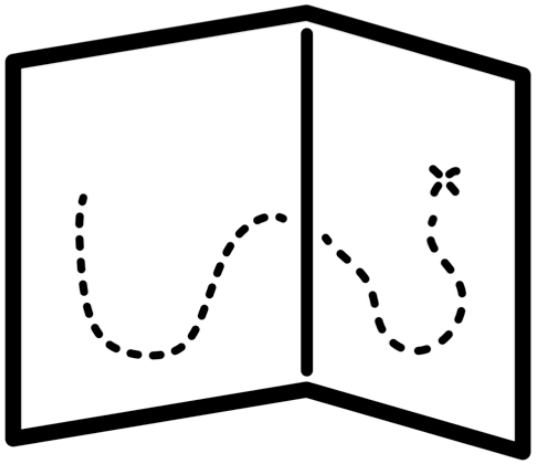
How to use Selenium, successfully



The Selenium Guidebook

Ruby Edition

by Dave Haeffner

Version 4.0.0

Preface

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

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

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

What This Book Will Cover

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

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

Who This Book Is For

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

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

About The Examples In This Book

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

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

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

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

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

How To Read This Book

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

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

Feedback

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

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

Table of Contents

Chapter 1
Selenium In A Nutshell

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

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

What Selenium Is and Is Not Good At

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

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

Selenium Highlights

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

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

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

Chapter 2
Defining A Test Strategy

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

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

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

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

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

What To Do With The Answers

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

Question 1: Money/Value

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

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

Question 2: Usage Data

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

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

Question 3: Browsers

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

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

Question 4: Risky Bits

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

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

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

Now You Are Ready

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

Chapter 3
Picking A Language

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

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

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

What To Do With The Answer

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

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

Some Additional Food For Thought

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

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

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

Outro

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

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

Chapter 4
A Programming Primer

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

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

Installation

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

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

- Linux
- OSX
- Windows

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

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

Installing Third-Party Libraries

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

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

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

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

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

Interactive Prompt

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

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

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

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

Choosing A Text Editor

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

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

Programming Concepts In A Nutshell

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

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

- Object Structures (Variables, Methods, and Classes)

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

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

Object Structures

Variables

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

A variable name:

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

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

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

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

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

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

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

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

Methods

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

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

example_method
```

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

```
def say(message)
  puts message
end

say 'Hello World!'

# outputs:
# Hello World!
```

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

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

say
say 'something else'

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

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

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

Classes

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

Class names:

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

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

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

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

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

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

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

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

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

Scope

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

Local Variables

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

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

Instance Variables

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

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

Constants

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

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

```
class Login

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

Environment Variables

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

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

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

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

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

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

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

Object Types

Strings

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

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

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

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

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

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

Numbers

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

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

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

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

Collections

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

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

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

```
# Array Example

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

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

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

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

```ruby
# Hash Example

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

# outputs:
String
Fixnum
Array
```

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

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

```ruby
class Login

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

Booleans

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

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

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

# returns: true
```

Actions

Assertions

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

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

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

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

# or

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

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

Conditionals

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

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

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

# outputs: The number is 10
```

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

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

# outputs: The number is 10
```

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

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

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

Iteration

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

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

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

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

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

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

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

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

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

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

Inheritance

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

Inheritance is done when declaring a child class by:

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

```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:

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

Chapter 5
Anatomy Of A Good Acceptance Test

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

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

Atomic & Autonomous Tests

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

Grouping Tests

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

Being Descriptive

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

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

Test Runners

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

Version Control

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

- Git
- Mercurial
- Subversion

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

Chapter 6
Writing Your First Test

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

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

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

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

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

How To Find Locators

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

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

How To Find Quality Elements

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

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

element you want to use.

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

An Example

Part 1: Find The Elements And Write The Test

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

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

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

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

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

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

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

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

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

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

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

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

require 'selenium-webdriver'

describe 'Login' do

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

  after(:each) do
    @driver.quit
  end

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

end
```

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

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

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

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

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

Part 2: Figure Out What To Assert

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

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

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

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

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

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

A Quick Primer on CSS Selectors

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

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

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

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

Part 3: Write The Assertion And Verify It

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

end
```

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

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

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

Just To Make Sure

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

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

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

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

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, try verifying them in the browser instead.

A Solution

Built into every major browser is the ability to verify locators from the JavaScript Console.

Simply open the developer tools in your browser and navigate to the JavaScript Console (e.g., right-click on an element, select `Inspect Element`, and click into the `Console` tab). From here it's a simple matter of specifying the CSS selector you want to look up by the `$$('')` command (e.g., `$$('#username')`) and hovering your mouse over what is returned in the console. The element that was found will be highlighted in the viewport.

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>
        <!-- ... -->
```

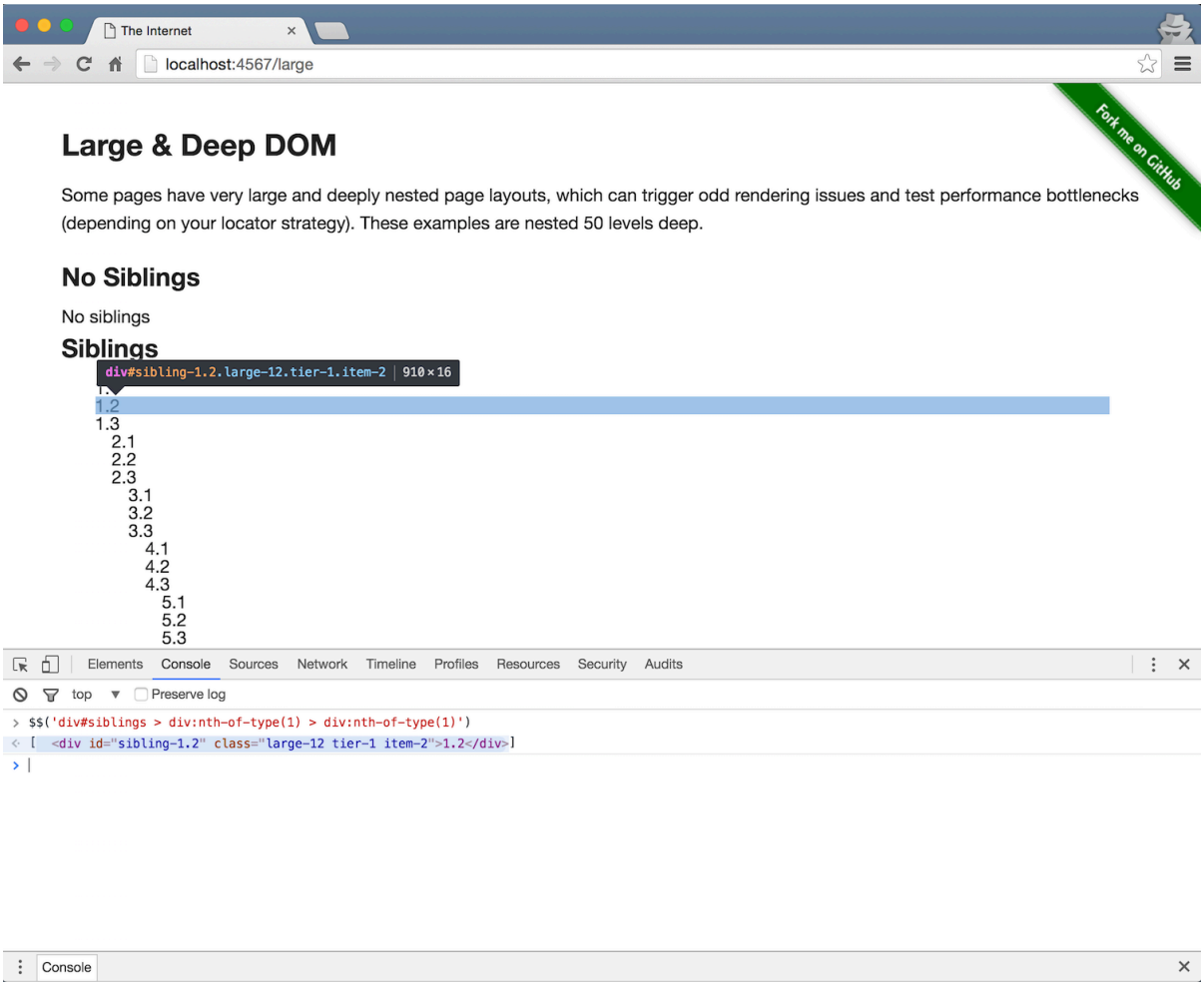
If we perform a `findElement` action using the following locator, it works.

```
driver.findElement({css: 'div#siblings > div:nth-of-type(1) > div:nth-of-type(1)'});
```

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

```
driver.findElement({css: 'div#siblings > div:nth-of-type(1) > div:nth-of-type(1) >
div:nth-of-type(1)'});
```

Fortunately with our in-browser approach to verifying our locators, we can quickly discern where the issue is. Here's what it shows us for the locators that "worked".

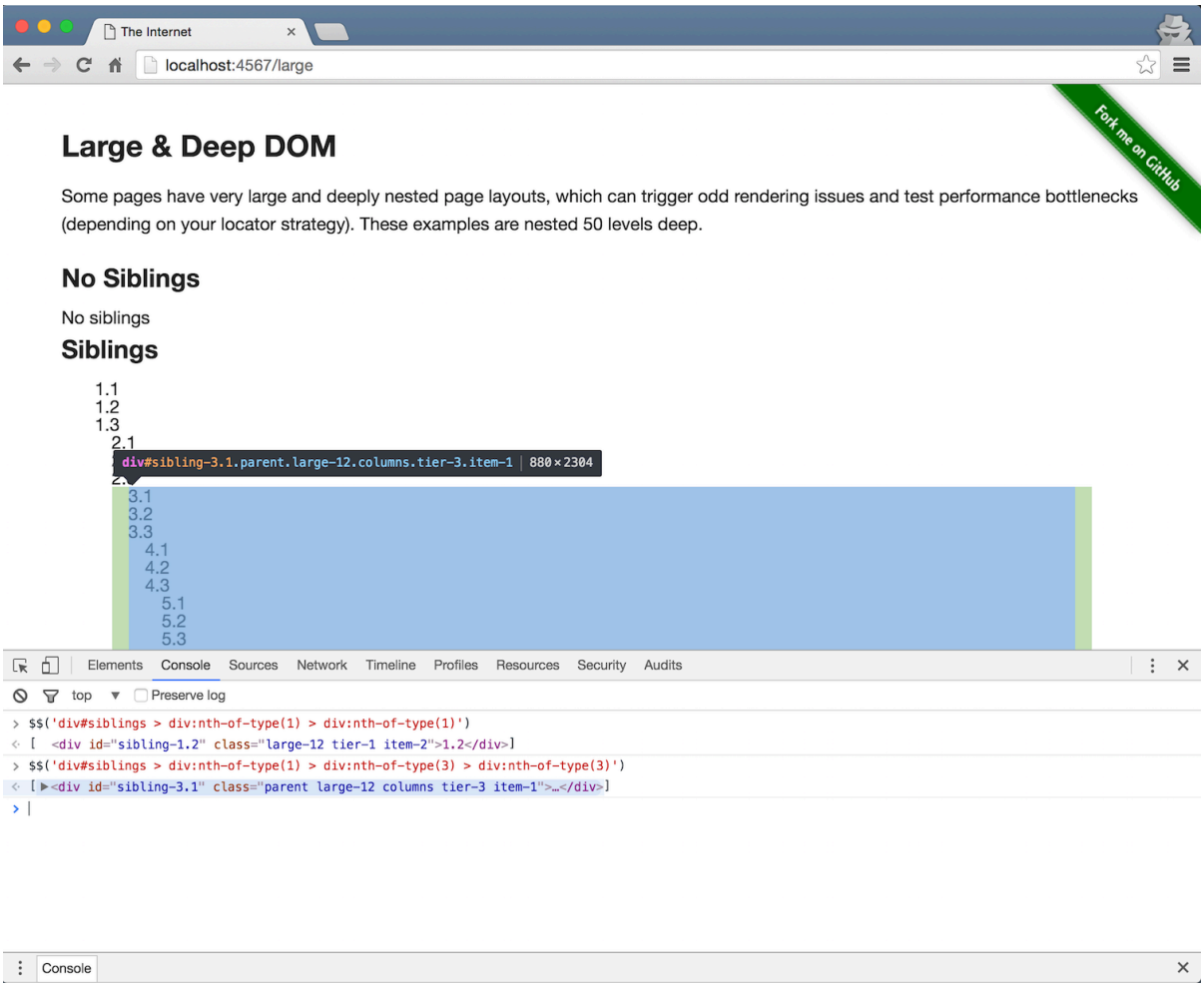


It looks like our locators are scoping to the wrong part of the first level (1.2). But we need to reference the third part of each level (e.g., 1.3, 2.3, 3.3) in order to traverse deeper since the nested divs live under the third part of each level.

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

```
driver.findElement({css: 'div#siblings > div:nth-of-type(1) > div:nth-of-type(3) > div:nth-of-type(3)'});
```

We can confirm that it works before changing any test code by looking in the JavaScript Console first.



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.

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.

To do that, let's create a new directory called `pages`, and add a new file called `login.rb` to it.

