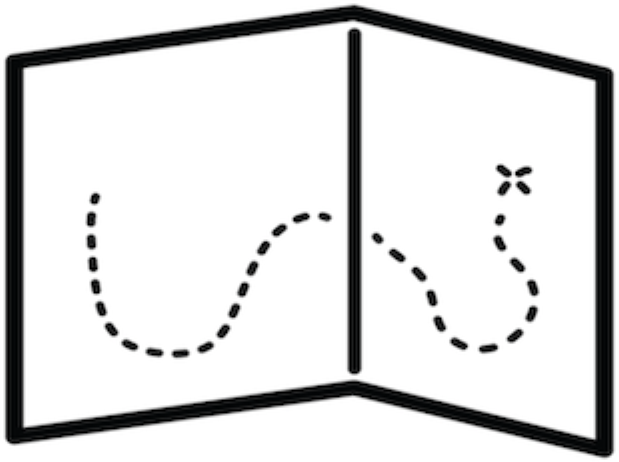
How to use Selenium, successfully



The Selenium Guidebook

by Dave Haeffner

Preface

This book is not a full and comprehensive treatise that outlines every possible permutation of Selenium. There are other books that already do this. My goal, instead, is to teach you the necessary bits to use Selenium successfully for your circumstance.

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

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

What This Book Will Cover

This book focuses primarily on Selenium WebDriver and it's use to test desktop browsers. Selenium RC will be covered only insofar as to help you upgrade to WebDriver. And mobile testing with Appium will also (briefly) be covered.

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

Who This Book Is For

This book is for anyone who wants to take automated acceptance testing seriously and isn't afraid to get their hands a little dirty. That is to say, this book is for anyone who wants to use computers for what they are good at, and free up you and people on your team to do what they are inherently good at (which does not include repetitive, mundane testing tasks). And don't worry if you're super new to programming, I'll cover the basics so you'll have a good platform to build from.

About The Examples In This Book

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

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

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

How To Read This Book

Chapters 1 through 5 are more focused on priming you with knowledge and questions to consider when it comes to strategy, language selection, and test design. Chapter 6 is where we first start to code. And the examples build upon each other up through chapter 14.

Chapter 15 is a culmination of 13 sub-chapters which outline things you will likely see and how to address sthem. They do not build upon each other or other parts of the book, and are meant to stand alone.

Table of Contents

Chapter 1
Selenium In A Nutshell

What Selenium Is And Is Not Good At

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

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

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

It's less ideal for checking lower-level functionality like HTTP status codes or HTTP headers. While it can be achieved, additional setup with a third-party tool (e.g. BrowserMob Proxy) is required since this functionality is not available out of the box -- and it can be a bit of a rabbit hole to fall down since there are numerous cases to consider.

Selenium Highlights

Selenium works on every major browser, for a majority of programming languages, and on every major operating system. Each language binding and browser are actively being developed for to stay current. Yes, even Internet Explorer (thanks to Jim Evans!).

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

One of the guiding philosophies of Selenium is to be able to write your tests once and run them across multiple browsers. While this is a rosy picture, it's not entirely accurate. There are plenty of gotchas to watch out for. We'll cover these in detail throughout the book.

Chapter 2

Defining a Test Strategy

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

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

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

After answering these questions you should have an understanding of the functionality and browsers that matter to the application you are testing. This will help you narrow down your efforts to the things that matter most.

What To Do With The Answers

Question 1 - Money/Value

Every company's application makes money (or generates value) through a core set of functionality -- a.k.a. a 'funnel'. Your answers to this question will help you determine what this is. And this will be your highest priority for automation. Start an automation backlog to keep track.

Question 2 - Usage Data

But odds are your application offers a robust set of functionality that grows well beyond the funnel. Your answers to this question will help highlight what it is. And if you're basing these answers on usage data then it will be broken down from highly used to lightly used. Tack these items onto your automation backlog based on order 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 ask around your team to get a full list. It typically reads like a laundry list of browser specific issues or functionality that has been flaky or forgotten about in the past. Be sure to check this list against your automation backlog. If it's not there, add it to the bottom of the backlog. If it is there, make a note in the backlog item that this has been an issue in the past. And if the issue has happened numerous times and has the potential to occur again, move that backlog item up in priority. And if issues keep cropping up that are related to a specific browser, compare this browser to your short list from question #3. If it's a browser that's not in your list but it's still important (e.g. a small pocket of users), track it on the backlog, but put it at the bottom.

Now Your Are Ready

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

Some Closing Thoughts

This strategy will help you focus your testing efforts, avoid wasting time, and increase your confidence in the approach you are taking.

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.

Chapter 3
Picking a Language

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

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

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

What To Do With The Answer

If you're a Tester (or a team of Testers) and you will be building and maintaining the test automation, then it's best to consider what languages you (and your team) already have experience with, or are interested in gaining, rather than what the application is built in. Have a discussion with your team to see what interests them.

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

But if you're a Developer that 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 grow 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 -- growing a framework (a.k.a. a test harness) and writing automated tests.

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

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

- ChemistryKit (Ruby)
- Geb (Groovy)
- Saunter (Python and PHP)

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

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

Chapter 4
A Programming Primer

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

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

Installation

Ruby is a fast changing ecosystem. In order to find the latest on how to install Ruby, I encourage you to go here.

Installing Third-party Libraries

One of the main benefits of Ruby is that it has a vibrant open source community with copious libraries (a.k.a. "gems") immediately available making it extremely simple to quickly build complex things. You can find out more information on gems here.

A helpful library called Bundler is a great way to easily manage gems within a project. This is what I use and what was used in building out the examples for this book.

Programming Concepts In A Nutshell

Programming can be a deep and indimidating rabbit hole if you're new to it. But don't worry. When it comes to testing, there is only a small subset that we really need to work with to get started. Granted, the more you know the better off you'll be. But you don't need to know a whole lot to be an effective test automator right now.

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

- Object Structures (Variables, Methods, and Classes)
- Object Types (Strings, Integers, Booleans, Collections)
- Assertions
- Conditionals
- Looping
- Inheritance

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

Object Structures

Variables

Variables are a helpful place to store things of various types (e.g. Strings, Integers, Booleans, Collections, etc).

They can be one or more words in length, start with a lowercase letter, and are often entirely lowercase. If they are more than one word long then an underbar is used to separate the words. You can place things into them by using an `=` sign. And the variable takes on the type of the object you store in it.

There are a handful of different variable types. But the ones we care most about for our test code are:

- local
- instance
- constants

Local and instance variables are similar except that they vary in scope. That is to say, local variables enable you to store and access things within a confined area, whereas instance variables (annotated with a prepended `@`) let you store and access things more broadly.

A common example of an instance variable is the instantiation of a Selenium WebDriver object

```
@driver = Selenium::WebDriver.for :firefox
```

This happens before any tests run, and within our test framework it is made available across all tests. If this were stored in a local variable, then the tests would not be able to use it.

And a common example of a local variable is storing an element (or a value from an element) in the beginning of a test to be referenced at the end of the test (e.g. for an assertion). In this scenario, the variable will only be available to that test when it is running.

```
it 'sample test' do
  @driver.get 'http://the-internet.herokuapp.com'
  title = @driver.title
  ...
```

Constants are for storing information that will not dynamically change. They are easy to spot since they start with a capital letter, and are often all uppercase. These are commonly used to store element locator information at the top of Page Objects.

```
class Login do

  LOGIN_FORM = '#login'
  USERNAME_INPUT = '#username'
  PASSWORD_INPUT = '#password'
  ...
```

Methods

One way to group common actions and behavior into helpful containers for easy reuse is to place them into methods.

An example of this would be the explicit wait action in Selenium.

