition to the dialog box problem it is a deep rabbit hole to fall down. But here is a reasonable approach that will get you there without too much fuss.

A Solution

We are going to side-step the dialog box by telling our browser to ignore it and specify a location where to automatically download files to. Depending on the file's type, we will need to provide some additional configuration -- overriding some sensible defaults.

Once the file is downloaded we will make a quick and dirty assertion on it to see that it exists and is not empty.

Let's dig in with an example in Ruby using Selenium WebDriver against our test app.

An Example

First we will include the requisite libraries and wire-up our setup, teardown, and run actions in helper methods.

We are using the selenium-webdriver gem so we can demonstrate the browser profile configuration inline. And rspec-expectations is for our assertion action.

```
require 'selenium-webdriver'
require 'rspec-expectations'

def setup
  @download_dir = "/Users/more/Desktop/tmp/#{Time.now.strftime("%m%d%y_%H%M%S")}"
  profile = Selenium::WebDriver::Firefox::Profile.new
  profile['browser.download.folderList'] = 2
  profile['browser.download.dir'] = @download_dir
  profile['browser.helperApps.neverAsk.saveToDisk'] = 'image/jpeg, application/pdf'
  profile['pdfjs.disabled'] = true

  @driver = Selenium::WebDriver.for :firefox, :profile => profile
end

def teardown
  @driver.quit
  system("rm -rf #{@download_dir}")
end

def run
  setup
  yield
  teardown
end
```

The setup method is where the magic is happening. In it we are creating an instance variable and setting it to an absolute path for the directory we would like to use (e.g., a uniquely named temp folder that exists on my Desktop). This will come in handy later when we need to access the downloaded file and make an assertion against it.

Next we create a Firefox profile object and ply it with some configuration parameters:

- browser.download.folderList tells Firefox where to place its downloads. A 2 tells it to use a custom download path whereas a 1 is the browser's default path and a 0 is the Desktop.
- browser.download.dir is where we set the custom download path.
- browser.helperApps.neverAsk.saveToDisk tells Firefox when NOT to prompt us for a file download. It requires us to pass in a comma-separated string of MIME types that we care about. You can find a full list of MIME types here. For this example we have a PDF and JPG files to play with.
- pdfjs.disabled is for when downloading PDFs. This overrides the sensible default set in Firefox that previews them in the browser.

After setting these bits we create an instance of Selenium WebDriver and pass in the now configured profile object.

Our teardown and run methods are pretty straightforward. Teardown quits the Selenium object

when done and cleans up the temp directory with a system call. And we wrap everything up neatly into a single run method that we can pass a block of commands to.

Now that we have setup Firefox appropriately we move on to our Selenium and assertion actions.

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

  downloaded_files = Dir.glob("#{@download_dir}/**/*")
  sorted_downloads = downloaded_files.sort_by { |file| File.mtime(file) }
  File.size(sorted_downloads.last).should > 0
}
```

After loading the page we find the first link available and click it. Alternatively, we could have clicked the second link

```
@driver.find_elements(css: 'a')[1].click
```

...or we could have clicked all of the links.

```
download_links = @driver.find_elements(css: 'a')
  download_links.each do |download_link|
    download_link.attribute('href').click
  end
```

All of these actions trigger an automatic download to the folder specified in setup. So we then check to see that a file is there and not empty.

This is done by getting a list of the files stored in the directory and sorting it from oldest to newest based on its modified date timestamp. This ensures that the last item in the list is the newest one.

Once we have it, we check its file size to make sure that it is greater than 0.

Expected Behavior

- Load the page
- Find the first download link
- Click it
- Download the file to a specified location on disk without prompting
- Check that the downloaded file is there and not empty

Outro

It's worth noting that the same approach can be applied to some browsers (e.g., Chrome) with a slightly different setup configuration, but not others (e.g., Internet Explorer). This is why I prefer to leverage a browser agnostic approach.

Chapter 17.5

File Download (browser agnostic)

While downloading files with a custom browser configuration works it is not portable from browser to browser.

And ultimately we shouldn't care if the browser can download something or not. Instead we should care that the file was downloaded and that it is of the correct type and size -- a subtle but important distinction.

Alongside our Selenium actions we can leverage an HTTP library to initiate a download and inspect the response header to confirm it is what we expect. And all this without actually downloading the file to disk. Such an approach enables us to keep with Selenium, but decouples us from a browser specific configuration.

Let's dig with an example.

An Example

In the first part of this example we take care of including our third-party libraries and setting up some helper methods (e.g., setup, teardown, and run).

The only thing that is different from most of our previous examples is that we have included the `rest-client` gem to handle the mechanics of our HTTP interactions for downloading files and viewing their response headers.

```
require 'selenium-webdriver'
require 'rspec-expectations'
require 'rest-client'

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

def teardown
  @driver.quit
end

def run
  setup
  yield
  teardown
end
```

In our run acton we use Selenium to load the page that has our download links and grab the URL for the first one listed.

We then use the `rest-client` gem to initiate the download and store its response into a variable called `response`.

From this object we are able to access the headers and perform checks against it. Primarily focusing on the `content_type` and `content_length` portions. For this example we are confirming that the file to be downloaded is an `image/jpeg` and that its contents are not empty.