```
Gemfile
pages
   login.rb
spec
   login_spec.rb
vendor
```

Here is the code for it.

```ruby
# filename: pages/login.rb
class Login

  USERNAME_INPUT  = { id: 'username' }
  PASSWORD_INPUT  = { id: 'password' }
  SUBMIT_BUTTON   = { css: 'button'  }
  SUCCESS_MESSAGE = { css: '.flash.success' }

  def initialize(driver)
    @driver = driver
    @driver.get 'http://the-internet.herokuapp.com/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(SUBMIT_BUTTON).click
  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 (a.k.a. a constructor). 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.

In our `with` method we are capturing the core behavior of the login page by accepting the username and password input values and putting them to use with the typing and clicking actions for the login form.

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. 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 with Selenium so that we can return a boolean (e.g., `@driver.find_element(SUCCESS_MESSAGE).displayed?`). 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: pages/login_spec.rb
require 'selenium-webdriver'
require_relative '../pages/login'

describe 'Login' do

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

  after(:each) do
    @driver.quit
  end

  it 'succeeded' do
    @login.with('tomsmith', 'SuperSecretPassword!')
    expect(@login.success_message_present?).to be_truthy
  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.

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 (`expect(@login.success_message_present?).to be_truthy`).

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 the mechanics will similar in our page object. 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: pages/login.rb

class Login

  USERNAME_INPUT  = { id: 'username' }
  PASSWORD_INPUT  = { id: 'password' }
  SUBMIT_BUTTON   = { css: 'button'  }
  SUCCESS_MESSAGE = { css: '.flash.success' }
  FAILURE_MESSAGE = { css: '.flash.error' }
  #...
```

Further down the file (just after the existing mutator method) we'll add a new method to check to see if the failure message is displayed.

```
# filename: pages/login.rb
# ...
  def success_message_present?
    @driver.find_element(SUCCESS_MESSAGE).displayed?
  end

  def failure_message_present?
    @driver.find_element(FAILURE_MESSAGE).displayed?
  end

end
```

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

```ruby
# filename: login_spec.rb
# ...
  it 'succeeded' do
    @login.with('tomsmith', 'SuperSecretPassword!')
    expect(@login.success_message_present?).to be_truthy
  end

  it 'failed' do
    @login.with('asdf', 'asdf')
    expect(@login.failure_message_present?).to be_truthy
  end

end
```

Now if we run our spec file (`rspec`) 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.

```ruby
it 'failed' do
    @login.with('asdf', 'asdf')
    expect(@login.success_message_present?).to be_falsey
  end
```

There are two problems with this approach. First, the absence of success message does not necessarily indicate a failed login. The assertion we ended up with is more concise. Second, our test will fail. This is because it errors out when looking for an element that's not present.

```
.F

Failures:

  1) Login failed
     Failure/Error: expect(@login.success_message_present?).to eql false

Selenium::WebDriver::Error::NoSuchElementError:
       Unable to locate element: {"method":"css selector","selector":".flash.success"}
     # ...
     # ./login.rb:21:in `success_message_present?'
     # ./login_spec.rb:23:in `block (2 levels) in <top (required)>'

Finished in 8.85 seconds (files took 0.18961 seconds to load)
2 examples, 1 failure

Failed examples:

rspec ./login_spec.rb:20 # Login failed
```

We'll address this limitation in the next chapter.

Part 3: Confirm We're In The Right Place

Before we can call our page object complete, there's one more addition we'll want to make. We'll want to add a check in the constructor to make sure the page is in the correct state. Otherwise the test should not proceed and fail.

As a rule, you want to keep assertions in your tests and out of your page objects. So we'll use an exception instead.

```ruby
# filename: login.rb

class Login

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

  def initialize(driver)
    @driver = driver
    @driver.get 'http://the-internet.herokuapp.com/login'
    raise 'Login page not ready' unless
          @driver.find_element(LOGIN_FORM).displayed?
  end
# ...
```

We add a new locator to the mix (the `id` of the login form) and add a new line to the end of our `initialize` method. In it we are checking to see if the login form is displayed. If it's not, we raise a custom exception, letting us know what the issue was before letting tests that use it proceed.

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, there's more we can do.

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 throughout our suite. If we leave this unchecked we will end up with duplicative code which will slowly make our page objects harder to maintain.

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:

- slower page object creation due to the lack of a simple Domain Specific Language (DSL)
- test maintenance issues if the Selenium API changes
- the inability to swap out the driver for your tests (e.g., mobile, REST, etc.)

With a facade layer we can easily side step these concerns by abstracting our common actions into a central place and leveraging it in our page objects.

An Example

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

Part 1: Create The Base Page Object

First we will need to create the base page object, which we'll put in the `pages` directory.

```
Gemfile
pages
   base_page.rb
   login.rb
spec
   login_spec.rb
vendor
    geckodriver
```

Next we'll create a class and populate it with methods for each of the Selenium commands we've been using.

```ruby
# filename: pages/base_page.rb
require 'selenium-webdriver'

class BasePage

  def initialize(driver)
    @driver = driver
  end

  def visit(url)
    @driver.get url
  end

  def find(locator)
    @driver.find_element locator
  end

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

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

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

end
```

First we pull in the Selenium Ruby bindings (`require selenium-webdriver`). By placing it here in this file we will no longer need to require 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 (`class 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 (e.g., `visit` , `find` , `type` , `submit` , and `is_displayed?`).

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

```ruby
# filename: pages/login.rb
require_relative 'base_page'

class Login < BasePage

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

  def initialize(driver)
    super
    visit 'http://the-internet.herokuapp.com/login'
    raise 'Login page not ready' unless
      is_displayed?(LOGIN_FORM)
  end

  def with(username, password)
    type username, USERNAME_INPUT
    type password, PASSWORD_INPUT
    click SUBMIT_BUTTON
  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 `require_relative`. And second, we need to connect the login page object with the base page object with 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.

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. So we replace all of the driver incantations in this class with calls to the methods we created in the base page object. Our tests remain unchanged, but our page objects are now more flexible and readable.

Running our tests (`rspec` from the command-line) will 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: expect(@login.success_message_present?).to eql false

Selenium::WebDriver::Error::NoSuchElementError:
       Unable to locate element: {"method":"css selector","selector":".flash.success"}
     # ...
     # ./login.rb:21:in `success_message_present?'
     # ./login_spec.rb:23:in `block (2 levels) in <top (required)>'

Finished in 8.85 seconds (files took 0.18961 seconds to load)
2 examples, 1 failure

Failed examples:

rspec ./login_spec.rb:20 # Login failed
```

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 catch it and return a `false` boolean instead of failing the test.

Let's update the `is_displayed?` method in our base page object to do just that.

```ruby
# filename: pages/base_page.rb
#...

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 attempt to see if an element is displayed to the user. If the element is not on the page Selenium will raise a `NoSuchElementError` exception. When that happens, our code will now `rescue` for that specific error condition, and return `false` instead of failing the test.

Now we can write a test to see if an element is not on the page 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: spec/login_spec.rb
#...
  it 'failed' do
    @login.with('asdf', 'asdf')
    #expect(@login.failure_message_present?).to be_truthy
    expect(@login.success_message_present?).to be_falsey
  end

end
```

Now when we run this (`rspec` 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 in your application.

By using high quality locators we're already in good shape, but there are still some 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 web 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 for elements you want to interact with. The best way to accomplish this is through the use of explicit waits.

An Example

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!'.

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

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: pages/dynamic_loading.rb
require_relative 'base_page'

class DynamicLoading < BasePage

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

  def load(example_number)
    visit 'http://the-internet.herokuapp.com/dynamic_loading/' + example_number
    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 method to load the example and start it (e.g., `load`), and a method to see if the finish text is present (e.g., `finish_text_present?`).

In `finish_text_present?` we are using a bit of aspirational code. We're referencing a method that hasn't been added to our base page object yet. So let's hop into our base page object and add it to the bottom of the class.

```
# filename: pages/base_page.rb
# ...
  def wait_for(seconds = 15)
    Selenium::WebDriver::Wait.new(timeout: seconds).until { yield }
  end

end
```

`wait_for` is where we are defining our explicit wait. In it we are accepting an integer for the number of seconds we'd like to wait. If nothing is provided, `15` seconds will be used. We then use this value to tell the explicit wait function that Selenium offers how long to wait, and what to wait for. By using a `yield` we are able to pass in a code block to run (e.g., a Selenium command

surrounded by brackets (`{}`)). If this passed in code doesn't evaluate to true, the explicit wait will retry it until it can either succeed, or it reaches the timeout.

More On Explicit Waits

The major benefit of explicit waits is that 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 a single wait time to fix the test -- rather than increase a blanket wait time (which impacts every test). And since the wait is dynamic (e.g., constantly polling), it won't take the full amount of time to complete (like a hard-coded sleep would).

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 create a new test to use it.

```
Gemfile
pages
   base_page.rb
   dynamic_loading.rb
   login.rb
spec
   dynamic_loading_spec.rb
   login_spec.rb
vendor
    geckodriver
```

```ruby
# filename: spec/dynamic_loading_spec.rb
require_relative '../pages/dynamic_loading'

describe 'Dynamic Loading' do

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

  after(:each) do
    @driver.quit
  end

  it 'Example 1: Hidden Element' do
    @dynamic_loading.load('1')
    expect(@dynamic_loading.finish_text_present?).to be_truthy
  end

end
```

When we run it (e.g., `rspec dynamic_loading_page.rb` from the command-line) it will visit the page, click the start button, wait for the text to appear, assert the text appeared, and close the browser.

Part 3: 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 nearly identical to the last one. The only difference is that it will render the final result after the progress bar completes. Otherwise, the markup is the same. So we can use the same locators.

This makes adding another test straight-forward.

```ruby
# filename: spec/dynamic_loading_spec.rb
# ...
  it 'Example 2: Rendered after the fact' do
    @dynamic_loading.load '2'
    expect(@dynamic_loading.finish_text_present?).to be_truthy
  end
end
```

When we run these tests (`rspec dynamic_loading_spec.rb` from the command-line) we can see that the explicit wait approach works for when the element is on the page but hidden and when it's rendered after the fact.

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 on various browsers.

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

Chrome execution can sometimes be faster than Firefox, so you could 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 Internet Explorer. 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 in a target browser and see which ones fail. Take each failed test, adjust your code as needed, and re-run it against the target browser until they all pass. Repeat for each browser you care about until everything is green.

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 to automate.

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

Now we're ready 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. This can live alongside the other test files in the `spec` directory.