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

We define a method with `def` and name it like we would a variable. Additionally, we can specify arguments we want to pass along when calling the method -- in this case, the amount of time we want to wait (in seconds). When setting this argument, we can also set a default to use if no argument is provided.

Classes

Classes are a useful way to represent complex, reusable concepts that will appear numerous times in our test code. They are large objects that can house multiple things (e.g. variables, methods, etc).

An example of this is the creation of a class that represents a page you are likely to use often (e.g. a login page). By using a class you can store elements you will interact with in constants, and create methods that capture the page's unique behavior. And then in your tests you simply create an instance of this class and call its methods to complete your test actions.

Object Types

Strings

Strings are alpha-numeric characters strung together (e.g. letters, numbers, and (most) special characters) surrounded by either single (`'`) or double (`"`) quotes. Single quotes are encouraged unless you intend to manipulate the string value when consuming it (a.k.a. interpolation). In that case, then use double-quotes.

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

```
it 'sample test' do
  @driver.get 'http://the-internet.herokuapp.com'
  heading_text = @driver.find_element(css: 'h2').text
  puts heading_text.class # this will tell you the object type
  ...
```

Integers

You can do a lot of math in Ruby. But the two common types of Integers you will likely run into with testing are Fixnum (whole numbers) and Float (decimals).

If you have a test that needs to pull some values from a page and add/subtract/multiply/divide/etc them, then this will come in handy. Although, you may need to convert the values from a String to an Integer first -- which is trivial to do in Ruby since it is a dynamically typed language.

```
integer_value = "8".to_i
```

Booleans

Booleans are binary values -- they are either `true` or `false`. You will see these when asking questions of your test. The answer returned (in most cases) will either be `true` or `false`. Without booleans you would not be able to make assertions or use conditionals.

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

Collections

Collections enable you to gather a set of data for later reference.

In Ruby there are two types of built-in collections -- Arrays and Hashes. Arrays are stored in an ordered listed with an index number. And Hashes store values in the order they were inputted -- but use a key, value pair to store and retrieve data.

You'll end up working with Arrays (or some type of enumerable object that is of a similar construct) if you need to test things like HTML Data Tables, Dropdown Lists, or if you need to take an action on a specific element in a list that is not easy to access based on how it's designed.

Looping

Collections wouldn't be nearly as valuable without the ability to loop over them. Alternatively, you can also use looping to a specific action a certain number of times, or until a certain precondition is met. This is useful if you want to make more resilient tests. But let's continue on with the example we started with Dropdowns.

Assertions

Assertions are a crucial element for all of your tests. They are what determines whether your test will pass or fail. And they are fueled by booleans.

```
it 'sample test' do
  @driver.get 'http://the-internet.herokuapp.com'
  @driver.title.include?('The Internet').should be_true
end
```

Conditionals

You can also put your booleans to work by making your tests do different things based on the boolean response. This will be done through the use of `if` and `case` statements.

We'll run into these when working with test setup to determine which browse to use and where to run our tests.

Inheritance

Within classes is the ability to connect them 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.

An example of this is writing all of the Selenium actions you intend to use into methods in a parent class and then using these methods in all of your child classes. This way if there are ever any changes to the Selenium API, you only have to change them in one place, and your code will be much more readable as a whole.

Additional Resources

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

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

Chapter 5
Anatomy Of A Good Acceptance Test

In order to write effective acceptance tests that are performant, maintainable, and resilient, there are some simple guidelines to follow:

- Write atomic and autonomous tests
- Group like tests together in small batches
- Be descriptive
- Use a Test Runner
- Store it 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 able to be run independently (e.g. sets up its own data rather than relying on a previous test to do it). Doing this may require a mental shift, discipline, and more up front effort. But it will make a dramatic impact on the quality, effectiveness, and maintainability of your tests.

Grouping Tests

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

Being Descriptive

Each test file should be named appropriately, and each test within it should have an informative name (even if it may be a bit verbose). Also, each test (or grouping of tests) should be tagged for flexible execution later (e.g. on a Continuous Integration server).

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

Test Runners

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

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

Version Control

In order to effectively collaborate with other Testers and Developers, your test code must live in a version control system of some sort (e.g. git, subversion, etc). Look to see what your Development Team uses and get a repository stood up there.

Keep in mind that your test code can live separate from the code of the application you're testing. There are cases and advantages where combining them may be advantageous. But if all your doing is writing and running tests against web endpoints (which is a majority of what your testing will be) then leaving your test code in a separate repository is a solid way to go.

Chapter 6
Writing Your First Test

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

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

1. Visit the main page of a site
2. Find the login button, and click it
3. Find the login form's userame field and input text
4. Find the login form's password field and input text
5. Find the login form and submit it

Selenium is able to find and interact with elements on a page by way of various locator strategies. The list includes Identifier, Id, Name, Link, DOM, XPath, and CSS.

While each serves a purpose there is one approach that is best of breed. One that is cross-browser performant, simpler to maintain, and leverages code on the page -- and that's CSS Selectors.

A Quick Primer on CSS Selectors

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

How To Find CSS Selectors

The simplest way to find CSS Selectors is to inspect the elements on a page. The best way to do this is from within your web browser. Fortunately popular browsers of today 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 attributes you can work with (e.g. `id`, `class`, etc). And at the bottom of the window a set of CSS selectors will be listed showing you the heirarchy of the elements on the page that lead up to your selection. In some cases this info may be useful, in others, not so much.

From here you should able to get started with constructing a CSS Selector to use in your test.

How To Find Quality Elements

Your focus with picking an effective element should be on finding something that is unique and unlikely to change. Ripe candidates for this are `id` and `class` attributes (in CSS Selector parlance `id`'s are prepended with a `#`, and `class`'s are prepended with a `.`). Whereas copy (e.g. 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, symantic 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

Let's take our login example from before and put it to test code.

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

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

Note the unique elements on the form. The username field has a unique `id`, as does the password field. The submit button doesn't, but the parent element does. Now let's put them to use in our first test (or 'spec' in RSpec parlance).

```ruby
# filename: login_spec.rb

require 'selenium-webdriver'

describe 'Login' do

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

  after(:each) do
    @driver.quit
  end

  it 'successful' do
    @driver.get 'http://the-internet.herokuapp.com/login'
    @driver.find_element(css: '#username').send_keys('username')
    @driver.find_element(css: '#password').send_keys('password')
    @driver.find_element(css: '#login').submit
  end

end
```

At the top of the file we are pulling in `selenium-webdriver` which is the third-party library (or 'gem' in Ruby parlance) that gives us the bindings necessary to use Selenium to drive the browser.

Next is the `describe`, which is effectively the title of the test. A simple and helpful name is chosen to note the intent of the file (e.g. 'Login').

In order to use Selenium, we need to instantiate it (with Firefox in this case). And when we're done, we don't want the browser we opened to hang around, so we close it. This is what we're doing in the `before(:each)` and `after(:each)` blocks. Note that we are storing our instance of Selenium in an instance variable (`@driver`) so it is available throughout our test.

And lastly is the `it` block, which is the test. And similar to the `describe` block, we can give it a simple and helpful name (e.g. 'successful').

In the test we are interacting with the unique CSS Selectors we identified from the markup by first finding them (e.g. `#username`, `#password`, and `#login`) and then taking an action against them. Which in this case is inputting text (with `send_keys`) and submitting the form (with `submit`).

If we run this (e.g. `rspec login_spec.rb` from the command-line), it will work and pass. But there's one thing missing -- an assertion. But in order to find an element to make an assertion against, we need to see what the markup is 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 it, there looks to be a couple of things we can key off of for our assertion. There's the flash message that appears at the top of the page (most appealing), the logout button (appealing), or the copy from the h2 (least appealing). Let's use the flash message.