```
run {
  @driver.get 'http://the-internet.herokuapp.com/download'
  link = @driver.find_element(css: 'a').attribute('href')
  response = RestClient.get link
  response.headers[:content_type].should == 'image/jpeg'
  response.headers[:content_length].to_i.should > 0
}
```

Expected Behavior

- Load the page
- Get the URL of the first download link
- Perform a GET action against the URL with an HTTP library
- Store the response
- Assert that the file type is correct
- Assert that the file is not empty

Chapter 17.6

File Upload

Normally when interacting with forms to upload a file you're prompted with a dialog box that is just out of reach for Selenium.

In these cases, people tend to turn to a tool to drive the GUI, such as Auto-IT. While this helps solve short-term pain, it sets you up for long term hurt later by chaining you to a specific platform -- limiting your ability to effectively test different browser/OS combinations.

A work-around for this problem is to side-step the dialog box experience entirely, and rather, inject the path of the file you want to upload into the form element, and then submit the form.

Let's break down this approach with some example Ruby code using Selenium WebDriver Remote and an example file upload form.

An Example

You can see the application we are running against here.

For our test script, first we'll include the requisite libraries and wire-up our setup and teardown actions in helper methods.

```
require 'selenium-webdriver'
require 'rspec-expectations'

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

def teardown
  @driver.quit
end
```

We are using the selenium-webdriver gem to drive the browser, and rspec-expectations for our assertion action (which is a subset of the RSpec library).

```
def run
  setup
  yield
  teardown
end

run do
  @driver.get 'http://the-internet.herokuapp.com/upload'
  uploader = @driver.find_element(id: 'file-upload')
  uploader.send_keys Dir.pwd + '/DaveHaeffner.jpg'
  uploader.submit

  uploaded_image = @driver.find_element(css: '.example img').attribute('src')
  uploaded_image.should =~ /DaveHaeffner.jpg/
end
```

We created a helper method called 'run' to easily call our setup and teardown actions without being overly verbose.

In our run block we are driving the test script using standard Selenium commands and ending things off with an RSpec assertion.

Expected Behavior

So the intended behavior is:

- Load the page
- Grab the form element we want to work with
- Inject the path to the file we want to upload
- Submit the form
- Grab the source path for the image once it renders in the browser
- Check to see that the image path contains the proper filename

Chapter 17.7
Frames

More often than not you'll run into a throwback relic of the front-end world -- in this case, frames. Rather than curse and spit when it comes to writing automated tests against them, there is a simple approach we can use to tap dance through them.

In order to access the elements within a frame we need to tell Selenium to switch to it.

Let's demonstrate this with a couple of test scripts.

An Example

Our general script setup is pretty straightforward. Nothing special -- just the standard setup, teardown, and run methods.

```
require 'selenium-webdriver'
require 'rspec-expectations'

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

def teardown
  @driver.quit
end

def run
  setup
  yield
  teardown
end
```

Our first example demonstrates how to deal with nested frames. You can see the page we are testing against here. But here is a simple representation of it:

```
| - frameset
    | - top frame
        | - frameset
            | - left
            | - middle
            | - right
    | - bottom frame
```

```
# Nested Frames
run {
  @driver.get 'http://the-internet.herokuapp.com/frames'
  @driver.switch_to.frame('frame-top')
    @driver.switch_to.frame('frame-middle')
      @driver.find_element(id: 'content').text.should =~ /MIDDLE/
}
```

In order to get the text of the middle frame, we need to switch to it. But in order for Selenium to see it, we need to switch to the parent frame first, then the child frame.

Once we've done that, we are able to freely find the element we want and assert that the relevant text appears.

While this example helps illustrate the point of frame switching, it is not very practical. So here is a more likely use case you'll run into -- working with the TinyMCE Editor.

Another Example

You can find the page we are testing against here.

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

  after_text.should_not == before_text
}
```

Once the page renders we switch into the TinyMCE frame. While the frame element does not have a `name` attribute it does have an `id`. This works just the same in the `switch_to.frame` call, so we

use it.

Now that Selenium is properly scoped we are able to use the top-level id of the TinyMCE widget to:

- grab its text
- clear its text
- input new text
- grab the altered text
- assert that the before and after 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, then we will need to switch to it. For instance...

```
@driver.switch_to.default_content
@driver.find_element(css: 'h3').text.should_not == ""
```

This switches us back to the main page and enables us to assert that the heading on the page is there.

Expected Behavior

- Load the page
- Switch to the frame you want
- Find and take an action against an element
- Switch to another frame or the main document if need-be
- Assert that the desired outcome has occurred

Chapter 17.8

HTTP Status Codes

There are times when you are testing a page (or numerous pages) and you want to verify that it responded correctly. A great way to handle this is by checking the HTTP Status Code that the browser received. But this functionality is not available in Selenium WebDriver.

