How to use Selenium, successfully

The Selenium Guidebook
JavaScript Edition

by Dave Haeffner

Version 1.0.0

Preface

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

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

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

What This Book Will Cover

This book focuses on the latest stable version of Selenium 2 (a.k.a. Selenium WebDriver) and its use to test desktop browsers. Its predecessor, Selenium RC, will not be covered. Mobile testing with Appium will also not be covered. But you can see a series of getting started with Appium posts I've written here.

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

Who This Book Is For

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

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

About The Examples In This Book

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

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

The test examples are written with Node.js and the officially supported Selenium JavaScript bindings to run on Mocha with npm managing the third-party dependencies.

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

How To Read This Book

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

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

Feedback

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

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

Acknowledgements

I intentionally saved JavaScript for last in my series of books because it is the most different of the languages to tackle given that it is fundamentally asynchronous (whereas all other officially supported programming languages for Selenium are synchronous). And with it's explosive popularity in Selenium over the last couple of years there are far more options to choose from with regards to available frameworks and language bindings. Frankly, the thought of getting started was intimidating.

Thanks to the work of Mek Stittri and Alister Scott, getting started was infinitely easier.

Mek Stittri spent a great deal of time looking through all of the prominent options in the JavaScript space for Selenium late last year. He provided a thorough overview in a meetup talk he gave at the San Francisco Selenium Meetup last November which confirmed a lot of my initial findings and suspicions about the tooling in the space. He also open-sourced the JavaScript code that they are using for testing with Selenium where he works (at Airware).

Alister Scott came to a similar conclusion as Mek with regards to his tool selection and published a multi-part series on getting started with the official Selenium JavaScript bindings and Mocha. You can see them here, here, here, and here. He works at Automattic (the company behind WordPress) and they open-sourced their Selenium test code as well earlier this year.

Thanks you two. You served as a great inspiration and helped make my entry into JavaScript & Selenium far less turbulent than I had anticipated.

Table of Contents

Chapter 1
Selenium In A Nutshell

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

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

What Selenium Is and Is Not Good At

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

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

Selenium Highlights

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

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

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

Chapter 2
Defining A Test Strategy

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

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

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

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

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

What To Do With The Answers

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

Question 1: Money/Value

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

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

Question 2: Usage Data

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

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

Question 3: Browsers

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

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

Question 4: Risky Bits

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

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

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

Now You Are Ready

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

Chapter 3
Picking A Language

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

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

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

What To Do With The Answer

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

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

Some Additional Food For Thought

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

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

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

Outro

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

With all that being said, the rest of this book will show you how to build your own test harness (in addition to writing well factored tests) in JavaScript with Node.js, Mocha, and the officially supported Selenium JavaScript bindings.

Chapter 4
A Programming Primer

This section will prime you with just enough programming concepts (and how they pertain to Selenium) so you have some working knowledge and a vocabulary. This will help you more effectively comprehend what you will see throughout this book (and in your work afterwards).

Don't get too hung up on the details though. 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

Installing Node.js is pretty straight-forward. There are installer packages available for Mac and Windows operating systems on the Node.js download page. And there are binary distributions for various Linux distributions as well (link).

If you're running on a Mac and you want to use Homebrew, then be sure to check out this write-up from Treehouse. And if you end up using WebStorm, check out the documentation on using it to work with Mocha (link).

Installing Third-Party Libraries

There are over 250,000 third-party libraries (a.k.a. "packages") available for Node.js through `npm`. `npm` is the Node Package Manager program that comes bundled with Node.

You can search for packages from `[npmjs.com](https://www.npmjs.com/)`. You don't need an account. Simply type into the search field at the top of the page and press Enter.

To install packages with it you type `npm install package-name` from the command-line. You can install a package globally using the `-g` flag. And you can auto-save the package to a local manifest file (e.g., `package.json`) which explicitly states the package name and version you are using with the `--save` flag.

Here is a list of the libraries we will be working with in this book:

- selenium-webdriver
- mocha
- grunt
- grunt-parallel
- grunt-shell

Interactive Prompt

Node.js comes with an interactive prompt (a.k.a. a REPL (record-eval-print loop)).

Just type `node` from the command-line. It will load a simple prompt that looks like this:

```
>
```

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

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

Choosing A Text Editor

In order to write Node.js code, you will need to use a text editor. Some popular ones are Vim, Emacs, and Sublime Text.

There's also the option of going for an IDE (Integrated Development Environment) like WebStorm. It's not free, but there is a free 30-day trial.

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 WebStorm.

Programming Concepts In A Nutshell

Programming can be a deep and intimidating rabbit hole if you're new to it. But don't worry. When it comes to automated browser testing there is only a small subset of programming that we really need to know in order to get started. Granted, the more you know, the better off you'll be. But you don't need to know a whole lot in order to be effective right now.

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

- Object Structures (Variables, Methods, and Classes)
- Scope
- Types of Objects (Strings, Integers, Data Structures, Booleans, etc.)
- Actions (Assertions and Conditionals)
- Inheritance
- Promises

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

Object Structures

Variables

Variables are places to store and retrieve values of various types (e.g., Strings, Integers, etc. -- more on these later). Variables are created and then referenced by their name.

A variable name:

- is prepended with the word `var`
- can be one or more words in length
- starts with a letter
- is not case sensitive
- should not be a keyword or reserved word in JavaScript

Since variable names are not case sensitive there are various ways you can write them (e.g., `camelCase`, `PascalCase`, `snake_case`). The general guidance across various style guides is to use `camelCase`.

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

```
> var exampleVariable = "42";
> Object.prototype.toString.call(exampleVariable);
// outputs: '[object String]'

> var exampleVariable = 42;
> Object.prototype.toString.call(exampleVariable);
// outputs: '[object Number]'
```

NOTE: In the above example **`Object.prototype.toString.call(exampleVariable)`** is used to find the object's type since there is no straight-forward way to get the object's type directly from the variable.

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

```
var pageTitle = driver.getTitle();
```

NOTE: **`driver`** is the variable we will use to interact with Selenium throughout the book. More on that later.

Methods

Throughout our tests we'll want to group common actions together for easy reuse. We do this by placing them into methods.

Method names follow the same rules as variables. The biggest differences between method and variable names are that method names tend to be a verb (since they denote some kind of an action to be performed), we use a function call when declaring them, and the contents of the method are wrapped in opening and closing brackets (e.g., `{}`).

```
var example_method = function() {
  // your code
  // goes here
};

example_method();
```

Additionally, we can specify arguments we want to pass into the method when calling it (a.k.a. specifying a parameter).

```
> var say = function(message) {
... console.log(message);
... };
> say('Hello World!');
// outputs: Hello World!
```

We can also specify a default value to use if no argument is provided.

```
> var say = function(message = 'Hello World!') {
... console.log(message);
... };
> say();
// outputs: Hello World!
> say('something else');
// outputs: something else
```

We'll see something like this used when we tell Selenium how to wait for things on the page to load (more on that in Chapter 10).

Classes

Classes are a useful way to represent concepts that will be reused numerous times in multiple places. They can contain variables and methods. To declare a class you specify the keyword `function` followed by the name you want.

NOTE: In JavaScript there is no such thing as a class. But there are functions (which are objects that can contain behavior and state) that you can use to create something very similar to a class. The word "class" will be used throughout the book as a function which has been adapted to behave like a class.

Class names:

- start with a capital letter
- should be PascalCase for multiple words (e.g., `ExampleClass`)
- should be descriptive (e.g., a noun, whereas methods should be a verb)

You first have to define the class. Then you can specify methods and variables for it. Method declarations in classes are done with `ClassName.prototype.methodName`. After that you can create an instance of the class (a.k.a. instantiation) to use it. Once you have an instance of it you can access the methods within it to trigger the behavior stored in them.

NOTE: The function block used to declare a class is a method in it's own right. It is considered a "constructor method". This is a method that gets automatically executed when a new instance of the class is created.

```
> function Messenger() {
... };
> Messenger.prototype.say = function(message) {
... console.log(message);
... };
> var messenger = new Messenger();
> messenger.say('This is an instance of a class');
// outputs: This is an instance of a class
```

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

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

function LoginPage(driver) {
}

LoginPage.prototype.with = function(username, password) {
// ...
```

The variables that are fully capitalized and separated by underscores (e.g., `_`) are called constants. They are variables that are unlikely to change. And the values in curly brackets (`{}`) are called object literals. They are simply a key/value pair.

Scope

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

Local Variables

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

Class Variables

Variables declared in a module (a.k.a. a class) will enable you to store and retrieve values more broadly (e.g., both inside and outside of methods). Essentially any variable that is declared outside of a function will be available throughout the entire class.

A common example you will see throughout the book is the usage of locators in page objects. These variables represent pieces of a web page we want to interact with. By storing them as broadly scoped variables we will be able to use them throughout an entire page object.

Global Variables

There are a few built-in global functions in Node.js. One of them is a `global` variable. Things stored in this variable are accessible across modules (e.g., throughout all of our test code).

We will use this sparingly to clean up our test code.

For more information on Global Variables, you can check out the Node.js documentation for them here.

Environment Variables

Environment variables are a way to pass information into our program from outside of it. They are also another way to make a value globally accessible (e.g., across an entire program, or set of programs). They can be set and retrieved from within your code by:

- using the `process.env` lookup function
- specifying the environment variable name with it

Environment variables are often used to retrieve configuration values that could change. A great example of this is the base URL and browser name we'll use in our tests.

```
module.exports = {
  baseUrl: process.env.BASE_URL || 'http://the-internet.herokuapp.com',
  browser: process.env.BROWSER || 'firefox'
};
```

Types of Objects

Strings

Strings are alpha-numeric characters packed together (e.g., letters, numbers, and most special characters) surrounded by either single (`'`) or double (`"`) quotes. Typically single-quotes.

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

Numbers

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

```
Number("42")
// outputs: 42
```

Collections

Collections enable you to gather a set of data for later use. In JavaScript there are two common collection types -- arrays and object literals. The one we'll want to pay attention to is object literals.

Object literals are an unordered set of data stored in key/value pairs. The keys are unique and are used to look up the data in the object.

```
> var example = {this: 'that', the: 'other'}
> example.this
// outputs: 'that'
> example.the
// outputs: 'other'
```

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

```
var LOGIN_FORM = {id: 'login'};
var USERNAME_INPUT = {id: 'username'};
var PASSWORD_INPUT = {id: 'password'};
var SUBMIT_BUTTON = {css: 'button'};
```

Booleans

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

There are numerous ways to ask questions. Some involve various comparison operators (e.g., `==`, `===`, `!=`, `<`, `>`). The response is either `true` or `false`.

```
> 2+2 === 4
// outputs: true
```

Selenium also has commands that return a boolean result when we ask questions of the page we're testing.

```
element.isDisplayed();
// returns true if the element is on the page and visible
```

Actions

A benefit of booleans is that we can use them to perform an assertion.

Assertions

Assertions are made against booleans and result in either a passing or failing test. In order to leverage assertions we will need to use an assertion library (e.g., the one built into Node.js or any of the assertion libraries Mocha supports). For the examples in this book we will be using the assertion library that comes with Node.js.

```
> var assert = require('assert');
> assert(2+2 === 5, 'incorrect')
// outputs: AssertionError: incorrect
```

For assertions in our Selenium tests we'll need to deal with Promises (more on them soon). Basically, we'll ask Selenium something about the page, expecting either a `true` or a `false` response. But it won't give it to use immediately. Instead we need to wait for the result and assert on that.

```
driver.findElement({css: '.flash.success'}).isDisplayed().then(function(
elementDisplayed) {
      assert.equal(elementDisplayed, true, 'Success message not displayed');
    });
```

If this is the only assertion in your test then this will result in a passing test. More on this and other good test writing practices in Chapter 5.

Conditionals

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

The most common conditionals in JavaScript are `if`, `else if`, and `else` statements.

```
var number = 10;
if (number > 10) {
  console.log('The number is greater than 10');
} else if (number < 10) {
  console.log('The number is less than 10');
} else if (number === 10) {
  consle.log('The number is 10');
} else {
  console.log('I don't know what the number is.');
};
// outputs: The number is 10
```

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

```
} else if (config.host === 'localhost') {
    if (config.browser === 'chrome') {
      var vendorDirectory = process.cwd() + '/vendor';
      process.env.PATH = vendorDirectory + ":$PATH";
    }
    builder = new webdriver.Builder().forBrowser(config.browser);
```

More on that in chapters 12 and 13.

Inheritance

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

Inheritance in JavaScript can be established by:

- importing the parent class (when in another file)
- calling the parent class constructor from the child class constructor
- setting the child's `prototype` value to the parent's
- setting the child's `prototype.constructor` to itself

```
function Parent() {
  this.hairColor = 'brown';
};

function Child() {
  Parent.call(this);
};

Child.prototype = Object.create(Parent.prototype);
Child.prototype.constructor = Child;

var child = new Child();
console.log(child.hairColor);
// outputs: brown
```

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

Promises

Test execution with Selenium is a fundamentally synchronous activity (e.g., visiting a page, typing text input a field, submitting the form, and waiting for the response). But JavaScript execution is inherently asynchronous, meaning that it will not wait for a command to finish executing before proceeding onto the next one. It will just keep going and the executed commands will eventually complete and return a result (a.k.a. a callback). Left unchecked it's obviously a non-starter for automated functional testing.

To account for this we enlist the help of Promises. Promises represent the result of each asynchronous action and can act as a blocking function that will wait for them to complete. Each promise comes in one of three states -- pending, fulfilled, or rejected. Thankfully, built into the Selenium Node.js bindings is a Promise handler that takes care of most of this for us (a.k.a. ControlFlow). So out of the box we get the appearance of a fairly synchronous test-writing experience by just using the built-in Selenium API functionality that is consistent across all of the language bindings.

However, there are some circumstances where we'll need to modify the Promise handler in our test code. More on that in chapters 8 and 9.

Additional Resources

Here are some additional resources that can help you continue your JavaScript/Node.js learning journey.

- codecademy JavaScript course
- Node.js Tutorals for Beginners (videos)
- NodeSchool
- Introduction to Object Oriented JavaScript
- JavaScript: The Good Parts (book)

Chapter 5

Anatomy Of A Good Acceptance Test

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

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

Atomic & Autonomous Tests

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

Grouping Tests

As your test suite grows you will have numerous test files. Each one containing a grouping of tests that exercise similar functionality. These test files should be grouped together in a simple directory structure as the groupings become obvious. If you want to create a test run of disparate tests, this is something that is easy to handle when using a Test Runner (covered briefly below, and in-depth in Chapter 15).

Being Descriptive

A test file should have a high level name that denotes what the tests within it are exercising. Each test should have an informative name (even if it is a bit verbose). Also, each test (or grouping of tests) should include some helpful metadata (e.g., Categories) which can provide additional information about the test as well as enable flexible test execution (more on that in Chapter 15). This way all or part of your test suite can be run, and the results will be informative thanks to helpful naming.

This enables developers to run a subset of tests to exercise functionality they just modified (as part of their pre-check-in testing) while also enabling you to intelligently wire your test suite up to a Continuous Integration (CI) server for fast and dynamic feedback (more on CI servers in Chapter 16).

Test Runners

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

Version Control

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

- Git
- Mercurial
- Subversion

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

Chapter 6
Writing Your First Test

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

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

1. Visit the login page of a site
2. Find the login form's username field and input the username
3. Find the login form's password field and input the password
4. Find the 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 (sorted alphabetically):

- Class
- CSS Selector
- ID
- Link Text
- Name
- Partial Link Text
- Tag Name
- 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 markup 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

You want to find an element that is unique, descriptive, and unlikely to change.

Ripe candidates for this are `id` and `class` attributes. Whereas text (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 element that you can use to start with and walk down to the child element you want to use.

When 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 typically a trivial thing for them to add helpful semantic markup to a page to make it more testable. This is especially true when they know the use case you're trying to automate. The alternative can be a lengthy and painful process which will probably yield working test code but it will be brittle and hard to maintain.

Once you've identified the target elements for your test, you need to craft a locator using one Selenium's strategies.

An Example

Part 1: Find The Elements And Write The Test

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

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

Note the unique elements on the form. The username input field has a unique `id`, as does the password input field. The submit button doesn't, but it's the only button on the page so we can easily find it and click it.

Let's put these elements to use in our first test. First we'll need to create a new folder called `test` in the root of our project directory. This is a default folder that Mocha will know to look for. In it we'll create a new test file called `LoginTest.js`. When we're done our directory structure should look like this (minus the requisite `node_modules` directory).