Now, with the flash message, there are two options. We can either leverage the unique class name, or the copy that it displays. Since the class name is descriptive and denotes success (which is the use case we are concerned with) let's go with that.

```
require 'selenium-webdriver'

describe 'Login' do

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

  after(:each) do
    @driver.quit
  end

  it 'successful' do
    @driver.get 'http://the-internet.herokuapp.com/login'
    @driver.find_element(css: '#username').send_keys('username')
    @driver.find_element(css: '#password').send_keys('password')
    @driver.find_element(css: '#login').submit
    @driver.find_element(css: '.flash.success').displayed?.should be_true
  end

end
```

Just To Make Sure

Now when we run this test 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. A simple fudging of the CSS Selector will suffice.

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

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

Chapter 7
Writing Re-usable Test Code

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

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

Enter Page Objects.

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

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

An Example

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

Let's start by creating the page object.

```
# filename: login.rb

class Login

  PAGE_HEADING = 'h2'
  LOGIN_FORM = '#login'
    USERNAME_INPUT = '#username'
    PASSWORD_INPUT = '#password'
  SUCCESS_MESSAGE = '.flash.success'

  attr_accessor :username, :password
  attr_reader :driver

  def initialize(driver)
    @driver = driver
    visit
    verify_page
  end

  def visit
    driver.get ENV['base_url'] + '/login'
  end

  def now
    driver.find_element(:css, USERNAME_INPUT).send_keys(username)
    driver.find_element(:css, PASSWORD_INPUT).send_keys(password)
    driver.find_element(:css, LOGIN_FORM).submit
  end

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

  private

    def verify_page
      driver.find_element(:css, PAGE_HEADING).text.should == 'Login Page'
    end

end
```

We roll our own class, naming it accordingly (`Login`), and store the locators we use on this page as constants. Next we use the helpful `attr` constructs available in Ruby. They are short-hand for creating values that can be stored, referenced, and manipulated (a.k.a. setters and getters) depending on which type of attr is used (e.g. accessor, reader, writer).

In the case of `:username, :password`, we are making it so we can store and use the username and password values from both outside the class (e.g. in our test) and inside (e.g. in our class's methods). As for `:driver`, we are using it for a stylistic advantage -- so we don't have to litter our page object with unneccessary instance variables (e.g. `@driver` becomes `driver`).

We then use an `initialize` method. This is a built in method for classes in Ruby that enable you to execute code upon instantiating the class (before you do anything else). And with it we are taking an argument -- the Selenium driver object. Our test has the driver object, and when we instantiate this class we want to pass it in so the class can then drive the browser.

In the `initialize` method itself we are setting an instance variable and calling two methods. The instance variable setting (`@driver = driver`) is how we are are passing the Selenium driver object into the `attr_reader :driver` object (so that we don't have to use an instance variable to reference it) in our class methods. The `visit` method is a way to cleanly abstract navigation to the page. In it we are referencing an environemnt variable that should contain the base URL of the application we want to test, and appending the path to the page for this page object. The use of an environment variable is a simple way to pass global information like this between the test and the page object without explicitly passing it in. And the `verify_page` is more for sanity. The rule of thumb is to keep your assertions in your tests (and out of your page objects), but this is an exception to that rule. With it we can make sure that the browser is in the right place before proceeding. If not, the test will fail. While this is a helpful method, it's really only useful within the class, so it's marked as `private` (effectively making it invisible to our tests) and placed at the bottom of the class.

This leaves us with the behavior and assertion methods -- `now` and `success_message_present?`.

In `now` we are excercising the login form by inputting the user credentials and submitting the form. It's named `now` for simplicity. You can break apart these actions (inputting and submitting) if you wish, or name it something more explicit, but there's no pressing need, so we'll leave it for now.

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

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

```
# filename: login_spec.rb

require 'selenium-webdriver'
require_relative 'login'

describe 'Login' do

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

  after(:each) do
    @driver.quit
  end

  let(:login) { Login.new(@driver) }

  it 'successful' do
    login.username = 'username'
    login.password = 'password'
    login.now
    login.success_message_present?.should be_true
  end

end
```

At the top of the file we include the page object file by using a `require_relative`. This enables us to reference a file relative to our present working diretory. We then insantiate the environment variable for the base_url (`ENV['base_url']`). Since we only have one test file, it will live here and be hard-coded for now. But don't worry, we'll address this later.

Next we set up our login page object for instantiation by using RSpec's `let` functionality. With it we can set the class to a variable that will have a local scope (`login` instead of `@login`) that will only be loaded if it's referenced in a test (a.k.a. memoized).

Last we wire up our test to use the page object and make an assertion against the final outcome (`login.success_message_present?.should be_true`).

Simpler Test Writing

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

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

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

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

First we add a new locator for the failure message at the top of the file, just below the success message locator.

```
# filename: login.rb

class Login

  PAGE_HEADING = 'h2'
  LOGIN_FORM = '#login'
    USERNAME_INPUT = '#username'
    PASSWORD_INPUT = '#password'
  SUCCESS_MESSAGE = '.flash.success'
  FAILURE_MESSAGE = '.flash.error'
```

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

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

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

Lastly, we add a new test in our spec file.

```ruby
it 'failed' do
  login.username = 'asdf'
  login.password = 'asdf'
  login.now
  login.failure_message_present?.should be_true
end
```

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

Chapter 8

Writing Really Re-usable Test Code

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

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

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

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

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

An Example

First we will need to create the base page object.

```
# filename: base_page.rb

require 'selenium-webdriver'

class BasePage

  attr_reader :driver

  def initialize(driver)
    @driver = driver
  end

  def find(locator)
    driver.find_element(:css, locator)
  end

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

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

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

  def text_of(locator)
    find(locator).text
  end

end
```

We first `require` our Selenium bindings. By placing them here, we no longer need to include it anywhere else in our test suite. And it makes more sense since this is the only file where we will be referencing Selenium commands directly. Next we declare our class and name it appropriately with camel-casing (`BasePage`).

True to how we did things in our original page object, we will use the `attr` construct here to store and reference the `driver` object. And this is the only thing that the `initialize` method is responsible for. That leaves the common methods that we will want to reference in our page objects -- `find` , `type` , `submit` , `is_displayed?` , and `text_of` .

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

```ruby
# filename: login.rb

require_relative 'base_page'

class Login < BasePage

  PAGE_HEADING = 'h2'
  LOGIN_FORM = '#login'
    USERNAME_INPUT = '#username'
    PASSWORD_INPUT = '#password'
  SUCCESS_MESSAGE = '.flash.success'
  FAILURE_MESSAGE = '.flash.error'

  attr_accessor :username, :password

  def initialize(driver)
    @driver = driver
    visit
    verify_page
  end

  def visit
    driver.get ENV['base_url'] + '/login'
  end

  def now
    type USERNAME_INPUT, username
    type PASSWORD_INPUT, password
    submit LOGIN_FORM
  end

  def success_message_present?
    is_displayed? SUCCESS_MESSAGE
  end

  def failure_message_present?
    is_displayed? FAILURE_MESSAGE
  end

  private

    def verify_page
      text_of(PAGE_HEADING).should == 'Login Page'
    end

end
```

In order to use the base page object we need to do two things. First we need to include the file, which is handled using the `require_relative` at the top of the file. And second, we need to connect them using inheritance. This 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.

`initialize` remains unchanged. But if we didn't have any actions that we wanted to run on load, then we wouldn't need `initialize` or the `attr_accessor`.

And to finish things off, we replace all of the driver incantations with calls to the methods we created in the base page object. Our test remains unchanged, but our page objects are now much more readable and flexible. And running our tests (`rspec login_spec.rb`) should still yield a passing result.

Chapter 9
Writing Resilient Test Code

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

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

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

And we accomplish this 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!'.