```
Gemfile
pages
   base_page.rb
   dynamic_loading.rb
   login.rb
spec
   dynamic_loading_spec.rb
   login_spec.rb
   spec_helper.rb
vendor
    geckodriver
```

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

RSpec.configure do |c|

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

  c.after do |example|
    @driver.quit
  end

end
```

We require the Selenium library here since we're working directly with it. By having it here we can remove it from our base page object. We can also remove our Selenium commands from the `before(:each)` and `after(:each)` in our tests -- replacing them with a simple require statement at the top of the file (`require_relative 'spec_helper'`). We will also be able to remove the `after(:each)` method from our tests, leaving just the `before(:each)` for use with our page objects.

Here's what our tests look like with these changes:

```ruby
# filename: spec/login_spec.rb
require_relative 'spec_helper'
require_relative '../pages/login'

describe 'Login' do

  before(:each) do
    @login = Login.new(@driver)
  end

  it 'succeeded' do
    @login.with('tomsmith', 'SuperSecretPassword!')
    expect(@login.success_message_present?).to be_truthy
  end

  it 'failed' do
    @login.with('asdf', 'asdf')
    expect(@login.failure_message_present?).to be_truthy
  end

end
```

```ruby
# filename: spec/dynamic_loading_spec.rb
require_relative 'spec_helper'
require_relative '../pages/dynamic_loading'

describe 'Dynamic Loading' do

  before(:each) do
    @dynamic_loading = DynamicLoading.new(@driver)
  end

  it 'Example 1: Hidden Element' do
    @dynamic_loading.load '1'
    expect(@dynamic_loading.finish_text_present?).to be_truthy
  end

  it 'Example 2: Rendered after the fact' do
    @dynamic_loading.load '2'
    expect(@dynamic_loading.finish_text_present?).to be_truthy
  end

end
```

Base URL

Up until now we've been hard-coding the URL we want to use for our application. But it's likely that the application could have numerous URL end-points. In the case of the-internet there is the production URL (e.g., `http://the-internet.herokuapp.com`) and then there's running the app locally (e.g., `http://localhost:4567`).

In order to service both of these URLs with our tests we'll want to make it so we can specify a base URL at runtime and update our page objects to use it. We'll also want to set a sensible default for the base URL so if we specify nothing, a valid URL will be provided to our tests. So let's create a central file to store configuration values like this and make sure it can receive values at runtime. Let's create a `config.rb` file in the `spec` directory and place it there.

```
Gemfile
pages
   base_page.rb
   dynamic_loading.rb
   login.rb
spec
   config.rb
   dynamic_loading_spec.rb
   login_spec.rb
   spec_helper.rb
vendor
    geckodriver
```

```ruby
# filename: config.rb
module Config
  def config
    { base_url: ENV['BASE_URL'] || 'http://the-internet.herokuapp.com' }
  end
end
```

By using a `module` we'll be able to easily mix it into where we need it. And with the help of a conditional when setting the `base_url` (e.g., `||=`) we're able to get our sensible default.

We are making it so we can override this value when launching our test suite (by specifying a value for the environment variable `BASE_URL`). It essentially means if the environment variable already exists and contains a value, use it. Otherwise, use `'http://the-internet.herokuapp.com'`. This will come in handy later, and is an approach we'll use frequently.

With the base URL in a central place we can now go and update our page objects to use it.

```ruby
# filename: pages/base_page.rb
require_relative '../spec/config'

class BasePage
  include Config

    # ...

  def visit(url)
    if url.start_with? 'http'
      @driver.get url
    else
      @driver.get config[:base_url] + url
    end
  end
# ...
```

In our base page object we put the config to use by mixing it in (e.g., `include Config`) and referencing it in `visit`. If the `url` provided starts with `http`, then we do what we've been doing. Otherwise, we create a URL with the base URL and what's provided in `url`.

Now to update our page objects to just provide a URL path when calling `visit`.

```ruby
# filename: pages/dynamic_loading.rb
# ...
  def example(example_number)
    visit "/dynamic_loading/#{example_number}"
  end
# ...
```

```ruby
# filename: pages/login.rb
# ...
  def initialize(driver)
    super
    visit '/login'
    raise 'Login page not ready' unless
      is_displayed?(LOGIN_FORM)
  end
# ...
```

Now our page objects are free from hard-coded URLs, making our tests more flexible.

Running Everything

Now that things are cleaned up, let's run everything to make sure the tests still pass (e.g., `rspec` from the command line).

To specify a different base URL, prepend the command with it (e.g., `base_url=http://localhost:4567 rspec` if you had a local instance of the-internet running).

Chapter 12

Running A Different Browser Locally

It's fairly straightforward to get your tests running locally against a single browser (that's what we've been doing up until now). But when you want to run them against additional browsers like Chrome, Safari, and Internet Explorer you quickly run into configuration overhead that can seem cumbersome and lacking in good documentation.

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 but not always) 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 existing `vendor` directory.

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

Just like with Firefox, 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 our system path,

or pass in the path to the file when launching Selenium. Let's do the latter.

NOTE: There is a different chromedriver binary for each major operating system. If you're using Windows, be sure to use the one that ends in .exe and specify chromedriver.exe in your configuration. This example was built to run on OSX (which does not have a specified file extension for chromedriver).

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

```
# filename: spec/config.rb
module Config
  def config
    {
      base_url:     ENV['BASE_URL']     || 'http://the-internet.herokuapp.com',
      browser_name: ENV['BROWSER_NAME'] || 'firefox'
    }
  end
end
```

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

RSpec.configure do |c|
  include Config

  c.before do |example|
    case config[:browser_name]
    when 'firefox'
      driver_path = File.join(Dir.pwd, 'vendor', 'geckodriver')
      if File.file? driver_path
        service = Selenium::WebDriver::Service.firefox(path: driver_path)
      end
    when 'chrome'
      driver_path = File.join(Dir.pwd, 'vendor', 'chromedriver')
      if File.file? driver_path
        service = Selenium::WebDriver::Service.chrome(path: driver_path)
      end
    end
    if service
      @driver = Selenium::WebDriver.for config[:browser_name].to_sym, service: service
    else
      @driver = Selenium::WebDriver.for config[:browser_name].to_sym
    end
  end

  config.after do |example|
    @driver.quit
  end

end
```

In `config.before(:each)` we're doing a conditional check against the browser environment variable we set in `config.rb` 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 loading an instance of Selenium for Chrome.

Now we can specify Chrome when launching our tests (e.g., `BROWSER=chrome rspec`) or Firefox (e.g., `BROWSER=firefox rspec`).

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. Be sure to

check out the documentation for the browser you care about to find out the specific requirements:

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

Chapter 13

Running Browsers In The Cloud

If you've ever needed to test features in an older browser like Internet Explorer 9 or 10 then odds are you ran a virtual machine (VM) on your computer with a "legit" version of Windows.

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 cover other browser and Operating System (OS) combinations? Now you're looking at provisioning, running, and maintaining your own set of machines and standing up something like Selenium Grid to coordinate tests across them.

Rather than take on the overhead of a test infrastructure you can easily outsource this to a third-party cloud provider like Sauce Labs.

A Selenium Grid Primer

At the heart of Selenium at scale is the use of Selenium Grid.

Selenium Grid lets you distribute test execution across several machines and you connect to it with Selenium. You tell the Grid which browser and OS you want your test to run on through the use of Selenium's `DesiredCapabilities`.

Under the hood this is how Sauce Labs works. They are ultimately running Selenium Grid behind the scenes, and they receive and execute tests through Selenium Remote and the `DesiredCapabilities` you set.

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 trial 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 check out their Open Sauce account.

With Sauce Labs we need to provide our credentials, specifics about what we want in our test environment, and configure Selenium a little bit differently than we have been. Let's start by updating our `config.rb` file.

```ruby
# filename: spec/config.rb
module Config
  def config
    {
      base_url:         ENV['BASE_URL']         || 'http://the-internet.herokuapp.com',
      host:             ENV['HOST']             || 'saucelabs',
      browser_name:     ENV['BROWSER_NAME']     || 'internet_explorer',
      browser_version:  ENV['BROWSER_VERSION']  || '11',
      platform_name:    ENV['PLATFORM_NAME']    || 'Windows 10',
      sauce_username:   ENV['SAUCE_USERNAME'],
      sauce_access_key: ENV['SAUCE_ACCESS_KEY']
    }
  end
end
```

Notice the use of a host environment variable (e.g., `ENV['HOST'] = 'saucelabs'`). This is what we'll use in our `spec_helper` file to determine whether to run things locally or in the cloud. We can specify our Sauce Labs credentials here by hard-coding them (not recommended) , or we can reference them through an environment variable we configure on our machine or specify them at run-time (recommended). And we'll use the `BROWSER_NAME`, `BROWSER_VERSION`, and `PLATFORM_NAME` environment variables to populate the `Capabilities` object.

Now to update our `spec_helper.rb` file.

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

RSpec.configure do |c|
  include Config

  c.before do |example|
    case config[:host] when 'saucelabs'
      caps = Selenium::WebDriver::Remote::Capabilities.send(config[:browser_name])
      caps[:browser_version] = config[:browser_version]
      caps[:platform_name] = config[:platform_name]
      url = "http://#{config[:sauce_username]}:#{config[:sauce_access_key]}
@ondemand.saucelabs.com:80/wd/hub"
      @driver = Selenium::WebDriver.for(
        :remote,
        url: url,
        desired_capabilities: caps)
    when 'localhost'
      case config[:browser_name]
      when 'firefox'
        driver_path = File.join(Dir.pwd, 'vendor', 'geckodriver')
        if File.file? driver_path
          service = Selenium::WebDriver::Service.firefox(path: driver_path)
        end
      when 'chrome'
        driver_path = File.join(Dir.pwd, 'vendor', 'chromedriver')
        if File.file? driver_path
          service = Selenium::WebDriver::Service.chrome(path: driver_path)
        end
      end
      if service
        @driver = Selenium::WebDriver.for config[:browser_name].to_sym, service:
service
      else
        @driver = Selenium::WebDriver.for config[:browser_name].to_sym
      end
    end
  end

  c.after do |example|
    @driver.quit
  end

end
```

We've taken our browser conditional and made it nested underneath one 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, our tests will run locally.

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

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, specifying `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`.

For the `url`, we are constructing a URL to Sauce's Selenium Grid which contains our account credentials (e.g., a basic auth URL) through what's known as 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`) 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 11.

Part 2: Test Name

In order to get the most out of our test runs in Sauce Labs, we'll want to pass some additional metadata like the test name. That way we'll be able to correlate a test run with the job in Sauce Labs so we can easily find it and view the reporting (e.g., video recording, screenshots, logs, etc.) if there was an issue found during the run.

To do that, it's a simple matter of using Selenium's JavaScript executor. We can pull the name of the test out of RSpec and pass it to Sauce Labs.

Let's update our teardown to handle this.

```ruby
# filename: spec/spec_helper.rb
# ...
  c.after do |example|
    begin
      if config[:host] == 'saucelabs'
        @driver.execute_script("sauce:job-name=#{example.full_description}")
      end
    ensure
      @driver.quit
    end
  end

end
```

Part 3: 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 Sauce Labs dashboard each of the test jobs will just say Finished. This will make our results less useful in the long run, so let's fix it.

This is also something we can handle with the JavaScript executor in our teardown.

Before we issue `@driver.quit` we will want to grab the session ID from our `@driver` object (which is also the job ID in Sauce Labs) and use it to set the status of the job based on the test result.

```ruby
# filename: spec/spec_helper.rb
#...
  c.after do |example|
    begin
      if config[:host] == 'saucelabs'
        test_passed = example.exception.nil?
        @driver.execute_script("sauce:job-name=#{example.full_description}")
        @driver.execute_script("sauce:job-result=#{test_passed}")
        if !test_passed
          puts "Watch a video of the test at https://saucelabs.com/tests/#{@driver.
session_id}"
        end
      end
    ensure
      @driver.quit
    end
  end

end
```

For bonus points, we've also added console output of the URL of the job in Sauce Labs when there is a failure. For good measure, we've also wrapped everything in a `begin` / `ensure` block to make sure that `driver.quit` always gets called, regardless of the outcome of the other commands in the teardown.

