How to use Selenium, successfully

The Selenium Guidebook
JavaScript Edition

by Dave Haeffner

Preface

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

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

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

What This Book Will Cover

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

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

Who This Book Is For

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

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

About The Examples In This Book

The examples in this book are written in 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 Chapter 9, Part 2 can be found in `09/02/` in the zip file).

How To Read This Book

Chapters 1 through 5 focus on the things you need to consider when it comes to test strategy, programming language selection, and good test design. Chapter 6 is where we first start to code. From there, the examples build upon each other through chapter 17.

Chapter 18 paints a picture of the Selenium landscape so you're better able to find information on your own.

Feedback

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

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

Acknowledgements

So, JavaScript...

It's a language that I've finally warmed to thanks to my work with [Selenium IDE](#).

When I wrote the last edition, I thought I had a decent handle on the language. But I realize now that there was a lot of opportunity for improvement. Partly because of the changes in the language (e.g., ES6 and the introduction of `async` / `await`) and partly because of what I've learned over the last year working full-time in the language.

As a result, there have been substantial changes to the examples in the book.

I'd like to thank my colleague [Tomer Steinfeld](#). We both work as full-time maintainers of Selenium IDE, and he's been a great teacher, sharing his expertise in the language.

Cheers,
Dave H

Table of Contents

Chapter 1
Selenium In A Nutshell

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

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

What Selenium Is and Is Not Good At

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

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

Selenium Highlights

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

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

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

Chapter 2
Defining A Test Strategy

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

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

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

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

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

What To Do With The Answers

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

Question 1: Money/Value

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

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

Question 2: Usage Data

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

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

Question 3: Browsers

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

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

Question 4: Risky Bits

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

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

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

Now You Are Ready

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

Chapter 3
Picking A Language

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

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

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

What To Do With The Answer

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

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

Some Additional Food For Thought

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

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

You can find a list of open source Selenium WebDriver frameworks and their respective languages [here](#).

Outro

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

With all that being said, the rest of this book will show you how to build your own test harness (in addition to writing well factored tests) in 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](#). There are also binary distributions avaialble for various Linux distros ([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](#).

Installing Third-Party Libraries

There are over 1 million 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](#). 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. You can also auto-save the package to a local 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 packages we will be working with in this book:

- [selenium-webdriver](#)
- [geckodriver](#)
- [chromedriver](#)
- [mocha](#)
- [mocha-parallel-tests](#)
- [@applitools/eyes-selenium](#)

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 [Atom](#), [Emacs](#), [Vim](#), and [Sublime Text](#).

There's also the option of going for an IDE (Integrated Development Environment) like [WebStorm](#) (it's not free, but has a free 30-day trial) or [Visual Studio Code](#).

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. If you end up using WebStorm be sure to check out the documentation on using it with Mocha ([link](#)).

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, Functions, Methods, and Classes)
- Scope
- Object Types (Strings, Integers, Collections, Booleans, etc.)
- Actions (Assertions and Conditionals)
- 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 `let`, or `const`
- 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`, `kebab-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).

```
> let exampleVariable = "42";
> typeof(exampleVariable)
// outputs: 'string'

> exampleVariable = 42;
> typeof(exampleVariable);
// outputs: 'number'
```

NOTE: `let` is used to declare a variable that can change, whereas `const` (short for "constant") is used to declare a variable that will not change.

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.

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

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

Functions

Functions are a way to group behavior together for easy reuse.

They can be specified and called on their own (e.g., they don't need to be part of a class -- more on classes soon).

When it comes to naming, they follow the same rules as variables. However, function names tend to be a verb (since they denote some kind of an action to be performed), and the contents of them are wrapped in opening and closing brackets (e.g., `{}`).

Functions can start either with the `function` keyword, or as an anonymous function (e.g., `() => { }`).

```
function doSomething() {
  // your code
  // goes here
};


doSomething();
```

We can specify arguments we want to pass into a function when calling it.

Here's an example.

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

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

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

We'll see functions used occasionally throughout the book. Most notably in the setup and teardown for the tests.

```
afterEach(async function() {
  const testPassed = this.currentTest.state === 'passed'
  await driverFactory.quit(testPassed)
})
```

Methods

Methods are functions that exist as part of an object (e.g., a class).

They can be specified as part of a class, don't require any special keywords, and follow the same naming conventions as functions.

Classes

Classes are a useful way to represent a concept that will be used more than once. They can contain variables and methods. They are useful when there is some underlying data (or state) that can change that you need to keep track of.

Class names:

- Starts with the `class` keyword followed by the name you want
- the name needs to start with a capital letter
- should be PascalCase for multiple words (e.g., `ExampleClass`)
- should be descriptive (e.g., a noun, whereas functions would be a verb)

You first have to define the class. Then you specify variables and methods for it. Once defined, you need to create an instance of it (a.k.a. instantiation). Once you have an instance of it you can access the methods within it to trigger the behavior in them.

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

An example of a class we'll see later in the book is storing information for how to interact with a web page -- also known as a page object. In it we'll store the locators for elements on the page we want to interact with in variables, and the behavior we intend to use on those variables in methods.

```
const LOGIN_FORM = {id: "login"};
const USERNAME_INPUT = {id: "username"};
const PASSWORD_INPUT = {id: "password"};

class Login {
  authenticate(username, password) {
// ...
}
```

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 function or a method is a classic example of this. It is useful within the area it was declared, but inaccessible outside of it.

Class Variables

Variables declared as a class property will enable you to store and retrieve them through the entire class (e.g., in all of its methods).

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 (e.g., class).

Constants

Variables that are fully capitalized and separated by underscores (e.g., `_` if more than one word) are called constants. They are variables that won't change. They are commonly used to locator variables within page objects.

Environment Variables

Environment variables are a way to pass information into our test code from outside of it at run-time. 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 when running your tests. A great example of this is which browser to run the tests against.

```
module.exports = {
  browser: process.env.BROWSER || 'firefox'
};
```

Types of Objects

Strings

Strings are alpha-numeric characters (e.g., letters, numbers, and most special characters) surrounded by either single (') or double (") 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 objects. The one we'll want to pay attention to is objects.

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

```
> const example = {tomato: 'tomahto', potato: 'potahto'}
> example.tomato
// outputs: 'tomahto'
> example.potato
// outputs: 'potahto'
```

You'll end up working with objects in page objects to store and retrieve a 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 `true` or `false` values that get returned when asking a question of your code. They are what enable us to complete assertions.

There are numerous ways to ask these questions. Some involve various [comparison operators](#) (e.g., `==`, `===`, `!=`, `<`, `>`). The response of which is `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

Assertions

With booleans we can perform assertions.

An assertion is used to check the that the actual result of something matches what you expected. This will result in either a passing or a failing test.

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.

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

Conditionals

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

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

```
const 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') {
    builder = new webdriver.Builder().forBrowser(config.browser);
   // ...
```

More on that in chapters 13 and 14.

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.

To account for this we enlist the help of a Promise.

Promises can take an asyncronous command and make our test code wait for it to complete. Thankfully, built into the Node.js there is now syntactic sugar to make working with promises very straigtforward. Through the user of the `async` and `await` keywords, we get all the benefit of promises with very little effort.

We'll see these keywords often in our Selenium test code, as they are fundamental to most of the commands we'll be using on a regular basis.

```
async authenticate(username, password) {
   await driver.findElement({id: 'username'}).sendKeys('tomssmith');
   // ...
}
```

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](#)
- [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 16).