```
package.json
test
    LoginTest.js
```

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

```
// filename: test/LoginTest.js
'use strict';
var webdriver = require('selenium-webdriver');
var test = require('selenium-webdriver/testing');

test.describe('Login', function() {
  this.timeout(30000);
  var driver;

  test.beforeEach(function() {
    driver = new webdriver.Builder().forBrowser('firefox').build();
  });

  test.afterEach(function() {
    driver.quit();
  });

  test.it('with valid credentials', function() {
    driver.get('http://the-internet.herokuapp.com/login');
    driver.findElement({id: 'username'}).sendKeys('tomsmith');
    driver.findElement({id: 'password'}).sendKeys('SuperSecretPassword!');
    driver.findElement({css: 'button'}).click();
  });

});
```

At the top of the file we specify `'use strict';`. This places our test code into a tighter, safer operating context. It will protect us by throwing exceptions with helpful information when we do something "unsafe" or make a blunder in our code. Next we import the requisite classes for Selenium. One is to create and control an instance of Selenium. The other is a wrapper for Mocha that will help ensure that the tests execute in a synchronous fashion.

We declare a test class with `test.describe('Login', function() {` and specify a timeout for Mocha in milliseconds (e.g., `this.timeout(30000)`). The default timeout for Mocha is `2000` milliseconds (or 2 seconds). If we don't change it then our test will fail before the browser finishes loading. Next we declare a `driver` variable where we'll store our instance of Selenium. We handle the setup and teardown of Selenium in `test.beforeEach` and `test.afterEach` methods. This ensures that a clean instance of Selenium is created before each test and destroyed after each test. To create an instance of Selenium we call `new webdriver.Builder().forBrowser('firefox').build();` and store it in a our `driver` variable.

Our test method starts with `test.it` and a helfpul name (e.g., `'with valid credentials'`). In this test we're visiting the login page by its URL (with `driver.get()`), finding the input fields by their ID (with `driver.findElement({id: 'username'});`), inputting text into them (with `.sendKeys();`), and submitting the form by clicking the submit button (e.g., `driver.findElement({css: 'button'}).click()`).

If we save this and run it (by running `mocha` from the command-line), it will run and pass. But there's one thing missing -- an assertion. In order to find an element to write an assertion against we need to see what the markup of the page is after submitting the login form.

Part 2: Figure Out What To Assert

Here is the markup that renders on the page after logging in.

```html
<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>
```

There are a couple of elements we can use for our assertion in this markup. There's the flash message class (most appealing), the logout button (appealing), or the copy from either the `h2` or the flash message (least appealing).

Since the flash message class name is descriptive, denotes a successful login, and is less likely to change than the copy, let's go with that.

```
class="flash success"
```

When we try to access an element like this (e.g., with a multi-worded class) we will need to use a CSS selector or an XPath.

NOTE: Both CSS selectors and XPath work well, but the examples throughout this book will focus on how to use CSS selectors.

A Quick Primer on CSS Selectors

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

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

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

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

Part 3: Write The Assertion And Verify It

Now that we have our locator, let's add an assertion that uses it.

```
// filename: test/LoginTest.js
'use strict';
var webdriver = require('selenium-webdriver');
var test = require('selenium-webdriver/testing');
var assert = require('assert');
// ...
  test.it('with valid credentials', function() {
    driver.get('http://the-internet.herokuapp.com/login');
    driver.findElement({id: 'username'}).sendKeys('tomsmith');
    driver.findElement({id: 'password'}).sendKeys('SuperSecretPassword!');
    driver.findElement({css: 'button'}).click();
    driver.findElement({css: '.flash.success'}).isDisplayed().then(function(
elementDisplayed) {
      assert.equal(elementDisplayed, true, 'Success message not displayed');
    });
  });
```

With `assert` we are checking for a `true` Boolean response. If one is not received the test will fail. With Selenium we are seeing if the success message element is displayed on the page (with `.isDisplayed()`). This Selenium command returns a promise which ultimately returns a boolean. To account for this we need to call `.then` and specify a variable. This variable will contain the boolean response. So we perform a check against it (e.g., `elementDisplayed`) instead of the Selenium command directly. If the element is rendered on the page and is visible (e.g., not hidden or covered up by an overlay), `true` will be returned, and our test will pass.

When we save this and run it (e.g. `mocha` from the command-line) it will run and pass just like before, but now there is an assertion which will catch a failure if something is amiss.

Just To Make Sure

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

```
driver.findElement({css: '.flash.successasdf'}).isDisplayed().then(function(
elementDisplayed) {
      assert.equal(elementDisplayed, true, 'Success message not displayed');
    });
```

If it fails then we can feel reasonably confident that the test is doing what we expect and 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.

Chapter 7
Verifying Your Locators

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

Instead of the painful and tedious process of trying out various locators in your tests until you get what you're looking for, try verifying them in the browser instead.

A Solution

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

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

An Example

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

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

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

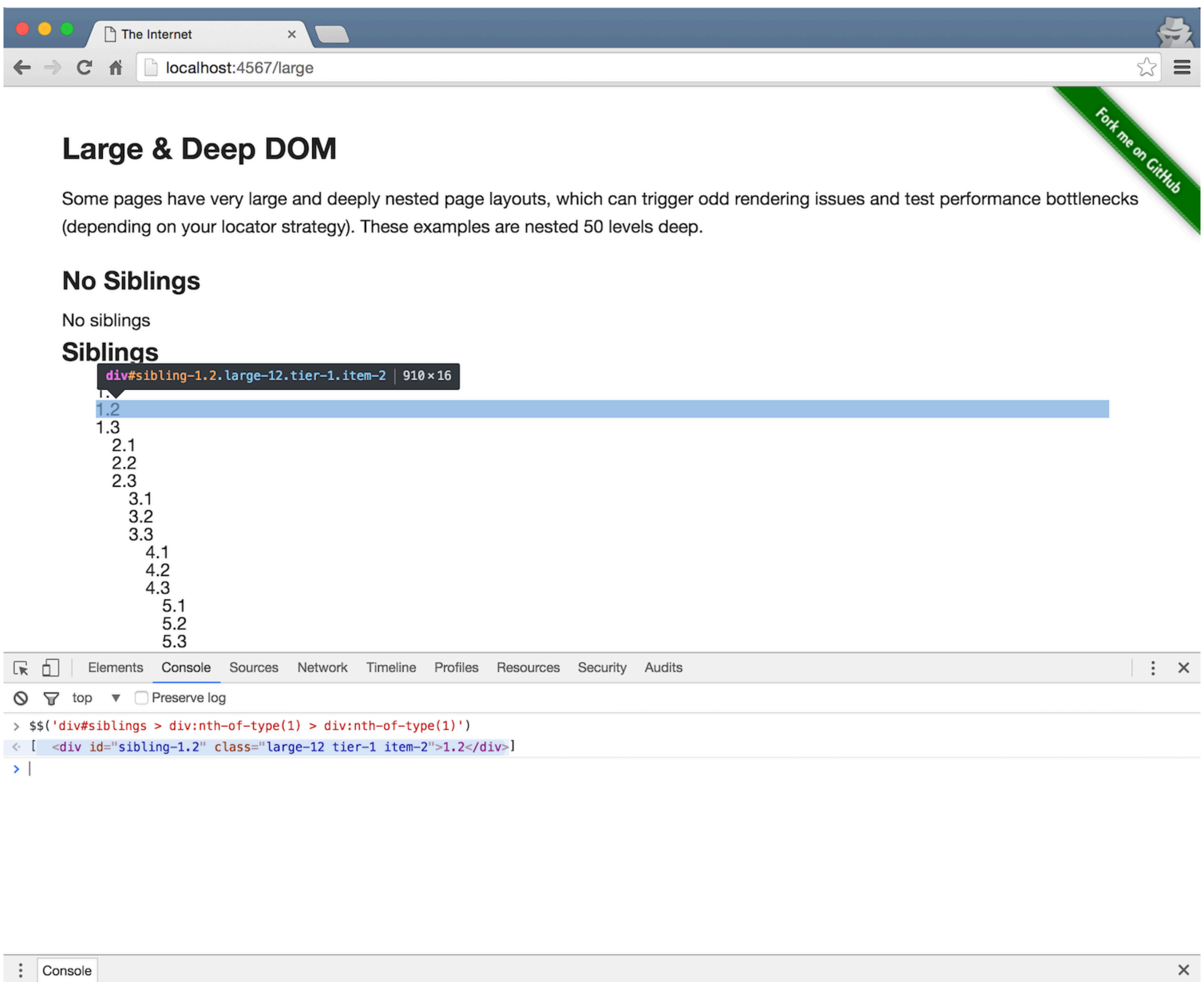
If we perform a `FindElement` action using the following locator, it works.

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

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

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

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

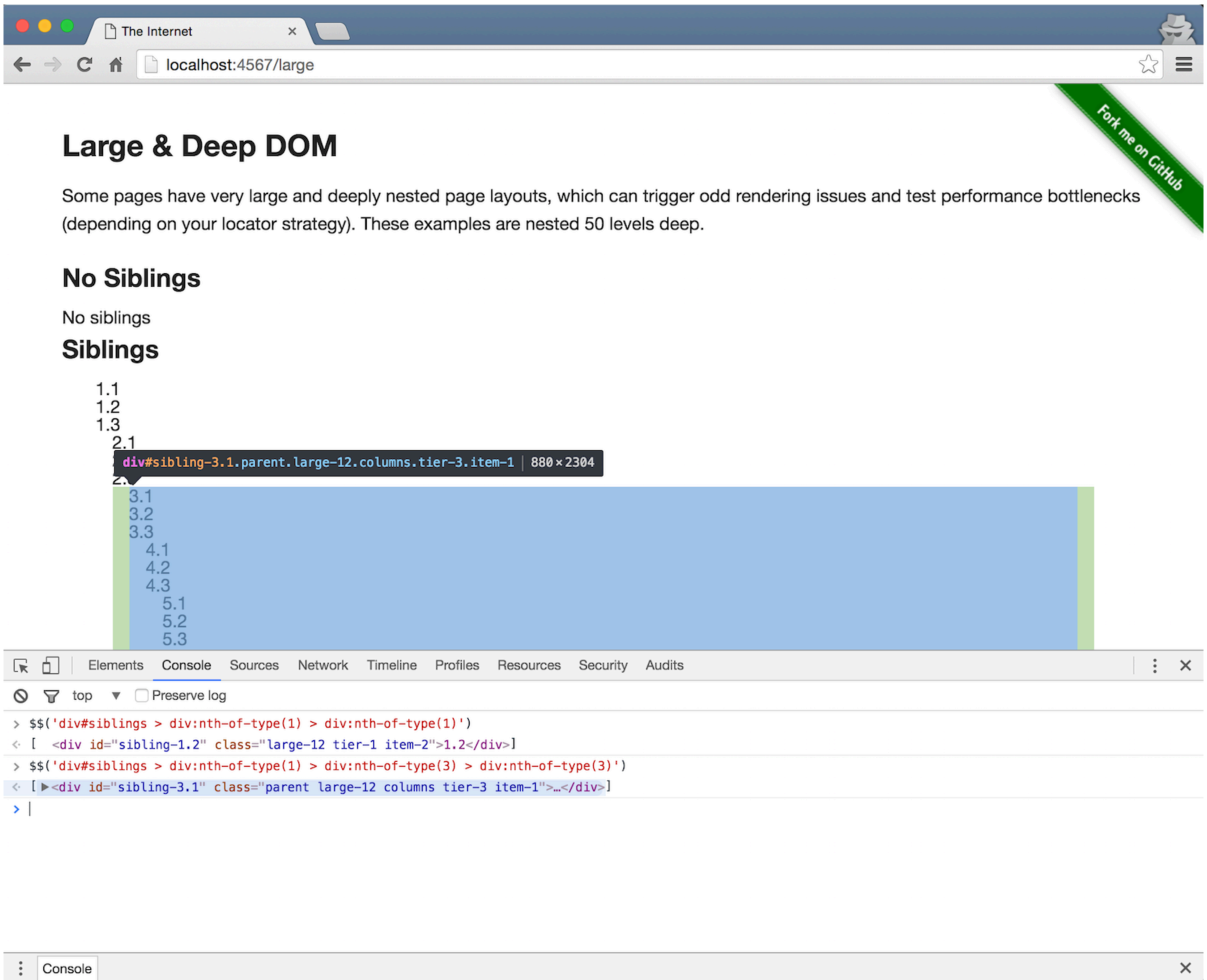


It looks like our locators are scoping to the wrong part of the first level (1.2). But we need to reference the third part of each level (e.g., 1.3, 2.3, 3.3) in order to traverse deeper since the nested divs live under the third part of each level.

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

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

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



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

Chapter 8

Writing Re-usable Test Code

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

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.

A Page Objects Primer

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

With this approach we not only get the benefit of controlled chaos, we also get reusable functionality across our suite of tests and more readable tests.

An Example

Part 1: Create A Page Object And Update Test

Let's take our login example from earlier, create a page object for it, and update our test accordingly.

First we'll need to create a new folder called `pages` in the root of our project (just like we did for `test`). In it we'll add a `LoginPage.js` file. When we're done our directory structure should look like this.

```
package.json
pages
   LoginPage.js
test
    LoginTest.js
```

And here's the code that goes with it.

```
//filename: pages/LoginPage.js
'use strict';

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

function LoginPage(driver) {
  this.driver = driver;
  this.driver.get('http://the-internet.herokuapp.com/login');
}

LoginPage.prototype.with = function(username, password) {
  this.driver.findElement(USERNAME_INPUT).sendKeys(username);
  this.driver.findElement(PASSWORD_INPUT).sendKeys(password);
  this.driver.findElement(SUBMIT_BUTTON).click();
};

LoginPage.prototype.successMessagePresent = function() {
  return this.driver.findElement(SUCCESS_MESSAGE).isDisplayed();
};

module.exports = LoginPage;
```

At the top of the file we specify `'use strict';` and some variables. One is for our instance of Selenium and the rest are for the locators we want to use on the page. We then declare the class by specifying its constructor (e.g., `function LoginPage(driver) {`). It will run whenever a new instance of this class is created. In order for this class to work we need access to the Selenium driver object, so we accept it as a parameter and store it in the `this.driver` class variable (so other methods in the class can access it). Then the login page is visited (with `driver.get()`).

The second method (e.g., `LoginPage.prototype.with = function(username, password) {`) is the core functionality of the login page. It's responsible for filling in the login form and submitting it. By accepting parameters for the username and password we're able to make the functionality here reusable for additional tests. Instead of the hard-coded locators, we updated the Selenium calls with the locator variables we specified near the top of the class.

The last method (e.g., `LoginPage.prototype.successMessagePresent`) is the display check from earlier that was used in our assertion. It will return a promise and ultimately a boolean result just like before. And the class ends with `module.exports = LoginPage;`. This makes it so the class gets returned as an object when it is required in our test.

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

```javascript
// filename: test/LoginTest.js
'use strict';
var webdriver = require('selenium-webdriver');
var test = require('selenium-webdriver/testing');
var assert = require('assert');
var LoginPage = require('../pages/LoginPage');

test.describe('Login', function() {
  this.timeout(30000);
  var driver;
  var login;

  test.beforeEach(function() {
    driver = new webdriver.Builder().forBrowser('firefox').build();
    login = new LoginPage(driver);
  });

  test.afterEach(function() {
    driver.quit();
  });

  test.it('with valid credentials', function() {
    login.with('tomsmith', 'SuperSecretPassword!');
    login.successMessagePresent().then(function(elementDisplayed) {
      assert.equal(elementDisplayed, true, 'Success message not displayed');
    });
  });

});
```

Before we can use the page object we first need to require it (e.g., `var LoginPage = require('../pages/LoginPage');`). Then it's a simple matter of updating our test setup to return an instance of the login page and updating the test method with the new page object actions.

Now the test is more concise and readable. And when you save everything and run it (e.g., `mocha` from the command-line), it will run and pass just like before.

Part 2: Write Another Test

Creating a page object may feel like more work than what we started with initially. But it's well worth the effort since we're in a much sturdier position and able to easily write follow-on tests (since the specifics of the page are abstracted away for simple reuse).

Let's add another test for a failed login to demonstrate.

First, let's take a look at the markup that gets rendered when we provide invalid credentials:

```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>
```

Here is the element we'll want to use in our assertion.

```
class="flash error"
```

Let's add a locator for this element to our page object along with a new method to perform a display check against it.

```javascript
// filename: pages/LoginPage.js
// ...
var SUCCESS_MESSAGE = {css: '.flash.success'};
var FAILURE_MESSAGE = {css: '.flash.error'};
// ...
LoginPage.prototype.failureMessagePresent = function() {
  return this.driver.findElement(FAILURE_MESSAGE).isDisplayed();
};

module.exports = LoginPage;
```

Now we're ready to add a test for failed login to our `test/LoginTest.js` file.

