Selenium Bootcamp

by Dave Haeffner

Table of Contents

1. The First Things You Need To Know
2. Writing Your First Selenium Test
3. How To Write Maintainable Tests
4. Writing Resilient Test Code
5. Packaging For Use

The First Things You Need To Know

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.

But before we dig in, there are a few things you'll want to know before you write your first test.

Define a Test Strategy

A great way to increase your chances of automated web testing success is to focus your efforts by mapping out a testing strategy. The best way to do that is to answer four questions:

1. How does your business make money (or generate value for the end-user)?
2. How do your users use your application?
3. What browsers are your users using?
4. What things have broken in the application before?

After answering these, you will have a good understanding of the functionality and browsers that matter most for the application you are testing. This will help you narrow down your initial efforts to the things that matter most.

From the answers you should be able to build a prioritized list (or backlog) of critical business functionality, a short list of the browsers to focus on, and include the risky parts of your application to watch out for. This prioritized list will help you make sure you're on the right track (e.g., focusing on things that matter for the business and its users).

Pick a Programming Language

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

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

As you are considering which language to go with, consider what open source frameworks already exist for the languages you're 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.

You can see a list of available open source Selenium frameworks here.

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 long term success.

For this course we'll be using the Ruby programming language. If you need help installing Ruby, then check out one of the following links:

- Linux
- Mac
- Windows

Choose an Editor

In order to write code, you will need to use some kind of an editor (e.g., text editor, or integrated development environment). There are plenty to choose from. Here are some of the more popular ones I've run into:

- Emacs
- IntelliJ
- Sublime Text
- Vim

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

Coming up next, I'll review how to write an effective acceptance test with Selenium.

Writing Your First Selenium Test

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 end result is what you intended.

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

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

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

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

How To Find Locators

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

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

How To Find Quality Elements

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

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

And if you can't find any unique elements, have a conversation with your development team

letting them know what you are trying to accomplish. It's not hard for them to add helpful, semantic markup to make test automation easier, especially when they know the use case you are trying to automate. The alternative can be a lengthy, painful process which will probably yield working test code -- but it will be brittle and hard to maintain.

Steps To Writing a Selenium Test

There are five parts to writing a Selenium test:

1. Find the elements you want to use
2. Write a test with Selenium actions that use these elements
3. Figure out what to assert
4. Write the assertion and verify it
5. Double-check the assertion by forcing it to fail

As part of writing your Selenium test, you will also need to create and destroy a browser instance. This is something that we will pull out of our tests in a future lesson, but it's worth knowing about up front.

Let's take our login example from above and step through the test writing process.

An Example

The best way to find the Selenium actions for your specific language is to look at the available language binding on the Selenium HQ wiki pages or look through the Selenium HQ getting started documentation for WebDriver.

For this example, we'll be using the Ruby programming language, RSpec version 2.14 (an open-source Ruby testing framework), and the-internet (an open-source web application).

Step 1: Find The Elements You Want To Use

Let's use the login example on the-internet). Here's the markup from the page.

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

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

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

2. Write A Test With Selenium Actions That Use These Elements

```ruby
# filename: login_spec.rb

require 'selenium-webdriver'

describe 'Login' do

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

  after(:each) do
    @driver.quit
  end

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

end
```

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

3. Figure Out What To Assert

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

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

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

After logging in, there looks to be a couple of things we can use for our assertion. There's the flash message class (most appealing), the logout button (appealing), or the copy from the `h2` or the flash message (least appealing).

Since the flash message class name is descriptive, denotes success, and is less likely to change than the copy, we'll go with that.

4. Write The Assertion And Verify It

```ruby
# filename: login_spec.rb

require 'selenium-webdriver'

describe 'Login' do

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

  after(:each) do
    @driver.quit
  end

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

end
```

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

5. Double-check The Assertion By Forcing It To Fail

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

```ruby
@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. Then we can change the assertion back to normal before committing our code. This trick will save you more trouble that you know. Practice it often.

How To Write Maintainable Tests

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 will change and break your tests.

But the reality of a software project is that change is a constant. So we need to account for this 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.

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

Step 1: Create A Page Object

```ruby
# filename: login.rb

class Login

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

  def initialize(driver)
    @driver = driver
    @driver.get 'http://the-internet.herokuapp.com/login'
  end

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

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

end
```

We start by creating our own class, naming it `Login`, and storing our locators at the top (in helpfully named constants). We then use an initializer to receive the Selenium driver object and visit the login page.

In our `with` method we are capturing the core behavior of the login page by accepting the username and password as arguments and housing the Selenium actions for inputting text and submitting the login form.

Since our locators and behavior now live in a page object, we want a clean way to make an assertion in our test. This is where our `success_message_present?` method comes in. Notice that it ends with a question mark. In Ruby, when methods end with a question mark, they imply that they will return a boolean value (e.g., `true` or `false`).

This enables us to ask a question of the page, receive a boolean response, and make an assertion against that response.

Step 2: Update The Login Test

```ruby
# filename: login_spec.rb

require 'selenium-webdriver'
require_relative 'login'

describe 'Login' do

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

  after(:each) do
    @driver.quit
  end

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

end
```

At the top of the file we include the page object with `require_relative` (this enables us to reference another file based on the current file's path).

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

Step 3: Write Another Test

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

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

```html
<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 the successful flash message, so let's mimic the behavior we used in our page object to create another method to help in our assertion.