Being Descriptive

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

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

Test Runners

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

Version Control

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

- [Git](#)
- [Mercurial](#)
- [Subversion](#)

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

Chapter 6
Writing Your First Test

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

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

1. Visit the 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. 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 might yield working test code but it will be brittle and hard to maintain.

Once you've identified the target elements and attributes you'd like to use for your test, you need to craft locators 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](#)).

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

Notice the element attributes 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`. We'll also create a `vendor` directory for third-party files and download `geckodriver` into it. Grab the latest release from here and unpack its contents into the `vendor` directory. This is a required file (known as a browser driver) in order to make Selenium work with Firefox. See Chapter 13 for more detail about browser drivers.

NOTE: As an alternative to using the `vendor` directory and manually downloading browser drivers for Firefox and Chrome, you can install them through `npm` packages (e.g., `npm install geckodriver` and `npm install chromedriver`).

When we're done our directory structure should look like this (not including the requisite `node_modules` directory).

```
package.json
test
    LoginTest.js
vendor
    geckodriver
```

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

```
// filename: test/LoginTest.js
const { Builder } = require('selenium-webdriver')
const path = require('path')

describe('Login', function() {
  this.timeout(30000)
  let driver

  beforeEach(async function() {
    const vendorDirectory =
      path.delimiter + path.join(__dirname, '..', 'vendor')
    process.env.PATH += vendorDirectory
    driver = await new Builder().forBrowser('firefox').build()
  })

  afterEach(async function() {
    await driver.quit()
  })

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

At the top of the file we import some dependencies. One is to create and control an instance of Selenium, the other is for working with file paths.

We declare a test class with `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 `beforeEach` and `afterEach` methods. This ensures that there is a clean instance of Selenium for each test (e.g., a new instance is created before a test, and destroyed after the test completes). To create an instance of Selenium we call `new Builder().forBrowser('firefox').build();` and store it in the `driver` variable. In order for Selenium to load an instance of Firefox we need to specify the path to the directory where the `geckodriver` file is. We do this by finding the path to the current working directory (e.g., `_dirname`, appending `/vendor` to it, and adding this to the execution path.

Our test method starts with `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, 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.

```javascript
// filename: test/LoginTest.js
// ...
var assert = require('assert');
// ...
  test.it('with valid credentials', function() {
    // ...
    assert(
      await driver.findElement({ css: '.flash.success' }).isDisplayed(),
      'Success message not displayed'
    )

  });
```

With `assert` we are checking for a `true` boolean response from Selenium on whether or not the element is displayed (e.g., `.isDisplayed()`). If it's not, we want to display a helpful failure message (e.g., `Success message not displayed`).

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 assertion to attempt to force a failure and run it again. Just add a `!` in front of the `driver` keyword (this will

invert the boolean -- e.g., `false` becomes `true`, `true` becomes `false`).

```
assert(
      await !driver.findElement({ css: '.flash.success' }).isDisplayed(),
      '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 CSS selector you want to look up by the `$$('')` command (e.g., `$$('#username')`) and hovering your mouse over what is returned in the console. The element that was found will be highlighted in the viewport.

An Example

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

```html
<!-- a snippet from http://the-internet.herokuapp.com/large -->

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

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

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

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

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

Fortunately with our in-browser approach to verifying our locators, we can quickly discern where the issue is. Here's what it shows us for the locators that "worked".
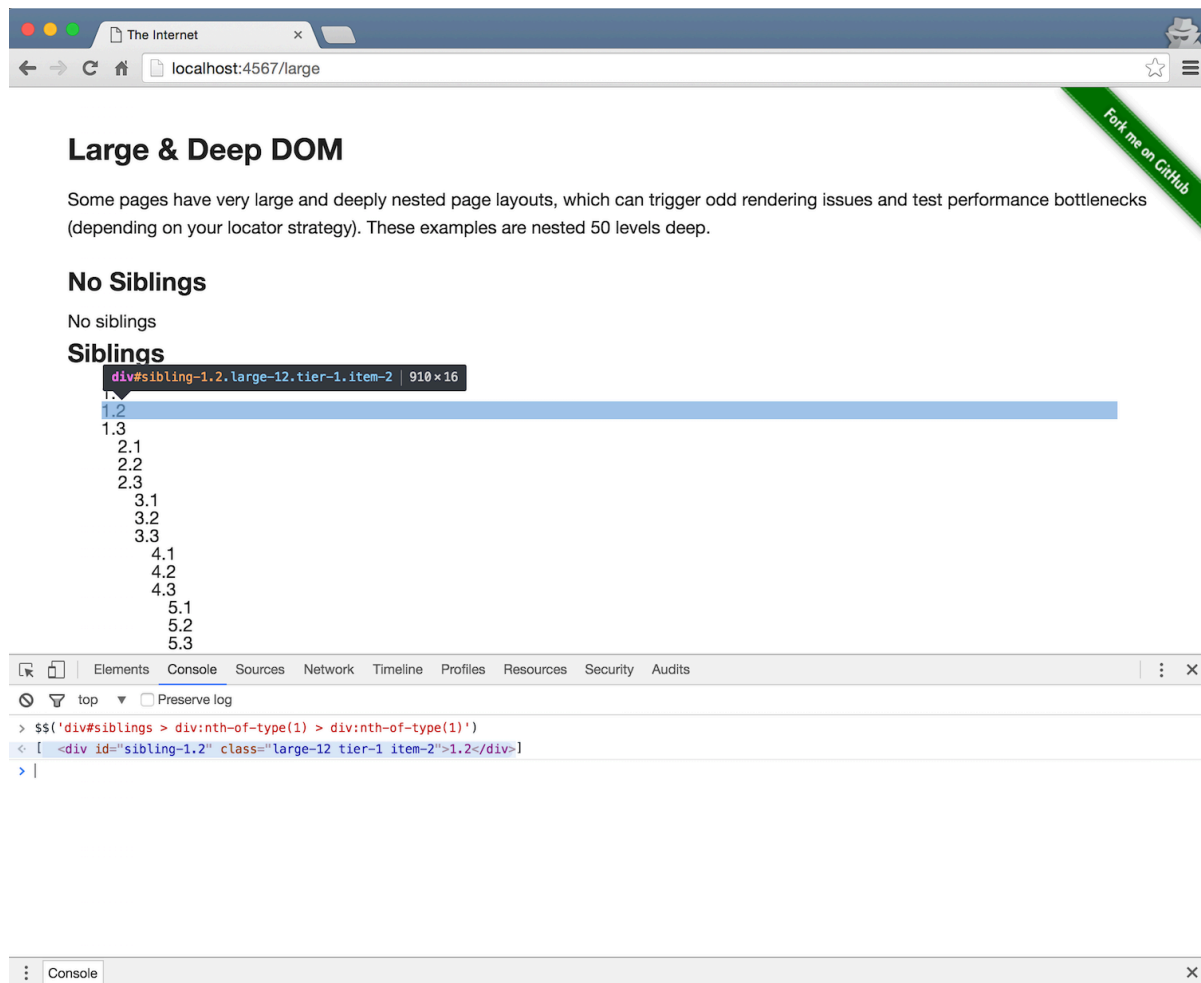
It looks like our locators are scoping to the wrong part of the first level (1.2). But we need to reference the third part of each level (e.g., 1.3, 2.3, 3.3) in order to traverse deeper since the nested divs live under the third part of each level.

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

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

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

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

Chapter 8
Leveling Up Your Assertions

As we saw in Chapter 6, you can find an element and assert something about it (e.g., is it displayed). This is a good start, but there's a lot of opportunity for improvement since assertions like this can give you a false sense of security.

Sure, the element is there, but what does it look like? The application could be working fine from a functional perspective but appear broken to the user due to visual anomalies in the application.

This is where automated visual testing comes in.

An Example

With automated visual testing we can gain the confidence that the feature we're testing appears as we expect. There are lots of solutions out there for visual testing (you can see a list here). We're going to step through using Applitools Eyes.

NOTE: You'll first need to sign up for an account. There's a free tier you can sign up for to get started. Go here for details.

Part 1: Initial Setup

There are a few things we'll need to do in our test.

First, we'll need to require the Eyes class out of the `eyes-selenium` library and declare an `eyes` variable. This is where we'll to store instances of Eyes that get created for the test.