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

```html
<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.

From here you could do one of two things -- either create a new page object to represent this page and its functionality, or create a quick and dirty test to get your approach down in code. After which, you abstract it out into a page object; adding any necessary common actions to your base page object.

It's really a matter of preference which you choose. For this example I'm going to go with the first approach and dive straight into a page object.

```ruby
# filename: dynamic_loading.rb

require_relative 'base_page'

class DynamicLoading < BasePage

  START_BUTTON = '#start button'
  FINISH_TEXT = '#finish'

  def visit(example_number)
    driver.get ENV['base_url'] + '/dynamic_loading/' + example_number
  end

  def start
    click START_BUTTON
  end

  def finish_text_present?
    wait_for(6) { 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 the common Selenium actions and access to the driver object. After that we wire up our CSS Selectors in constants and drop in our methods for visiting, starting, and grabbing a boolean for an assertion.

`visit` is set to take an argument because there are actually two dynamic loading examples. But they are both built with the same markup. They just have slightly different behavior and slightly different URLs. So we can actually capture them both in this one page object. And making the `visit` method take an example number as an argument is the only thing we need to do to accommodate that.

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

```ruby
# filename: base_page.rb

require 'selenium-webdriver'

class BasePage
  ...

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

  def wait_for(seconds = 2)
    Selenium::WebDriver::Wait.new(timeout: seconds).until { yield }
  end

end
```

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

With `wait_for` we are accepting an integer for the number of seconds that we would like to wait. If nothing is provided, 2 seconds will be used. We then use this value to tell the explicit wait mechanism that Selenium offers how long to wait, and what to wait for. By using a `yield` we are able to easily pass in a code block to exercise.

So in our page object when we're using `wait_for(6) { is_displayed? FINISH_TEXT }` we are telling Selenium to to see if the finish text is displayed. It will keep trying until either it returns `true` or reaches six seconds and times out -- whichever comes first. This provides a huge benefit to the longer term reliability of our test, because 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). That, and it's dynamic, so it won't always take the full amount of time.

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

```ruby
require_relative 'dynamic_loading'

describe 'Dynamic Loading' do

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

  after(:each) do
    @driver.quit
  end

  let(:dynamic_loading) { DynamicLoading.new(@driver) }

  it 'Example 1: Hidden Element' do
    dynamic_loading.visit_example "1"
    dynamic_loading.start
    dynamic_loading.finish_text_present?
  end

end
```

And when we run it (`rspec dynamic_loading_page.rb`) it should pass.

But Wait, There's More

Remember when I mentioned there were actually two dynamic loading examples? Well, the first one was built to demonstrate a page that has the final result we want already on the page, but hidden until we want it displayed. The second one is the same but slightly different -- the final result we want is not on the page. Instead it gets rendered after the progress bar completes.

Let's see if our approach for the first example holds steady for the second.

Here's the markup for the second example.

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

  <br>

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

  <br>
</div>
```

Note that there isn't any finish text on the page. In order to find the selector for the finish text element (or in this case, just double check it) we need to inspect the page after the loading bar sequence finishes. And it looks like this.

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

Now that we have that, we can wire up a second test in our spec file.

```
it 'Example 2: Renderd after the fact' do
    dynamic_loading.visit_example "2"
    dynamic_loading.start
    dynamic_loading.finish_text_present?
  end
```

And if we run the spec (`rspec dynamic_loading_spec.rb`) the same approach should work for both cases.

By checking to see if something is displayed (rather than just doing a simple find for an element) we are placed in a better position since Selenium will keep trying for the specified amount of time rather than throwing an exception.

Browser Timing

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

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

In my experience, Chrome execution is inherently faster, so you will see some odd timeout failures. This is an indicator that you need to add explicit waits to parts of your page objects that don't already have them. And the inverse is true when running things against older version of Internet Explorer (e.g. IE 8). This is an indicator that your explicit wait times are not long enough. The browser is taking longer to respond and so your tests timeout.

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

Chapter 10
Prepping For Use

Now that we have tests, page objects, and a base page object, let's package things into a more useful structure and add the beginnings of our own lightweight framework.

Global Setup and Teardown

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

```ruby
# filename: spec_helper.rb

require 'selenium-webdriver'

RSpec.configure do |config|

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

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

end
```

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

Base URL

As for setting our base URL, we want to do two things. First, put it someplace that's a little more configurable. So we'll create a 'config.rb' file and place it there.

```ruby
# filename: config.rb

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

And second, we only need to reference the base_url when visiting a page. So rather than have it in each page object, let's create a generic visit action in our base page object and reference that instead.

```
# filename: base_page.rb

class BasePage
  ...

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

end
```

Then it's just a simple matter of updating our existing page objects usage of visit.

```
# filename: dynamic_loading.rb

def visit_example(example_number)
   visit('/dynamic_loading/' + example_number)
end
```

```
# filename: login.rb

def initialize(driver)
  @driver = driver
  visit '/login'
  verify_page
end

# and removed the visit method
```

Now instead of each page object having its own visit method, we are leveraging the one in the base page object which accepts the URL path as an argument.

Folder Organization

It's about time we create some folders for our specs and page objects. It's only a matter of time until things become cluttered and untenable. To do this, we'll need to update the page object require statements in our tests. To err on the side of simplicity, let's call the folders 'spec' and 'pages'. We are using 'spec' in the singular form since that is an RSpec default.

Here's everything we should have once we implement our new folders:

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

And here are what the updated specs look like.

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

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

describe 'Login' do

  let(:login) { Login.new(@driver) }

  it 'successful' do
    login.username = 'username'
    login.password = 'password'
    login.now
    login.success_message_present?.should be_true
  end

  it 'failed' do
    login.username = 'asdf'
    login.password = 'asdf'
    login.now
    login.failure_message_present?.should be_true
  end

end
```

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

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

describe 'Dynamic Loading' do

  let(:dynamic_loading) { DynamicLoading.new(@driver) }

  it 'Example 1: Hidden Element' do
    dynamic_loading.visit_example "1"
    dynamic_loading.start
    dynamic_loading.finish_text_present?
  end

  it 'Example 2: Renderd after the fact' do
    dynamic_loading.visit_example "2"
    dynamic_loading.start
    dynamic_loading.finish_text_present?
  end

end
```

Note the use of `..` in the page object require statement in our specs. This is because we need to traverse up a directory before we can access the page objects folder from our spec directory. And the spec_helper require remains unchanged since this file lives in the spec directory.

Now that things are cleaned up, we can run everything while specifying the config file (`rspec --require ./config.rb`), and everything should pass.

Chapter 11

Running A Different Browser Locally

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

A Brief Primer on Browser Drivers

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

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

Let's step through an example using ChromeDriver.

An Example

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

So now our directory tree looks like this:

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

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

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

```
# filename: config.rb

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

```
# filename: spec/spec_helper.rb

require 'selenium-webdriver'

RSpec.configure do |config|

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

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

end
```

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

Assuming Chrome is already installed on the system (and it's in a standard location) when we run our tests (`rspec -r ./config.rb`), we should see them execute in Chrome. If that's not the case for you, then check out the ChromeDriver documentation site to see if there are additional requirements for your specific system setup.

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

Multiple Config Files

If you want to run Firefox, now all you need to do is update the browser value in 'config.rb'. Alternatively, you can create a second config file and use that at run time. Let's do that.

I'll rename config.rb to configchrome.rb and add a second file called 'configfirefox.rb'.