But fear not because there is more than one way to fall down this deep rabbit hole.

The tried and true approach that Selenium Committers and Practitioners recommend is to use a proxy server. With it, we will be able to watch and manipulate network traffic between our tests and the application under test -- giving us access to a whole host of functionality that we wouldn't have otherwise had.

Let's step through an example using BrowserMob Proxy.

An Example

First we have to download BrowserMob Proxy to our working directory (available on their website). With that in hand, we are ready to wire up our script.

At the top of the file we include the libraries we need -- `selenium-webdriver` to drive the browser, `rspec-expectations` for our assertions, and `browsermob/proxy` to use the proxy server. You can find more info on the `browsermob-proxy` gem we are using here.

```
require 'selenium-webdriver'
require 'rspec-expectations'
require 'browsermob/proxy'
```

Next we create a `configure_proxy` method to prepare the proxy for use with Selenium. In it we start the proxy server and create a configurable instance of it. We then create a browser profile (Firefox in this case) and set it to use the proxy server.

Once that's done, we wire up the `configure_proxy` method into the setup action, and pass in the profile object to our Selenium instance.

```
def configure_proxy
  server = BrowserMob::Proxy::Server.new('./browsermob-proxy/bin/browsermob-proxy')
  server.start
  @proxy = server.create_proxy
  @profile = Selenium::WebDriver::Firefox::Profile.new
  @profile.proxy = @proxy.selenium_proxy
end

def setup
  configure_proxy
  @driver = Selenium::WebDriver.for :firefox, :profile => @profile
end
```

After that, we wire up our teardown and run actions.

```
def teardown
  @driver.quit
  @proxy.close
end

def run
  setup
  yield
  teardown
end
```

And to make things clean we wrap up our proxy calls to get the status code from the HTTP Archive (a.k.a. HAR) in a `visit` action that returns it after it visits the page.

```
def visit(url)
  @proxy.new_har
  @driver.get url
  @proxy.har.entries.first.response.status
end
```

All that's left is our run action, which is simple and to the point.

```
run do
  status_code = visit 'http://the-internet.herokuapp.com/status_codes/404'
  status_code.should == 404
end
```

Expected Behavior

- Visit the URL
- Retrieve the HTTP Status Code
- Assert that it is the correct status code

Outro

This was inspired by Jim Evans' multi-part blog post series on doing the same thing in C# with Fiddler (1, 2, 3) which was in response to Selenium Issue 141. Thanks Jim!

Chapter 17.9
Mobile

Mobile is kind of a big deal. And if you haven't needed to test a mobile app yet, it's only a matter of time. But where do you start? And how do you do it in a way that is complimentary to your exisitng automation and experience?

Give Appium a try. It's open source, works for both iPhone & Android, and works through the use of the Selenium WebDriver API (without having to recompile the app in order to test it).

That is to say -- you can write tests pretty much the same way as you do now, there are just some aditional things to think about -- like new functionality (e.g., pinch, swipe, etc.) and setup (e.g., serving your app, running simulators and a server locally, or sending things off to the cloud).

To simplify things, let's step through 2 examples using Sauce Labs to test native apps -- one for iPhone and another for Android.

NOTE: These examples were borrowed from the Appium project and modified for simplicity.

A Testing Native Apps In Sauce Labs Primer

In order to test your app in Sauce Labs you will need to zip it up and serve it somehow. Here are some options on how to do that:

- make it publically available (e.g., from AWS)
- use Sauce Labs' temporary storage
- use Sauce Connect and grab it from a machine behind your firewall

Once you have that you just need to set your test's desired capabilities accordingly. You can see a list of options here.

An iPhone Example

In this example we will be testing an application that has a form that takes 2 numbers, adds them together, and displays the result.

NOTE: Contrary to the Appium philosophy of not needing to recompile your app, there is one SMALL tweak you'll need to make before your app is ready for Sauce (if you care about testing against an iPhone). It's to make sure that the correct simulator gets loaded. Instructions available here.

We kick off our iPhone example by requiring our dependent libraries (`selenium-webdriver` to drive the browser, and `rspec-expectations` for handling assertions). After that, we wire up our

setup, teardown, and run methods.

```
require 'selenium-webdriver'
require 'rspec-expectations'

def setup
  caps = {
    'platform'            => 'Mac 10.8',
    'version'             => '6.1',
    'device'              => 'iPhone Simulator',
    'app'                 => 'http://appium.s3.amazonaws.com/TestApp6.0.app.zip',
    'name'                => 'Ruby/iPhone Example for Appium'
  }

  @driver = Selenium::WebDriver.for(
    :remote,
    :url => "http://SAUCE_USERNAME:SAUCE_API_KEY@ondemand.saucelabs.com:80/wd/hub",
    :desired_capabilities => caps)
end

def teardown
  @driver.quit
end

def run
  setup
  yield
  teardown
end
```

Note that in `setup` we configure our capabilities by setting them to a hash and storing them in a variable called `caps`. In it we are specifying the platform we would like (`Mac 10.8`), the iOS version (`6.1`), the device (`iPhone Simulator`), and the app we would like to load (zipped up and served from AWS).