Now when we run our tests (`rspec`) 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').

And if a test fails, a URL to the Sauce Labs job will appear in the console output.

> Watch a video of the test at
> https://saucelabs.com/tests/8b075a49cb20477f9aa820de4d196ac5

Part 4: 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 certificate based authentication). Others keep it behind their firewall. For those that stay behind a firewall, Sauce Labs has you covered.

They have a program called Sauce Connect Proxy that creates a secure tunnel between your machine and their private cloud. With it you can run tests in Sauce Labs and test applications that are only available on your private network.

To use Sauce Connect you need to download and run it. There's a copy for each operating system -- get yours here and run it from the command-line. In the context of our existing test code let's download Sauce Connect, unzip its contents, and store it in our `vendor` directory.

```
Gemfile
pages
   base_page.rb
   dynamic_loading.rb
   login.rb
spec
   config.rb
   dynamic_loading_spec.rb
   login_spec.rb
   spec_helper.rb
vendor
    chromedriver
    geckodriver
    sc
```

Now we just need to launch the application while specifying our Sauce account credentials.

```
vendor/sc -u $SAUCE_USERNAME -k $SAUCE_ACCESS_KEY
// ...
Starting Selenium listener...
Establishing secure TLS connection to tunnel...
Selenium listener started on port 4445.
Sauce Connect is up, you may start your tests.
```

Now that the tunnel is established, we could run our tests against a local instance of our application (e.g., the-internet). Assuming the application was set up and running on our local machine, we run our tests against it by specifying a different base URL at runtime (e.g., `BASE_URL=http://localhost:4567 rspec`) and they would work.

To see the status of the tunnel, we can view it on the tunnel page of the account dashboard. To shut the tunnel down, we can do it manually from this page. Or we can issue a `Ctrl+C` command to the terminal window where it's running.

When the tunnel is closing, here's what you'll see.

```
Got signal 2
Cleaning up.
Removing tunnel 21ff9664b06c4edaa4bd573cdc1fbac1.
All jobs using tunnel have finished.
Waiting for the connection to terminate...
Connection closed (8).
Goodbye.
```

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 it will take longer and longer for it to finish running, which puts a real damper on our ability to get feedback quickly.

With parallelization we can 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 paralleltests provides -- `parallelrspec`.

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

gem 'rspec', '~>3.8.0'
gem 'selenium-webdriver', '~>4.0.0.alpha.2'
gem 'parallel_tests', '~>2.29.1'
```

`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` for a config file, we just have to specify `--test-options` before-hand.

To run our tests in parallel with Sauce Labs (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 small price to pay for gaining parallel test execution.

Randomizing

A great way to make sure your tests don't have any inter-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 at runtime.

```
parralel_rspec --test-options '--order random'
```

This coupled with parallelization will make your tests really work for you.

Chapter 15

Flexible Test Execution

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 and informative (so people on your team will care about them). 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 critical acceptance tests (e.g., smoke or sanity tests). Assuming those pass, we can have another job run the remaining tests after that (e.g., the less critical and longer running tests). Adam Goucher refers to this strategy as a 'shallow' and 'deep' tagging model. To demonstrate this, let's add tags to our tests.

Part 1: Update Tests With Tag Metadata

RSpec comes built in with tagging support. It's a simple matter of adding a key/value pair or a symbol to denote what you want. You can place it on individual tests, or a group of tests. And you can use 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
# ...
describe 'Dynamic Loading', :deep do
# ...
```

```
# filename: spec/login_spec.rb
# ...
describe 'Login', :shallow do
# ...
```

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

```
it 'succeeded', :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` followed by the tag value (e.g., `--tag shallow` or `--tag deep`).

Part 2: Simple Command-line Execution

To simplify test execution from the command-line we can use a library called `Rake` to help.

With it we can specify tasks which will enable us to abstract away some of the complexity we'll need to remember to run our tests.

First, we'll need to add `rake` to our `Gemfile` and perform a `bundle install`.

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

gem 'rspec', '~>3.8.0'
gem 'selenium-webdriver', '~>4.0.0.alpha.2'
gem 'parallel_tests', '~>2.29.1'
gem 'rake', '~>12.3.2'
```

Next we'll want to create a `Rakefile` in the root directory of our project and add some tasks.

```
Gemfile
Rakefile
pages
   base_page.rb
   dynamic_loading.rb
   login.rb
spec
   config.rb
   dynamic_loading_spec.rb
   login_spec.rb
   spec_helper.rb
vendor
```

```
# filename: Rakefile
def launch_in_parallel
  tags      = ENV['TAG']
  processes = ENV['PROCESSES']
  system(
    "parallel_rspec " +
    "#{'-n ' + processes if processes} " +
    "--test-options '--order random " + "#{'--tag ' + tags if tags}' " +
    "spec"
  )
end

desc 'Run tests locally'
task :local, :browser_name do |task, args|
  ENV['HOST'] = 'localhost'
  ENV['BROWSER_NAME'] = args[:browser_name]
  launch_in_parallel
end

desc 'Run tests on Sauce Labs'
task :sauce, :browser_name, :browser_version, :platform_name do |t, args|
  ENV['BROWSER_NAME']  = args[:browser_name]
  ENV['BROWSER_VERSION'] = args[:browser_version]
  ENV['PLATFORM_NAME'] = args[:platform_name]
  launch_in_parallel
end
```

In the `launch_in_parallel` method we've abstracted launching the tests in parallel. It uses Ruby's `system` command launch a command in a shell process.

The tasks `local` and `cloud` use the `launch_in_parallel` method after setting environment variables with the values passed in.

When we save this file and run `rake -T` from the command-line, he is what we'll see.

```
> rake -T
rake local[browser_name]                                  # Run tests locally
rake sauce[browser_name,browser_version,platform_name]  # Run tests on Sauce Labs
```

Here are some examples of the tasks being used.

```
rake local['chrome']
rake sauce['safari','12.0','macOS 12.14']
```

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 tell people when there are issues. But that only helps you solve part of the problem.

The real goal in test automation is to find issues reliably, quickly, and automatically. We've built things to be reliable and quick. Now we need to make them run on their own, and ideally, in sync with the development workflow you are a part of.

To do that we need to use a Continuous Integration server.

A Continuous Integration Server Primer

A Continuous Integration server (a.k.a. CI) is responsible for merging code that is actively being developed into a central place (e.g., "trunk", "head", or "master") frequently (e.g., several times a day, or on every code commit) to find issues early so they can be addressed quickly — all for the sake of releasing working software in a timely fashion.

With CI we can automate our test runs so they can happen as part of the development workflow. 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 Selenium tests too.

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

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 all CI Servers is JUnit XML.

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'.

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

gem 'rspec', '~>3.8.0'
gem 'selenium-webdriver', '~>4.0.0.alpha.2'
gem 'parallel_tests', '~>2.29.1'
gem 'rake', '~>12.3.2'
gem 'rspec_junit_formatter', '~>0.4.1'
```

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 this with our parallel test execution we need to create a uniquely named XML output file for each test process. Thankfully its trivial to accomplish with `parallel_tests` because of an environment variable they use during execution.

We'll handle this in our `Rakefile` by updating the `launch_in_parallel` method.

```
# Rakefile
def launch_in_parallel
  tags      = ENV['TAG']
  processes = ENV['PROCESSES']
  ci        = "--format RspecJunitFormatter " +
              "--out tmp/result#{ENV['TEST_ENV_NUMBER']}.xml" if ENV['CI']
  system(
    "parallel_rspec " +
    "#{'-n ' + processes if processes} " +
    "--test-options '--order random #{'--tag ' + tags if tags} #{ci if ci}' " +
    "spec"
  )
end
```

When we run our tests with the `CI` environment variable set (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.

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 name="rspec" tests="2" failures="0" errors="0" time="25.890860" timestamp=
"2015-11-21T11:35:05-05:00">
  <!-- Randomized with seed 5604 -->
  <properties/>
  <testcase classname="spec.dynamic_loading_spec" name="Dynamic Loading Example 1:
Hidden Element" file="./spec/dynamic_loading_spec.rb" time="13.115928"/>
  <testcase classname="spec.dynamic_loading_spec" name="Dynamic Loading Example 2:
Rendered after the fact" file="./spec/dynamic_loading_spec.rb" time="12.774309"/>
</testsuite>
```

A CI Example

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

- Bamboo
- CircleCI
- Jenkins
- TravisCI

Let's pick one and step through an example.

Jenkins is a fully functional, widely adopted, free and open-source CI server. Its a great candidate for us to try.

Lets start by setting it up on the same machine as our test code. Keep in mind that this isn't the "proper" way to go about this — its merely beneficial for this example. To do it right, the Jenkins server (e.g., master node) would live on a machine of its own.

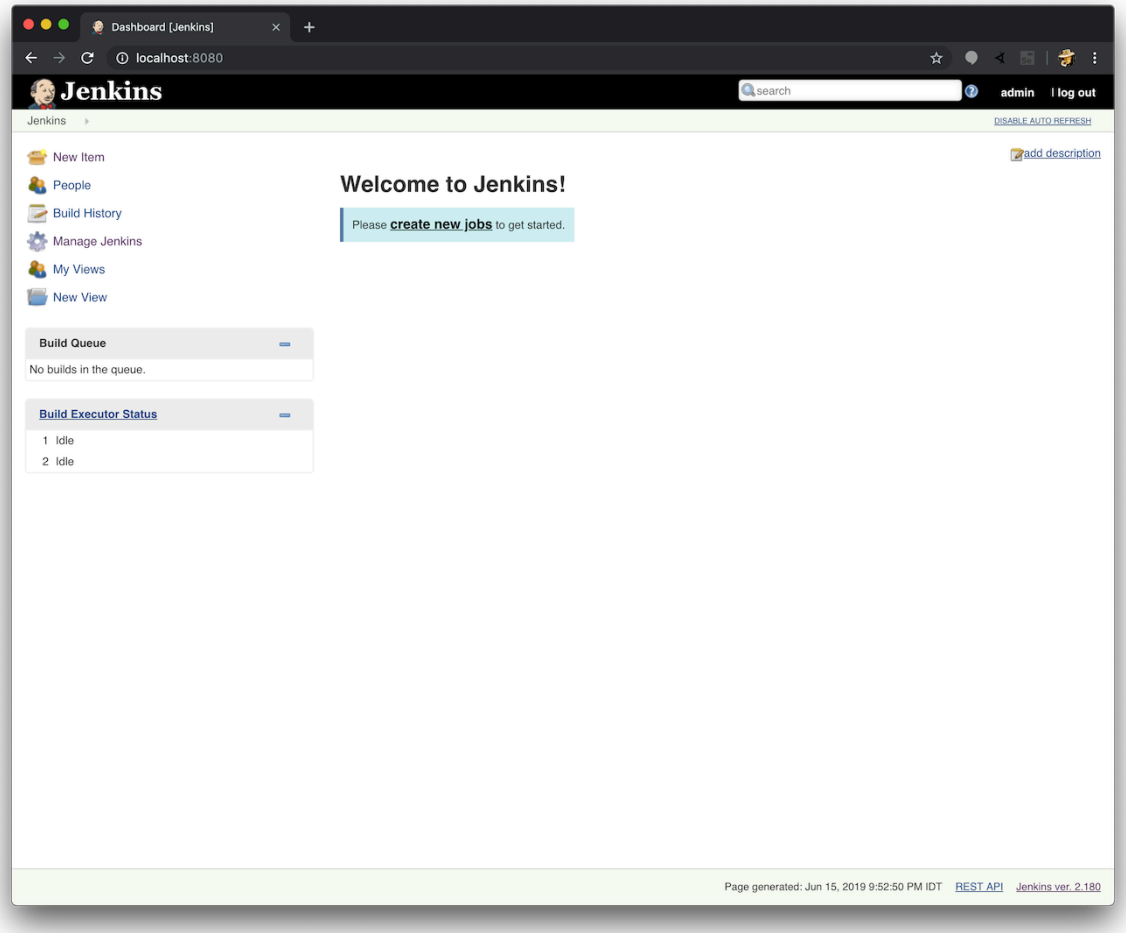
Part 1: Quick Setup

A simple way to get started is to grab the latest Jenkins war file. You can grab it from the Jenkins download page.