```
# filename: config_chrome.rb

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

```
# filename: config_firefox.rb

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

Now all we need to do is change which file we use at runtime. For Firefox, we would use `rspec -r ./config_firefox.rb` and for Chrome, `rspec -r ./config_chrome.rb`.

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

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

Additional Browsers

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

- ChromeDriver
- FirefoxDriver
- InternetExplorer Driver
- OperaDriver
- SafariDriver

Chapter 12

Running Browsers In The Cloud

If you've ever needed to test features in an older browser like Internet Explorer 6 then odds are you ran a virtual machine (VM) on your computer with a "legit" version of Windows XP.

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

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

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

Why Sauce Labs?

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

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

A brief primer on Selenium Remote, Selenium Grid, and Sauce Labs

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

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

Let's dig in with an example.

An Example

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

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

```ruby
# filename: config_cloud.rb
ENV['base_url']         = 'http://the-internet.herokuapp.com'
ENV['host']             = 'saucelabs'
ENV['operating_system'] = 'Windows XP'
ENV['browser']          = 'internet_explorer'
ENV['browser_version']  = '8'
ENV['sauce_username']   = 'your_username'
ENV['sauce_access_key'] = 'your_api_key'
```

Notice the use of a host variable. This is what we'll use in our 'spec_helper' file to determine whether to run things locally or in the cloud. That means we'll need to add a host variable to our local configs as well.

```ruby
# filename: config_firefox.rb
ENV['base_url'] = 'http://the-internet.herokuapp.com'
ENV['host']     = 'localhost'
ENV['browser']  = 'firefox'
```

```ruby
# filename: config_chrome.rb
ENV['base_url'] = 'http://the-internet.herokuapp.com'
ENV['host']     = 'localhost'
ENV['browser']  = 'chrome'
```

Now that we have that sorted, let's update our 'spec_helper' file.

```ruby
require 'selenium-webdriver'

RSpec.configure do |config|

  config.before(:each) do
    case ENV['host']
    when 'localhost'
      case ENV['browser']
      when 'firefox'
        @driver = Selenium::WebDriver.for :firefox
      when 'chrome'
        Selenium::WebDriver::Chrome::Service.executable_path = File.join(Dir.pwd,
'vendor/chromedriver')
        @driver = Selenium::WebDriver.for :chrome
      end
    when 'saucelabs'
      caps = Selenium::WebDriver::Remote::Capabilities.send(ENV['browser'])
      caps.version = ENV['browser_version']
      caps.platform = ENV['operating_system']
      caps[:name] = self.example.metadata[:full_description]

      @driver = Selenium::WebDriver.for(
        :remote,
        :url => "http://#{ENV['sauce_username']}:#{ENV['sauce_access_key']}
@ondemand.saucelabs.com:80/wd/hub",
        :desired_capabilities => caps)
    end
  end

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

end
```

We've taken our original conditional and made it nested underneath a check for the host environment variable. If the host is set to 'localhost', then we check the browser type and execute things accordingly. If the host is set to 'saucelabs' then we configure the capabilities for Selenium Remote, passing in the requisite informaiton that we will need for our Sauce Labs session.

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

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

First, we are using some metaprogramming when we are instantiating

`Selenium::WebDriver::Remote::Capabilities`. We are using the `.send` method to pass in the environment variable. The value of which, in this case, is the same name as the method to configure Selenium Remote to use Internet Explorer. So, we are in effect, creating `Selenium::WebDriver::Remote::Capabilitites.internet_explorer`. And if we were to change `ENV['browser']` to 'chrome', then it would give us `Selenium::WebDriver::Remote::Capabilities.chrome`.

Second, for `caps[:name]` we are using an available hook in RSpec that gives us the name of each test as it is being run. This will make it so each individual job that executes in Sauce Labs will have the name of your test.

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

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

Test Status

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

And thanks to Sauce Labs' 'sauce_whisk' library, it's really simple to do.

After we install it we will need to require it somewhere, and our 'config_cloud.rb' file seems like a logical place. Also, we'll need to adjust our Sauce credential environment variables slightly since the library expects them to be capitalized.

```ruby
# filename: config_cloud.rb

require 'sauce_whisk'

ENV['base_url']          = 'http://the-internet.herokuapp.com'
ENV['host']              = 'saucelabs'
ENV['operating_system'] = 'Windows XP'
ENV['browser']           = 'internet_explorer'
ENV['browser_version']  = '8'
ENV['SAUCE_USERNAME']   = 'the-internet'
ENV['SAUCE_ACCESS_KEY'] = '26bd4eac-9ef2-4cf0-a6e0-3b7736bd5359'
```

All that's left is to add an action to our `after(:each)` block in our 'spec_helper' file. Before we issue `@driver.quit` we will want to grab the job ID from our `@driver` object and set the job status based on the test result. Also, we'll want to make sure that it only executes when running tests against Sauce Labs -- so we'll want to wrap it in a conditional check against the host environment variable.

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

RSpec.configure do |config|
  ...

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

    @driver.quit
  end

end
```

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

Accessing Secure Apps

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

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

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

Let's step through an example using the 'sauce-connect' library.

An Example

First we'll need to install the sauce-connect library. Once we have that we'll want to create a new config file for this since there will be some differences (e.g. baseurl, specifying a tunnel value, etc). So lets' create one called 'configcloud_tunnel.rb'.

```ruby
# filename: config_cloud_tunnel.rb

require 'sauce_whisk'
require 'sauce'

ENV['base_url']         = 'http://the-internet-local:4567'
ENV['host']             = 'saucelabs'
ENV['operating_system'] = 'Windows XP'
ENV['browser']          = 'internet_explorer'
ENV['browser_version']  = '8'
ENV['SAUCE_USERNAME']   = 'the-internet'
ENV['SAUCE_ACCESS_KEY'] = '26bd4eac-9ef2-4cf0-a6e0-3b7736bd5359'
ENV['tunnel']
```

Compared to our previous cloud configuration, this one is slightly different.

In addition to the 'sauce_whisk' library we are requiring 'sauce'. And we're doing this instead of requiring 'sauce-connect' because 'sauce-connect' is not built to be required. But since it is a subset of the 'sauce' library, we can include that instead and get access to the functionality that we will need.

At the bottom of the file we are including a tunnel environment variable. We only need this variable to exist, it doesn't have to contain a value. And we don't need it to exist in the other config files.

And notice that our base_url is different. It's set to `the-internet-local:4567`. This is because we're going to be run a copy of 'the-internet' locally. To do that we'll first need to download and upzip a copy of the master branch from its GitHub repo and place it in 'vendor/the-internet'.

In order to take advantage of this we need to update our local hosts file to map 127.0.0.1 to a DNS helper name ('the-internet-local' in this case). This is so our tests running Sauce Labs will be able to find our local server. If we just provide 'localhost' or 127.0.0.1, it won't work (as noted in this Sauce Support post. Updating the hosts file varies depending on your operating system. So if you're not sure how to do it, or what that means, then read this.

Now that we have that sorted, we can open a new terminal window, navigate into the 'vendor/the-internet' directory, and start the app (`ruby server.rb`) -- which will fire up on port 4567.

All that's left to do now is tell RSpec how to connect and disconnect from the tunnel -- and we can do that in our 'spec_helper' file.

```
# filename: spec/spec_helper.rb

require 'selenium-webdriver'

RSpec.configure do |config|

  config.before(:all) do
    Sauce::Connect.connect if ENV['tunnel']
  end

  ...

  config.after(:all) do
    Sauce::Connect.connect if ENV['tunnel']
  end

end
```

We're effectively telling RSpec to start the tunnel before all of our tests, and close it down after they are all done. And in order to make sure this only occurs when we intend to use Sauce Connect we add a conditional that keys off of the tunnel environment variable, which only exists in our 'configcloudtunnel.rb' file.

Now if we run our tests (`rspec -r ./config_cloud_tunnel.rb`) it will connect the tunnel and run the tests in Sauce which will then connect and exercise our local app.