We then instantiate Selenium Remote in a `@driver` object; pointing it at Sauce Labs, and passing in our desired capabilities.

And in our test we wire up interaction with the app using familiar Selenium WebDriver API actions and an assertion.

```
run do
    values = [rand(10), rand(10)]
    expected_sum = values.reduce(&:+)

    elements = @driver.find_elements(:tag_name, 'textField')
    elements.each_with_index do |element, index|
      element.send_keys values[index]
    end

    @driver.find_element(:tag_name, 'button').click
    @driver.find_element(:tag_name, 'staticText').text.should == expected_sum.to_s
end
```

Expected Behavior

- Open a job with Sauce Labs
- Load an iPhone simulator on a Mac
- Load the test app
- Input two random numbers
- Sum them them together
- Assert that they are correctly added together

NOTE: You can see the test result complete with video here.

An Android Example

In this example we will be testing a simple note taking app.

We kick off our Android example by requiring our dependent libraries (`selenium-webdriver` to drive the browser, and `rspec-expectations` for handling assertions). After that, we wire up our setup, teardown, and run methods.

```
require 'selenium-webdriver'
require 'rspec-expectations'

def setup
  caps = {
    'platform'            => 'Linux',
    'version'             => '4.2',
    'device'              => 'Android',
    'app'                 => 'http://appium.s3.amazonaws.com/NotesList.apk',
    'app-package'         => 'com.example.android.notepad',
    'app-activity'        => '.NotesList',
    'name'                => 'Ruby/Android Example for Appium'
  }

  @driver = Selenium::WebDriver.for(
    :remote,
    :url => "http://SAUCE_USERNAME:SAUCE_API_KEY@ondemand.saucelabs.com:80/wd/hub",
    :desired_capabilities => caps)
end

def teardown
  @driver.quit
end

def run
  setup
  yield
  teardown
end
```

In `setup`, we configure our capabilities by setting them to a hash and storing them in a variable called `caps`. In it we are specifying the platform (`Linux`), the Android version (`4.2`), the device (`Android`), and where to find the app (zipped and served from AWS). We then instantiate Selenium Remote in a `@driver` object; pointing it at Sauce Labs, and passing in our desired capabilities.

The additional bits of info we need to provide in `caps` (versus in our iPhone example) are `app-package` and `app-activity`. `app-package` being the package name of your app, and `app-activity` being the name of the activity you would like to start. `app-activity` gets appended to `app-package` and is used at Sauce run time to launch your app (e.g., `com.example.android.notepad.NotesList`).

And in our test we wire up interaction with the app using familiar Selenium WebDriver API actions and an assertion.

```
run do
  @driver.find_element(:name, 'New note').click
  @driver.find_element(:tag_name, 'textfield').send_keys 'This is a new note, from
Ruby'
  @driver.find_element(:name, 'Save').click

  notes = @driver.find_elements(:tag_name, 'text')
  notes[2].text.should == 'This is a new note, from Ruby'
end
```

Expected Behavior

- Open a job with Sauce Labs
- Load an Android simulator on a Linux machine
- Load the test app
- Create a new note
- Fill in the new note
- Grab the note text and assert that it's what we inputted

NOTE: You can see the test result complete with video here.

Other Helpful Tidbits

Inspecting Elements

In order to easily inspect and identify the elements in your app, you can use Appium's Inspector which comes in appium-dot-app and requires you to set up Appium locally.

NOTE: You can see a demo of it in action with an iOS app here.

Mobile Web Apps

If instead you have a mobile web app, you can launch mobile Safari by switching out the URL in `app` with `safari`, like so:

```
caps = {
    'platform'            => 'Mac 10.8',
    'version'             => '6.1',
    'device'              => 'iPhone Simulator',
    'app'                 => 'safari',
    'name'                => 'Ruby/iPhone Example for Appium',
  }
```

But it's worth noting that Sauce Labs does not currently support the ability to launch a mobile

browser on Android (e.g., `chrome`). This functionality is, however, available when running Appium locally (as documented here).

Hybrid Apps

If you have a hybrid mobile app (e.g., an app with a native container that leverages the built in browser engine), then you should be good to go both locally and in Sauce. For details, go here.

Location Services

If you have an iOS app that deals with geolocation, then working with the Location Services dialog can be a little tricky.

You can either do a blanket dismiss (which requires some elbow grease -- see this Appium Google Groups thread for details), or capture and dismiss the dialog as it appears (using functionality found in a library like ruby_lib).

Sauce Labs now supports Location Services (as noted in this support thread comment). You can set it in your desired capabilities with `locationServicesEnabled` and a `true` or `false` value.

Outro

In order to take full advantage of what Appium has to offer, getting it set up locally is key. I encourage you to check out the installation docs (available for OS X, Linux, and Windows).

And if you're curious about other mobile automation frameworks, Alister Scott has a good breakdown of them.