```javascript
// filename: test/LoginTest.js
// ...
  test.it('with invalid credentials', function() {
    login.with('tomsmith', 'bad password');
    login.failureMessagePresent().then(function(elementDisplayed) {
      assert.equal(elementDisplayed, true, 'Failure message not displayed');
    });
  });
```

If we save these changes and run our tests (e.g., `mocha` from the command-line) we will see two browser windows open (one after the other) testing for successful and failure login scenarios.

Why Asserting False Won't Work (yet)

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

```
login.successMessagePresent().then(function(elementDisplayed) {
      assert.equal(elementDisplayed, false, "Success message displayed");
    });
```

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

```
NoSuchElementError: Unable to locate element: {"method":"css selector","selector":
".flash.success"}
```

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

Second, the absence of a success message doesn't necessarily indicate a failed login. The assertion we ended up with originally is more effective.

Part 3: Confirm We're In The Right Place

Before we can call our page object finished, there's one more addition we should 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 resiliency to our test.

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

```
// filename: pages/LoginPage.js
'use strict';
var assert = require('assert');

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

function LoginPage(driver) {
  this.driver = driver;
  this.driver.get('http://the-internet.herokuapp.com/login');
  this.driver.findElement(LOGIN_FORM).isDisplayed().then(function(elementDisplayed) {
    assert.equal(elementDisplayed, true, 'Login form not loaded');
  });
}
// ...
```

After requiring the assertion library and adding a new locator to the page object for the login form we add an assertion to the constructor, just after Selenium visits the login page. This will check that the login form is displayed. If it is the tests using this page object will proceed. If not the test will fail and provide an output message stating that the login form wasn't loaded.

Now when we save everything and run our tests they will run just like before. But now we can feel confident that the tests will only proceed if login page is in a ready state.

Outro

With Page Objects you'll be able to easily maintain and extend your tests. But how you write your Page Objects may vary depending on your preference and experience. The example demonstrated above is a simple approach. It's worth taking a look at the Selenium project wiki page for Page Objects here (even if its examples are only written in Java). There's also Martin Fowler's seminal blog post on the topic as well (link).

Chapter 9
Writing Really Re-usable Test Code

In the previous chapter we stepped through creating a simple page object to capture the behavior of the page we were interacting with. While this was a good start, it leaves some 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 throughout our suite. If we leave this unchecked we will end up with duplicative code which will slowly make our page objects harder to maintain.

Right now we are using Selenium actions directly in our page object. While on the face of it this may seem fine, it has some long term impacts, like:

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

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 class and leveraging it in our page objects.

An Example

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

Part 1: Create The Base Page Object

First let's create the base page object by adding a file called `BasePage.js` to the `pages` directory.

```
package.json
pages
   BasePage.js
   LoginPage.js
test
    LoginTest.js
```

Next let's populate the file.

```
// filename: pages/BasePage.js
'use strict';

var driver;

function BasePage(driver) {
  this.driver = driver;
}

BasePage.prototype.visit = function(url) {
  this.driver.get(url);
};

BasePage.prototype.find = function(locator) {
  return this.driver.findElement(locator);
};

BasePage.prototype.click = function(locator) {
  this.find(locator).click();
};

BasePage.prototype.type = function(locator, inputText) {
  this.find(locator).sendKeys(inputText);
};

BasePage.prototype.isDisplayed = function(locator) {
  return this.find(locator).isDisplayed();
};

module.exports = BasePage;
```

After declaring the class by specifying the constructor (e.g., `function BasePage(driver) {`) we receive and store an instance of Selenium just like in our Login page object. But what's different here is the methods that come after the constructor (e.g., `visit`, `find`, `click`, `type`, and `isDisplayed`). Each one stores a specific behavior we've used in our tests. Some of the names are the same as you've seen in Selenium, others renamed (for improved readability).

Now that we have all of our Selenium actions in one place, let's update our login page object to leverage this facade.

```
// filename: pages/LoginPage.js
'use strict';
var BasePage = require('./BasePage');
var assert = require('assert');

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

function LoginPage(driver) {
  BasePage.call(this, driver);
  this.visit('http://the-internet.herokuapp.com/login');
  this.isDisplayed(LOGIN_FORM).then(function(elementDisplayed) {
    assert.equal(elementDisplayed, true, 'Login form not loaded');
  });
}

LoginPage.prototype = Object.create(BasePage.prototype);
LoginPage.prototype.constructor = LoginPage;

LoginPage.prototype.with = function(username, password) {
  this.type(USERNAME_INPUT, username);
  this.type(PASSWORD_INPUT, password);
  this.click(SUBMIT_BUTTON);
};

LoginPage.prototype.successMessagePresent = function() {
  return this.isDisplayed(SUCCESS_MESSAGE);
};

LoginPage.prototype.failureMessagePresent = function() {
  return this.isDisplayed(FAILURE_MESSAGE);
};

module.exports = LoginPage;
```

Two fundamental things have changed in our Login page object. We've established inheritance between `BasePage` and `LoginPage`, and we've swapped out all of our Selenium commands with the methods we created in `BasePage`.

Inheritance isn't straight-forward in JavaScript since the notion of classes is not really concrete. There are three pieces to it. With `LoginPage.prototype = Object.create(BasePage.prototype);` we are setting the type of this class to `BasePage`. With `Login.prototype.constructor =`

`LoginPage;` we are making sure that the `LoginPage` constructor is still called when we create a new instance of this class. And in the constructor, we are calling the constructor in `BasePage` before we do anything else. Effectively, this means that `LoginPage` is now a child of `BasePage`, which is what enables us to swap out all of the Selenium actions to use the methods we just specified in `BasePage`.

If we save everything and run our tests they will run and pass just like before. But now our page objects are more readable, simpler to write, and easier to maintain and extend.

Part 2: Add Some Error Handling

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

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

```
NoSuchElementError: Unable to locate element: {"method":"css selector","selector":
".flash.success"}
```

The important thing to note is the name of the exception Selenium offered up -- `NoSuchElementError`. Let's modify the `isDisplayed` method in our base page object to handle it.

```
// filename: pages/BasePage.js
'use strict';

var driver,
    Promise = require('selenium-webdriver').promise;
// ...
BasePage.prototype.isDisplayed = function(locator) {
  var defer = Promise.defer();
  this.find(locator).isDisplayed().then(function(isDisplayed) {
    defer.fulfill(isDisplayed);
  }, function(error) {
    if (error.name === 'NoSuchElementError') {
      defer.fulfill(false);
    } else {
      defer.reject(error);
    }
  });
  return defer.promise;
};
```

Since JavaScript is asynchronous we can't rely on something like a `try` / `catch` to handle our Selenium exceptions. Instead we need to leverage the promise handler that's built into the Selenium bindings and manually take control of it when doing an `isDisplayed` lookup.

First we require the promise class and store it in a `Promise` variable. Rather than declare `var` for it we leverage a short-hand by using a comma and specifying it on a new line (tacking it onto the `var` declaration for `driver`).

Next we call `Promise.defer();` in the `isDisplayed` method (storing it in a local variable). This tells Selenium that we're going to manually step through a promise. We then update `this.find(locator).isDisplayed()` to start the promise with a variable that represents it (e.g., `.then(function(isDisplayed) {`). If no exception is thrown then we consider the promise "fulfilled", specifying it with `defer.fulfill(isDisplayed)`.

What comes next is the error callback (e.g., `}, function(error) {`). In this callback we have access to the error that was thrown. So we can check its name and determine the return response based on it. So when the error name is `NoSuchElementError` (e.g., `if (error.name === 'NoSuchElementError')`) we can fulfill the promise with `false` (e.g., `defer.fulfill(false);`) instead of throwing the exception. This ensures that we only return `false` on this specific exception since we want all other exceptions to get thrown normally. We handle all other exceptions in the `else` case of our conditional by calling `defer.reject(error);`.

At the end of the method we make sure to return the promise (e.g., `return defer.promise;`). This makes sure that the caller (e.g., any page object that uses the `isDisplayed` method) gets a promise that it can use regardless if an exception was thrown or not.

With this new handling in place, let's revisit our `'with invalid credentials'` login test and alter it so it checks to see if the success message is not present (which would normally trigger a `NoSuchElementException` exception) to make sure things work as we expect.

```
// filename: test/LoginTest.js
//  ...
  test.it('with invalid credentials', function() {
    login.successMessagePresent().then(function(elementDisplayed) {
      assert.equal(elementDisplayed, false, "Success message displayed");
    });
  });
});
```

When we save our changes and run this test it will run and pass without throwing an exception.

Chapter 10
Writing Resilient Test Code

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

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

But there is a simple approach that makes up the bedrock of reliable and resilient Selenium tests -- and that's how you wait 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 you want to accomplish.

Selenium will repeatedly try this action until either it can be accomplished, or until 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. After that it disappears and is replaced with the text `Hello World!`.

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

Here's the markup from 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 find and click on the start button and verify the finish text.

When writing automation for new functionality like this, you may find it easier to write the test first (to get it working how you'd like) and then create a page object for it (pulling out the behavior and locators from your test). There's no right or wrong answer here. Do what feels intuitive to you. But for this example, we'll create the page object first, and then write the test.

Let's create a new page object file called `DynamicLoadingPage.js` in the `pages` directory.

```
package.json
pages
   BasePage.js
   DynamicLoadingPage.js
   LoginPage.js
test
    LoginTest.js
```

In this file we'll establish inheritance to the base page object and specify the locators and behavior we'll want to use.

```javascript
// filename: pages/DynamicLoadingPage.js
'use strict';
var BasePage = require('./BasePage');

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

function DynamicLoadingPage(driver) {
  BasePage.call(this, driver);
}

DynamicLoadingPage.prototype = Object.create(BasePage.prototype);
DynamicLoadingPage.prototype.constructor = DynamicLoadingPage;

DynamicLoadingPage.prototype.loadExample = function(exampleNumber) {
  this.visit('http://the-internet.herokuapp.com/dynamic_loading/' + exampleNumber);
  this.click(START_BUTTON);
};

DynamicLoadingPage.prototype.finishTextPresent = function() {
  return this.waitForIsDisplayed(FINISH_TEXT, 10000);
};

module.exports = DynamicLoadingPage;
```

Since there are two dynamic loading examples to choose from on the-internet we created the method `loadExample`. It accepts a number as an argument so we can specify which of the examples we want to visit and start.

And similar to our Login page object, we have a display check for the finish text (e.g., `finishTextPresent`). This check is slightly different though. Aside from the different name we're using (e.g., `waitForIsDisplayed`), it has a second argument (an integer value of `10000`). This is a bit of aspirational code that we'll need to write. And this millisecond argument is how we'll specify how long we'd like Selenium to wait for an element to be displayed before giving up. Let's update our base page object to add this new `waitForIsDisplayed` method which will enable explicit waits.

```
// filename: pages/BasePage.js
'use strict';

var driver,
    Promise = require('selenium-webdriver').promise,
    Until = require('selenium-webdriver').until;
// ...
BasePage.prototype.waitForIsDisplayed = function(locator, timeout) {
  var defer = Promise.defer();
  var driver = this.driver;
  driver.wait(Until.elementLocated(locator), timeout).then(function() {
    var element = driver.findElement(locator);
    driver.wait(Until.elementIsVisible(element), timeout).then(function() {
      defer.fulfill(true);
    }, function(error) {
      if (error.name === 'NoSuchElementError') {
        defer.fulfill(false);
      } else {
        defer.reject(error);
      }
    });
  });
  return defer.promise;
};
```

Selenium comes with a wait function which we require and store in a method (e.g., `Until = require('selenium-webdriver').until;`).

The `waitForIsDisplayed` method takes a locator and a timeout. Inside the method we take advantage of `Promise.defer();` again. And we also store the Selenium instance in a local variable (more on that soon). Next we call `driver.wait`, provide an initial condition that we want to wait for (e.g., wait until the element is located), the locator to wait for, and the timeout (e.g., `(Until.elementLocated(locator), timeout)`). We then trigger a promise (e.g., `.then(function() {`). In this new function we start to lose scope of the class that we're in. As a result we're not able to access other class methods (like `this.find`), but we can still access local variables in the method (like the local `driver` instance we created at the top of the method).

In this new promise we do an element lookup which gets passed into another wait condition (e.g., wait until the element is visible -- which is the same thing as `.isDisplayed()` but through the explicit wait API). We then initiate a new promise, this time to handle the promise outcome. If things go as planned we fulfill the promise with `true` (e.g., `defer.fulfill(true);`). If there's a `NoSuchElementError` exception we fulfill the promise with `false` (e.g., `defer.fulfill(false);`). For all other errors we reject the promise and return the error (e.g., `defer.reject(error);`). And at the end of the method we return the promise.

NOTE: This tap-dance of waiting for an element to be present before doing a display lookup is necessary since the display lookup requires the use of a found element.

There are other conditions you can wait for besides `elementLocated` or `elementIsVisible`. You can find a list [here in the API documentation]().

More On Explicit Waits

The major benefit of explicit waits is that if the behavior on the page takes longer than we expect (e.g., due to slow load times, or a feature change), we can simply adjust a single wait time to fix the test -- rather than increase a blanket wait time (which impacts every test). And since the wait is dynamic (e.g., constantly polling), it won't take the full amount of time to complete (like a hard-coded sleep would).

If you're thinking about mixing explicit waits with an implicit wait -- DON'T. If you use both together you're going to run into issues later on due to inconsistent implementations of the implicit wait functionality across local and remote browser drivers. Long story short, you'll end up with randomly failing tests that will be hard to debug. You can read more about the specifics [here]().

A better approach would be to set a default timeout on the explicit wait method (e.g., `BasePage.prototype.waitForIsDisplayed = function(locator, timeout = 15000) {`) and use it where your tests need to account for some delay. This would make specifying a timeout optional (e.g., only necessary when you need a different time than this default). But if you do this it's important to set to a reasonably sized timeout. You want to be careful not to make it too high. Otherwise the tests that use it and don't specify their own timeout can take longer than they need to. But set it too low and your tests can be brittle, forcing you to run down trivial and transient issues.

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

Now that we have our new page object and an updated base page, it's time to write our test to use it.

Let's create a new file called `DynamicLoadingTest.js` in the `test` directory.

```
package.json
pages
   BasePage.js
   DynamicLoadingPage.js
   LoginPage.js
test
    DynamicLoadingTest.js
    LoginTest.js
```

The contents of this test file are similar to `LoginTest.js` with regards to its setup and structure.

```
// filename: test/DynamicLoadingTest.js
'use strict';
var webdriver = require('selenium-webdriver');
var test = require('selenium-webdriver/testing');
var assert = require('assert');
var DynamicLoadingPage = require('../pages/DynamicLoadingPage');

test.describe('Dynamic Loading', function() {
  this.timeout(30000);
  var driver;
  var dynamicLoading;

  test.beforeEach(function() {
    driver = new webdriver.Builder().forBrowser('firefox').build();
    dynamicLoading = new DynamicLoadingPage(driver);
  });

  test.afterEach(function() {
    driver.quit();
  });

  test.it('hidden element', function() {
    dynamicLoading.loadExample('1');
    dynamicLoading.finishTextPresent().then(function(elementDisplayed) {
      assert.equal(elementDisplayed, true, 'Finish text not displayed');
    });
  });
});
```

In our test (e.g., `'hidden element'`) we are visiting the first dynamic loading example and clicking the start button (which is accomplished in `dynamicLoading.loadExample('1');`). We're then asserting that the finish text gets displayed.

When we save this and run it (e.g., `mocha test/DynamicLoadingTest.js` from the command-line) it will:

- Launch a browser
- Visit the page
- Click the start button
- Wait for the loading bar to complete
- Find the finish text
- Assert that it is displayed.
- Close the browser

Part 3: Update Page Object And Add A New Test

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

[The second dynamic loading example]() is laid out similarly to the last one. The only difference is that it renders the final text after the progress bar completes (whereas the previous example had the element on the page but it was hidden).

Here's the markup for it.

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

  <br>

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

  <br>
</div>
```

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

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

Let's add a second test to `DynamicLoadingTest.js` called `'rendered element'` that will load this second example and perform the same check as we did for the previous test.

```
// filename: test/DynamicLoadingTest.js
  test.it('rendered element', function() {
    dynamicLoading.loadExample('2');
    dynamicLoading.finishTextPresent().then(function(elementDisplayed) {
      assert.equal(elementDisplayed, true, 'Finish text not displayed');
    });
  });
```

When we run both tests (e.g., `mocha test/DynamicLoadingTest.js` from the command-line) we will see that the same approach will work in both cases of how the page is constructed.

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 on 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.

Chrome execution can sometimes be faster than Firefox, so you could see some odd timeout failures. This is an indicator that you need to add explicit waits to parts of your page objects that don't already have them. And the inverse is true when running things against older version of Internet Explorer (e.g., IE 8, 9, etc.). This is an indicator that your explicit wait times are not long enough since the browser is taking longer to respond -- so your tests timeout.

The best approach to solve this is an iterative one. Run your tests in a target browser and see which ones fail. Take each failed test, adjust your code as needed, and re-run it against the target browser until they all pass. Repeat for each browser you care about until everything is green.

Closing Thoughts

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

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

Chapter 11

Prepping For Use

Now that we have some tests and page objects, we'll want to start thinking about how to structure our test code to be more flexible. That way it can scale to meet our needs.

Part 1: Global Setup & Teardown

We'll start by pulling the Selenium setup and teardown out of our tests and into a central location.

We'll create two things. A class that will contain the creation and destruction of our Selenium instances (a.k.a. a Driver Factory) and a base test that all tests will pull from (which is similar to the concept of a Base Page Object).