Once downloaded, launch it from the command-line and follow the setup steps provided.

```
> java -jar jenkins.war
// ...
hudson.WebAppMain$3 run
INFO: Jenkins is fully up and running
```

You will now be able to use Jenkins by visiting http://localhost:8080/ in your browser.

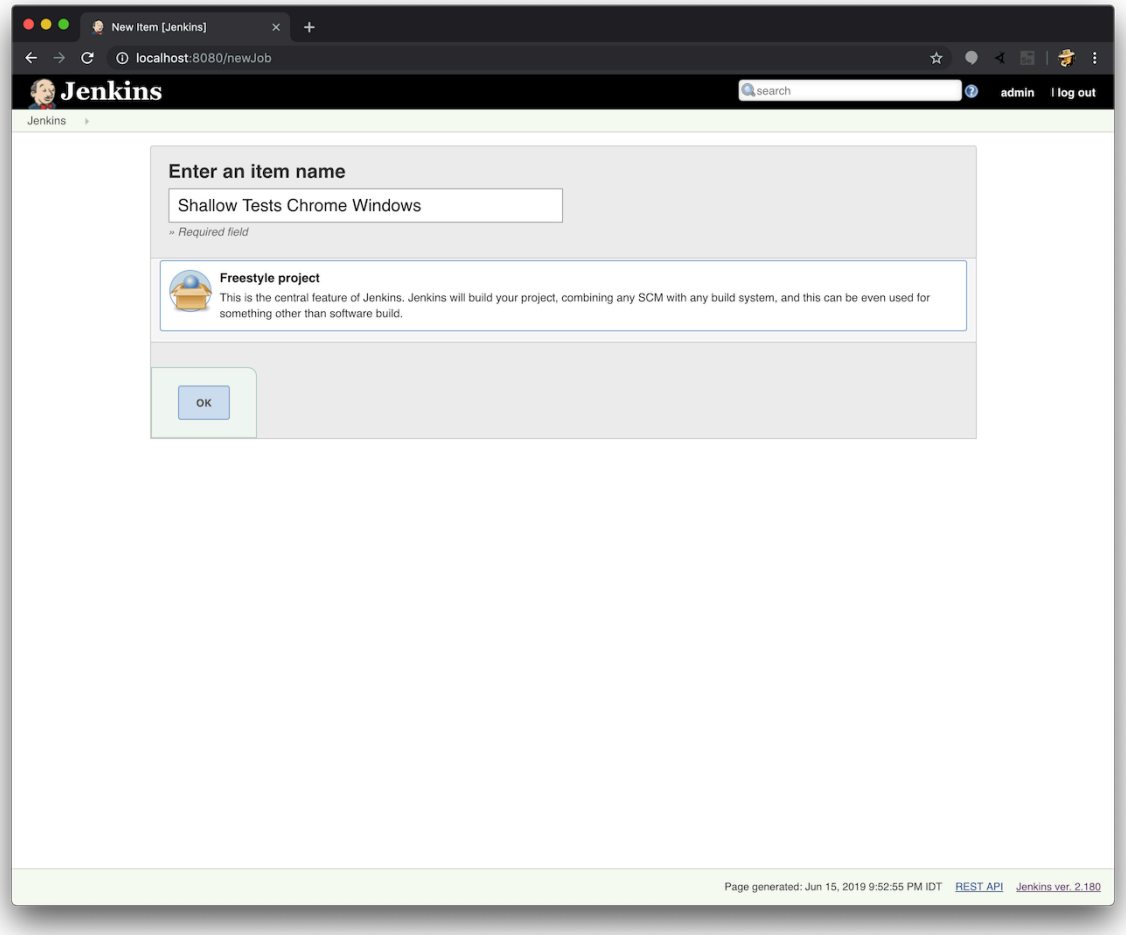


NOTE: Before moving to the next step, click **`ENABLE AUTO-REFRESH`** at the top right-hand side of the page. Otherwise you'll need to manually refresh the page (e.g., when running a job and waiting for results to appear).

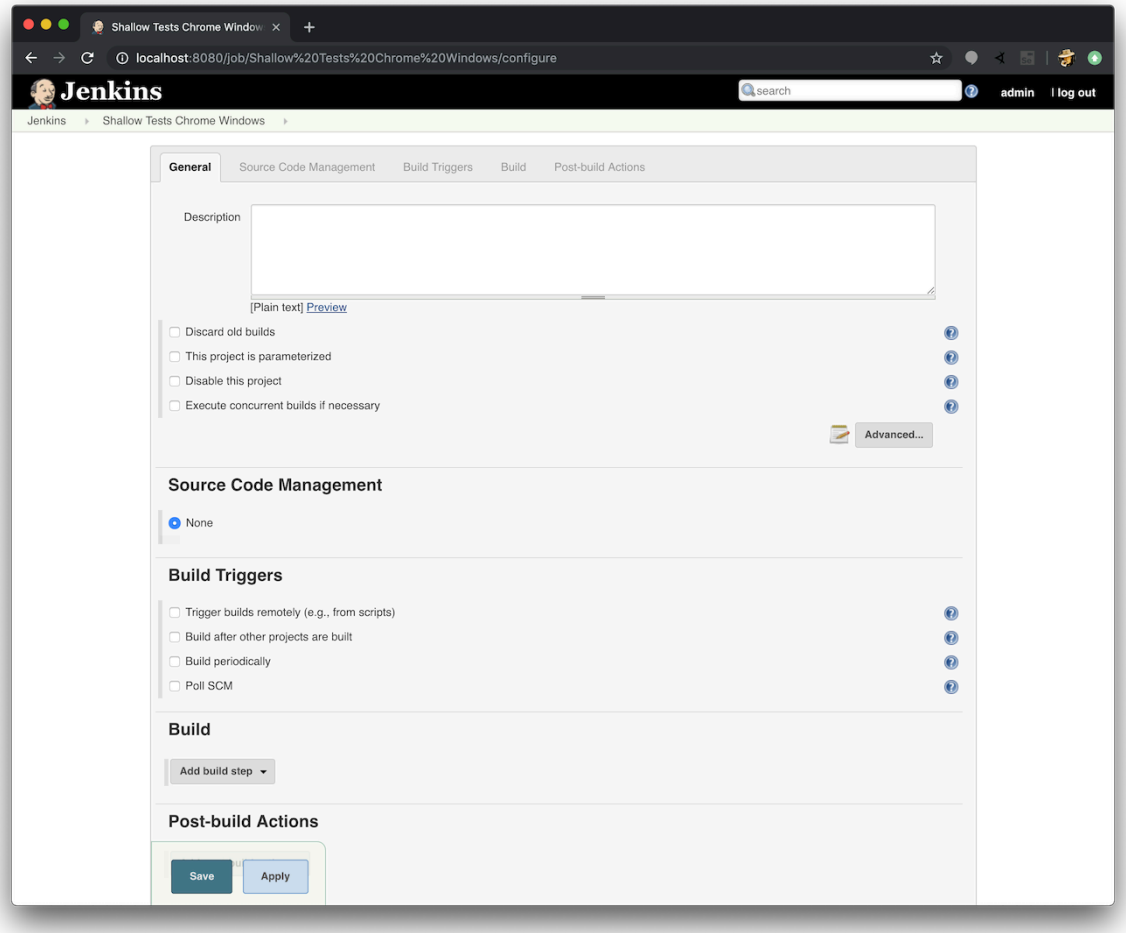
Part 2: Job Creation And Configuration

Now that Jenkins is loaded in the browser, let's create a Job and configure it to run our `shallow` tests against Chrome on Windows 10.

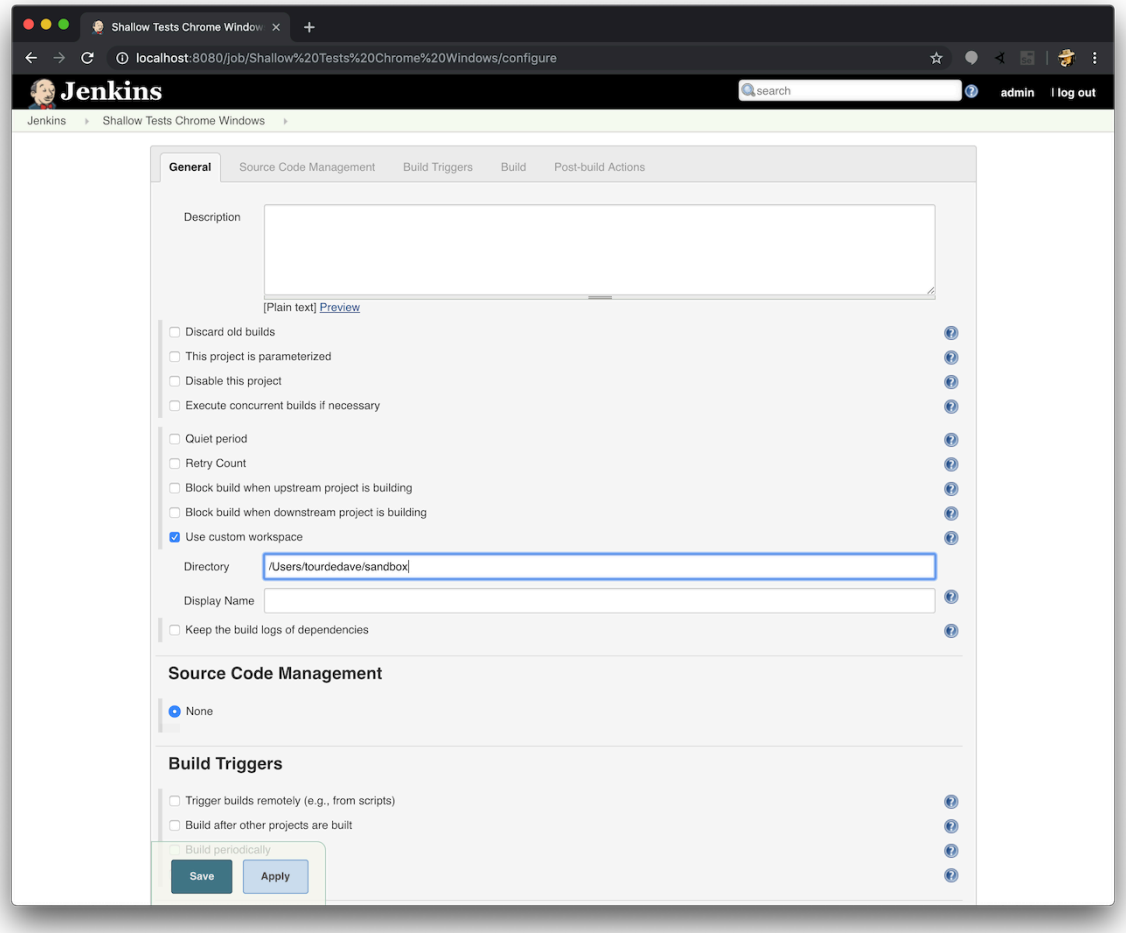
- Click `New Item` from the top-left of the Dashboard
- Give it a name (e.g., `Shallow Tests Chrome 50 Windows 10`)
- Select the `Freestyle project` option
- Click `OK`



This will load a configuration screen for the Jenkins job.