```javascript
// filename: test/LoginTest.js
// ...
const { Eyes } = require('@applitools/eyes-selenium')

describe('Login', function() {
  this.timeout(30000)
  let driver
  let eyes
// ...
```

Next, in the test's `beforeEach` we'll wire up the code which connects the test to Applitools.

```
// filename: test/LoginTest.js
// ...
  beforeEach(async function() {
    // ...
    eyes = new Eyes()
    eyes.setApiKey(process.env.APPLITOOLS_API_KEY)
    driver = await eyes.open(
      driver,
      'the-internet',
      this.currentTest.fullTitle(),
      {
        width: 1024,
        height: 768,
      }
    )
  })
```

We create an instance of Eyes, specify our account API key (by pulling it from an environment variable), and open a session with Applitools (e.g., `eyes.open`). When opening a session we provide the instance of Selenium, some metadata (e.g., the application name and the name of the test that's running), and the viewport size we'd like the browser to use. We store the end result of `eyes.open` in the `driver` variable.

NOTE: For details on how to get your API key, go here.

Now to add a visual assertion to our test.

```
// filename: test/LoginTest.js
// ...
  it('with valid credentials', async function() {
    // ...
    await eyes.checkWindow('Logged in')
    await eyes.close()
  })
})
```

With `eyes.checkWindow` we're able to take a screenshot of the entire page our test is on, establish a baseline for the page, and use that baseline as the point of comparison for future test runs. With this command, you can either specify a descriptor (e.g., `Logged in`), or nothing.

At the end of the test we need to call `eyes.close()` to end the Eyes session. We put it at the end of the test (and not in the `afterEach`) because if there are visual differences found we want the test to fail along with a URL in the failure output pointing to the job in the Applitools Eyes dashboard for further review.

As a matter of cleanup, we'll also want to update our test's teardown as well.

```
// filename: test/LoginTest.js
// ...
  afterEach(async function() {
    await driver.quit()
    await eyes.abortIfNotClosed()
  })
// ...
```

If for some reason an Eyes session wasn't successfully closed, then we'll want to abort it after quitting the Selenium instance (e.g., `eyes.abortIfNotClosed()`).

Part 2: Targeted Runs

Not all tests might have visual assertions. If that's the case, then we don't want to start an Eyes session needlessly. To account for this, we'll want to make it so an Eyes session is opened only if a test contains Eyes commands.

```
// filename: test/LoginTest.js
// ...
  beforeEach(async function() {
    // ...
    const hasEyesCommands = this.currentTest.body.match(/eyes\./)
    if (hasEyesCommands) {
      eyes = new Eyes()
      eyes.setApiKey(process.env.APPLITOOLS_API_KEY)
      driver = await eyes.open(
        driver,
        'the-internet',
        this.currentTest.fullTitle(),
        {
          width: 1024,
          height: 768,
        }
      )
    }
  })
```

Similar to how we accessed the test's name (to pass to `eyes.open`), we can also access the `body` of the test.

This contains the test's commands, which we can search through and look to see if there are any which match `eyes.` . If there is a match, the search will return `true` . If no, then it will return `false` . We store the result in a variable called `hasEyesCommands` (for readability) and use it in a

conditional. If it's `true`, we create the Eyes instance. If it's `false`, we don't.

Outro

If you're interested in learning more about "why" and "when" to use automated visual testing, you can check out [this introductory blog post series I wrote on the topic](#).

Chapter 9
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
vendor
    geckodriver
```

Here's the code that goes with it.

```javascript
// filename: pages/LoginPage.js
const USERNAME_INPUT = { id: 'username' }
const PASSWORD_INPUT = { id: 'password' }
const SUBMIT_BUTTON = { css: 'button' }
const SUCCESS_MESSAGE = { css: '.flash.success' }

class LoginPage {
  constructor(driver) {
    this.driver = driver
  }

  async load() {
    await this.driver.get('http://the-internet.herokuapp.com/login')
  }

  async authenticate(username, password) {
    await this.driver.findElement(USERNAME_INPUT).sendKeys(username)
    await this.driver.findElement(PASSWORD_INPUT).sendKeys(password)
    await this.driver.findElement(SUBMIT_BUTTON).click()
  }

  async successMessagePresent() {
    return await this.driver.findElement(SUCCESS_MESSAGE).isDisplayed()
  }
}

module.exports = LoginPage
```

At the top of the file we specify some variables. These are for the locators we want to use on the page. We then declare the class by specifying its constructor (e.g., `concstructor(driver) {`). This block will run whenever a new instance of the 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 in the constructor and store it in the a class variable (e.g., `this.driver`). This enables the rest of the class to use it.

The second method (e.g., `load()`) is responsible for navigating to the page. Since it's asynchronous we can't have this behavior happen in the constructor.

Thie third method (e.g., `authenticate(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. Also, instead of the hard-coded locators, we updated the Selenium calls with the locator variables we specified at the top of the class.

The last method (e.g., `successMessagePresent()`) is the display check from earlier that was used in our assertion. It will return a boolean result just like before.

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
const { Builder } = require('selenium-webdriver')
const { Eyes } = require('@applitools/eyes-selenium')
const path = require('path')
const assert = require('assert')
const LoginPage = require('../pages/LoginPage')

describe('Login', function() {
  this.timeout(30000)
  let driver
  let eyes
  let login

  beforeEach(async function() {
    const vendorDirectory =
      path.delimiter + path.join(__dirname, '..', 'vendor')
    process.env.PATH += vendorDirectory
    driver = await new Builder().forBrowser('firefox').build()
    const hasEyesCommands = this.currentTest.body.match(/eyes\./)
    if (hasEyesCommands) {
      eyes = new Eyes()
      eyes.setApiKey(process.env.APPLITOOLS_API_KEY)
      driver = await eyes.open(
        driver,
        'the-internet',
        this.currentTest.fullTitle(),
        {
          width: 1024,
          height: 768,
        }
      )
    }
    login = new LoginPage(driver)
    await login.load()
  })

  afterEach(async function() {
    await driver.quit()
    if (eyes) await eyes.abortIfNotClosed()
  })

  it('with valid credentials', async function() {
```

```
    await eyes.checkWindow('Login')
    await login.authenticate('tomsmith', 'SuperSecretPassword!')
    assert(await login.successMessagePresent(), 'Success message not displayed')
    await eyes.checkWindow('Logged in')
    await eyes.close()
  })
})
```

Before we can use the page object we first need to require it (e.g., `const LoginPage = require('../pages/LoginPage');`). Then it's a simple matter of updating our test setup to create an instance of the login page (storing it in a `login` variable) and updating the test method to use the page object instead.

Now the test is more concise and readable. 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.