So let's create a new directory call `lib` and create a new files in it called `DriverFactory.js`. And then let's create a new file in the `test` directory called `BaseTest.js`.

```
lib
   DriverFactory.js
package.json
pages
   BasePage.js
   DynamicLoadingPage.js
   LoginPage.js
test
    BaseTest.js
    DynamicLoadingTest.js
    LoginTest.js
```

Here are the initial contents of the Driver Factory.

```javascript
// filename: lib/DriverFactory.js
'use strict';
var webdriver = require('selenium-webdriver');
var driver;

function DriverFactory() {
  this.build();
}

DriverFactory.prototype.build = function() {
  var builder = new webdriver.Builder().forBrowser('firefox');
  this.driver = builder.build();
};

DriverFactory.prototype.quit = function() {
  this.driver.quit();
};

module.exports = DriverFactory;
```

After requiring the Selenium library and specifying a class variable to store the Selenium instance in we create three methods -- a constructor, `build`, and `quit`. `build` is responsible for creating an instance of Selenium and storing it in the `driver` variable. It gets called by the constructor. `quit` is responsible for destroying the Selenium instance (which relies on the `driver` class variable). The class ends with `module.exports`, just like in previous classes we've created.

Now let's put it to use in our base test.

```javascript
// filename: test/BaseTest.js
'use strict';
var test = require('selenium-webdriver/testing');
var DriverFactory = require('../lib/DriverFactory'),
    driverFactory;
global.testTimeout = 30000;

test.beforeEach(function() {
  this.timeout(global.testTimeout);
  driverFactory = new DriverFactory();
  global.driver = driverFactory.driver;
});

test.afterEach(function() {
  this.timeout(global.testTimeout);
  driverFactory.quit();
});
```

In Mocha, when you specify before and after hooks outside of a test class they are used globally for all tests. These are referred to as root-level hooks. In order to make them work with our existing tests we need to leverage the Selenium Mocha wrapper (e.g., `var test = require('selenium-webdriver/testing');`, `test.beforeEach`, and `test.afterEach`). We also need to specify the Mocha timeout like we were doing in our tests, since these actions are called before our test. So rather than duplicate the timeout number, we store it in a global variable and reference it throughout our test code.

In `beforeEach` we set the Mocha timeout and create a new instance of the Driver Factory, storing it in a variable. Then we grab the driver instance from the Driver Factory and store it in a global variable. This global variable is what we'll use in our tests. In `afterEach` we set the Mocha timeout again and then call the `quit()` method in the Driver Factory. The Driver Factory is aware of the Selenium instance since it stores it locally in a variable within the class.

Now to update our tests.

```
// filename: test/LoginTest.js
'use strict';
var test = require('selenium-webdriver/testing');
var assert = require('assert');
var BaseTest = require('./BaseTest');
var LoginPage = require('../pages/LoginPage');

test.describe('Login', function() {
  this.timeout(global.testTimeout);
  var login;

  test.beforeEach(function() {
    login = new LoginPage(global.driver);
  });

  test.it('with valid credentials', function() {
    login.with('tomsmith', 'SuperSecretPassword!');
    login.successMessagePresent().then(function(elementDisplayed) {
      assert.equal(elementDisplayed, true, 'Success message not displayed');
    });
  });

  test.it('with invalid credentials', function() {
    login.with('tomsmith', 'bad password');
    login.failureMessagePresent().then(function(elementDisplayed) {
      assert.equal(elementDisplayed, true, 'Failure message not displayed');
    });
  });

});
```

```
// filename: test/DynamicLoadingTest.js
'use strict';
var test = require('selenium-webdriver/testing');
var assert = require('assert');
var BaseTest = require('./BaseTest');
var DynamicLoadingPage = require('../pages/DynamicLoadingPage');

test.describe('Dynamic Loading', function() {
  this.timeout(global.testTimeout);
  var dynamicLoading;

  test.beforeEach(function() {
    dynamicLoading = new DynamicLoadingPage(global.driver);
  });

  test.it('hidden element', function() {
    dynamicLoading.loadExample('1');
    dynamicLoading.finishTextPresent().then(function(elementDisplayed) {
      assert.equal(elementDisplayed, true, 'Finish text not displayed');
    });
  });

  test.it('rendered element', function() {
    dynamicLoading.loadExample('2');
    dynamicLoading.finishTextPresent().then(function(elementDisplayed) {
      assert.equal(elementDisplayed, true, 'Finish text not displayed');
    });
  });
});
```

In order to use the Base Test we need to require it. Then we're able to remove the creation and storing of a driver instance in `beforeEach` and leverage the global driver variable instead. We're also able to remove the `afterEach` method entirely.

If we save our files and run our tests (e.g., `mocha` from the command-line) they should work just like before.

Part 2: Base URL

It's a given that we'll need to run our tests against different environments (e.g., local, test, staging, production, etc.). So let's make it so we can specify a different base URL for our tests at runtime.

First, let's create a file called `config.js` in the `lib` directory.

```
lib
   DriverFactory.js
   config.js
package.json
pages
   BasePage.js
   DynamicLoadingPage.js
   LoginPage.js
test
    BaseTest.js
    DynamicLoadingTest.js
    LoginTest.js
```

In it we'll specify a variable for `baseUrl` that will grab and store an environment. If one is not provided then a sensible default will be used.

```
// filename: lib/config.js
module.exports = {
  baseUrl: process.env.BASE_URL || 'http://the-internet.herokuapp.com'
};
```

Now let's update the visit method in our Base Page object to use this config object.

```
// filename: pages/BasePage.js
'use strict';

var driver,
    Promise = require('selenium-webdriver').promise,
    Until = require('selenium-webdriver').until,
    config = require('../lib/config');
// ...
BasePage.prototype.visit = function(url) {
  if (url.startsWith('http')) {
    this.driver.get(url);
  } else {
    this.driver.get(config.baseUrl + url);
  }
};
```

In `visit` there could be a case where we'll want to navigate to a full URL so to be safe we've added a conditional check of the `url` parameter to see if a full URL was passed in. If so, we'll visit it. If not, the `config.baserl` will be combined with the URL path that was passed (e.g., `url`) to create a full URL (e.g., `config.baseUrl + url`) and visit it.

Now all we need to do is update our page objects so they're no longer using hard-coded URLs when calling `visit`.

```
// filename: pages/LoginPage.js
// ...
function LoginPage(driver) {
  BasePage.call(this, driver);
  this.visit('/login');
// ...
```

```
// filename: pages/DynamicLoadingPage.js
// ...
DynamicLoadingPage.prototype.loadExample = function(exampleNumber) {
  this.visit('/dynamic_loading/' + exampleNumber);
// ...
```

Outro

Now when running our tests, we can specify a different base URL by providing some extra information at run-time (e.g., `BASE_URL=url mocha`). We're also in a better position now with our setup and teardown abstracted into a central location. Now we can easily extend our test framework to run our tests on other browsers.

Chapter 12
Running A Different Browser Locally

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

A Brief Primer On Browser Drivers

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

WebDriver works with each of the major browsers through a browser driver which is (ideally but not always) maintained by the browser manufacturer. It is an executable file (consider it a thin layer or a shim) that acts as a bridge between Selenium and the browser.

Let's step through an example using ChromeDriver.

An Example

Before starting, we'll need to download the latest ChromeDriver binary executable for our operating system from here (pick the highest numbered directory) and store the unzipped contents of it somewhere. The simplest thing to do is create a new folder for it (and other things like it) in our test code.

Let's create a `vendor` directory in the root of our project and place the ChromeDriver binary file there.

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

```
lib
   DriverFactory.js
   config.js
package.json
pages
   BasePage.js
   DynamicLoadingPage.js
   LoginPage.js
test
   BaseTest.js
   DynamicLoadingTest.js
   LoginTest.js
vendor
    chromedriver
```

In order for Selenium to use this binary we have to make sure it knows where it is. There are two ways to do that. We can either manually add `chromedriver` to our system path, or we can update our system path for the current terminal session automatically in our Driver Factory. For simplicity, let's go with the latter option.

We'll also want to make sure our test suite can run either Firefox or Chrome. To do that, we'll need to make a couple of changes.

First, let's add a `browser` value to our `config.js` file that will check for the existing of a `BROWSER` environment variable. If there isn't one then we will default to `'firefox'`.

```
// filename: lib/config.js
module.exports = {
  baseUrl: process.env.BASE_URL || 'http://the-internet.herokuapp.com',
  browser: process.env.BROWSER || 'firefox'
};
```

Now to update our Driver Factory to use the browser value and add the vendor directory to the system path.

```
// filename: lib/DriverFactory.js
'use strict';
var webdriver = require('selenium-webdriver');
var config = require('./config');
var driver;

function DriverFactory() {
  this.build();
}

DriverFactory.prototype.build = function() {
  if (config.browser === 'chrome') {
    var vendorDirectory = process.cwd() + '/vendor';
    process.env.PATH = vendorDirectory + ":$PATH";
  }
  var builder = new webdriver.Builder().forBrowser(config.browser);
  this.driver = builder.build();
};

DriverFactory.prototype.quit = function() {
  this.driver.quit();
};

module.exports = DriverFactory;
```

We first require `config.js` and store it in a class variable. We then update our `build` method with a conditional to check on which browser value was passed in. If `'chrome'` was provided, we add the full path to the vendor directory (which contains the `chromedriver` binary) to our system path. After that we create an of Selenium, passing in the value for `config.browser`.

Now we can specify Chrome as our browser when launching our tests (e.g., `BROWSER=chrome mocha`).

NOTE: 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 ways to set this up, but this is good enough to see where our tests fall down in Chrome (and it will not be the primary way we will run our tests a majority of the time anyway -- more on that later in the book).

Additional Browsers

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

- ChromeDriver
- EdgeDriver
- FirefoxDriver
- InternetExplorer Driver
- SafariDriver

Chapter 13
Running Browsers In The Cloud

If you've ever needed to test features in an older browser like Internet Explorer 8 or 9 then odds are you ran a virtual machine (VM) on your computer with a "legit" version of Windows.

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

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

A Selenium Remote, Selenium Grid, And Sauce Labs Primer

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

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

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

Let's dig in with an example.

An Example

Part 1: Initial Setup

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

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

```javascript
// filename: lib/config.js
module.exports = {
  baseUrl: process.env.BASE_URL || 'http://the-internet.herokuapp.com',
  host: process.env.HOST || 'saucelabs',
  browser: process.env.BROWSER || 'internet explorer',
  browserVersion: process.env.BROWSER_VERSION || '11.0',
  platform: process.env.PLATFORM || 'Windows 7',
  sauceUsername: process.env.SAUCE_USERNAME,
  sauceAccessKey: process.env.SAUCE_ACCESS_KEY
};
```

In addition to the `baseUrl` and `browser` variables, we've added some more (e.g., `host`, `browserVersion`, `platform`, `sauceUsername`, and `sauceAccessKey`).

`host` enables us to specify whether our tests run locally or on Sauce Labs.

With `browser`, `browserveresion`, and `platform` we can specify which browser and operating system combination we want our tests to run on. You can see a full list of Sauce's available platform options [here](). They also have a handy configuration generator (which will tell you what values to plug into your test suite at run-time) [here]().

Now we can update the `build` method in our Driver Factory to work with Selenium Remote.

```js
// filename: lib/DriverFactory.js
// ...
DriverFactory.prototype.build = function() {
  var builder;
  if (config.host === 'saucelabs') {
    var url = 'http://ondemand.saucelabs.com:80/wd/hub';
    builder = new webdriver.Builder().usingServer(url);
    builder.withCapabilities({
      browserName: config.browser,
      browserVersion: config.browserVersion,
      platform: config.platform,
      username: config.sauceUsername,
      accessKey: config.sauceAccessKey
    });
  } else if (config.host === 'localhost') {
    if (config.browser === 'chrome') {
      var vendorDirectory = process.cwd() + '/vendor';
      process.env.PATH = vendorDirectory + ":$PATH";
    }
    builder = new webdriver.Builder().forBrowser(config.browser);
  }
  this.driver = builder.build();
  this.driver.getSession().then(function(sessionid){
      sessionId = sessionid.id_;
  });
};
// ...
```

We wrap everything in another condition to check `config.host`. If it's set to `'saucelabs'` then we specify the `url` for their on-demand end-point, create a new `webdriver` object that points to their end-point, and pass in the capabilities that we want and need (e.g., `config.browser`, `config.browserVersion`, `config.platform`, `config.sauceUsername`, and `config.sauceAccessKey`). Otherwise, if `config.host` is set to `'localhost'` then we handle browser execution just like before (checking `config.browser` to see if `'chrome'` was provided and launching a browser locally).

One additional thing we're doing is grabbing and storing the session ID, which we'll use in a later section.

If we save everything and run our tests they will execute in Sauce Labs and on the account dashboard we'll see our tests running in Internet Explorer 11 on Windows 7. To run the tests on different browser and operating system combinations, then simply provide their values as command-line options (e.g., `BROWSER=name --BROWSER_VERSION=version --PLATFORM=os mocha`). For a full list of possible options be sure to check out [the Sauce Labs Platform Configurator].

Part 2: Test Name

It's great that our tests are running on Sauce Labs. But we're not done yet because the test name in each Sauce job is getting set to `unnamed job`. This makes it extremely challenging to know what test was run in the job. To remedy this we'll need to pass the test name to Sauce Labs somehow.

Given the order of operations of our test code, we only have access to the test name after the test has completed. So we'll account for this in both the `quit` method of our Driver Factory and the global `afterEach` in our Base Test. Let's start with the Driver Factory first.

```
// filename: lib/DriverFactory.js
DriverFactory.prototype.quit = function(testName) {
  if (config.host === 'saucelabs') {
    this.driver.executeScript('sauce:job-name=' + testName);
  }
  this.driver.quit();
};
```

With Selenium we have access to execute JavaScript directly in the browser session. When executing tests in Sauce Labs we have access to pass information to them about the current job through JavaScript calls. We take advantage of this fact by specifying the name of the job for the session. And we only want this to happen when our tests are executing in Sauce Labs, so we wrap this in a conditional check.

The test name (e.g., `testName`) gets passed in as an argument for this method. So let's hop over to our Base Test to update the `afterEach` method to pass in the test name.

```
// filename: test/BaseTest.js
// ...
test.afterEach(function() {
  this.timeout(global.testTimeout);
  var testName = this.currentTest.fullTitle();
  driverFactory.quit(testName);
});
```

In Mocha we have access to the test name by virtue of `this.currentTest.fullTitle();`. This will give us both the test class name and the test method name as a string. Which we use to update our call to `driverFactory.quit` so the `testName` is passed in as an argument.

Now when we run our tests in Sauce Labs, [the account dashboard] will show the tests running with a correct name.

Part 3: Test Status

There's still one more thing we'll need to handle, and that's setting the status of the Sauce Labs job after it completes.

Right now regardless of the outcome of a test, the job in Sauce Labs will register as `Finished`. Ideally we want to know if the job was a `Pass` or a `Fail`. That way we can tell at a glance if a test failed or not. And with a couple of tweaks we can make this happen easily enough.

First we need to update our `quit` method in the Driver Factory one more time.

```
// filename: lib/DriverFactory.js
// ...
DriverFactory.prototype.quit = function(testName, testResult) {
  if (config.host === 'saucelabs') {
    this.driver.executeScript('sauce:job-name=' + testName);
    this.driver.executeScript('sauce:job-result=' + testResult);
  }
  this.driver.quit().then(function() {
    if (config.host === 'saucelabs' && testResult === false) {
      console.log('https://saucelabs.com/beta/tests/' + sessionId);
    }
  });
};
```

With the JavaScript executor we're able to pass in the test result, which we're getting as a second parameter on this method. And for good measure we've also put the `testResult` to good use by outputting a URL of the Sauce Labs job to the console if there is a test failure. That way we'll have easy access to the direct URL of the job to review what happened in the test.

Now let's update the `afterEach` in our Base Test.

```
// filename: test/BaseTest.js
test.afterEach(function() {
  this.timeout(global.testTimeout);
  var testName = this.currentTest.fullTitle(),
      testResult = (this.currentTest.state === 'passed') ? true : false;
  driverFactory.quit(testName, testResult);
});
```

We grab the state of the current test (e.g., `this.currentTest.state`) and check to see if it passed (e.g., `=== 'passed'`). Depending on the result we store an appropriate boolean (e.g., `true` or `false`) in `testResult` and pass it into `driverFactory.quit`.

Now when we run our tests in Sauce Labs and navigate to [the Sauce Labs Account dashboard](), we will see our tests running like before. But now there will be a proper test status when they finish (e.g., `Pass` or `Fail`) and we'll see the URL for the job in the console output as well. This enables us to easily jump to the specific job in Sauce Labs.

Part 4: Sauce Connect

There are various ways that companies make their pre-production application available for testing. Some use an obscure public URL and protect it with some form of authentication (e.g., Basic Auth, or certificate based authentication). Others keep it behind their firewall. For those that stay behind a firewall, Sauce Labs has you covered.