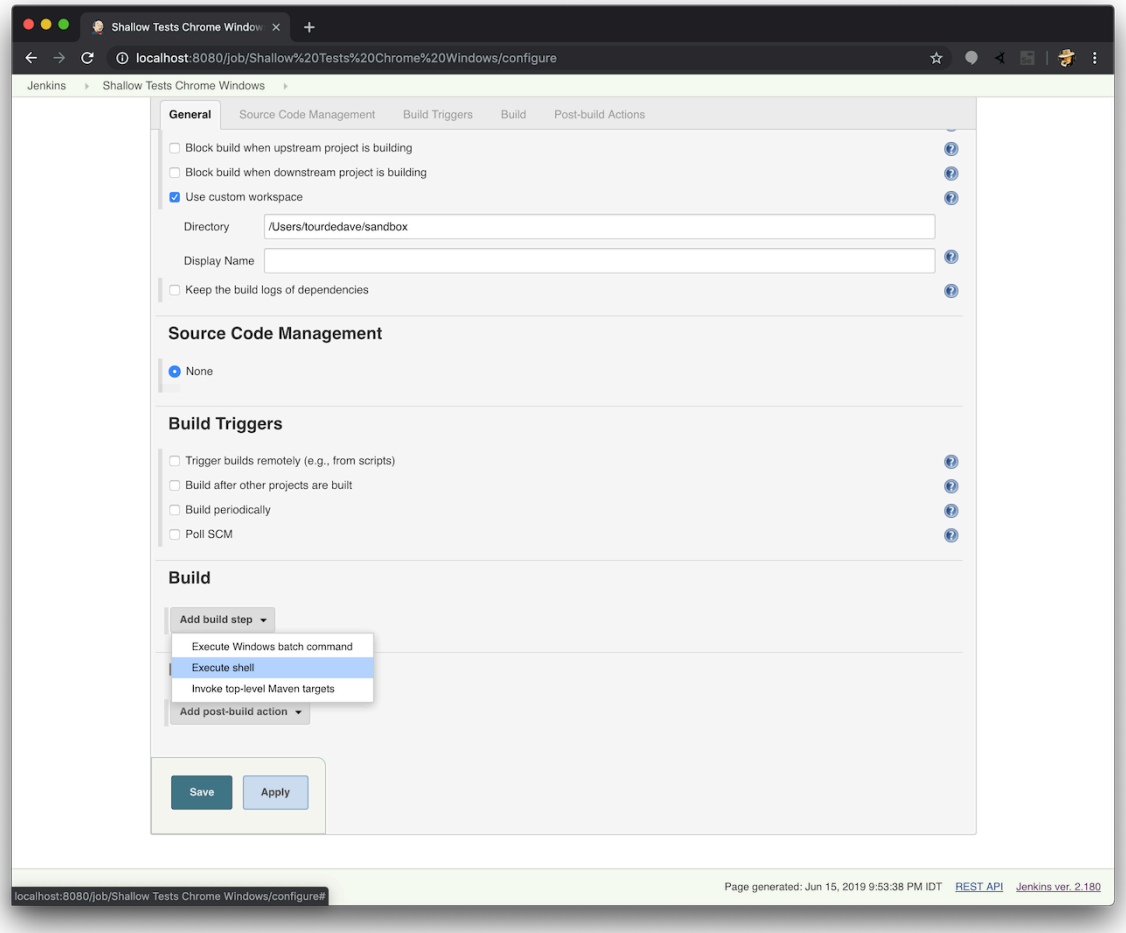


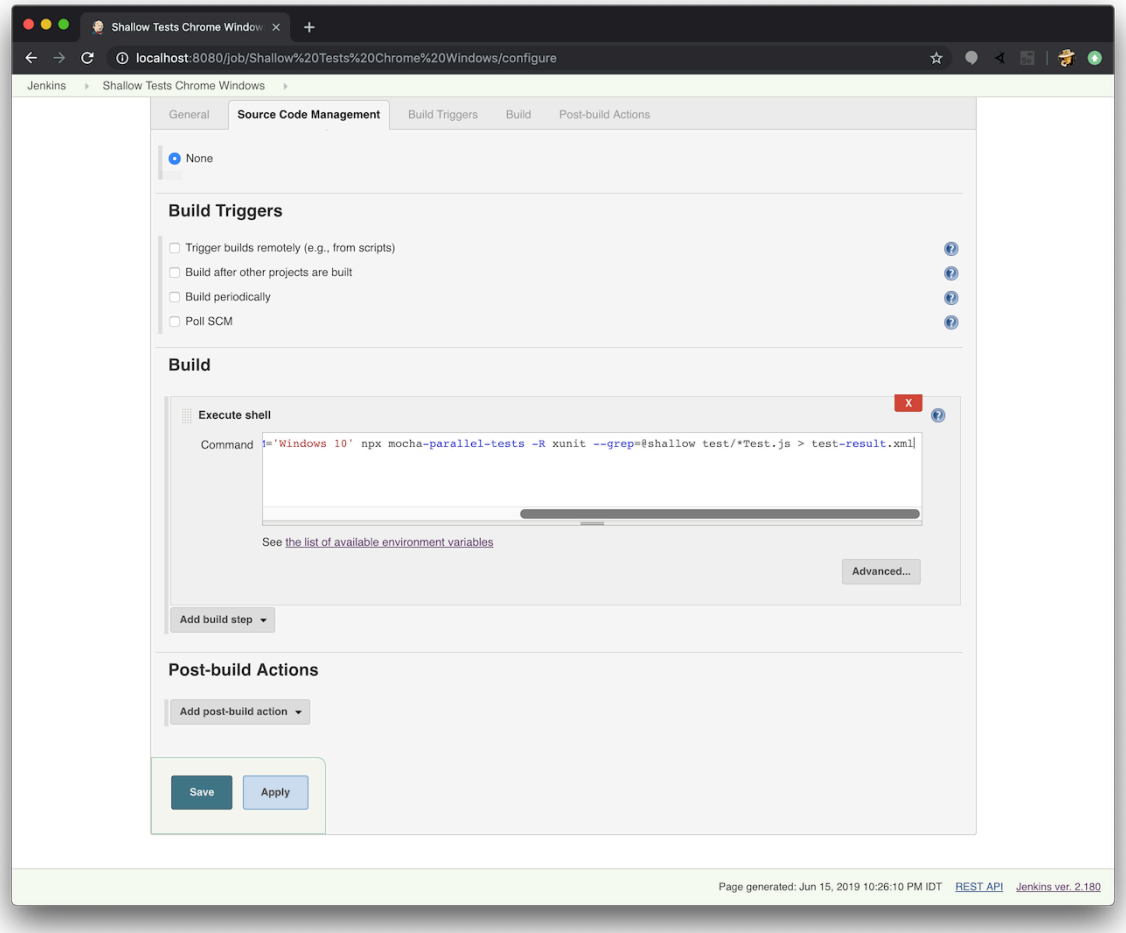
- In the `Advanced Project Options` section select the `Advanced` button
- Choose the checkbox for `Use custom workspace`
- Provide the full path to your test code
- Leave the `Display Name` field blank



NOTE: Ideally, your test code would live in a version control system and you would configure your job (under **`Source Code Management`**) to pull it in and run it. To use this approach you may need to install a plugin to handle it. For more info on plugins in Jenkins, go here.

- Scroll down to the `Build` section and select `Add build step`
- Select `Execute shell`
- Specify the commands needed to launch the tests

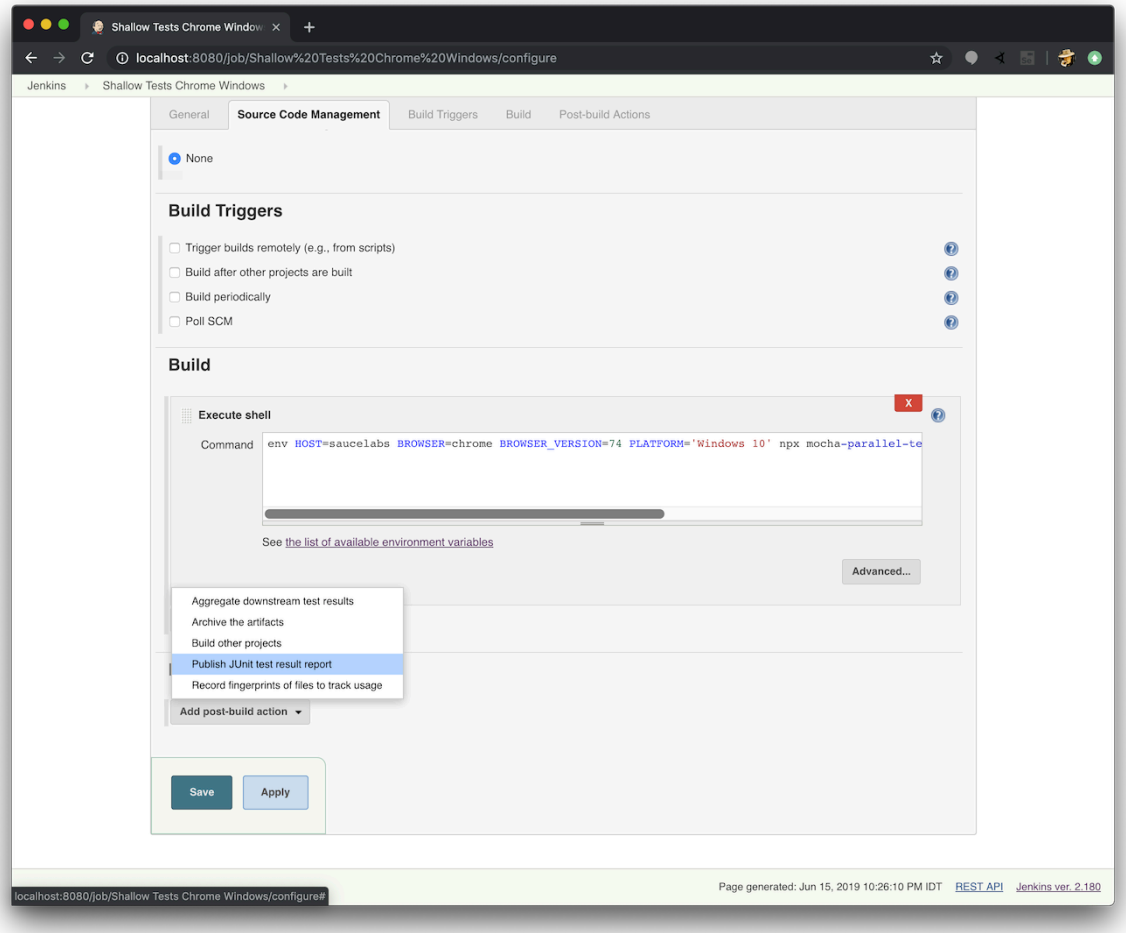

Shallow Tests Chrome Windows
localhost:8080/job/Shallow%20Tests%20Chrome%20Windows/configure
Jenkins
Shallow Tests Chrome Windows
General
Source Code Management
Build Triggers
Build
Post-build Actions
Block build when upstream project is building
Block build when downstream project is building
Use custom workspace
Directory
/Users/tourdedave/sandbox
Display Name
Keep the build logs of dependencies
Source Code Management
None
Build Triggers
Trigger builds remotely (e.g., from scripts)
Build after other projects are built
Build periodically
Poll SCM
Build
Add build step
Execute Windows batch command
Execute shell
Invoke top-level Maven targets
Add post-build action
Save
Apply
Page generated: Jun 15, 2019 9:53:38 PM IDT
REST API
Jenkins ver. 2.180
localhost:8080/job/Shallow Tests Chrome Windows/configure#