```
// filename: pages/LoginPage.js
// ...
var SUCCESS_MESSAGE = {css: '.flash.success'};
var FAILURE_MESSAGE = {css: '.flash.error'};
// ...
  async failureMessagePresent() {
    return await 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.

```
// filename: test/LoginTest.js
// ...
  it('with invalid credentials', async function() {
    await login.authenticate('tomsmith', 'SuperSecretPassword!')
    assert(await login.failureMessagePresent(), '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 Checking For The Absence Of An Element 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.

```
assert(await !login.successMessagePresent(), 'Success message displayed')
```

There are two problems with this approach. First, our test will fail 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 accurate.

```
// filename: test/LoginTest.js
// ...
  it('with invalid credentials', async function() {
    await login.authenticate('tomsmith', 'bad password')
    assert(await login.failureMessagePresent(), 'Failure message not displayed')
    await eyes.checkWindow('Incomplete Login')
    await eyes.close()
  })
})
```

But for the best results, we also add a visual check (e.g., `eyes.checkWindow` with a descriptive name, and calling `eyes.close` at the end of the test).

Part 3: Confirm We're In The Right Place

Before we can call our page object complete, there's one more addition we should make. We'll want to add a check to make sure that Selenium is in the right place before proceeding, which will in turn, add some resiliency to our tests.

```
// filename: pages/LoginPage.js
// ...
  async load() {
    await this.driver.get('http://the-internet.herokuapp.com/login')
    if (await !this.driver.findElement(LOGIN_FORM).isDisplayed())
      throw new Error('Login form not loaded')
  }
// ...
```

In our `load` method of the login page we want to check to see that the login form is displayed after navigating there. If not, we will throw an exception with the message `'Login form not loaded'`, which will error the test before it attempts to log in (and provide this message in the output). Otherwise, the test will proceed without issue.

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 10
Writing Really Re-usable Test Code

In the previous chapter we stepped through creating a simple page object to capture the behavior of the page we were interacting with. While this was a good start, there's more we can do.

As our test suite grows and we add more page objects we will start to see common behavior that we will want to use over and over again throughout our suite. If we leave this unchecked we will end up with duplicative code which will slowly make our page objects harder to maintain.

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

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

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

An Example

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

Part 1: Create The Facade Layer

First let's add a new file called `BasePage.js` in `pages` directory.

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

Next let's populate the file.

```javascript
// filename: pages/BasePage.js
class BasePage {
  constructor(driver) {
    this.driver = driver
  }

  async visit(url) {
    await this.driver.get(url)
  }

  find(locator) {
    return this.driver.findElement(locator)
  }

  async click(locator) {
    await this.find(locator).click()
  }

  async type(locator, inputText) {
    await this.find(locator).sendKeys(inputText)
  }

  async isDisplayed(locator) {
    return await find(locator).isDisplayed()
  }
}

module.exports = BasePage
```

In this module we declare a `BasePage` class along with methods for all of the common behavior we use with Selenium (e.g., `visit`, `find`, `click`, `type`, and `isDisplayed`). We also have a constructor that enables us to pass in and store an instance of the driver, so we don't have to explicitly pass it to the methods whenever we call them.

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

```javascript
// filename: pages/LoginPage.js
const BasePage = require('./BasePage')

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

class LoginPage extends BasePage {
  constructor(driver) {
    super(driver)
  }

  async load() {
    await this.visit('http://the-internet.herokuapp.com/login')
    if (await !this.isDisplayed(LOGIN_FORM, 1000))
      throw new Error('Login form not loaded')
  }

  async authenticate(username, password) {
    await this.type(USERNAME_INPUT, username)
    await this.type(PASSWORD_INPUT, password)
    await this.click(SUBMIT_BUTTON)
  }

  successMessagePresent() {
    return this.isDisplayed(SUCCESS_MESSAGE, 1000)
  }

  failureMessagePresent() {
    return this.isDisplayed(FAILURE_MESSAGE, 1000)
  }
}

module.exports = LoginPage
```

A few things have changed in our Login page object. We've imported the base page class we want to use, established inheritance between the two classes, and we've swapped out all of our Selenium commands with calls to the methods in the base page object (e.g., `this.visit`, `this.type`, `this.click`, etc.).

To establish inheritance we used the `extends` keyword when declaring the class (e.g., `class LoginPage extends Page {`) and called `super` from the constructor (e.g., `super(driver)`). This passes the instance of Selenium to the base page object, and makes all of the base page object's

methods available to our login page object (though `this.`).

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.

```javascript
// filename: pages/BasePage.js
// ...
async isDisplayed(locator) {
  try {
    return await this.find(locator).isDisplayed()
  } catch (error) {
    return false
  }
}
// ...
```

Thanks to `async` / `await` we can reliabily use a `try` / `catch` block to account for the `NoSuchElementError` and return false instead.

Now let's revisit our `'with invalid credentials'` login test and alter it so it checks to see if the success message is not present to make sure things work as we expect.

```javascript
// filename: test/LoginTest.js
//  ...
  it('with invalid credentials', async function() {
    await login.authenticate('tomsmith', 'bad password')
    assert(await !login.successMessagePresent(), 'Success message displayed')
    // ...
  })
})
```

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

this time. Feel free to keep the test as-is, or change it back to what it was before.

Chapter 11
Writing Resilient Test Code

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

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

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

An Explicit Waits Primer

Explicit waits are applied to individual commands in a test. 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.

```
<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
vendor
     geckodriver
```

Next we'll specify the locators and behavior we'll want to use on this page.

```javascript
// filename: pages/DynamicLoadingPage.js
const BasePage = require('./BasePage')

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

class DynamicLoadingPage extends BasePage {
  constructor(driver) {
    super(driver)
  }

  async loadExample(exampleNumber) {
    await this.visit(
      'http://the-internet.herokuapp.com/dynamic_loading/' + exampleNumber
    )
    await this.click(START_BUTTON)
  }

  async isFinishTextPresent() {
    return this.isDisplayed(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.

Similar to our Login page object, we have a display check for the finish text (e.g., `finishTextPresent`). This check is slightly different though. Notice that it has a second argument (an integer value of `10000`). This is a bit of aspirational code that we'll need to write. It's how we'll tell Selenium to wait (in milliseconds) for an element to be displayed before giving up.

Let's update the `isDisplayed` method our base page object to offer this behavior.

```
// filename: pages/BasePage.js
const Until = require('selenium-webdriver').until
// ...
  async isDisplayed(locator, timeout) {
    if (timeout) {
      await this.driver.wait(Until.elementLocated(locator), timeout)
      await this.driver.wait(
        Until.elementIsVisible(this.find(locator)),
        timeout
      )
      return true
    } else {
      try {
        return await this.find(locator).isDisplayed()
      } catch (error) {
        return false
      }
    }
  }
}
// ...
```

Selenium comes with an `until` module which is a collection of functions that can be used with Selenium's `wait` function. We require and store it in a variable. We then update the `isDisplayed` function to take an additional argument and use the `Until` variable we just created.