A small bit of clean-up

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

```
# filename: .gitignore

*.log
```

This leaves our directory tree structure looking like this:

```
.
|-- config_chrome.rb
|-- config_cloud.rb
|-- config_cloud_tunnel.rb
|-- config_firefox.rb
|-- Gemfile
|-- pages
|   |-- base_page.rb
|   |-- dynamic_loading.rb
|   `-- login.rb
|-- spec
|   |-- dynamic_loading_spec.rb
|   |-- login_spec.rb
|   `-- spec_helper.rb
`-- vendor
    |-- chromedriver
    `-- the-internet/
```

Chapter 13
Speeding Up Your Test Runs

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

But with parallelization we can quickly remedy this. And in Ruby, there is a library that makes this easy to accomplish -- and that's 'parallel_tests'. After we install it all we need to do is change how we execute our test runs. And that's done through the use of the executable command that the library has installed for use with RSpec -- `parallel_rspec`.

`parallel_rspec` is effectively a wrapper around RSpec. It's responsible for breaking our spec files out into groups and launching each of them in separate system processes along with any arguments we pass in. So if we run `parallel_rspec --test-options '-r ./config_cloud.rb' spec` our tests will execute in Sauce Labs, except this time they will be running in parallel -- finishing much faster than previous runs.

Notice that we have to pass in `--test-options` and then our normal RSpec arguments (`-r ./config_cloud.rb`), along with where our specs live (`specs`). It can be a little verbose, but this is a necessary evil with this library.

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

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

Randomizing

A great way to make sure your tests are truly atomic and to potentially ferret out anomalies in your application under test is to run your tests in a random order. Within RSpec this is an easy thing to accomplish. It's just another command-line argument to pass in on test execution `--order random`.

Now our test launch sequence looks like this:

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

That's a lot to remember and type out every time we want to run our tests. Let's clean things up with some simple Rake tasks.

Rake Tasks

Rake is a simple library in Ruby that enables us to capture and run repetitive tasks.

To set it up, we just need to install the library, add a 'Rakefile', and fill it in with our tasks.

```
# filename: Rakefile

require 'rake'

def launch_in_parallel(config_file)
  system("parallel_rspec --test-options '--require #{Dir.pwd}/#{config_file} --order
random' spec")
end

desc 'Run tests locally in parallel'
task :local, :browser do |t, args|
  launch_in_parallel("config_#{args[:browser]}")
end

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

desc "Run tests in Sauce Connect in parallel"
task :cloud_tunnel do
  launch_in_parallel('config_cloud_tunnel.rb')
end
```

The first thing to note is that this looks very similar to a standard Ruby file. We are requiring a library ('rake') and declaring a method. The rest is what's a little bit different. Task declaration in Rake has its own Domain Specific Language which reads similar to Ruby code. For more of a primer on Rake, check out this tutorial.

In our `launch_in_parallel` method we pulled out the executor string, wrapped it in a `system` command to launch it, and made it configurable by adding an argument for the config file (injecting it into the string with interpolation). This keep our tasks clean by removing redundancy and makes them more flexible to change if we have additional arguments we want to start using with RSpec.

The only task that's different is `local`. In it we are taking a browser as an additional argument. This enables us to only have one task for local runs, switching between the local configs we have for each browser.

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

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

And to run one (e.g. `rake local['firefox']`) then it should run everything in parallel, in random order, and the tests should all pass.

The same pattern applied to our local runs (for browser selection) can easily be done for our cloud configuration as well. It's just a simple matter of updating our Rakefile cloud tasks, adding additional config files as needed.

Chapter 14
Automating Your Test Runs

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

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

A primer on Continuous Integration

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

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

There are numerous CI Servers available for use today, most notably -- Bamboo, Jenkins, and TravisCI.

Tagging

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

The workflow is pretty straightforward. The CI Server pulls the latest code, merges it, and runs unit tests. We can then have the CI Server execute another job to run a subset of our acceptance tests -- our critical ones (e.g. smoke or sanity tests). And assuming those pass, then we can have another job run the remaining tests -- our longer running tests. Adam Goucher refers to this strategy as a 'shallow' and 'deep' tagging model.

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

A Tagging Example

RSpec comes built in with tagging support. It's a simple matter of adding a key and value to denote what you want. And 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. So let's add some to our specs -- following Adam Goucher's shallow and deep approach.

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

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

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

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

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

If we wanted to apply them directly to a test, then we could have done the following instead:

```ruby
it 'successful', depth: 'shallow' do
```

Now to run our tests and take advantage of these tags we just need to pass along `--tag` and the key:value information when we run our tests (e.g. `--tag depth:shallow` or `--tag depth:deep`).

Let's update one of our 'Rakefile' tasks to demonstrate how to do this.

```
# filename: Rakefile

require 'rake'

def launch_in_parallel(config_file, tags)
  if tags
    system("parallel_rspec --test-options '--require #{config_file} --order random
--tag #{tags}' spec")
  else
    system("parallel_rspec --test-options '--require #{config_file} --order random'
spec")
  end
end

desc 'Run tests locally in parallel'
task :local, :browser, :tags do |t, args|
  launch_in_parallel("#{Dir.pwd}/config_#{args[:browser]}.rb", args[:tags])
end
```

By updating our `launch_in_parallel` method to take a second argument for tags, we are able to add a conditional to alter our execution string based on whether or not tags are passed in.

In our rake task, we add an additional argument to consume tags, passing it in when we call `launch_in_parallel`. And as a matter of cleanup, we've pulled the `Dir.pwd` incantation down to the rake task.

Now when we run `rake -T` we should see the following:

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

Now we can run our tests locally both with and without tags against the browsers we've configured -- e.g. `rake local['chrome','depth:shallow']` or `rake local['firefox','depth:deep']`.

The same pattern can be applied to our cloud tasks. So while we're in here, let's do that, and also make additional configs for them as well.

The first thing we should do is create a 'configs' directory and move our configs there. This way our root folder doesn't get cluttered and our approach becomes more manageable. Once we have that, we can create additional cloud configs for both cloud and cloud_tunnel.

And since we now have multiple Sauce Labs files, we should abstract out our access credentials into a separate config.

```
# filename: configs/config_sauce.rb

require 'sauce_whisk'

ENV['SAUCE_USERNAME']   = 'the-internet'
ENV['SAUCE_ACCESS_KEY'] = '26bd4eac-9ef2-4cf0-a6e0-3b7736bd5359'
```

We can safely move the require statement for our 'sauce_whisk' library here. But we'll want to keep the require for the 'sauce' library in the tunnel specific configs. Otherwise it will trigger the tunnel when we don't intend to.

Next we need to update all of our cloud and cloudtunnel configs to require our configsauce file and remove the environment variables for `SAUCE_USERNAME` and `SAUCE_ACCESS_KEY`.

Here's what a config_cloud file should look like:

```
# filename: configs/config_cloud_firefox.rb

require_relative 'config_sauce'

ENV['base_url']         = 'http://the-internet.herokuapp.com'
ENV['host']             = 'saucelabs'
ENV['operating_system'] = 'Windows XP'
ENV['browser']          = 'firefox'
ENV['browser_version']  = '23'
```

And here's what a configcloudtunnel file should look like:

```
require_relative 'config_sauce'
require 'sauce'

ENV['base_url']         = 'http://the-internet-local:4567'
ENV['host']             = 'saucelabs'
ENV['operating_system'] = 'Windows XP'
ENV['browser']          = 'firefox'
ENV['browser_version']  = '23'
ENV['tunnel']
```

And we can rename our local config files to make it obvious that they are meant for local runs.