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

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

```
lib
   DriverFactory.js
   config.js
package.json
pages
   BasePage.js
   DynamicLoadingPage.js
   LoginPage.js
test
   BaseTest.js
   DynamicLoadingTest.js
   LoginTest.js
vendor
    chromedriver
    sc-4.3.16-osx
        bin
           sc
           sc.dSYM
               Contents
                   Info.plist
                   Resources
                       DWARF
                           sc
        include
           sauceconnect.h
        lib
           libsauceconnect.a
           libsauceconnect.la
        license.html
```

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

```
vendor/sc-4.3.16-osx/bin/sc -u $SAUCE_USERNAME -k $SAUCE_ACCESS_KEY
Sauce Connect 4.3.16, build 2399 c7e5fec
*** WARNING: open file limit 7168 is too low!
*** Sauce Labs recommends setting it to at least 8000.
Starting up; pid 58426
Command line arguments: vendor/sc-4.3.16-osx/bin/sc -u the-internet -k ****
Log file: /tmp/sc.log
Pid file: /tmp/sc_client.pid
Timezone: EDT GMT offset: -4h
Using no proxy for connecting to Sauce Labs REST API.
Resolving saucelabs.com to 162.222.75.243 took 68 ms.
Started scproxy on port 49310.
Please wait for 'you may start your tests' to start your tests.
Starting secure remote tunnel VM...
Secure remote tunnel VM provisioned.
Tunnel ID: 21ff9664b06c4edaa4bd573cdc1fbac1
Secure remote tunnel VM is now: booting
Secure remote tunnel VM is now: running
Using no proxy for connecting to tunnel VM.
Resolving tunnel hostname to 162.222.76.147 took 55ms.
Starting Selenium listener...
Establishing secure TLS connection to tunnel...
Selenium listener started on port 4445.
Sauce Connect is up, you may start your tests.
```

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

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

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

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

Chapter 14

Speeding Up Your Test Runs

We've made huge strides by leveraging page objects, a base page object, explicit waits, and connecting our tests to Sauce Labs. But we're not done yet. Our tests still take a good deal of time to run since they're executing in series (e.g., one after another). As our suite grows this slowness will grow with it.

With parallelization we can easily remedy this pain before it becomes acute by executing multiple tests at the same time. Unfortunately in the Node.js ecosystem there are few current and supported parallel execution libraries for Mocha that will work with the official JavaScript Selenium bindings.

Thankfully, with `grunt` and some plugins for it (e.g., `grunt-parallel` and `grunt-shell`) we can set up a workable solution.

Configuration

First we need to install the additional libraries we need. So let's update our `package.json` file and use `npm install` to install them.

```
// filename: package.json
{
  "name": "selenium-guidebook-examples",
  "dependencies": {
    "grunt": "^1.0.1",
    "grunt-parallel": "^0.5.1",
    "grunt-shell": "^1.3.0",
    "mocha": "2.5.3",
    "selenium-webdriver": "2.53.3"
  }
}
```

NOTE: Alternatively we could have installed the library and have our package.json file auto-updated by using **`npm install package-name --save`**.

Now we have access to Grunt and these supporting libraries. Grunt requires the use of a `Gruntfile.js` file. This is where we'll configure our parallel execution. In the beginning of the file we'll want to grab a collection of the test files and clean it up a bit.

```
// filename: Gruntfile.js
'use strict';

module.exports = function (grunt) {
    var testFiles = grunt.file.expand('test/*.js');
    testFiles.shift();
    testFiles = testFiles.map(function(testFile) {
      return testFile.replace(/test\//, '');
    });
// ...
```

With `grunt.file.expand` we can provide a pattern to match for, which we use to grab all of the JavaScript files in the test directory (e.g., `'test/*.js'`). We used `.shift()` to remove the first entry which is `test/BaseTest.js` (since it's alphabetical). We then iterate over the collection and remove `test/` from each result. This way we're left with just the filenames (e.g., `LoginTest.js`, `DynamicLoadingTest.js`, etc.).

Now to specify the shell execution command we want Grunt to use, and specify all of the parallel tasks we want.

```
// filename: Gruntfile.js
    grunt.initConfig({
      shell: {
        runTests: {
          command: function(testFile) {
            return 'mocha test/'+testFile+'';
          }
        }
      },
      parallel: {
        assets: {
          options: {
            grunt: true
          },
          tasks: testFiles
        }
      }
    });
// ...
```

Within `grunt.initConfig` we configure the core of what we need. Under `shell` we create a `runTests` task that will execute a command that will launch a specific test file (which accepts a test file name as an argument). Under `parallel` we pass in the collection of `testFiles` to denote that we want a task for each test file to be executed in parallel.

Now onto loading and registering these tasks to tie everything together.

```
// filename: Gruntfile.js
// ...
    grunt.loadNpmTasks('grunt-parallel');
    grunt.loadNpmTasks('grunt-shell');
    grunt.registerTask('default', ['parallel']);

    testFiles.forEach(function(testFile) {
        grunt.registerTask(testFile, ['shell:runTests:'+testFile+'']);
    });
};
```

For the `shell` and `parallel` tasks to work we need to first load `grunt-parallel` and `grunt-shell`, which we accomplish with the `grunt.loadNpmTasks` commands. Next we iterate over the collection of test files and register a task for each. Each time specifying a shell command to run (which passes the filename to the shell command).

When we save this file we can now launch our tests using Grunt (e.g., `grunt` from the command-line) and our tests will execute in parallel.

NOTE: If you're using Sauce Labs you'll have a concurrency limit (e.g., number of available concurrent virtual machines you can use). It's listed on the My Account page in the Account Dashboard. This number will be the limiter to how many parallel tests you can run at once. The general recommendation is to limit the number of processes for your test runs to equal the concurrency limit. This solution does not support that. But Sauce Labs will queue the excess sessions that you throw at it and make sure that they get executed once other tests of yours finish.

Random Order Execution

When enabling parallel execution in your tests you may start to see odd, inconsistent behavior that is hard to track down.

This is often due to dependencies between tests that you didn't know were there. A great way to expose these kinds of issues and ensure your tests are ready for prime time is to execute them in a random order. This also has the added benefit of exercising the application you're testing in a random order (which could unearth previously unnoticed bugs).

This is functionality which is still being built for Mocha, so it's not available yet. But you can follow along with its progress here.

Chapter 15
Flexible Test Execution

In order to get the most out of our tests we'll want a way to break them up into relevant, targeted chunks. Running tests in smaller groupings like this (along with parallel execution) will help keep test run times to a minimum and help enhance the amount of feedback you get in a timely fashion.

With [Mocha's `--grep` feature]() we're able to easily achieve test grouping (a.k.a. tags).

Let's step how to set this up.

Specifying Tags

Grep is a simple text match function. It will look through the test files and execute them based on if it found a match or not. To make this helpful we'll want to specify simple names as meta-data which are prepended with a special character to denote that this is what they are.

Some simple examples of this are `@shallow` and `@deep`. `@shallow` tests are roughly equivalent to "smoke" or "sanity" tests. These should pass before you can consider running other tests which aren't as mission critical and may take longer to run (e.g., `@deep` tests).

Let's update our tests to apply these "tags".

```
// filename: test/LoginTest.js
// ...
  test.it('with valid credentials @shallow', function() {
    login.with('tomsmith', 'SuperSecretPassword!');
    login.successMessagePresent().then(function(elementDisplayed) {
      assert.equal(elementDisplayed, true, 'Success message not displayed');
    });
  });

  test.it('with invalid credentials @deep', function() {
    login.with('tomsmith', 'bad password');
    login.failureMessagePresent().then(function(elementDisplayed) {
      assert.equal(elementDisplayed, true, 'Failure message not displayed');
    });
  });
// ...
```

In `LoginTest.js` we updated the test names directly. The happy path test is now marked as

`@shallow` and the invalid credentials test as `@deep`. Now let's apply the `@deep` marker to the entire class in `DynamicLoadingTest.js`.

```
// filename: test/DynamicLoadingTest.js
// ...
test.describe('Dynamic Loading @deep', function() {
// ...
```

Tags are powerful since they can be applied across different test files, enabling you to create a dynamic grouping of tests at runtime.

Running Tags

With Mocha we can specify which marker to launch at runtime. This is handled as another runtime flag on the command-line using `--grep`. But in order to use it in the context of our parallel execution we'll need to update our Gruntfile.

```
// filename: Gruntfile.js
// ...
    grunt.initConfig({
      shell: {
        runTests: {
          command: function(testFile, testOptions) {
            return 'mocha test/'+testFile+' '+testOptions+'';
          }
        }
      },
// ...
    var tag = grunt.option('tag'),
        testOptions = '';

    if (tag) {
      testOptions = '--grep '+tag+'';
    }

    testFiles.forEach(function(testFile) {
        grunt.registerTask(testFile, ['shell:runTests:'+testFile+':'+testOptions+'']);
    });
};
```

First we updated our `shell` execution command to take another argument for `testOptions`. We then specified a Grunt run-time option so we can specify a tag when running our tests. If we provide one, then we store it in `testOptions` and pass it into our `shell` tasks so they get executed with our test suite.

Now we can run our tests with tags by specifying `--tag=tag-name`.

```
grunt --tag=@shallow
```

For more info on this functionality and other available options, check out the documentation. There's also additional tagging functionality in the works for Mocha. You can learn more about that here.

Chapter 16
Automating Your Test Runs

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

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

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

A Continuous Integration Server Primer

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

With CI we can automate our test runs so they can happen as part of the development workflow. The lion’s share of tests that are typically run on a CI Server are unit (and potentially integration) tests. But we can very easily add in our Selenium tests too.

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

- Bamboo
- Jenkins
- Solano Labs
- TravisCI

Let's pick one and step through an example.

A CI Example

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

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

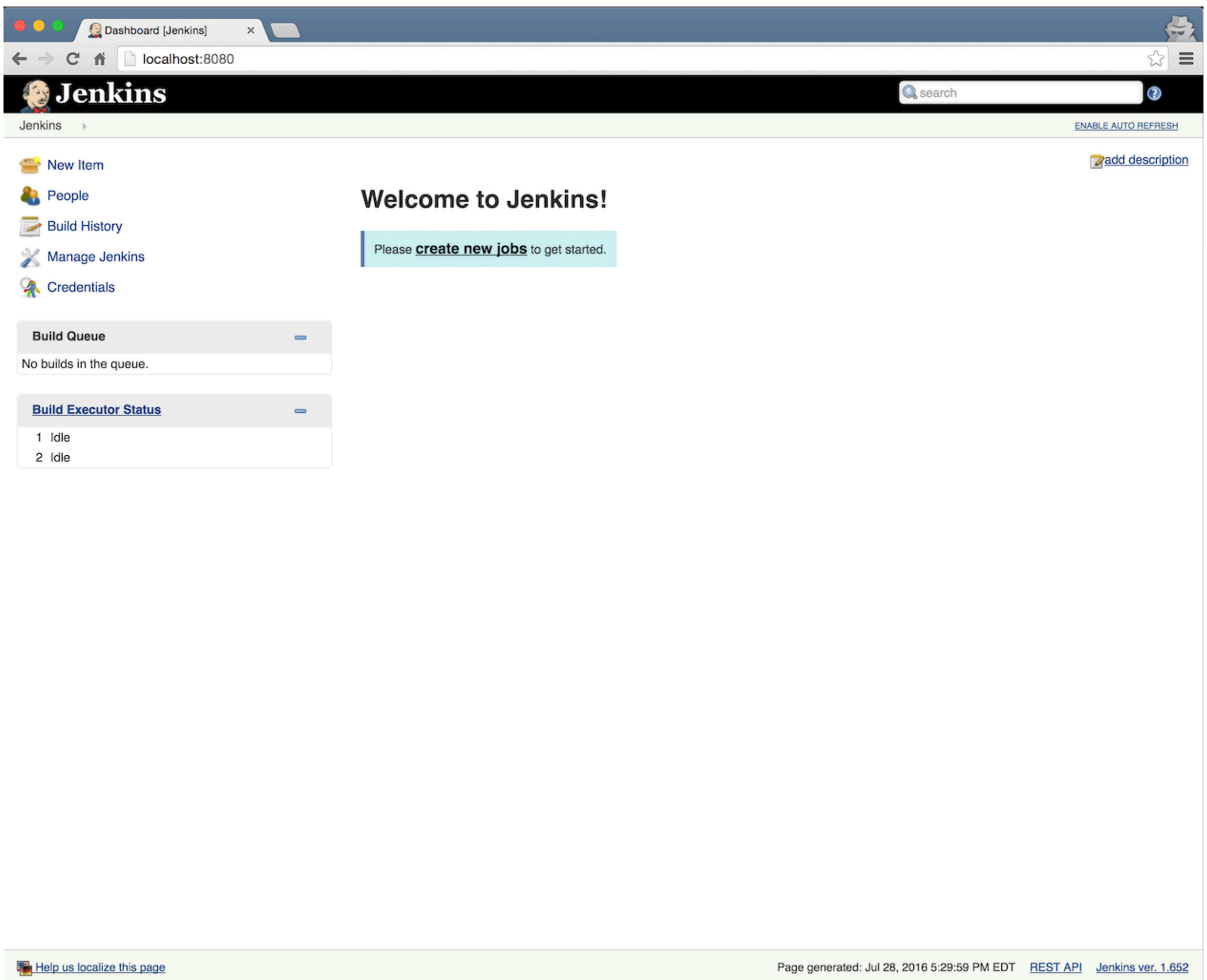
Part 1: Quick Setup

A simple way to get started is to grab the latest Jenkins war file. You can grab it from the [Jenkins homepage](), or from [the direct download link on the homepage]().

Once downloaded, launch it from the command-line.

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

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

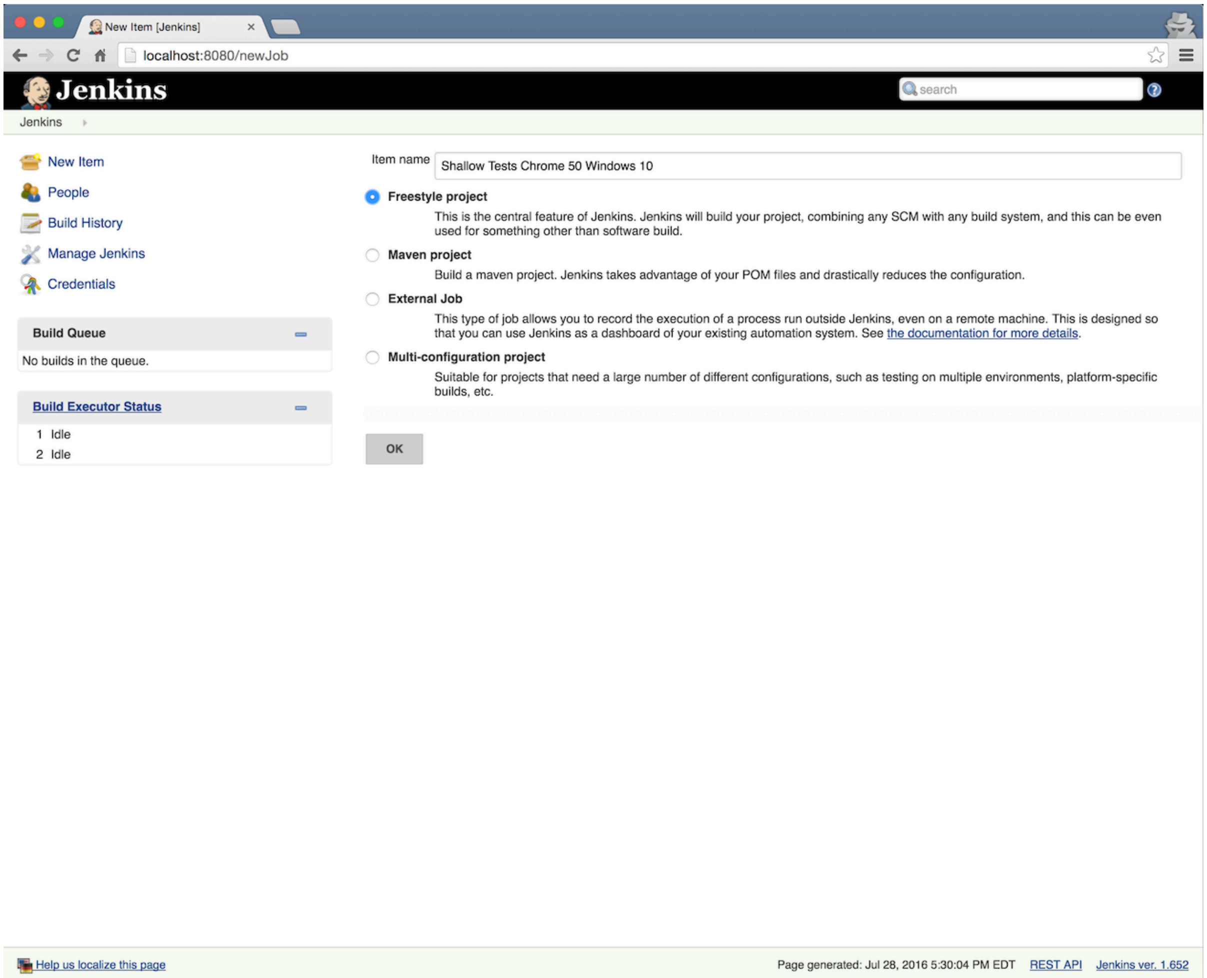


NOTE: Before moving to the next step, click `ENABLE AUTO-REFRESH` at the top right-hand side of the page. Otherwise you'll need to manually refresh the page (e.g., when running a job and waiting for results to appear).