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 wait for the element to be displayed (e.g., `(Until.elementIsVisible(find(locator))`). If it's successful, then we return `true`, and if it's unsuccessful an error will be raised (which will fail the test).

NOTE: We have to wait for an element to be present before doing a display lookup because the display lookup requires the use of a found element (not a locator).

In the `else` block of `isDisplayed` we account for the case where no timeout is provided when calling this function. When that happens the original behavior will be used (e.g., see if an element is displayed without waiting and return false if it's not present).

There are other conditions you can wait for besides `elementLocated` or `elementIsVisible`. You can find a full 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 could run into issues later on due to inconsistent implementations of the implicit wait functionality across local and remote browser drivers. Long story short, you could 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., `async function isDisplayed(locator, timeout = 5000) {`) 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 timeout 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 a long time to fail if there's an issue. 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
vendor
    geckodriver
```

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

```javascript
// filename: test/DynamicLoadingTest.js
const { Builder } = require('selenium-webdriver')
const path = require('path')
const assert = require('assert')
const DynamicLoadingPage = require('../pages/DynamicLoadingPage')

describe('Dynamic Loading', function() {
  this.timeout(30000)
  let dynamicLoading

  beforeEach(async function() {
    const vendorDirectory =
      path.delimiter + path.join(__dirname, '..', 'vendor')
    process.env.PATH += vendorDirectory
    driver = await new Builder().forBrowser('firefox').build()
    dynamicLoading = new DynamicLoadingPage(driver)
  })

  afterEach(async function() {
    await driver.quit()
  })

  it('hidden element', async function() {
    await dynamicLoading.loadExample('1')
    assert(
      await dynamicLoading.isFinishTextPresent(),
      true,
      'Finish text not displayed'
    )
  })
})
```

In `beforeEach` we've omitted the Eyes configuration, since we won't be using visual validations in these tests.

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: 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 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 until the progress bar finished).

Here's the markup for it.

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


  <br>


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


  <br>
</div>
```

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

```html
<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
// ...
  it('rendered element', async function() {
    await dynamicLoading.loadExample('2')
    assert(
      await dynamicLoading.isFinishTextPresent(),
      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 one browser and assume you're all set. But once you start to run things against other browsers you may be in for a surprise. The first thing you're likely to run into is the speed of execution. A lot of your tests may start to fail when you point them at either Chrome or Internet Explorer, and likely for different reasons.

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

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

Closing Thoughts

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

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

Chapter 12
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 three things. A class that will contain the creation and destruction of our Selenium instances (a.k.a. a Driver Factory), a helper that all tests will pull from, and an option file Mocha uses to store commonly used command-line arguments.

In the `lib` directory we'll create a new file called `DriverFactory.js`, and in the `test` directory we'll create files called `spec_helper.js` and `mocha.opts`.

```
lib
    DriverFactory.js
package.json
pages
    BasePage.js
    DynamicLoadingPage.js
    LoginPage.js
test
    DynamicLoadingTest.js
    LoginTest.js
    mocha.opts
    spec_helper.js
vendor
    geckodriver
```

Here are the initial contents of the Driver Factory.

```
// filename: lib/DriverFactory.js
const path = require('path')
const { Builder } = require('selenium-webdriver')
const { Eyes } = require('@applitools/eyes-selenium')

class DriverFactory {
  async _openEyes() {
    this.eyes = new Eyes()
    this.eyes.setApiKey(process.env.APPLITOOLS_API_KEY)
    return await this.eyes.open(this.driver, 'the-internet', this.testName, {
      width: 1024,
      height: 768,
    })
  }

  async build(testName, hasEyesCommands = false) {
    this.testName = testName
    process.env.PATH += path.delimiter + path.join(__dirname, '..', 'vendor')
    this.driver = await new Builder().forBrowser('firefox').build()
    if (hasEyesCommands) this.driver = await this._openEyes()
    return this.driver
  }

  async quit() {
    await this.driver.quit()
    if (this.eyes) await this.eyes.abortIfNotClosed()
  }
}

module.exports = DriverFactory
```

After requiring our requisite libraries, we declare a class along with three methods -- `_openEyes`, `build`, and `quit`.

NOTE: In JavaScript, an underscore prefix in a variable or method name is used to denote that something is meant to be private (e.g., **_openEyes**). It's not meant to be used outside of the class.

`build` is responsible for creating an instance of Selenium. `_openEyes` is responsible for creating an instance of Eyes for visual testing. `quit` is responsible for destroying the Selenium instance and terminating the Eye session if it was started and not properly closed.

The class ends with `module.exports`, just like in previous classes we've created.

Now to update our `mocha.opts` file. It's a small change that will help us clean up the hard-coded timeout that we've needed to specify in each test.

```
// filename: test/mocha.opts
-t 60000
```

Now let's put everything to use in our spec helper.

```javascript
// filename: test/spec_helper.js
const DriverFactory = require('../lib/DriverFactory')
const driverFactory = new DriverFactory()

beforeEach(async function() {
  const testName = this.currentTest.fullTitle()
  const hasEyesCommands = this.currentTest.body.match(/this.eyes/)
  this.driver = await driverFactory.build(testName, hasEyesCommands)
  this.eyes = driverFactory.eyes
})

afterEach(async function() {
  await 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.

At the top of the spec helper we require the Driver Factory and create a new instance of it, storing it in a variable.

In `beforeEach` we grab the name of the currently running test, check whether or not there are any visual testing commands using Applitools Eyes, and then we create a driver instance and store it in a variable on `this` (which will make it accessible to the test). We also store an instance of Eyes on `this`.

In `afterEach` we call the `quit` method in the Driver Factory to destroy the Selenium instance and abort any lingering Eyes sessions (if they exist).

Now to update our tests.

```javascript
// filename: test/LoginTest.js
require('./spec_helper')
const assert = require('assert')
const LoginPage = require('../pages/LoginPage')

describe('Login', function() {
  let login

  beforeEach(async function() {
    login = new LoginPage(this.driver)
    await login.load()
  })

  it('with valid credentials', async function() {
    await login.authenticate('tomsmith', 'SuperSecretPassword!')
    assert(await login.successMessagePresent(), 'Success message not displayed')
    await this.eyes.checkWindow('Logged in')
    await this.eyes.close()
  })

  it('with invalid credentials', async function() {
    await login.authenticate('tomsmith', 'bad password')
    assert(await login.failureMessagePresent(), 'Failure message not displayed')
    await this.eyes.checkWindow('Incomplete Login')
    await this.eyes.close()
  })
})
```

```javascript
// filename: test/DynamicLoadingTest.js
require('./spec_helper')
const assert = require('assert')
const DynamicLoadingPage = require('../pages/DynamicLoadingPage')

describe('Dynamic Loading', function() {
  let dynamicLoading

  beforeEach(async function() {
    dynamicLoading = new DynamicLoadingPage(this.driver)
  })

  it('hidden element', async function() {
    await dynamicLoading.loadExample('1')
    assert(
      await dynamicLoading.isFinishTextPresent(),
      true,
      'Finish text not displayed'
    )
  })

  it('rendered element', async function() {
    await dynamicLoading.loadExample('2')
    assert(
      await dynamicLoading.isFinishTextPresent(),
      true,
      'Finish text not displayed'
    )
  })
})
```

In order to use the spec helper we just need to require it.