When we're done, here is what our final directory structure should look like:

```
.
|-- configs
|   |-- config_cloud_firefox.rb
|   |-- config_cloud_internet_explorer.rb
|   |-- config_cloud_tunnel_firefox.rb
|   |-- config_cloud_tunnel_internet_explorer.rb
|   |-- config_local_chrome.rb
|   |-- config_local_firefox.rb
|   `-- config_sauce.rb
|-- Gemfile
|-- pages
|   |-- base_page.rb
|   |-- dynamic_loading.rb
|   `-- login.rb
|-- Rakefile
|-- spec
|   |-- dynamic_loading_spec.rb
|   |-- login_spec.rb
|   `-- spec_helper.rb
`-- vendor
    |-- chromedriver
    `-- the-internet/
```

And now we just need to update the tasks in our 'Rakefile'.

```
# filename: Rakefile
...

desc 'Run tests locally in parallel'
task :local, :browser, :tags do |t, args|
  launch_in_parallel("#{Dir.pwd}/configs/config_local_#{args[:browser]}.rb", args[:tags
])
end

desc "Run tests in Sauce in parallel"
task :cloud, :browser, :tags do |t, args|
  launch_in_parallel("#{Dir.pwd}/configs/config_cloud_#{args[:browser]}.rb", args[:tags
])
end

desc "Run tests in Sauce Connect in parallel"
task :cloud_tunnel, :browser, :tags do |t, args|
  launch_in_parallel("#{Dir.pwd}/configs/config_cloud_tunnel_#{args[:browser]}.rb",
args[:tags])
end
```

By having a consistent naming convention across our config files we can easily concatenate the config file name by using the browser provided as an argument in the rake task and apply this pattern across all of our rake tasks.

So now when we run `rake -T` we get the following:

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

Now we're able to easily run our tests against the browsers we care about locally, in the cloud, or in the cloud across a secure tunnel with the use of tags.

A More Configurable Base URL

But there's still one thing we're missing -- the ability to easily specify a base URL without having to update our configs. You will likely have multiple base URLs to test against, so having multiple config permutations or updating files all the time feels a bit heavy. And it's a simple fix, so let's add it.

We just need to update the base_url environment variable instantiation in our config files so they are open to receiving alternative input. Here's what it looks like:

```
# filename: configs/config_cloud_internet_explorer.rb

require_relative 'config_sauce'

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

By adding a `||` in front of the `=` we are effectively saying "if the baseurl environment variable already exists then use it, otherwise, let's set it to this instead". So now when we run our tests we have the option of overriding our baseurl without having to update our configs -- `rake cloud['internet_explorer'] base_url='http://the-internet-staging.heroku-app.com'`

While there are more sophisticated approaches we can take to handle our configs, this one is simple and has everything nailed down and obvious for others to see and use.

Reporting

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

Let's implement it in our test harness.

A JUnit XML Example

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

After we have it installed we need to specify the formatter type and the output file we want. This is done through the use of two new arguments during test execution -- `--format RspecJunitFormatter` and `--out results.xml`.

Let's update our Rakefile to use these.

```
# filename: Rakefile

require 'rake'

def launch_in_parallel(config_file, tags)
  if tags
    system("parallel_rspec --test-options '--require #{config_file} --order random
--tag #{tags} --format RspecJunitFormatter --out results.xml' spec")
  else
    system("parallel_rspec --test-options '--require #{config_file} --order random
--format RspecJunitFormatter --out results.xml' spec")
  end
end
...
```

Now when we run our tests, we should see a results.xml file in our root folder. It should contain a bunch of info the tests ran. This is what our CI server will use to track test passes, failures, time to complete, and trend it over time.

But we don't want to commit this file to our repository, so let's add it to our ignore file.

```
# filename: .gitignore

*.log
*.xml
```

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

A Primer On CI Server Job Configuration

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

If your team already uses Jenkins and has a job that runs unit or integration tests, you can configure your job to key off of that one -- e.g. if the unit tests pass, then run your shallow tests. This is ideal since their job is already tightly coupled to their workflow (e.g. triggering every time code is pushed to the mainline), or some kind of time delayed polling (e.g. see if new code was pushed every 5 minutes). This means code is compiled and tests are run often. So by keying your job off of theirs, you are plugging your test suite directly into the development workflow. This is why running shallow tests to start is important -- we want only fast and critical tests to run.

If this is not how things are set up, then look to set up your own polling mechanism for the codebase so that your tests can be coupled to their workflow.

You can also setup scheduled runs for different times of the day or week. But these are more ideal for longer running tests to run at off hours.

Let's take a look at some examples.

A Jenkins Example

Coming soon...

A TravisCI Example

Coming soon...

Chapter 15
Things You will See And How To Work With Them

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

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

Enjoy!

Chapter 15.1
A/B Testing

Sometimes when running tests you may hit unexpected behavior due to A/B testing being done in the application you're working with.

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

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

An Example

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

And on the page we are testing, the heading text will change based on which test group we are in.

You can see the example application we are testing here.

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

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

def teardown
  @driver.quit
end

def run
  setup
  yield
  teardown
end
```

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

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

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

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

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

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

Expected Behavior

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

Chapter 15.2
Basic Auth

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

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

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

An Example

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

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

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

def teardown
  @driver.quit
end

def run
  setup
  yield
  teardown
end
```

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

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

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

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

Here's what that would look like.

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

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

def teardown
  @driver.quit
end

def run
  setup
  yield
  teardown
end

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

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

Expected Behavior

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

Chapter 15.3
Dropdown Lists

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

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

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

An Example

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

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

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

def teardown
  @driver.quit
end

def run
  setup
  yield
  teardown
end
```

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

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

  options = drop_down_list.find_elements(tag_name: 'option')
  options.each { |option| option.click if option.text == 'Option 1' }

  selected_option = options.map { |option| option.text if option.selected? }.join
  selected_option.include?('Option 1').should == true
}
```

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

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

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

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

Another Example

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

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

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

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

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

Expected Behavior

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

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

Chapter 15.4
File Download (browser specific)

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

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

A Solution

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

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

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

An Example

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

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

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

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

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

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

def run
  setup
  yield
  teardown
end
```

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

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

After setting these bits we instantiate Selenium WebDriver and pass in the now configured profile object.

Our teardown and run methods are pretty straightforward. Teardown quits the Selenium object when done and cleans up the temp directory with a system call. And we wrap everything up neatly into a single run method that we can pass a block of commands to.

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

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

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

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

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

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

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

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

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

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

Expected Behavior

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

Outro

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

Chapter 15.5
File Download (browser agnostic)

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

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

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

Let's dig with an example.

An Example

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

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

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

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

def teardown
  @driver.quit
end

def run
  setup
  yield
  teardown
end
```

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

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

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

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

Expected Behavior

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

Chapter 15.6
File Upload

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

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

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

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

An Example

You can see the application we are running against here, or view the code for it here.

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

```
require 'selenium-connect'
require 'rspec-expectations'

def setup
  SeleniumConnect.configure do |c|
    c.host    = 'localhost'
    c.browser = 'chrome'
  end

  @driver = SeleniumConnect.start
end

def teardown
  SeleniumConnect.finish
end
```

We are using the selenium-connect gem to make configuring and running a local instance of Selenium Remote a snap. And we are using rspec-expectations for our assertion action.

```
def run
  setup
  yield
  teardown
end

run {
  @driver.get 'http://localhost:4567/upload'
  uploader = @driver.find_element(id: 'file-upload')
  uploader.send_keys '/Users/more/Desktop/daveHaeffner.jpg'
  uploader.submit

  uploaded_image = @driver.find_element(css: 'img').attribute('src')
  uploaded_image.should =~ /daveHaeffner.jpg/
}
```

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

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

Expected Behavior

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

Chapter 15.7
Frames

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

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

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

An Example

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

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

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

def teardown
  @driver.quit
end

def run
  setup
  yield
  teardown
end
```

Our first example demonstrates how to deal with nested frames. You can see the page we are testing against here.