Chapter 17.10

Multiple Windows

Occasionally you will run into a link or action on a site that opens a new window. In order to work within this new window, or the originating one, you will need to switch between them.

On the face of it, this is a pretty straightforward thing. But lurking within it is a small gotcha to watch out for.

Let's step through it.

An Example

NOTE: You can find a link to our example application here.

```
require 'selenium-webdriver'
require 'rspec-expectations'

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

def teardown
  @driver.quit
end

def run
  setup
  yield
  teardown
end
```

Nothing fancy here. Just loading the libraries necessary to run Selenium Webdriver and perform our assertions. That and wiring up some simple setup, teardown, and run actions to keep things clean.

```
run {
  @driver.get 'http://the-internet.herokuapp.com/windows'
  @driver.find_element(css: '.example a').click
  @driver.switch_to.window(@driver.window_handles.first)
  @driver.title.should_not =~ /New Window/
  @driver.switch_to.window(@driver.window_handles.last)
  @driver.title.should =~ /New Window/
}
```

After loading the page and clicking the link (which spawns a new window) we grab the window handles (a.k.a. non-sensical identifier strings which represent each open browser window) and traverse between them based on the order of the list -- making assertions based on each page's title.

While this may feel intuitively correct, this approach is going to bite you later.

The order of the window handles is not consistent across all browsers. Some return in the order opened, others alphabetically.

Here's a more effective approach.

A Better Example

```
run {
  @driver.get 'http://the-internet.herokuapp.com/windows'
  main_window = @driver.window_handle
  @driver.find_element(css: '.example a').click
  windows = @driver.window_handles
  windows.each do |window|
    if main_window != window
      @new_window = window
    end
  end
  @driver.switch_to.window(main_window)
  @driver.title.should_not =~ /New Window/
  @driver.switch_to.window(@new_window)
  @driver.title.should =~ /New Window/
}
```

After we load the page, but before we click the new window link, we get the window handle for the main window. Then we click the link and get the window handles for both open windows.

Since this returns an array we iterate through it and find the window handle that is different -- a.k.a. the new window -- and store its value.

Once we have that it's all down hill from there. We're able to switch between the main window and the new window and assert that the page title contains the correct text.

Expected Behavior

- Load the page
- Get the window handle for the current window
- Take an action that opens a new window
- Get the window handle for the new window
- Switch between the windows as needed

Chapter 17.11

Secure File Downloads

In a previous chapter we covered how to write tests to download files in a browser agnostic way by leveraging Selenium Webdriver and an HTTP library in tandem. This approach is great, but there are often times where the file you want to download is behind authentication, presenting a hurdle to overcome.

In order to access secure files with an HTTP library, we want to pull the authenticated session information out of Selenium's cookie store and pass it into the HTTP library when we perform the download action.

Let's dig in with an example.

An Example

We start by requiring our libraries (Selenium to drive the browser, RSpec for our assertions, and RestClient for our download action) and wire up our setup, teardown, and run actions.

```
require 'selenium-webdriver'
require 'rspec-expectations'
require 'rest-client'

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

def teardown
  @driver.quit
end

def run
  setup
  yield
  teardown
end
```

Next we access a page of download links that is behind Basic HTTP Authentication, grab the first download link, and pull the authentication session cookie from the cookie store. Once we have that we fire up RestClient and perform a HEAD request against the download link with the cookie information and then check the headers to make sure it has the correct type and is not empty.

```
run do
  @driver.get 'http://admin:admin@the-internet.herokuapp.com/download_secure'
  link = @driver.find_element(css: 'a').attribute('href')
  cookie = @driver.manage.cookie_named 'rack.session'
  response = RestClient.head link, cookie: "#{cookie[:name]}=#{cookie[:value]};"
  response.headers[:content_type].should == 'application/pdf'
  response.headers[:content_length].to_i.should > 0
end
```

In order for things to work correctly with RestClient we have to stringify the cookie values (due to an open issue in the library). But your cookie configuration will likely vary depending on your HTTP library of choice.

And note that we are using a HEAD request instead of a GET request. Since we only care about the header information this will perform a partial fetch of data, avoiding a full download of the file.

Expected Behavior

- Load the page
- Get the first download link
- Pull the authenticated cookie information
- Use it to perform a HEAD request against the download link
- Check the headers to make sure the file is the correct type and not empty

Outro

While this example demonstrates accessing files behind Basic HTTP Authentication it should also work with form-based authentication.

Chapter 17.12
Tables

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, enabling them to sort and manipulate it. Depending on your application it can be quite common and something you will want to write automation for.

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

All hope is not lost because you can traverse the table through the use of CSS Pseudo-classes.

But keep in mind that if you care about older browsers, then this approach won't work on them. In those cases it's better to get something working in the short term and file a fix with your front-end developer(s) to update the table with helpful attributes.

A Tables & CSS Pseudo-classes Primer

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 represented as...