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

```
# filename: login.rb

class Login

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

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

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

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

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

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

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

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

Step 4: Confirm We're In The Right Place

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

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

```
class Login

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

  def initialize(driver)
    @driver = driver
    @driver.get 'http://the-internet.herokuapp.com/login'
    @driver.find_element(LOGIN_FORM).displayed?.should == true
  end
  ...
```

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

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

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

Writing Resilient Test Code

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

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

But there is a simple approach that makes up the bedrock of reliable and resilient Selenium tests -- and that's how you wait and interact with elements. The best way to accomplish this is through the use of explicit waits.

An Explicit Waits Primer

Explicit waits are applied to individual test actions. Each time you want to use one you specify an amount of time (in seconds) and the Selenium action action you want to accomplish.

Selenium will repeatedly try this action until either it can be accomplished, or the amount of time specified has been reached. If the latter occurs, a timeout exception will be thrown.

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.

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

  <br>

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

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

</div>
```

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

Let's add a page object for Dynamic Loading.

```ruby
# filename: dynamic_loading.rb

class DynamicLoading

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

  def initialize(driver)
    @driver = driver
    @driver.get "http://the-internet.herokuapp.com/dynamic_loading/1"
  end

  def start
    @driver.find_element(START_BUTTON).click
  end

  def finish_text_present?
    wait_for { is_displayed? FINISH_TEXT }
  end

  def is_displayed?(locator)
    @driver.find_element(locator).displayed?
  end

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

end
```

This approach should look familiar from the last lesson. The thing which is new is the `wait_for` method. In it we are using the built-in mechanism Selenium has for explicit waits.

More On Explicit Waits

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

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

```ruby
# filename: dynamic_loading_spec.rb

require 'selenium-webdriver'
require_relative 'dynamic_loading'

describe 'Dynamic Loading' do

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

  after(:each) do
    @driver.quit
  end

  it 'Waited for Hidden Element' do
    @dynamic_loading.start
    @dynamic_loading.finish_text_present?.should be_true
  end

end
```

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

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

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

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

  <br>

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

  <br>
</div>
```

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

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

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

```
# filename: dynamic_loading.rb

class DynamicLoading

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

  def initialize(driver)
    @driver = driver
  end

  def visit_example(example_number)
    @driver.get "http://the-internet.herokuapp.com/dynamic_loading/#{example_number}"
  end

  ...
```

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

```ruby
# filename: dynamic_loading_spec.rb

require_relative 'dynamic_loading'

describe 'Dynamic Loading' do

  ...

  it 'Waited for Hidden Element' do
    @dynamic_loading.visit_example 1
    @dynamic_loading.start
    @dynamic_loading.finish_text_present?.should be_true
  end

  it 'Waited for Element To Render' do
    @dynamic_loading.visit_example 2
    @dynamic_loading.start
    @dynamic_loading.finish_text_present?.should be_true
  end

end
```

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

Explicit waits are one of the most important concepts in testing with Selenium. Use them often.

Packaging For Use

In order to get the most out of our tests and page objects, we'll need to package them into a more useful structure. Once that's done, we'll be able to add in the ability to run our tests against different browser and operating system combinations.

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

```
# filename: spec_helper.rb

require 'selenium-webdriver'

RSpec.configure do |config|

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

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

end
```

We need to include the Selenium library here, and by doing so, can remove it from our tests. And by having our test configuration here, we can also clean up the `before(:each)` and `after(:each)` in our tests by removing the `after(:each)` completely, but keeping the `before(:each)` around for setting up our page objects.

To use the `spec_helper` we'll need to require it in our tests. Here's an example of the login test after these changes have been made.

```
require_relative 'spec_helper'
require_relative 'login'

describe 'Login' do

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

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

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

end
```

Folder Organization

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

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

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

Updating Require Statements

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

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

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

describe 'Login' do
...
```

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

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

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

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

Now that things are cleaned up, we can run everything with the `rspec` command. Give it a shot. All of the tests should run and pass just like before.

Now we're ready to run our tests against different browser and operating system combinations.

Running Tests On Any Browser

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

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

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

An Example

Step 1: Initial Setup

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

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

```
# filename: config_cloud.rb

ENV['host']             = 'saucelabs'
ENV['operating_system'] ||= 'Windows XP'
ENV['browser']          ||= 'internet_explorer'
ENV['browser_version']  ||= '8'
ENV['SAUCE_USERNAME']   ||= 'your-sauce-username'
ENV['SAUCE_ACCESS_KEY'] ||= 'your-sauce-access-key'
```

Notice the use of environment variables (most notably the host environment variable). This is what we'll use in our `spec_helper` file to determine whether to run things locally or in the cloud -- and we'll use the other environment variables to our Sauce Labs session.

For a full list of available browser and operating system combinations supported by Sauce Labs, go here.

NOTE: Be sure to update this file with you Sauce Username and Sauce Access Key, or, specify them externally (e.g., at run time, on the command-line, or in your bash profile). To get your Sauce Access Key, go to the bottom-left corner of your Sauce Account page.

Now we'll need to update our `spec_helper` to connect to Sauce Labs and use these variables.

```
# filename: spec/spec_helper.rb

require 'selenium-webdriver'

RSpec.configure do |config|

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

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

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

end
```

Notice that we've added a conditional to check on the host environment variable. If the host is set to 'saucelabs', then we configure our tests to point at Sauce Labs (passing in the requisite information that we will need for the session). Otherwise, it will run our tests locally using Firefox.

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

And to run the tests on an alternate browser (or operating system), you can either update the values in the config_cloud file, or, you can specify them at runtime like so:

```
browser='chrome' browser_version='31' rspec -r ./config_cloud.rb
```