Then we're able to remove the creation and storing of a driver instance in `beforeEach` and leverage the driver variable stored on `this` 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., localhost, 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
    config.js
    DriverFactory.js
package.json
pages
    BasePage.js
    DynamicLoadingPage.js
    LoginPage.js
test
    DynamicLoadingTest.js
    LoginTest.js
    mocha.opts
    spec_helper.js
vendor
    geckodriver
```

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.

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

Now let's update the visit method in the base page object to use this config object.

```javascript
// filename: pages/BasePage.js
const config = require('./config')
// ...
async function visit(url) {
  if (url.startsWith('http')) {
    await this.driver.get(url)
  } else {
    await 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 visit it. If not, `config.baseUrl` is combined with the URL path passed in as an argument to create the 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.

```
// filename: pages/LoginPage.js
// ...
  async load() {
    await this.visit('/login')
    if (await !this.isDisplayed(LOGIN_FORM, 1000))
      throw new Error('Login form not loaded')
  }
// ...
```

```
// filename: pages/DynamicLoadingPage.js
// ...
  async loadExample(exampleNumber) {
    await this.visit('/dynamic_loading/' + exampleNumber)
    await this.click(START_BUTTON)
  }
// ...
```

Step 3: Viewport Size

When running tests with visual validations we'll eventually want to specify different viewport sizes. Right now this value is hard-coded in our setup code. We can move it to our config file, and make it so the sensible defaults can be overridden at run-time if needbe.

```
// filename: lib/config.js
module.exports = {
  baseUrl: process.env.BASE_URL || 'http://the-internet.herokuapp.com',
  viewportSize: { width: 1024, height: 768 } || {
    width: process.env.WIDTH,
    height: process.env.HEIGHT,
  },
}
```

Now to update our DriverFactory to receive and use the config.

```
// filename: lib/DriverFactory.js
// ...
class DriverFactory {
  constructor(config) {
    this.config = config
  }

  async _openEyes() {
    this.eyes = new Eyes()
    this.eyes.setApiKey(process.env.APPLITOOLS_API_KEY)
    return await this.eyes.open(
      this.driver,
      'the-internet',
      this.testName,
      this.config.applitools.viewportSize
    )
  }
// ...
```

```
// filename: test/spec_helper.js
const DriverFactory = require('../lib/DriverFactory')
const config = require('../lib/config')
const driverFactory = new DriverFactory(config)
// ...
```

We made it so the DriverFactory can take a config object as part of instantiation and store it in a class variable. We then updated our spec helper to require the config file and pass it to the DriverFactory.

Outro

Now when running our tests, we can specify a different base URL or viewport size by providing some extra information at run-time (e.g., `BASE_URL=url HEIGHT=1234 WIDTH=567 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 13
Running A Different Browser Locally

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

NOTE: An alternative to downloading browser drivers manually is to use the compendium `npm` packages that fetch the latest version for you and manage updating the path. There is one for each browser driver (e.g., `npm intall geckdriver`, `npm install chromedriver`, etc.). But if you need a specific browser driver version, then downloading them manually is the way to go.

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 in our `vendor` directory.

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
    config.js
    DriverFactory.js
package.json
pages
    BasePage.js
    DynamicLoadingPage.js
    LoginPage.js
test
    DynamicLoadingTest.js
    LoginTest.js
    mocha.opts
    spec_helper.js
vendor
    chromedriver
    geckodriver
```

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 DriverFactory. Let's go with the latter option (which we're already using for `geckodriver`).

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 existence 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',
  viewportSize: { width: 1024, height: 768 } || {
    width: process.env.WIDTH,
    height: process.env.HEIGHT,
  },
}
```

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

```
// filename: lib/DriverFactory.js
// ...
class DriverFactory {
// ...
  async build(testName, hasEyesCommands = false) {
    this.testName = testName
    process.env.PATH += path.delimiter + path.join(__dirname, '..', 'vendor')
    this.driver = await new Builder().forBrowser(this.config.browser).build()
    if (hasEyesCommands) this.driver = await this._openEyes()
    return this.driver
  }
// ...
}
```

By leveraging the fact that we have the config settings on `this.config`, we used it to update our browser incantation to grab the browser value (e.g., `.forBrowser(this.config.browser)`).

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

Additional Browsers

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

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

Chapter 14
Running Browsers In The Cloud

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

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

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

A Selenium Grid Primer

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

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

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

Let's dig in with an example.

An Example

Part 1: Initial Setup

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

With Sauce Labs we need to provide 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.

```
// filename: lib/config.js
module.exports = {
  baseUrl: process.env.BASE_URL || 'http://the-internet.herokuapp.com',
  browser: process.env.BROWSER || 'firefox',
  viewportSize: { width: 1024, height: 768 } || {
    width: process.env.WIDTH,
    height: process.env.HEIGHT,
  },
  host: process.env.HOST || "localhost",
  sauce: {
    username: process.env.SAUCE_USERNAME,
    accessKey: process.env.SAUCE_ACCESS_KEY,
    platform: process.env.PLATFORM || "Windows 7",
    browserName: process.env.BROWSER || 'internet explorer',
    version: process.env.BROWSER_VERSION || "11.0"
  },
}
```

In addition to the `baseUrl`, `browser`, and `viewportSize` variables, we've added some more (e.g., `host`, `username`, `accessKey`, `platform`, `browserName`, and `version`).

`host` enables us to specify whether our tests run locally or on Sauce Labs. The others are stored under a key `sauce` key to make their use explicit.

With the combination of `platform`, `browserName`, and `version` 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 our Driver Factory to work with these new values and connect to Sauce Labs.

```javascript
// filename: lib/DriverFactory.js
// ...
class DriverFactory {
  constructor(config) {
    this.config = config
  }

  _configure() {
    let builder = new Builder()
    switch (this.config.host) {
      case 'saucelabs':
        const url = 'http://ondemand.saucelabs.com:80/wd/hub'
        builder.usingServer(url)
        builder.withCapabilities(this.config.sauce)
        break
      case 'localhost':
        process.env.PATH +=
          path.delimiter + path.join(__dirname, '..', 'vendor')
        builder.forBrowser(this.config.browser)
        break
    }
    return builder
  }
// ...
  async build(testName, hasEyesCommands = false) {
    this.testName = testName
    process.env.PATH += path.delimiter + path.join(__dirname, '..', 'vendor')
    this.driver = await this._configure().build()
    if (hasEyesCommands) this.driver = await this._openEyes()
    return this.driver
  }
}

module.exports = DriverFactory
```

We create a method to configure the builder object for Selenium (e.g., `_configure()`), wrapping everything in a conditional check against `config.host`. If it's set to `'saucelabs'` then we specify the `url` for their on-demand end-point and pass in the capabilities that we want (e.g., everything specified under `sauce` in config.js). If `config.host` is set to `'localhost'` then we handle browser execution just like before (adding the path to the vendor directory to the execution path and launching a browser locally).

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](#).

NOTE: There are some changes to DesiredCapaibilties in Selenium 4. Instead of `browser`, `platform`, and `version`, you will need to specify these values as `browserName`, `browserVersion`, and `platformName`. Also, vendor specific options will need to be specified in their own key (e.g., `vendor:options`). For Sauce, this would be `sauce:options`. Details for this new configuration are documented [here](#). These changes are not required yet since Selenium is still 4 alpha and Sauce Labs has only rolled out this functionality as a beta. When they become the new normal on Sauce Labs, it will just be a small matter of updating your config file.