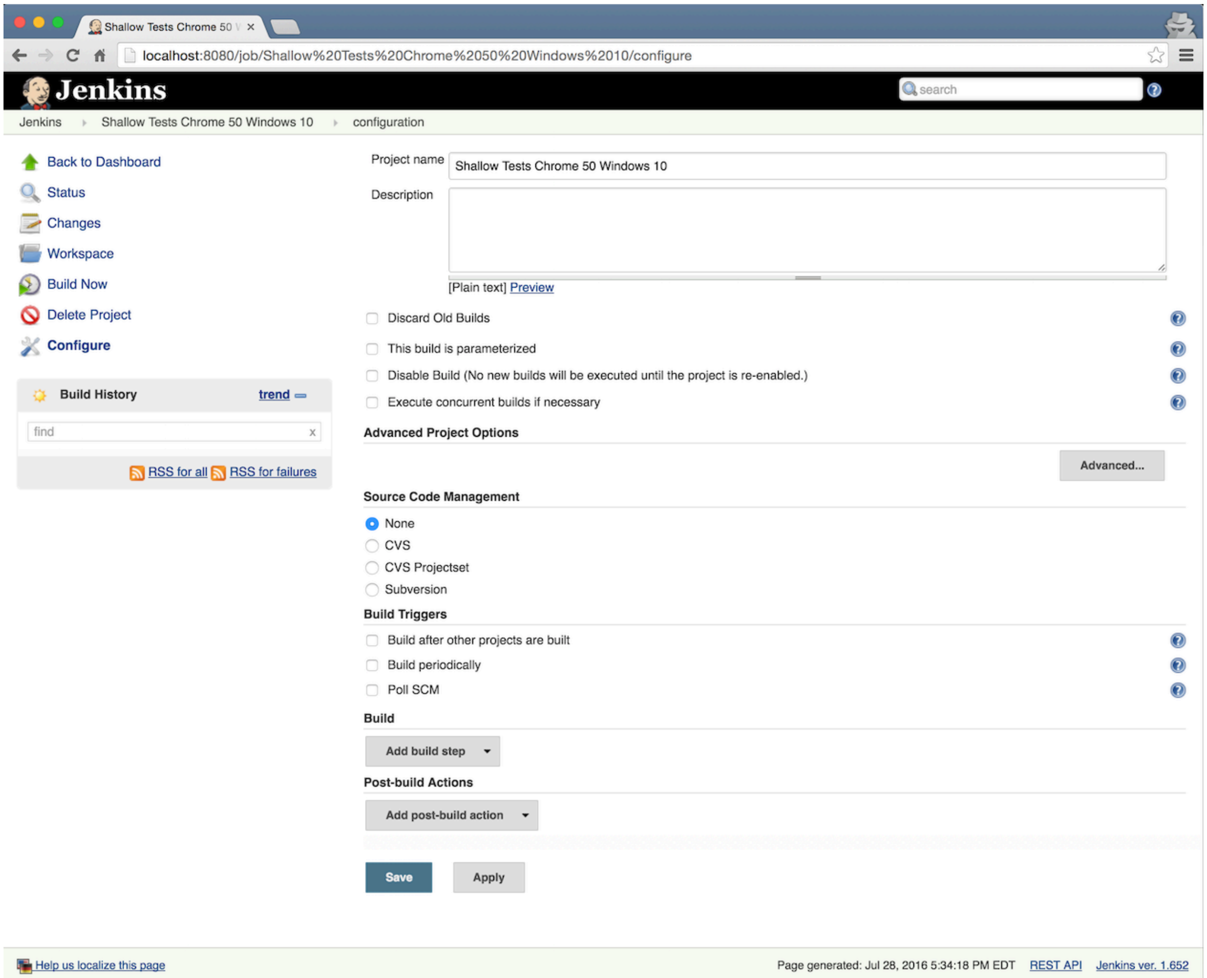
Part 2: Job Creation And Configuration

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

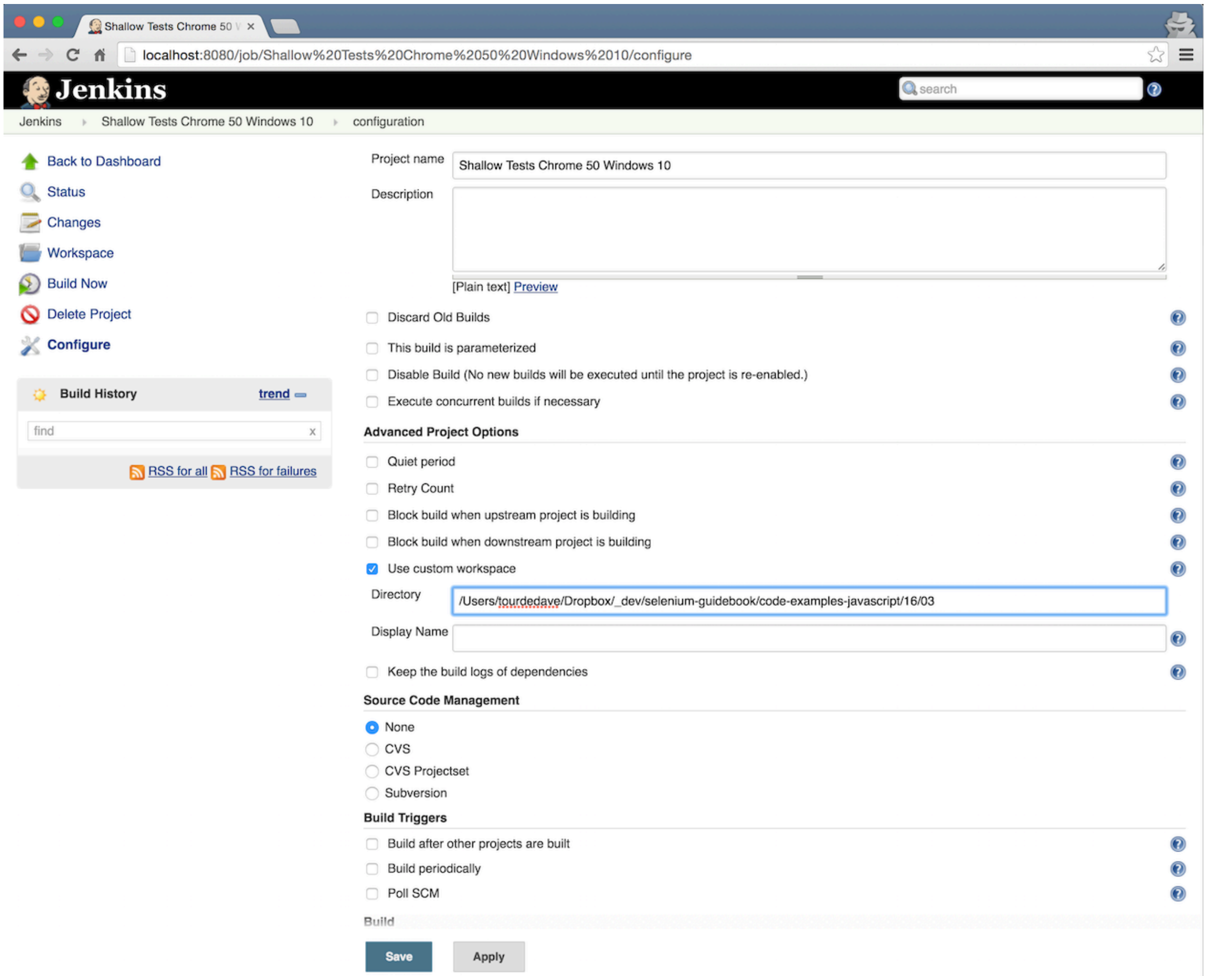
- Click `New Item` from the top-left of the Dashboard
- Give it a name (e.g., `Shallow Tests Chrome 50 Windows 10`)
- Select the `Freestyle project` option
- Click `OK`



This will load a configuration screen for the Jenkins job.

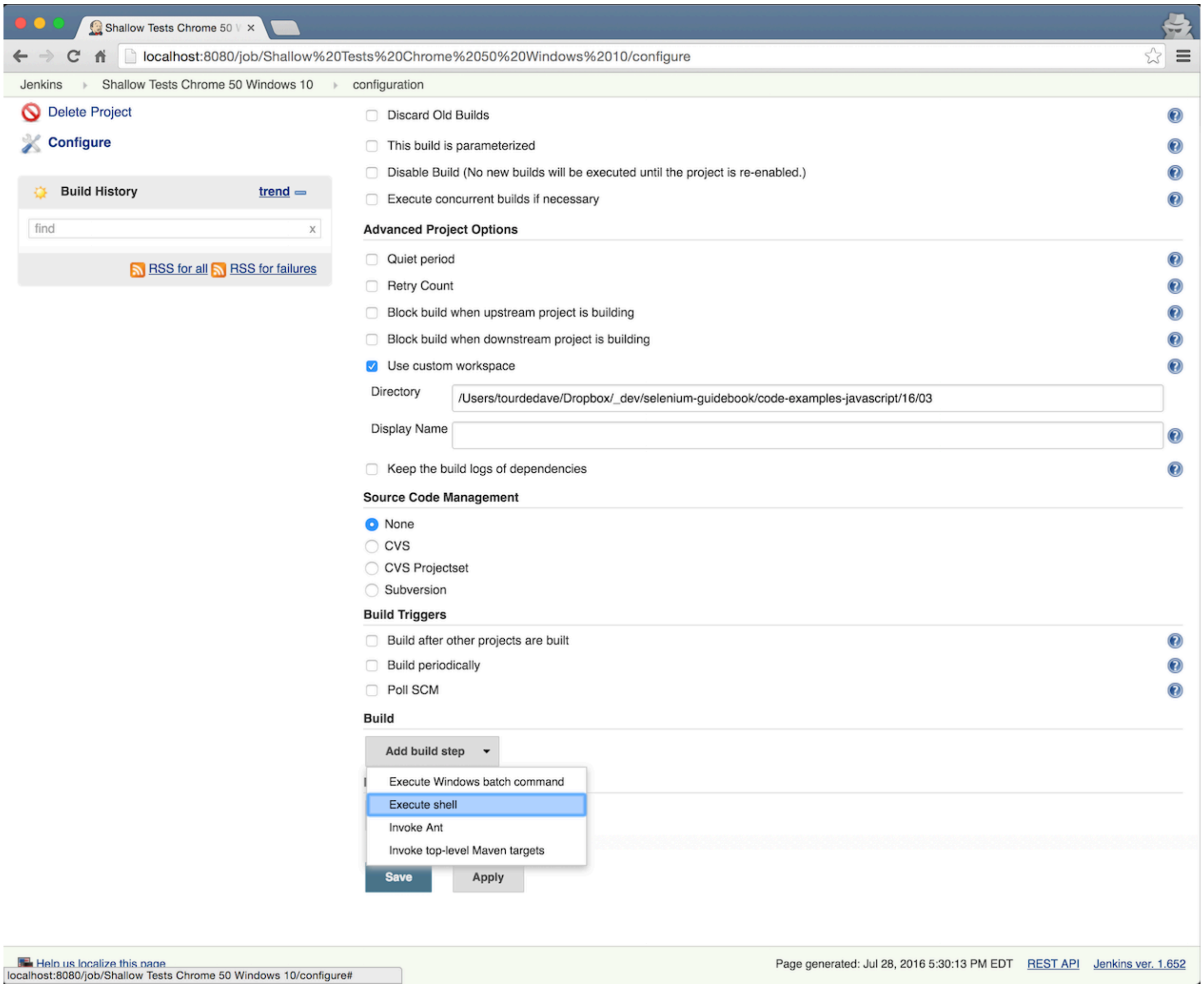


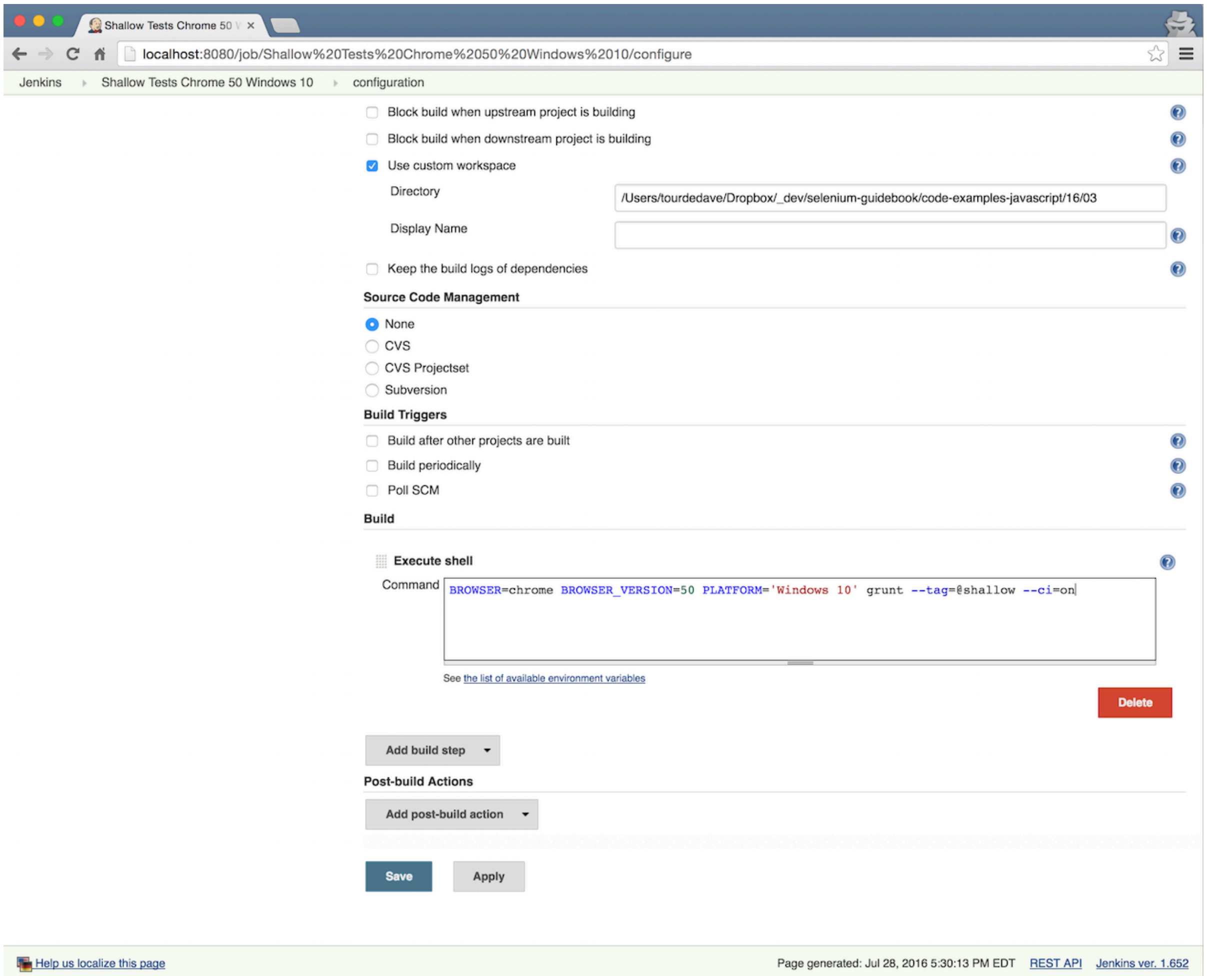
- In the `Advanced Project Options` section select the `Advanced` button
- Choose the checkbox for `Use custom workspace`
- Provide the full path to your test code
- Leave the `Display Name` field blank



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

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


Shallow Tests Chrome 50
localhost:8080/job/Shallow%20Tests%20Chrome%2050%20Windows%2010/configure
Jenkins
Shallow Tests Chrome 50 Windows 10
configuration
Delete Project
Configure
Build History
trend
find
RSS for all
RSS for failures
Discard Old Builds
This build is parameterized
Disable Build (No new builds will be executed until the project is re-enabled.)
Execute concurrent builds if necessary
Advanced Project Options
Quiet period
Retry Count
Block build when upstream project is building
Block build when downstream project is building
Use custom workspace
Directory
/Users/tourdedave/Dropbox/_dev/selenium-guidebook/code-examples-javascript/16/03
Display Name
Keep the build logs of dependencies
Source Code Management
None
CVS
CVS Projectset
Subversion
Build Triggers
Build after other projects are built
Build periodically
Poll SCM
Build
Add build step
Execute Windows batch command
Execute shell
Invoke Ant
Invoke top-level Maven targets
Save
Apply
Help us localize this page
localhost:8080/job/Shallow Tests Chrome 50 Windows 10/configure#
Page generated: Jul 28, 2016 5:30:13 PM EDT
REST API
Jenkins ver. 1.652




```
BROWSER=chrome BROWSER_VERSION=50 PLATFORM='Windows 10' grunt --tag=@shallow --ci=on
```

We haven't written a Grunt task for `--ci=on` yet (listed in the command execution string above). It's responsible for outputting our test results in XML so Jenkins can consume them appropriately. Let's hop over to `Gruntfile.js` and do that now.

```
// filename: Gruntfile.js
// ...
    grunt.initConfig({
      shell: {
        runTests: {
          command: function(testFile, testOptions) {
            return 'mocha test/'+testFile+' '+testOptions+'';
          }
        },
        runTestsOnCI: {
          command: function(testFile, testOptions) {
            return 'mocha test/'+testFile+' '+testOptions+' -R xunit > test-result-'+
testFile+'.xml';
          }
        }
      },
// ...
```

We added a new `shell` execution block for running tests on CI (e.g., `runTestsOnCi`). In it we're specifying some additional command-line options for outputting XML that Jenkins can consume. Specifically `-R xunit > test-result-'+testFile+'.xml'`. The `-R` is used to specify a different reporter (e.g., `xunit` in this case). And to output the results to a file we use `>` and specify a file name. Since each test file will run in a concurrent thread we need to make sure each result file is uniquely named. And since no two files will ever have the same name, we use the test filename to make each result file unique (e.g., `test-result-LoginTest.js.xml`, etc.).