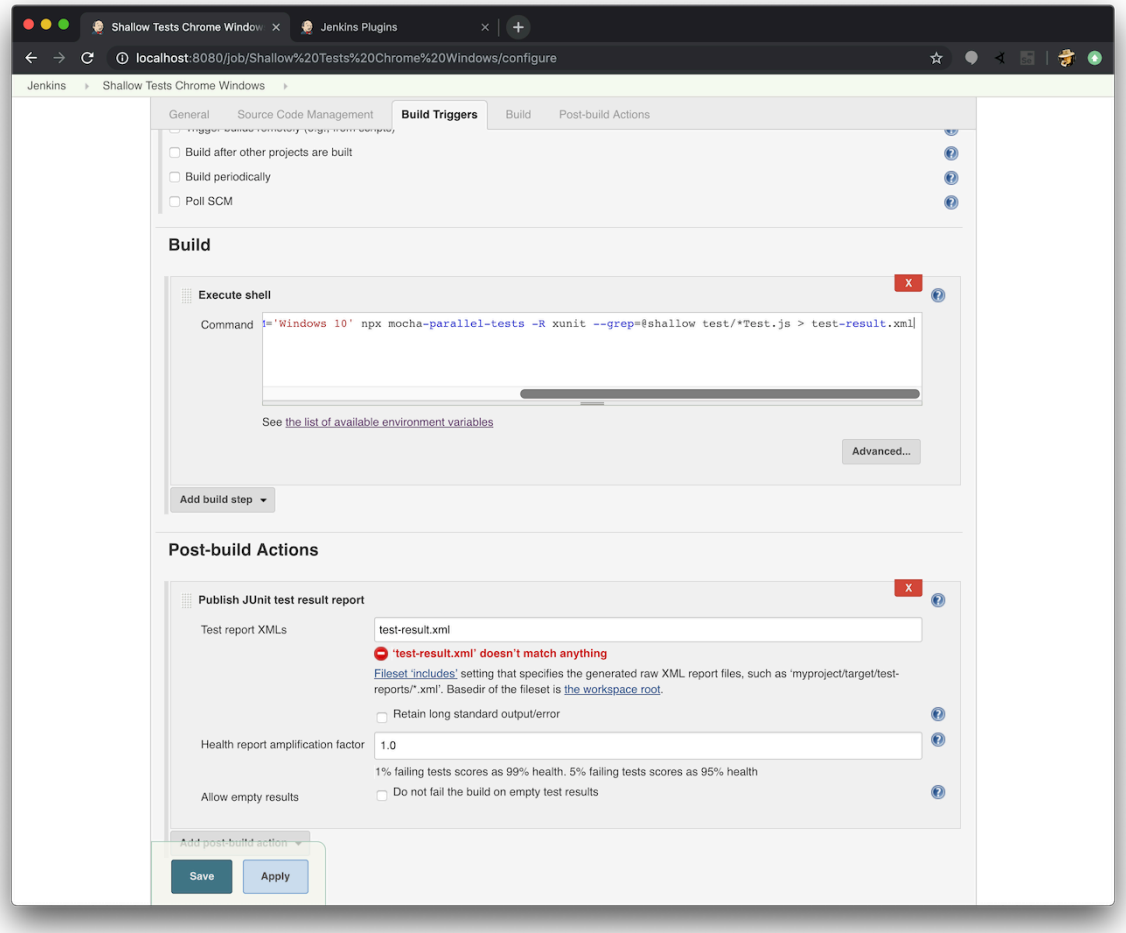



```
CI=on TAG=shallow rake cloud['chrome','75','Windows 10']
```

- Under `Post-build Actions` select `Add post build action`
- Select `Publish JUnit test result report`
- Add the pattern for the result files outputted from a test run -- `tmp/*.xml`
- Click `Save`

NOTE: If this post build action isn't available to you, you will need to install the JUnit Jenkins plugin.


Shallow Tests Chrome Window
localhost:8080/job/Shallow%20Tests%20Chrome%20Windows/configure
Jenkins
Shallow Tests Chrome Windows
General
Source Code Management
Build Triggers
Build
Post-build Actions
None
Build Triggers
Trigger builds remotely (e.g., from scripts)
Build after other projects are built
Build periodically
Poll SCM
Build
Execute shell
Command
env HOST=saucelabs BROWSER=chrome BROWSER_VERSION=74 PLATFORM='Windows 10' npx mocha-parallel-te
See the list of available environment variables
Advanced...
Aggregate downstream test results
Archive the artifacts
Build other projects
Publish JUnit test result report
Record fingerprints of files to track usage
Add post-build action
Save
Apply
Page generated: Jun 15, 2019 10:26:10 PM IDT
REST API
Jenkins ver. 2.180
localhost:8080/job/Shallow Tests Chrome Windows/configure#



Now our tests are ready to be run, but before we do, let's go ahead and add a failing test so we can demonstrate the test report.

Part 3: Force A Failure

Let's add a new test method to `LoginTest.js` that will fail every time we run it.

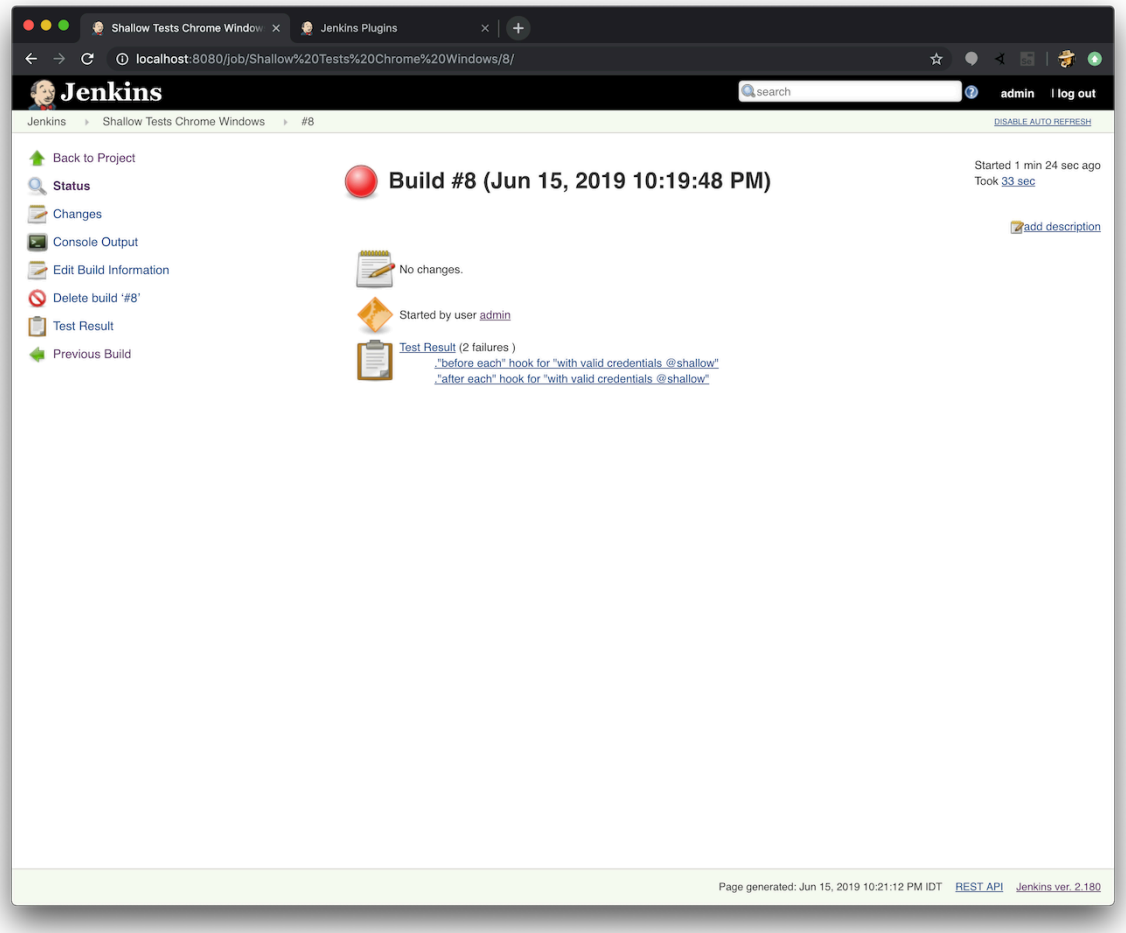
```
# filename: spec/login_spec.rb
## ...
  it 'forced failure' do
    @login.with('asdf', 'asdf')
    expect(@login.failure_message?).to be_falsey
  end

end
```

This test mimics our `'failed'` test by visiting the login page and providing invalid credentials. The differences here are in the assertion and the tag. It will fail since the failure message will be displayed, and we want it to run as part of our `@shallow` suite.

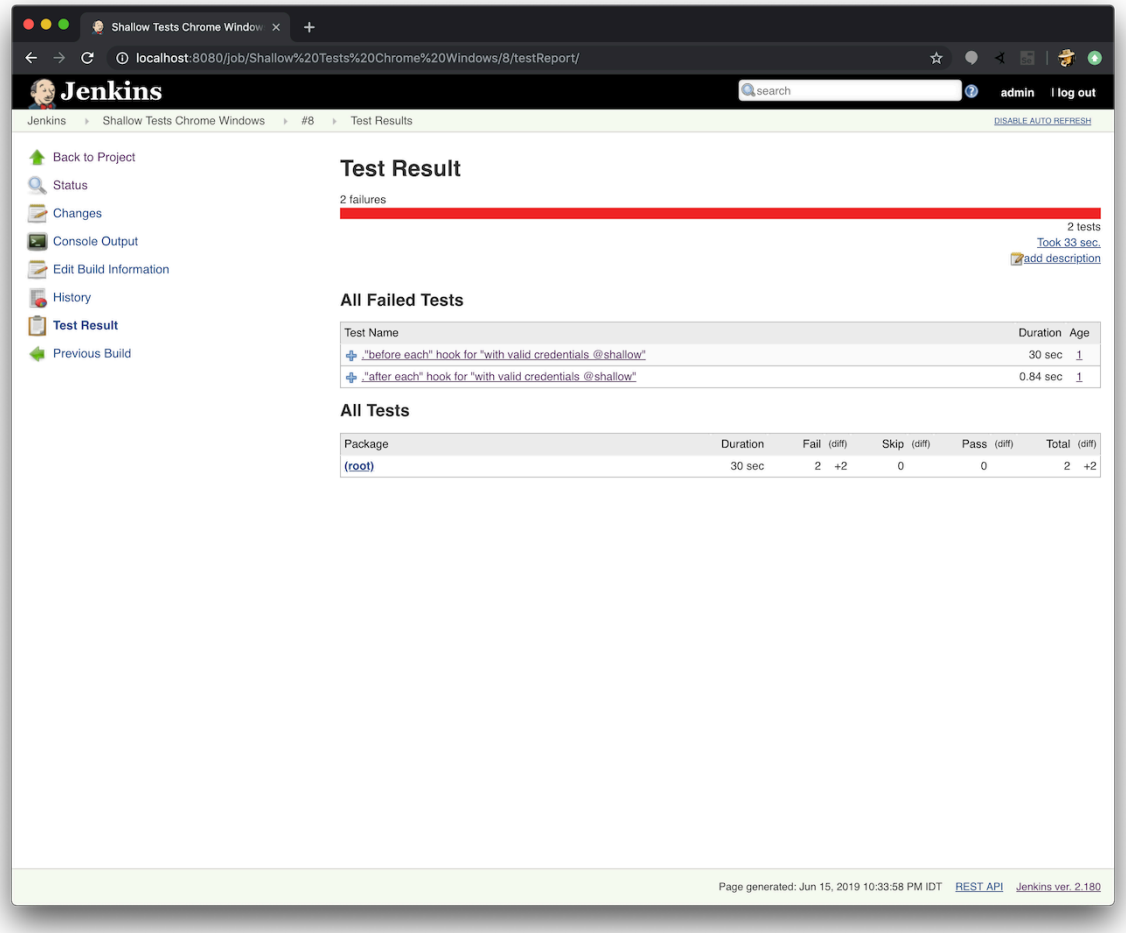
Now let's run our Jenkins job by clicking `Build Now` from the left-hand side of the screen.