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
// ...
  async quit() {
    if (this.config.host === 'saucelabs') {
      this.driver.executeScript('sauce:job-name=' + this.testName)
    }
    await this.driver.quit()
    if (this.eyes) await this.eyes.abortIfNotClosed()
  }
}

module.exports = DriverFactory
```

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`) was stored earlier as an argument to `build` (e.g., from the `beforeEach`) which we stored in a class variable (e.g., `this.testName`).

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. With a couple of tweaks we can make this happen easily enough.

First we need to update our `build` method in the Driver Factory to grab the session ID from Selenium.

```javascript
// filename: lib/DriverFactory.js
// ...
  async build(testName, hasEyesCommands = false) {
    // ...
    this.driver = await this._configure().build()
    const { id_ } = await this.driver.getSession()
    this.sessionId = id_
    // ...
  }
```

Next, we need to update the `quit` method in the Driver Factory.

```javascript
// filename: lib/DriverFactory.js
// ...
  async quit(testPassed) {
    if (this.config.host === 'saucelabs') {
      this.driver.executeScript('sauce:job-name=' + this.testName)
      this.driver.executeScript('sauce:job-result=' + testPassed)
      if (!testPassed)
        console.log(
            'See a video of the run at https://saucelabs.com/tests/' +
              this.sessionId
        )
    }
    await this.driver.quit()
    if (this.eyes) await this.eyes.abortIfNotClosed()
  }
}

module.exports = DriverFactory
```

With the JavaScript executor we're able to pass in the test result just like the name, which we're getting as a 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 spec helper.

```
// filename: test/spec_helper.js
afterEach(async function() {
  const testPassed = this.currentTest.state === 'passed'
  await driverFactory.quit(testPassed)
})
```

We grab the state of the current test (e.g., `this.currentTest.state`), check to see if it passed (e.g., `=== 'passed'`), 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 Proxy](#) that creates a secure tunnel between your machine and their private cloud. With it you can run tests in Sauce Labs and test applications that are only available on your private network.

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

```
  lib
      DriverFactory.js
      config.js
  package.json
  pages
      BasePage.js
      DynamicLoadingPage.js
      LoginPage.js
  test
      DynamicLoadingTest.js
      LoginTest.js
      mocha.opts
      spec_helper.js
  vendor
       chromedriver
       geckodriver
       sc
```

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

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

Now that the tunnel is established, we could run our tests against a local instance of our application (e.g., the-internet). Assuming the application was set up and running on our local machine, we run our tests against it by specifying a different base URL at runtime (e.g., `BASE_URL=http://localhost:4567 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 15
Speeding Up Your Test Runs

We've made huge strides by leveraging page objects, a base page object, explicit waits, visual testing, 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.

Thankfully, this is simple to accomplish with `mocha-parallel-tests` .

Setup

First we need to install mocha-parallel-tests. So let's update our `package.json` file and use `npm install` to install it.

```
// filename: package.json
{
  "name": "selenium-guidebook-examples",
  "dependencies": {
    "@applitools/eyes-selenium": "^4.16.0",
    "mocha": "^6.1.4",
    "mocha-parallel-tests": "^2.1.0",
    "selenium-webdriver": "^4.0.0-alpha.1"
  }
}
```

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

Once installed you can run mocha tests in parallel through the binary executor provided by mocha-parallel-tests.

```
mocha-parallel-tests test/*Test.js
```

The concurrency limit for parallel execution can be controlled with the `--max-parallel` command-line flag. By default it's set to the number of CPU cores on your computer.

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. If you send more jobs than your concurrency limit, Sauce Labs will queue the excess and run them as the initial batch of jobs 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 can be 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 16
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 through how to set this up.

Specifying Tags

Grep is a simple text match function. It will look through the test files and execute them if it found a match for a given string pattern. To make this work we'll want to add some metadata to our test names with with a special character to make it stand out as metadata so it's easier to search for.

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
// ...
  it('with valid credentials @shallow', async function() {
    await login.authenticate('tomsmith', 'SuperSecretPassword!')
    assert(await login.successMessagePresent(), 'Success message not displayed')
    await this.eyes.checkWindow('Logged in')
    await this.eyes.close()
  })

  it('with invalid credentials @deep', async function() {
    await login.authenticate('tomsmith', 'bad password')
    assert(await login.failureMessagePresent(), 'Failure message not displayed')
    await this.eyes.checkWindow('Incomplete Login')
    await this.eyes.close()
  })
// ...
})
```

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` .

```javascript
// filename: test/DynamicLoadingTest.js
// ...
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 both Mocha and mocha-parallel-tests we can specify which marker to launch at runtime. This is handled as another runtime flag on the command-line using `--grep` .

```
mocha --grep @shallow
mocha-parallel-tests --grep @shallow
```

Chapter 17
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](#)
- [CircleCI](#)
- [Jenkins](#)
- [TravisCI](#)

Let's pick one and step through an example.

A CI Example

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

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

Part 1: Quick Setup

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

Once downloaded, launch it from the command-line.

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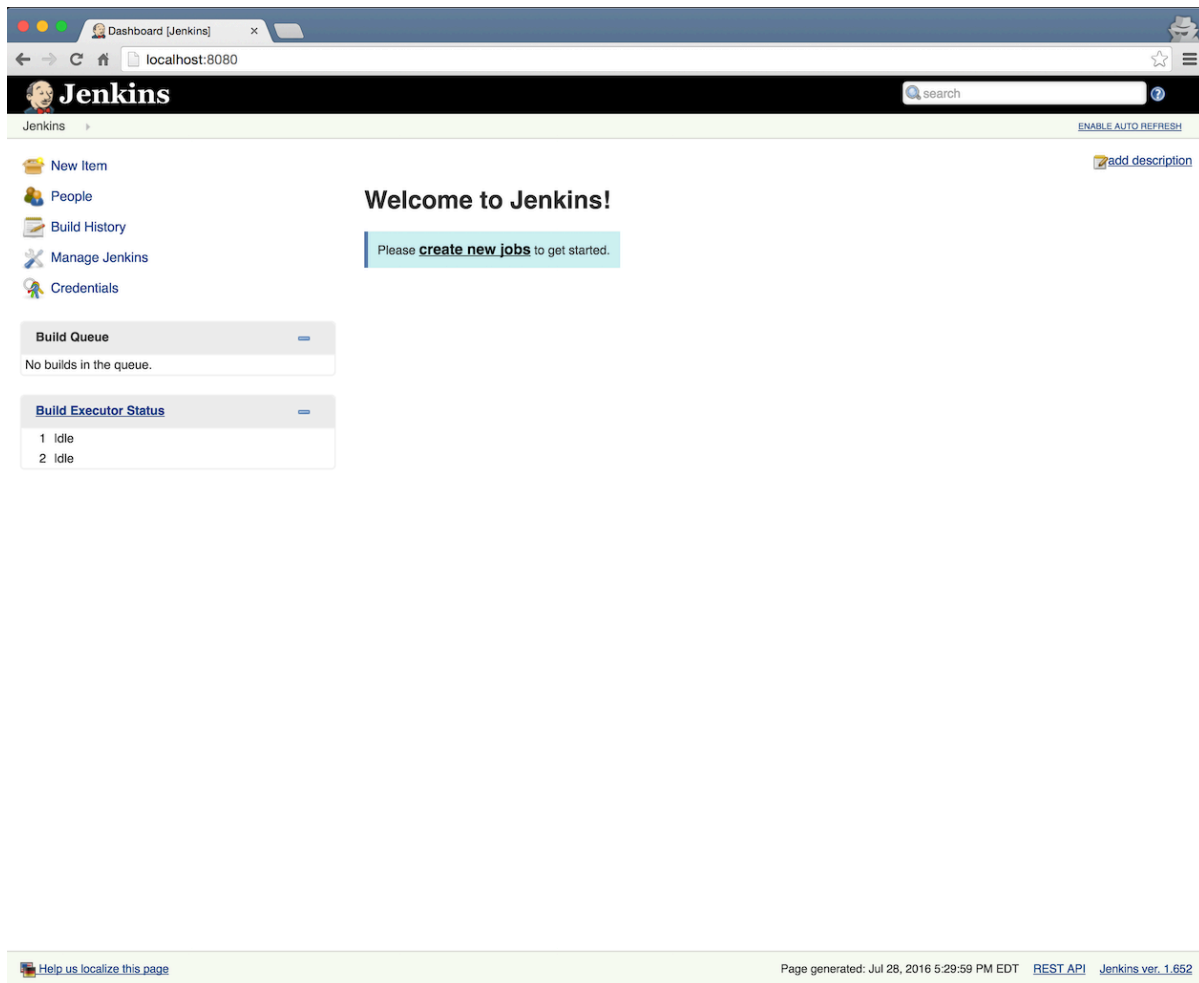
NOTE: Before moving to the next step, click `ENABLE AUTO-REFRESH` at the top right-hand side of the page. Otherwise you'll need to manually refresh the page (e.g., when running a job and waiting for results to appear).

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

Delete Project

Configure

Build History   trend —

find   x

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]