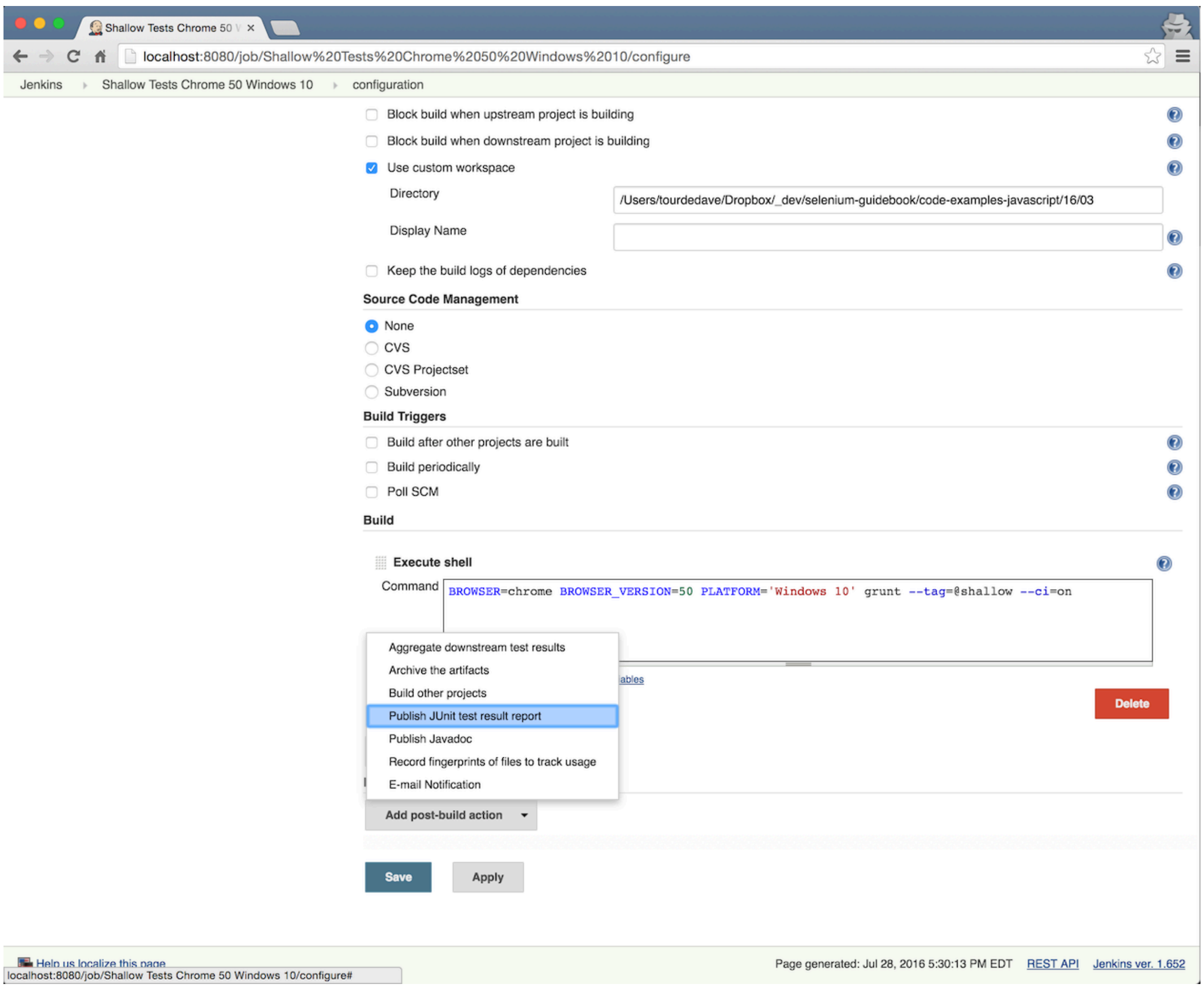
Now to add a Grunt run-time option for `--ci` and update the task registration.

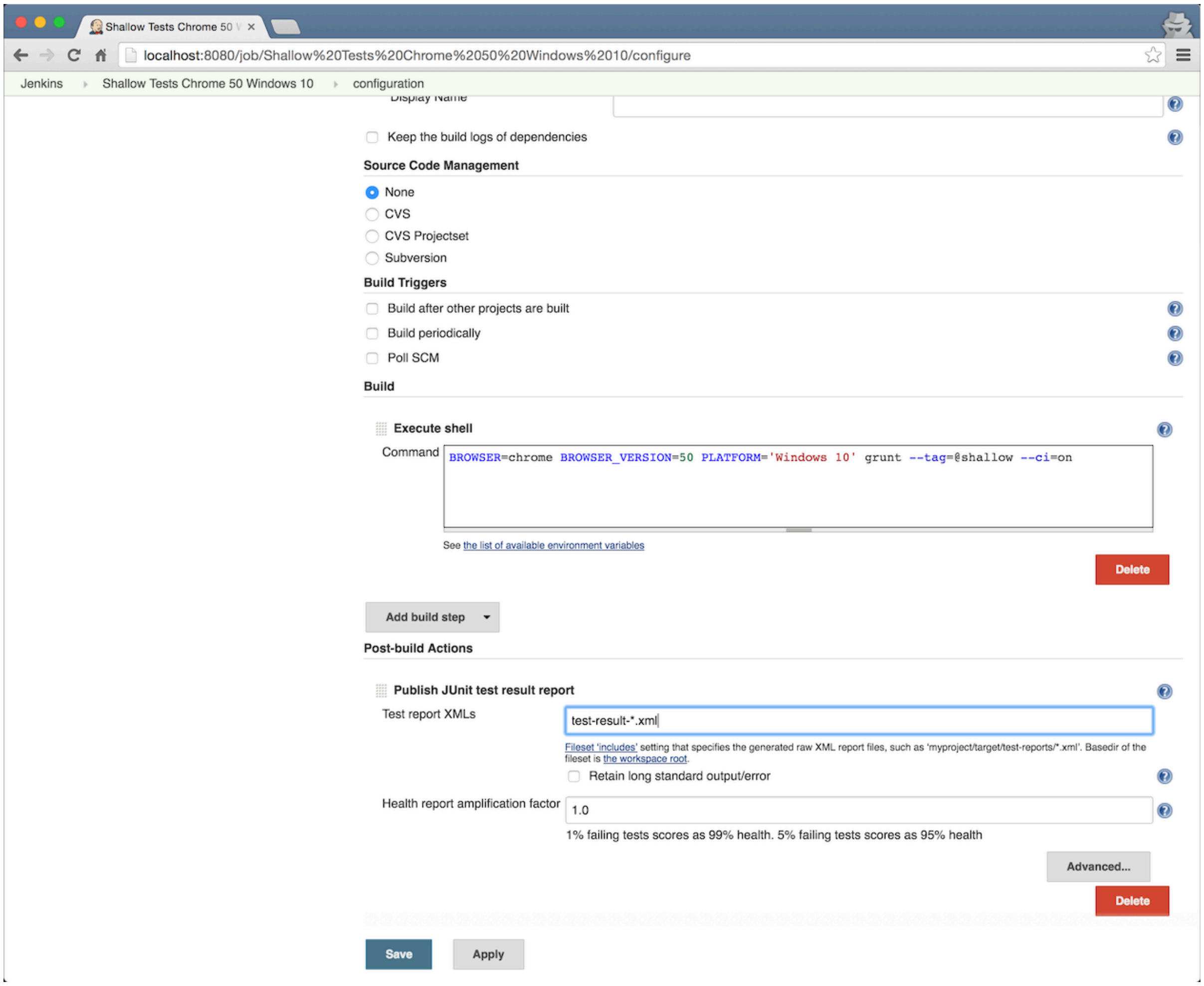
```
// filename: Gruntfile.js
// ...
    var tag = grunt.option('tag'),
        testOptions = '',
        ci = grunt.option('ci') || 'off';
// ...
    testFiles.forEach(function(testFile) {
      if (ci === 'on') {
        grunt.registerTask(testFile, ['shell:runTestsOnCI:'+testFile+':'+testOptions+''
]);
      } else {
        grunt.registerTask(testFile, ['shell:runTests:'+testFile+':'+testOptions+'']);
      }
    });
};
```

With `grunt.option('ci')` we're checking to see if a value is passed in with `--ci` and storing the result in a `ci` variable. If something is passed in then we use it. If not then we set it to `'off'`. We then use the `ci` variable in a conditional when registering our tasks. If `ci` is set to `'on'` we use the `runTestsOnCI` `shell` executor. Otherwise we use our regular `runTest` `shell` executor.

Now let's hop back over to Jenkins and configure the job to consume the test results.

- Under `Post-build Actions` select `Add post build action`
- Select `Publish JUnit test result report`
- Add the name of the result file specified in the command -- `test-result-*.xml`
- Click `Save`





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

Part 3: Force A Failure

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

```
// filename: test/LoginTest.js
// ...
  test.it('forced failure @shallow', function() {
    login.with('tomsmith', 'bad password');
    login.successMessagePresent().then(function(elementDisplayed) {
      assert.equal(elementDisplayed, true, "Success message displayed");
    });
  });
```

This test mimics our `'with invalid credentials @deep'` test by visiting the login page and providing invalid credentials. The differences here are in the assertion and the tag. It will fail since a success message won't be present after attempting to login with bogus credentials. And we want it to run as part of our `@shallow` suite.

One more thing we'll want to do is update how we're outputting the Sauce Labs job URL when there's a test failure. Right now we're outputting it to the console, but with the XML report generation this information will be hard to find in our Jenkins job. So let's make sure it shows up in the stack trace, and ultimately, the final test result report.

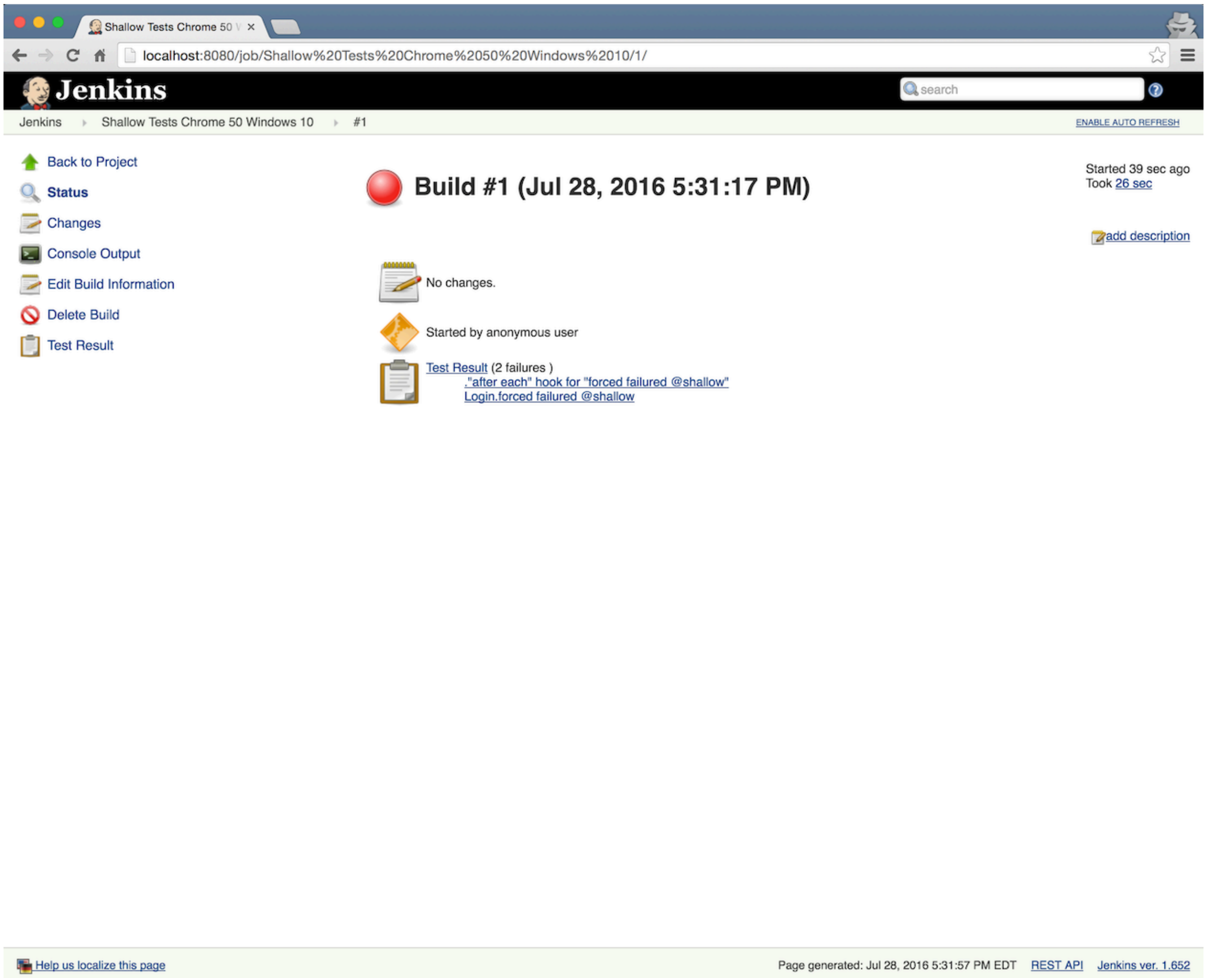
```
// filename: lib/DriverFactory.js
// ...
DriverFactory.prototype.quit = function(testName, testResult) {
  if (config.host === 'saucelabs') {
    this.driver.executeScript('sauce:job-name=' + testName);
    this.driver.executeScript('sauce:job-result=' + testResult);
  }
  this.driver.quit().then(function() {
    if (config.host === 'saucelabs' && testResult === false) {
      throw new Error('https://saucelabs.com/beta/tests/' + sessionId);
    }
  });
};

module.exports = DriverFactory;
```

In the `quit` method of our Driver Factory we make it so we throw an exception with the Sauce Labs job URL when there's a test failure. This will make sure it shows up in a relevant spot in the XML test report.