- `<th>` in a header
- `<td>` in a body (also known as 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 couple it with Selenium to grab all instances of a target (e.g., the third `<td>` cell from all rows in the table body) -- which gives us all of the data for a specific column.

Let's dig in with an example for a common set of table functionality -- sorting columns in ascending and descending order.

An Example

NOTE: You can see the application under test here. There are 2 tables. We will start with Example 1 and then work with Example 2

We kick things off by requiring our dependent libraries (`selenium-webriver` to drive the browser and `rspec-expectations` to handle our assertions) and wiring up our `setup`, `teardown`, and `run` actions.

```
require 'selenium-webdriver'
require 'rspec-expectations'

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

def teardown
  @driver.quit
end

def run
  setup
  yield
  teardown
end
```

Here is a snippet from the first table we are 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. The first click should sort them in ascending order, the second click in descending order.

And there is a small sampling of data in the table to work with (4 rows worth), so we should be able to sort and confirm that it actually worked. So let's do that -- starting with the `Due` column.

```
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.each { |due| due_values << due.text.gsub(/\$/,'').to_f }

  (due_values == due_values.sort).should == true
```

After navigating to 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)`) which is effectively saying 'give me the 4th `th` element in the table heading' which is the placement of `Due` .

We then use another pseudo-class to find all of the `td` elements found within each row of the table body that are `Due`s -- which is effectively the 4th column.

Once we have them we create an array and iterate over the collection of `td` elements we found, stripping off the dollar sign, converting them into a decimal number, and shoveling them into the array. With this array of numbers we can then compare it against a sorted version of itself to make sure that they match. If they do then the table data on the page was sorted in ascending order.

If we click the `Due` heading again it should sort in descending order. The code for that is identical except for the assertion which is checking the same thing but seeing if it's false.

```
@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.each { |due| due_values << due.text.gsub(/\$/,'').to_i }

  (due_values == due_values.sort).should == false
```

We can then expand our sort check to a different column and see that it gets sorted correctly. So let's focus on the `Email` column.

```
@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.each { |email| email_values << email.text }

  (email_values == email_values.sort).should == true
end
```

The mechanism for this is the same except that we are not manipulating the text before checking it. We are just doing a straight comparision. This is because Ruby will sort a collection of numbers that are represented as strings in descending order. Whereas all other comparisons will sort in ascending order.

But What About Older Browsers?

So, now we have a working example that will load the page and check sorting three different ways. Great! But as soon as we run this againt an older browser (like Internet Explorer 8) it will throw an exception stating `Unable to find element`. This is because older browsers don't support CSS Pseudo-classes.

You've already come a long way, so it's best to get value out of what you've just written. You can

run these tests on current browsers, file a fix with your front-end developers to improve the table design with some `class` attributes, and then update these tests once they've made the necessary changes.

Here is a snippet of what our original table would look like with helpful attributes added in.

```
<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 that, the selectors for our sorting tests become a lot simpler and more expressive. Here's an example of the `Due` ascending test being written against it.

```
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.each { |due| due_values << due.text.gsub(/\$/,'').to_f }
  (due_values == due_values.sort).should == true
end
```

Not only will these selectors work in both current and older browsers, but they are more resilient to changes in the table column order since they're not hard-coded values any more.

Expected Behavior

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

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 an automation strategy that rules out older browsers).

For more info on CSS Pseudo-classes you should check out a nice write-up by Sauce Labs. And maybe the W3C spec, but that may be a bit more of a rabbit hole than you may want to fall down.

And for a more in-depth walk-through on HTML Table design you can check out Treehouse's write-up.

Chapter 17.13

Upgrading From Selenium RC

As the Selenium project continues to evolve, Selenium RC is moving closer to end-of-life. This is especially true with the upcoming release of Selenium 3 where minimal support will be offered for RC (details of which you can learn more about from a recent Selenium Hangout).

But how do you approach an upgrade like this? Especially if you have an enormous amount of tests that you rely on?

A Solution

Take Jason Leyba's advice. He works at Google and oversaw their transition from Selenium RC to WebDriver. And he posits that it can be done by following 4 simple steps:

1. Clean up your tests
2. Swap in WebDriver-backed Selenium
3. Use WebDriver for all new features
4. Replace RC usage as tests break

To give you some context, Google's Selenium tests spawn something on the order of 3 million unique browser sessions per day. And they were able to make this transition happen with minimal disruption to this. If they were able to make this happen at such a scale, then hopefully this serves as inspiration for you to do the same at your organization.

NOTE: Jason discussed this in detail in his keynote at Selenium Conf 2013. His talking points and code examples are recapped below. You can view his full talk here.

4 Simple Steps

1. Clean Up Your Tests

> "Turns out that tests that are easy to maintain are tests that are easy to migrate." -- Jason Leyba

The best thing you can do to keep your tests clean is to practice good abstractions. In the simplest form this means not referencing Selenium commands directly in your tests -- pulling these out into something like page objects instead.

2. Swap In WebDriver-backed Selenium

With WebDriver-backed Selenium you get two driver instances to use in your tests -- one for

Selenium RC and another for WebDriver. This enables you to keep your Selenium RC tests running while simultaneously building out WebDriver functionality as you transition things over.

You can see an example of what this looks like here.

3. & 4. Improve Things Incrementally

For every new feature you write tests for, use WebDriver. And as your Selenium RC tests break go back and replace the broken bits with WebDriver.

Chip away at this over time and before you know it you'll be upgraded!

Common Pitfalls

There are a handfull of pitfalls to be cognizant of when stepping through this upgrade.

Alert Handling

WebDriver handles JavaScript alerts in a fundamentally different and incompatible way between Selenium RC and WebDriver that won't jibe in WebDriver-backed Selenium.

The best way to address this is to port your alert handling from Selenium RC to WebDriver -- which is pretty straightforward and should be easy to accomplish assuming you have things abstracted (re: Step #1 above).

Here is an example of the difference:

```
// Selenium RC
selenium.chooseCancelOnNextConfirmation();
selenium.click("id=foo");

// WebDriver
webdriver.findElement(By.id("foo")).click();
webdriver.switchTo().alert().dismiss();
```

Waiting For The Page To Load

There's no need to explicitly wait for the page to load anymore since WebDriver is implicitly doing this. Save your tests some time and remove it. It's also not a bad idea to explicitly wait for the element you want to interact with.

Here's an example:

```
// Selenium RC
selenium.open("http://www.google.com");
selenium.waitForPageToLoad();
selenium.typeKeys("name=q", "bears");

// WebDriver
selenium.open("http://www.google.com");
selenium.typeKeys("name=q", "bears");
```

Getting Text From The Page

It's an expensive operation (and overkill) to get ALL of the text from the page to make a verification. And it takes longer to execute this in WebDriver.

A better approach is to find the element that has the text you want and check its text instead. Alternatively, if you want to search the entire page, it's better to get the page's source and parse that.

```
String text = selenium.getBodyText();
assertTrue(text.contains("hello"));

WebElement body = webdriver.findElement(
     By.tagName("body"));
String text = body.getText();
assertTrue(text.contains("hello"));
```

Be Careful With JavaScript

WebDriver-backed Selenium is not built for performance, it's meant to be a transitional tool. So things will run slower in it. The worst offender being issuing JavaScript commands directly to the Selenium Core API. So do your best to avoid things like this.