```
BROWSER=chrome BROWSER_VERSION=74 PLATFORM='Windows 10' mocha-parallel-tests
--grep=@shallow -R xunit test/*Test.js
```

`-R xunit` specifies the reporter we'd like to use for the test results. `xunit` will give us an XML report that Jenkins expects. 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 -- `*.xml`
- Click `Save`

localhost:8080/job/Shallow%20Tests%20Chrome%2050%20Windows%2010/configure

☐ 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

▦ **Execute shell**

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

Aggregate downstream test results
Archive the artifacts
Build other projects
Publish JUnit test result report
Publish Javadoc
Record fingerprints of files to track usage
E-mail Notification

Add post-build action   ▾

Save   Apply

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.

```javascript
// filename: test/LoginTest.js
// ...
  it.only('forced failure @shallow', async function() {
    await login.authenticate('tomsmith', 'bad password')
    assert.equal(false, true)
  })
```

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 `false` is not `true`, 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.

```javascript
// filename: lib/DriverFactory.js
// ...
  async quit(testPassed) {
    if (this.config.host === 'saucelabs') {
      this.driver.executeScript('sauce:job-name=' + this.testName)
      this.driver.executeScript('sauce:job-result=' + testPassed)
    }
    await this.driver.quit()
    if (this.config.host === 'saucelabs' && !testPassed) {
      throw new Error(
        'See a video of the run at https://saucelabs.com/tests/' +
          this.sessionId
      )
    }
    if (this.eyes) await this.eyes.abortIfNotClosed()
  }
}

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. We also move it to fire after calling `driver.quit()` to ensure that the job in Sauce Labs terminates correctly. This change will make the job URL show 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. If we click on the failure from the teardown we can see the URL to the job in Sauce Labs.

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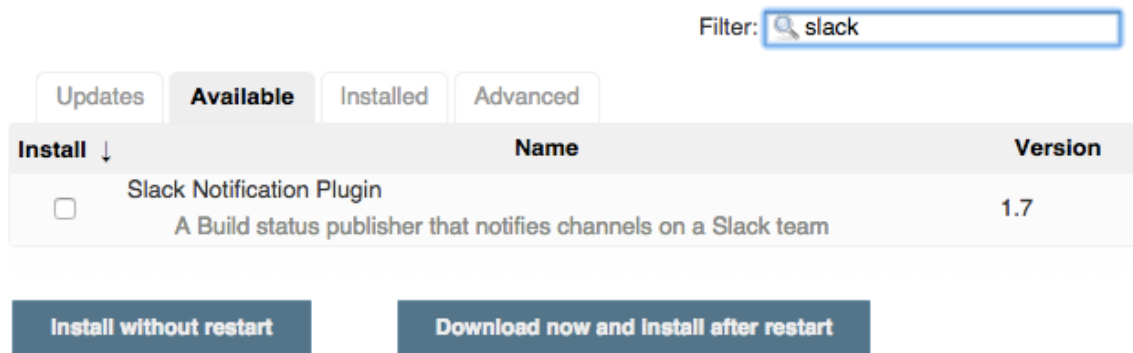
Notifications

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

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

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

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

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

Ideal Workflow

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

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

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

Outro

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

Chapter 18
Finding Information On Your Own

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

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

Documentation & Tips

- [Selenium HQ](#)

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

- [The Selenium Wiki](#)

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

- [Elemental Selenium Archives](#)

Every tip I've written is freely available on the tips archive page. There are over 70 different Selenium problems and solutions covered. They're in Ruby, but the code has been open-sourced with a fair number of them being ported into other programming languages. You can find the code for them [here](#).

Blogs

- [The official Selenium blog](#)

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

- [A list of "all" Selenium WebDriver blogs](#)

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

Other Books

- [Selenium Testing Tools Cookbook](#)

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

- [Selenium Design Patterns and Best Practices](#)

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

Meetups

- [All Selenium Meetups listed on Meetup.com](#)

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

- [How to start your own Selenium Meetup](#)

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

Conferences

- [Selenium Conf](#)

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

- [Selenium Camp](#)

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

- [List of other testing conferences](#)

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

Videos

- [Selenium Conference Talks](#)

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

- [Selenium Meetup Talks](#)

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

Mailing Lists

- [Selenium Developers List](#)

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

- [Selenium Users Google Group](#)
- [Selenium LinkedIn Users Group](#)

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

Forums

- [Stack Overflow](#)
- [Quora](#)
- [Reddit](#)

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

Issues

- [Selenium Issue Tracker](#)

If you're running into a specific and repeatable issue that just doesn't make sense, you may have found a bug in Selenium. You'll want to check the Selenium Issue Tracker to see if it has already been reported. If not, then create a new issue -- assuming you're able to provide a short and self-contained example that reproduces the problem.

This is known as [SSCCE](#) (a Short, Self Contained, Correct (Compilable), Example). For a

tongue-in-cheek take on the topic, see [this post](#).

Chatting With the Selenium Community

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

You can connect either through Slack or IRC. Details on how to connect are available [here](#).

Once connected, feel free to say hello and introduce yourself. But more importantly, ask your question. If it looks like no one is chatting, ask it anyway. Someone will see it and eventually respond. They always do. In order to get your answer, you'll probably need to hang around for a bit. But the benefit of being a fly on the wall is that you gain insight into other problems people face, possible solutions, and the current state of the Selenium project and its various pieces.

Chapter 19
Now You Are Ready

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

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

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

Cheers,
Dave H