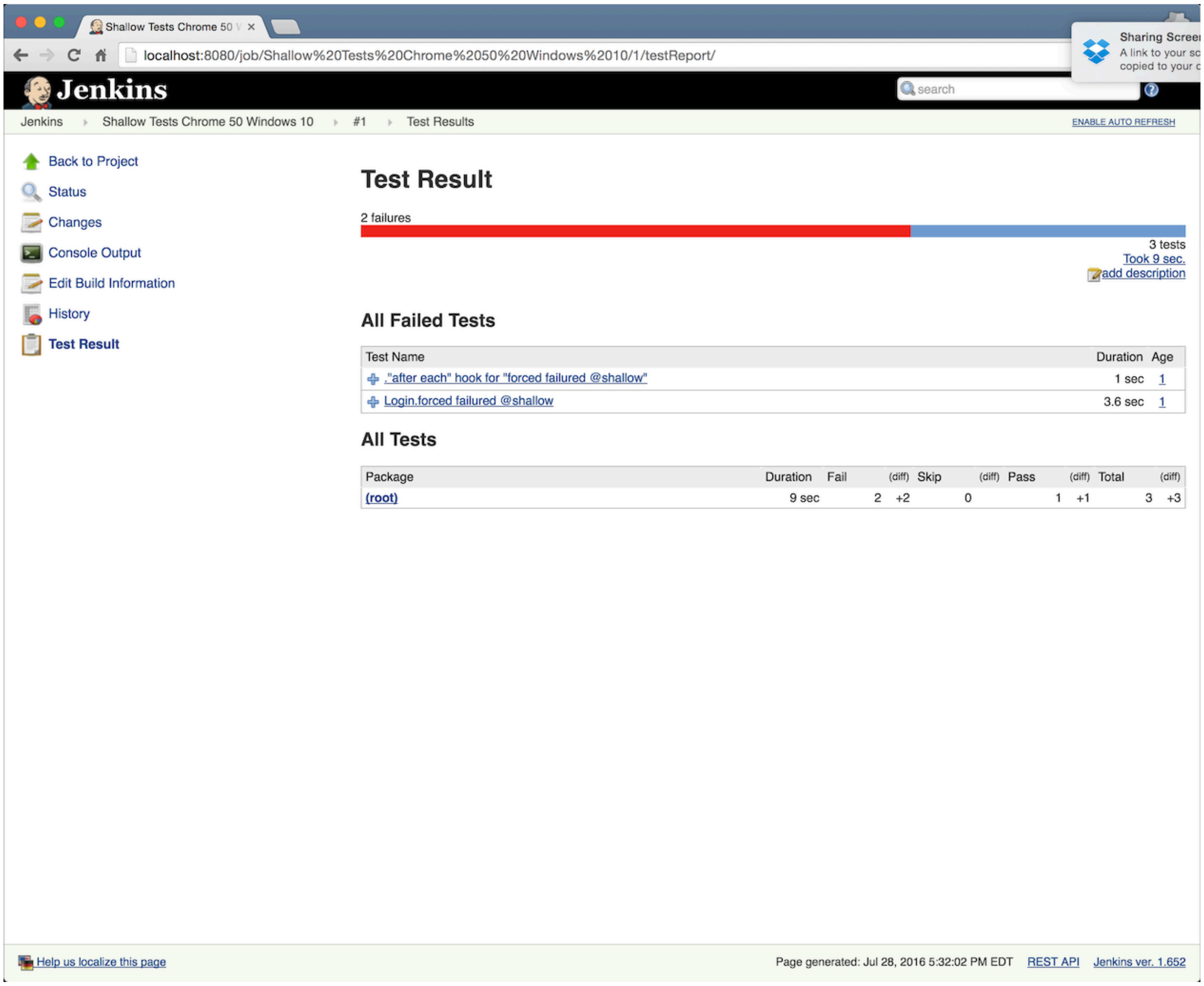
Now let's run our Jenkins job by clicking `Build Now` from the left-hand side of the screen.

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

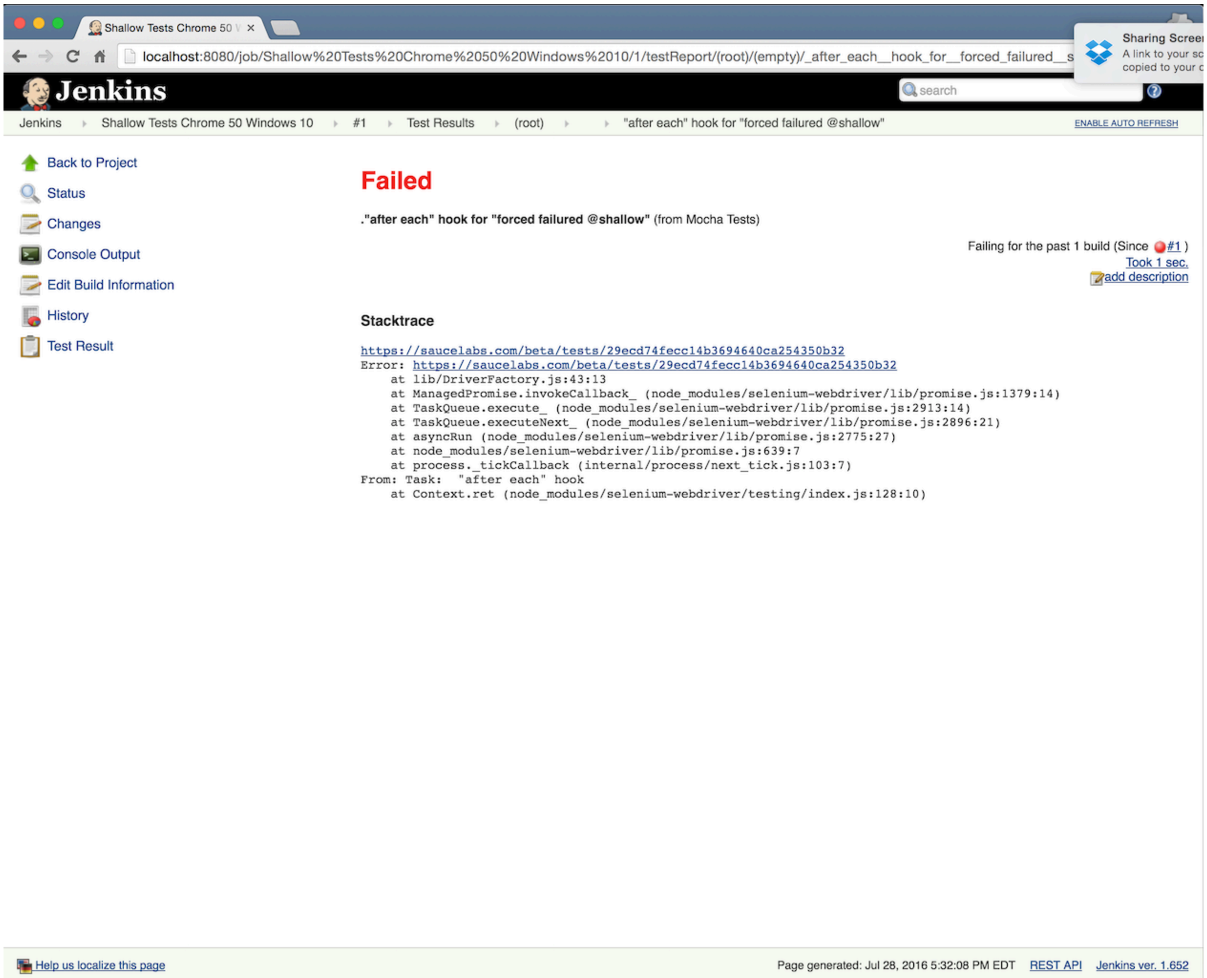
When the test completes, it will be marked as failed.



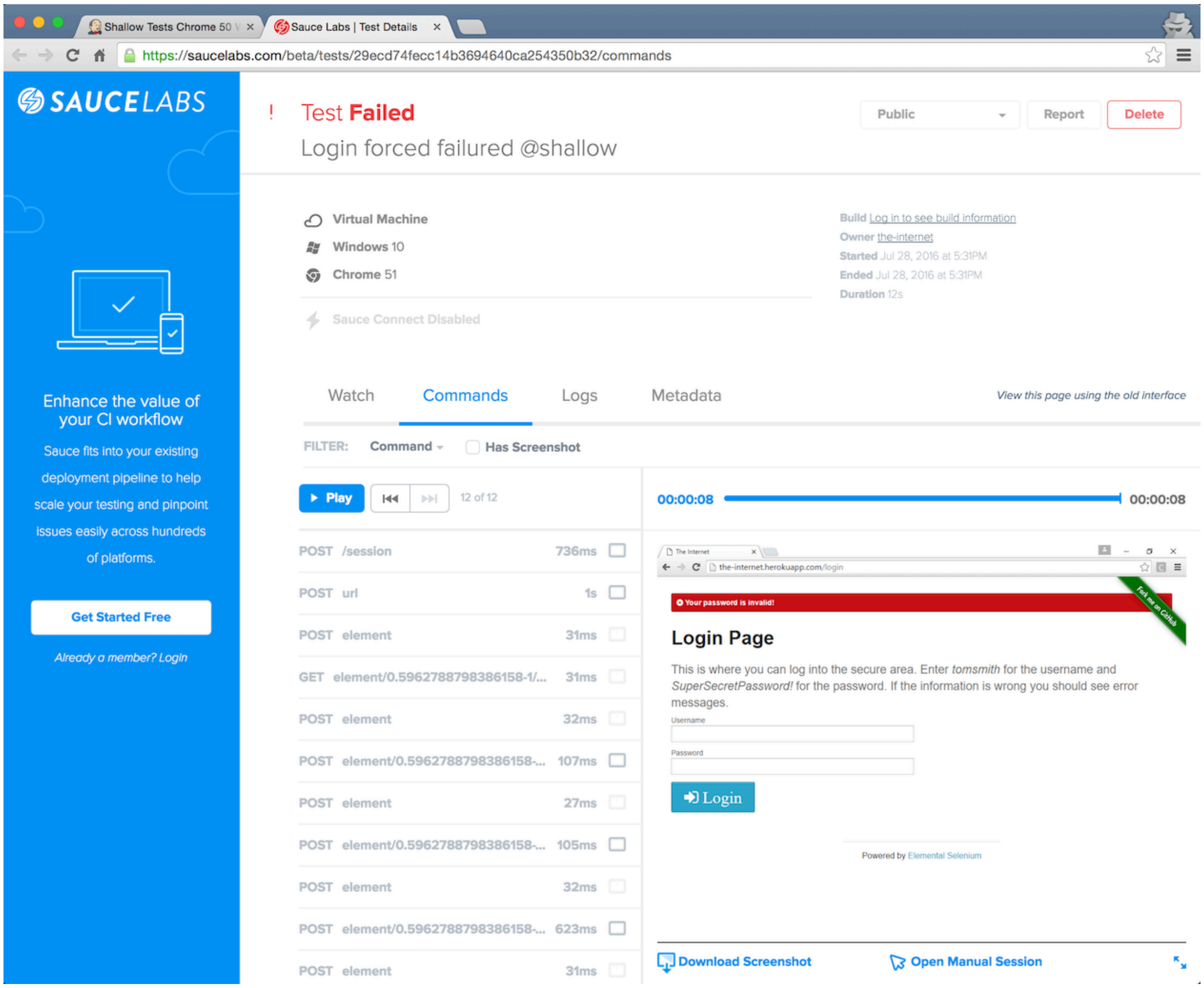
When we click on `Latest Test Result` we can see the test that failed (e.g., `Login.forced failure @shallow`). The other failure listed (e.g., `,"after each" hook for "forced failure @shallow"`) is from the teardown of our test. It contains the Sauce Labs job URL.



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



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

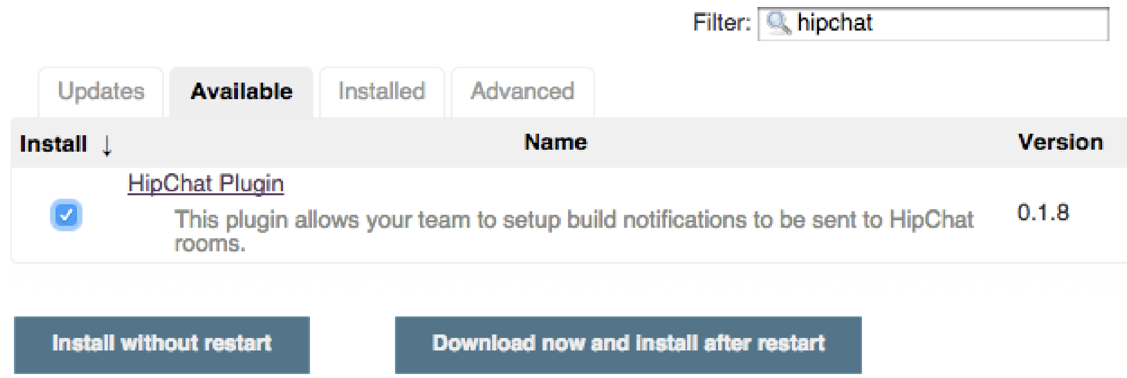


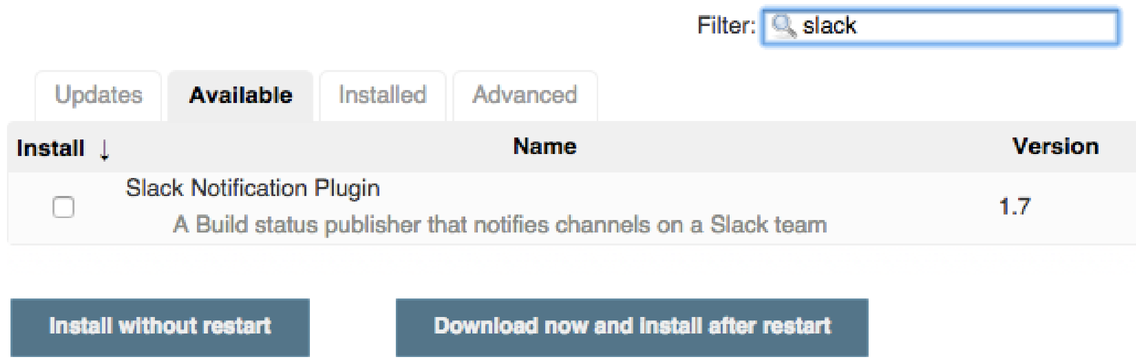
Notifications

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

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

For instance, if you wanted to use chat notifications and you use a service like HipChat or Slack, you would do a plugin search and find the following plugins:





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

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

Ideal Workflow

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

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

NOTE: You may need to incorporate a code deployment action as a preliminary step before your "Shallow" and "Deep" jobs can be run. Consult a developer on your team for help if that's the case.

Outro

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

Chapter 17
Finding Information On Your Own

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

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

Documentation & Tips

- Selenium HQ

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

- The Selenium Wiki

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

- Elemental Selenium Archives

Every tip I've written is freely available on the tips archive page. There are over 70 different Selenium problems and solutions covered.

Blogs

- The official Selenium blog

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

- A list of "all" Selenium WebDriver blogs

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

Other Books

- Selenium 2 Testing Tools

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

- Selenium Testing Tools Cookbook

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

- Selenium Design Patterns and Best Practices

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

Meetups

- The Selenium Hangout

This is an entirely online meetup that is run by me and David Burns where we talk about the latest in the Selenium community (e.g., upcoming conferences, the status of Selenium 3, the W3C spec, etc.) and answer questions from the community. Videos are recorded and made available (along with notes) on the official Selenium blog.

- All Selenium Meetups listed on Meetup.com

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

- How to start your own Selenium Meetup

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

Conferences

- Selenium Conf

This is the official annual conference of the Selenium project where practitioners and committers gather and share their latest knowledge and experiences with testing. The conference location changes every year (e.g., it's been in San Francisco, London, Boston, Bangalore, and soon -- Portland).

- Selenium Camp

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

- List of other testing conferences

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

Videos

- Selenium Conference Talks

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

- Selenium Meetup Talks

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

- Selenium Hangout

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

Mailing Lists

- Selenium Developers List

This is where developers discuss changes to the Selenium project.

- Selenium Users Google Group
- Selenium LinkedIn Users Group The signal to noise ratio in these groups can be challenging at times. But you can occasionally find some answers to your questions.

Forums

- Stack Overflow
- Quora

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

Issues

- Selenium Issue Tracker

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

Chatting With the Selenium Community on IRC

The Selenium IRC Chat Channel is arguably the best way to connect with the Selenium community and get questions answered. This is where committers and practitioners hang out day-in and day-out.

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 networks you can connect to. Each one containing people and bots logged in and joined to one or more chat rooms talking, sharing files, etc.

One of the beautiful things about IRC is that there is no registration 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 network, and pick a unique nickname for yourself on that network.

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 download one that supports it (or one that is built specifically for IRC).

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

OSX

- LimeChat
- Textual ($4.99 to buy)

Windows

- mIRC (free for 30 days, $20 for a single-user license)

Linux

- Irssi

Web-only

- Webchat
- IRCCloud (free with paid tier as well, Android and iOS apps available as well)

Step 2: Connect To The Proper Server

The Selenium chat channel lives on the Freenode network. 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 will open a new chat window to the Selenium chat channel. Woohoo!

Step 4: Talk And Hangout

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

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

Chapter 18
Now You Are Ready

The journey for doing Selenium successfully can be long and arduous. But by adhering to the principals in this book, you will avoid a majority of the pitfalls around you. You're also in a better position now -- armed with all of the information necessary to continue your Selenium journey.

You are ready. Keep going, and best of luck!

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

- E-mail: dhaeffner@gmail.com
- Twitter: @TourDeDave
- Office hours: SoHelpful.me/TourDeDave

My office hours are free (read: completely free -- no strings attached). You can grab a 30-minute time slot when they're available to hop on a Skype or Google Hangout call with me to talk about Selenium. And if we both happen to be at the same conference or meetup, please find me and say hello!

Cheers,
Dave H