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

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

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

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

Another Example

You can find the page we are testing against here.

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

  after_text.should_not == before_text
}
```

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

Now that Selenium is properly scoped we are able to use the top-level id of the TinyMCE widget to: + grab its text + clear its text + input new text + grab the altered text + assert that the before and after texts are not the same

Keep in mind that if we need to access a part of the page outside of the frame we are currently in then we will need to switch to it. For instance...

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

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

And that's pretty much all there is to it.

Expected Behavior

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

Chapter 15.8
HTTP Status Codes

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

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

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

Let's step through an example using BrowserMob Proxy.

An Example

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

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

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

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

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

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

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

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

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

def run
  setup
  yield
  teardown
end
```

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

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

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

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

Expected Behavior

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

Outro

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

Chapter 15.9
Mobile

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

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

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

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

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

A brief primer on Native App Testing In SauceLabs

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

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

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

An iPhone Example

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

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

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

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

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

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

def teardown
  @driver.quit
end

def run
  setup
  yield
  teardown
end
```

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

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

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

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

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

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

Expected Behavior

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

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

An Android Example

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

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

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

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

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

def teardown
  @driver.quit
end

def run
  setup
  yield
  teardown
end
```

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

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

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

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

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

Expected Behavior

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

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

Other Helpful Tidbits

Inspecting Elements

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

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

Mobile Web Apps

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

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

But it's worth noting that Sauce Labs does not currently support the ability launch a mobile browser on Android (e.g. `chrome`). This functionality is, however, available when running Appium locally (as documented here).

Hybrid Apps

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

Location Services

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

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

And in Sauce Labs, Location Services is disabled by default on iOS 6.1, but enabled by default on iOS 7.1. This is not configurable, but may be in the future through desired capabilities (there's an active thread about this on Sauce's support site, feel free to chime in!).

Outro

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

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

Chapter 15.10
Multiple Windows

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

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

Let's step through it.

An Example

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

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

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

def teardown
  @driver.quit
end

def run
  setup
  yield
  teardown
end
```

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

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

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

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

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

Here's a more effective approach.

A Better Example

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

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

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

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

Expected Behavior

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

Chapter 15.11
Secure File Downloads

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

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

Let's dig in with an example.

An Example

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

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

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

def teardown
  @driver.quit
end

def run
  setup
  yield
  teardown
end
```

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

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

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

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

Expected Behavior

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

Outro

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

Chapter 15.12
Tables

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

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

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

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

A quick primer on Tables and CSS Pseudo-classes

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

A table has...

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

Columns are made up of cells which are represented as...

- `<th>` in a header
- `<td>` in a body (also known as table data)

CSS Pseudo-classes work by walking through the structure of an object and targeting a specific part of it based on a relative number (e.g. the third `<td>` cell from a row in the table body). This works well with tables since we can couple it with Selenium to grab all instances of a target (e.g. the third `<td>` cell from all rows in the table body) -- which gives us all of the data for a specific column.

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

An Example

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

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

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

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

def teardown
  @driver.quit
end

def run
  setup
  yield
  teardown
end
```

Here is a snippet from the first table we are working with. Note that it does not have any `id` or `class` attributes.

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

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

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

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

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

  dues = @driver.find_elements(css: '#table1 tbody tr td:nth-of-type(4)')
  due_values = []
  dues.each { |due| due_values << due.text.gsub(/\$/,'').to_f }

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

After navigating to the page we find and click the column heading that we want with a CSS Pseudo-class (e.g. `#table1 thead tr th:nth-of-type(4)`) which is effectively saying 'give me the 4th `th` element in the table heading' which is the placement of `Due` .

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

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

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

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

  dues = @driver.find_elements(css: '#table1 tbody tr td:nth-of-type(4)')
  due_values = []
  dues.each { |due| due_values << due.text.gsub(/\$/,'').to_i }

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

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

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

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

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

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

But What About Older Browsers?

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

You've already come a long way, so it's best to get value out of what you've just written. You can run these tests on current browsers, file a fix with your front-end developers to improve the table design with some `class` attributes, and then update these tests once they've made the necessary changes.

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

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

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

```
run do
  @driver.get 'http://the-internet.herokuapp.com/tables'
  @driver.find_element(css: '#table2 thead .dues').click
  dues = @driver.find_elements(css: '#table2 tbody .dues')
  due_values = []
  dues.each { |due| due_values << due.text.gsub(/\$/,'').to_f }
  (due_values == due_values.sort).should == true
end
```

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

Expected Behavior

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

Outro

CSS Pseudo-classes are a great resource and unlock a lot of potential for your tests -- enabling a bit of CSS gymnastics (assuming you've come up with an automation strategy that rules out older browsers).

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

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

Chapter 15.13

Upgrading From Selenium RC

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

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

A Solution

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

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

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

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

4 Simple Steps

1. Clean Up Your Tests

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

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

2. Swap in WebDriver-backed Selenium

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

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

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

3. & 4. Improve Things Incrementally

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

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

Common Pitfalls

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

Alert Handling

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

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

Here is an example of the difference:

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

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

Waiting For The Page To Load

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

Here's an example:

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

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

Getting Text From The Page

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

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

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

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

Be careful with JavaScript

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

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

Inertia

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

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

Chapter 16

Finding Information On Your Own

Available Resources

Coming soon...

Chatting With the Community

You can go to the heart of the Selenium Community and talk directly with the Committers of the Selenium project and various practioners without ever leaving your house. You can do this by hopping on to the Selenium IRC Chat Channel! It is a great place to ask questions, find answers, and get pointed in the right direction when you're stuck.

Brief Intro to IRC

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

Within the realm of IRC there are numerous servers you can connect to. Each one containing people and bots logged in and joined to 1 or more chat rooms talking, sharing files, etc.

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

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

How To Get Connected

Step 1 -- Get an IRC Chat Client

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

You may already have one and not even know. For example, Adium (for OSX) supports a staggering number of chat protocols. If you already have it (or something like it) then use that to connect. If you don't, then you'll need to grab one (or one that is built specifically for IRC).

Here are some worthwhile IRC chat clients (broken out by platform -- and are free unless otherwise noted).

OSX

- LimeChat
- Textual ($4.99 to buy)

Windows

- mIRC (free, but asks for donations)

Linux

- Irssi

Web-only

- Webchat

Step 2 -- Connect to the proper server

The Selenium chat channel lives on the Freenode server. To connect to it directly you would use `irc.freenode.net`.

Before connecting you should be able to set a nickname and perhaps even specify which channel you would like to connect to after connecting (e.g. `#selenium`). If you don't see these bits, don't sweat it. Connect and proceed to the next step.

Note: If you're using Webchat, it will automatically connect you to Freenode.

Step 3 -- Join the chat channel

In IRC parlance channels are prepended with a `#` and are lower-case. So the Selenium channel is `#selenium`.

If you were able to configure your chat client to join the channel for you, then proceed to the next step. Otherwise, you'll need to issue a command in the status window. There are a series of commands you can issue in IRC. They are all prepended with a `/`. To join a chat channel type `/join #selenium` and press `Enter`.

This should open a new window in the app and in it should be the Selenium chat channel. Woohoo!

Step 4 -- Talk and hangout!

Feel free to say hello and introduce yourself. But more importantly, ask your question. And if it looks like no one is chatting, ask it anyway. Someone will see it and eventually respond. They always do.

In order to get your answer, you'll probably want to hang around for a bit. But the benefit of being a fly on the wall is that you quickly gain insight into other problems people face, possible solutions, and the current status of the Selenium project and its various pieces.

Getting Involved

Coming soon...