```
selenium.getEval(
     "selenium.isElementPresent('id=foo') && " +
     "selenium.isVisible('id=foo')");
```

Inertia

It's worth noting that the hardest part of Google's transition wasn't technical -- it was inertia. Teams were really slow to adopt WebDriver -- even when the tools were readily available, well documented, and easy to use. Jason and his team were able to perservere and succeed through various means (e.g carrot, stick, elbow grease, etc.), but this really slowed their progress.

If you find yourself in a similar situation at your organization, it's worth giving Jason's talk a watch

for inspiration.

Chapter 18

Finding Information On Your Own

There is information all around us when it comes to Selenium. But it it can be challenging to sift through it, or know where to look.

Here is a list breaking down a majority of the Selenium resources available, and what they're useful for.

Documentation & Tips

- Selenium HQ

This is the official Selenium project documentation site. It's a bit dated, but there is loads of helpful information here. You just have to get the hang of how the navigate site to find what you need.

- The Selenium Wiki

This is where all the good stuff is -- mainly, documentation about the various language bindings and browser drivers. If you're not already familiar with it, take a look.

- Elemental Selenium Archives

Every tip I've written is freely available on the tips archive page. There are over 35 different Selenium problems and solutions covered.

Blogs

- The official Selenium blog

This is where news of the Selenium project gets announced, and there's also the occasional round-up of what's going on in the tech space (as it relates to testing). Definitely worth a look.

- A list of "all" Selenium WebDriver blogs

At some point, someone rounded up a large list of blogs from Selenium practitioners and committers. It's a pretty good list.

Other Books

- Selenium 2 Testing Tools

This book is by David Burns and it is a tremendous and thorough resource. It outlines how to use Selenium, top-to-bottom. It's definitely worth having on your shelf.

- Selenium Testing Tools Cookbook

This is another good book outlining loads of great ways to leverage Selenium. While I haven't had a chance to finish reading it, it's clear that Gundecha has a very pragmatic approach that will yield great results.

Meetups

- The Selenium Hangout

This is an entirely online meetup that is run by me and David Burns where we talk about the latest in the Selenium community (e.g., the upcoming conference, the status of Selenium 3, etc.) and answer questions from the community. Videos are recorded and made available (along with notes) on the official Selenium blog.

- All Selenium Meetups listed on Meetup.com

A listing of all in-person Selenium Meetups are available on Meetup.com. If you're near a major city, odds are there's one waiting for you.

- How to start your own Selenium Meetup

If there's not a Selenium Meetup near you, start one! Sauce Labs has a great write up on how to do it.

Conferences

- Selenium Conf

This is a tremendous annual conference where practitioners and committers gather and share knowledge. All of the talks are top notch, as is the hallway track. I highly recommend it.

- Selenium Camp

This is an annual Selenium conference in Europe. I haven't been yet, but it sounds like a good event.

- List of other testing conferences

Alister Scott has a good write-up of the testing conferences coming up this year.

Videos

- Selenium Conference Talks

All of the talks from The Selenium Conference are recorded and made freely available online. This is a tremendous resource.

- Selenium Meetup Talks

Some of the Selenium Meetups make it a point to record their talks and publish them afterwards. Here are some of them. They are a great way to see what other people are doing and pick up some new tips.

- Selenium Hangout

All of the Selenium Hangout Meetups are recorded and made available. A lot of great stuff is discussed in them.

Mailing Lists

- Selenium Users Google Group
- Selenium LinkedIn Users Group

The signal to noise ratio in these groups can be challenging at times. But you can occasionally find some answers to your questions.

- Agile Testing Yahoo Group

This list is less about Selenium and more about "testing" (in the checking versus testing sense). It's a great place to ask for sage advice, and to hang out and listen. You will learn some valuable insights here.

Forums

- Stack Overflow
- Quora

These are the usual forums where you can go looking for answers to questions you're facing (in addition to the mailing lists above).

Issues

- Selenium Issue Tracker

If you're running into a specific and repeatable issue that just doesn't make sense, you may have found a bug in Selenium. You'll want to check the Selenium Issue Tracker to see if it has already been reported. If not, then create a new issue. But be sure to read this post before you do (so you can be sure that you have provided enough information).

Chatting With the Selenium Community on IRC

The Selenium IRC Chat Channel is arguably the best way to connect with the Selenium community and get questions answered. This is where committers and practitioners hang out day-in and day-out. Let's step through what it is and how to access it.

Brief Intro To IRC

IRC (short for Internet Relay Chat) is a protocol that freely enables live chatting (both in groups and person to person) and file sharing. It's been around for a while (circa 1988) and is the preferred mode of communication among certain tech circles.

Within the realm of IRC there are numerous servers you can connect to. Each one containing people and bots logged in and joined to one or more chat rooms talking, sharing files, etc.

One of the beautiful things about IRC is that there is no sign-up required to join the party. You just need to download a client that handles the IRC protocol (there's at least one available for every platform), point it at a server, and pick a unique nickname for yourself on that server.

Once you're on you can join a chat room and start jib-jabbing.

How To Get Connected

Step 1: Get An IRC Chat Client

First thing's first, get a chat client that supports IRC.

You may already have one and not even know. For example, Adium (for OSX) supports a staggering number of chat protocols. If you already have it (or something like it) then use that to connect. If you don't, then you'll need to grab one (or one that is built specifically for IRC).

Here are some worthwhile IRC chat clients (broken out by platform -- and are free unless otherwise noted).

OSX

- LimeChat
- Textual ($4.99 to buy)

Windows

- mIRC (free for 30 days, $20 for a single-user license)

Linux

- Irssi

Web-only

- Webchat
- IRCCloud (free with paid tier as well, Android and iOS apps available as well)

Step 2: Connect To The Proper Server

The Selenium chat channel lives on the Freenode server. To connect to it directly you would use `irc.freenode.net`.

Before connecting you should be able to set a nickname and perhaps even specify which channel you would like to connect to after connecting (e.g., `#selenium`). If you don't see these bits, don't sweat it. Connect and proceed to the next step.

Note: If you're using Webchat, it will automatically connect you to Freenode.

Step 3: Join The Chat Channel

In IRC parlance channels are prepended with a `#` and are lower-case. So the Selenium channel is `#selenium`.

If you were able to configure your chat client to join the channel for you, then proceed to the next step. Otherwise, you'll need to issue a command in the status window. There are a series of commands you can issue in IRC. They are all prepended with a `/`. To join a chat channel type `/join #selenium` and press `Enter`.

This should open a new window in the app and in it should be the Selenium chat channel. Woohoo!

Step 4: Talk And Hangout

Feel free to say hello and introduce yourself. But more importantly, ask your question. And if it looks like no one is chatting, ask it anyway. Someone will see it and eventually respond. They always do.

In order to get your answer, you'll probably want to hang around for a bit. But the benefit of being a fly on the wall is that you quickly gain insight into other problems people face, possible solutions, and the current status of the Selenium project and its various pieces.

Chapter 19
Now You Are Ready

The journey for doing Selenium successfully can be long and arduous. But by adhering to the principals in this book, you will avoid a majority of the pitfalls around you. You're also in a better position now -- armed with all of the information necessary to continue your Selenium journey.

You are ready. Keep going, and best of luck!

And if you have any questions, feedback, or want help -- get in touch!

- E-mail: dhaeffner@gmail.com
- Twitter: @TourDeDave
- Office hours: SoHelpful.me/TourDeDave

My office hours are free (read: completely free -- no strings attached). You can grab a 30-minute time slot to hop on a Skype or Google Hangout call with me to talk about testing. And if we're both at a conference or meetup, feel free to stop me and say hello!