NOTE: You can peer behind the scenes of a job while it's running (and after it completes) by clicking on the build you want from `Build History` and selecting `Console Output`. This output will be your best bet in tracking down an unexpected result.

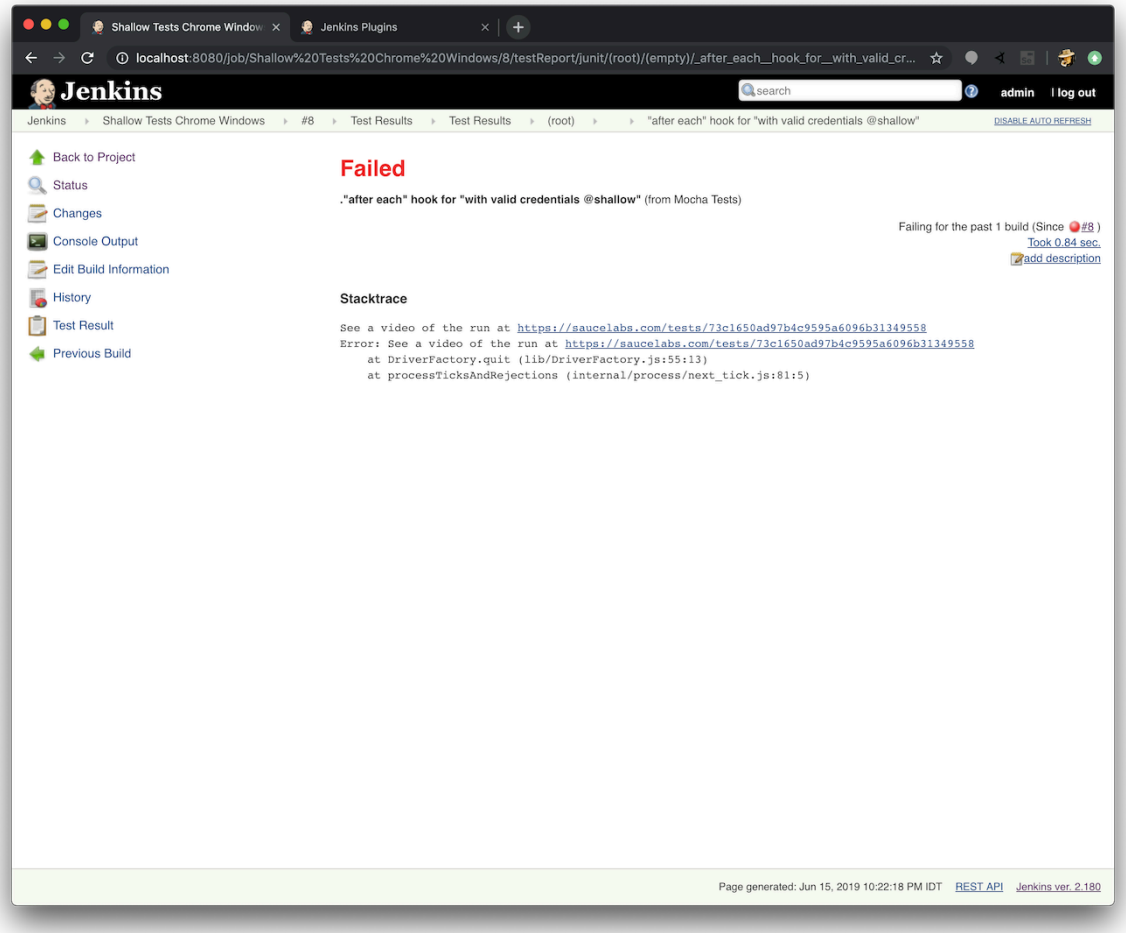
When the test completes, it will be marked as failed.



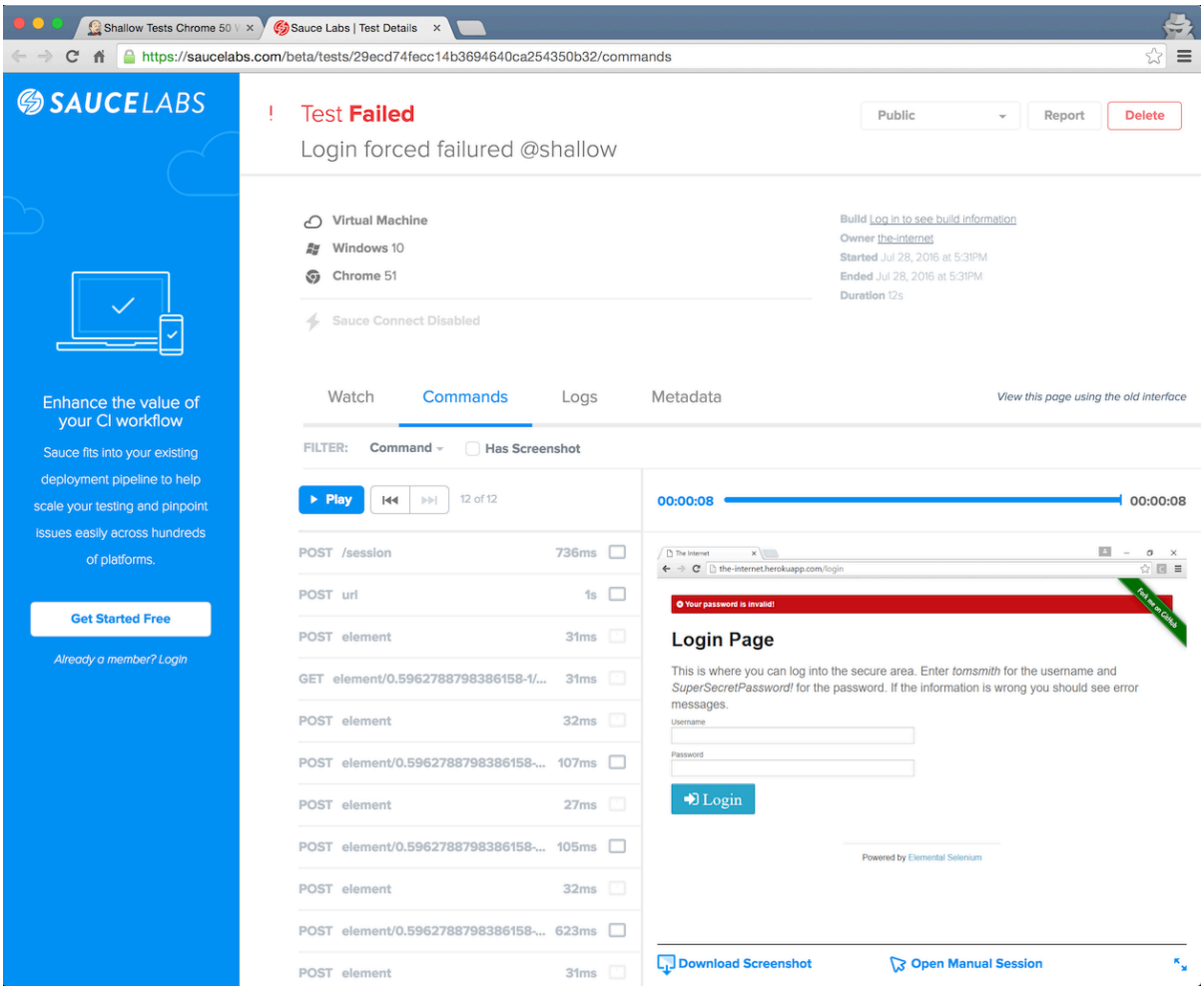
When we click on `Latest Test Result` we can see the test that failed (e.g., `Login.forced failure @shallow`). The other failure listed (e.g., `,"after each" hook for "forced failure @shallow"`) is from the teardown of our test. It contains the Sauce Labs job URL.



If we click on the failed test we can see the stack trace from the test failure. If we click on the failure from the teardown we can see the URL to the job in Sauce Labs.



When we follow the URL to the Sauce Labs job we're able to see what happened during the test run (e.g., we can replay a video of the test, see what Selenium commands were issued, etc.).



Notifications

In order to maximize your CI effectiveness, you'll want to send out notifications to alert your team members when there's a failure.

There are numerous ways to go about this (e.g., e-mail, chat, text, co-located visual cues, etc). Thankfully there are numerous, freely available plugins that can help facilitate whichever method you want. You can find out more about Jenkins' plugins here.

For instance, if you wanted to use chat notifications and you use a service like Slack, you would do a plugin search:

After installing the plugin, you will need to provide the necessary information to configure it (e.g., an authorization token, the channel/chat room where you want notifications to go, what kinds of notifications you want sent, etc.) and then add it as a `Post-build Action` to your job (or jobs).

After installing and configuring a plugin, when your CI job runs and fails, a notification will be sent to the chat room you configured.

Ideal Workflow

In the last chapter we covered test grouping with categories and applied some preliminary ones to our tests (e.g., "Shallow" and "Deep"). These categories are perfect for setting up an initial acceptance test automation workflow.

To start the workflow we'll want to identify a triggering event. Something like a CI job for unit or integration tests that the developers on your team use. Whenever that runs and passes, we can trigger our "Shallow" test job to run (e.g., our smoke or sanity tests). If the job passes then we can trigger a job for "Deep" tests to run. Assuming that passes, we can consider the code ready to be promoted to the next phase of release (e.g., manual testing, push to a staging, etc.) and send out a relevant notification to the team.

NOTE: You may need to incorporate a deployment action as a preliminary step before your "Shallow" and "Deep" jobs can be run (to make sure your tests have an environment available to be run against). Consult a developer on your team for help if that's the case.

Outro

By using a CI Server you're able to put your tests to work by using computers for what they're good at -- automation. This frees you up to focus on more important things. But keep in mind that there are numerous ways to configure your CI server. Be sure to tune it to what works best for you and your team. It's well worth the effort.

Chapter 17
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 70 different Selenium problems and solutions covered. They're in Ruby, but the code has been open-sourced with a fair number of them being ported into other programming languages. You can find the code for them here.

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 Testing Tools Cookbook

This book outlines some great ways to leverage Selenium. It's clear that Gundecha has a very pragmatic approach that will yield great results.

- Selenium Design Patterns and Best Practices

Dima Kovalenko's book covers useful tactics and strategies for successful test automation with Selenium. I was a technical reviewer for the book and think it's a tremendous resource. The book covers Ruby, but he has ported the examples to Java. You can find them here.

Meetups

- 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 the official conference of the Selenium project where practitioners and committers gather and share their latest knowledge and experiences with testing. There are two conferences a year, with the location changing every time (e.g., it's been in San Francisco, London, Boston, Bangalore, Portland, Austin, Berlin, Chicago, and Tokyo).

- Selenium Camp

This is an annual Selenium conference in Eastern Europe (in Kiev, Ukraine) organized by the folks at XP Injection. It's a terrific conference. If you can make the trip, I highly recommend it.

- List of other testing conferences

A helpful website that lists all of the testing conferences out there.

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.

Mailing Lists

- Selenium Developers List

This is where developers discuss changes to the Selenium project, both technically and administratively.

- 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.

Forums

- Stack Overflow
- Quora
- Reddit

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 -- assuming you're able to provide a short and self-contained example that reproduces the problem.

This is known as SSCCE (a Short, Self Contained, Correct (Compilable), Example). For a tongue-in-cheek take on the topic, see this post.

Chatting With the Selenium Community

The Selenium 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.

You can connect either through Slack or IRC. Details on how to connect are available here.

Once connected, feel free to say hello and introduce yourself. But more importantly, ask your question. 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 need to hang around for a bit. But the benefit of being a fly on the wall is that you gain insight into other problems people face, possible solutions, and the current state of the Selenium project and its various pieces.

Chapter 18
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!

If you have any questions, feedback, or want help -- get in touch!

Cheers,
Dave H