Chapter 1
Selenium In A Nutshell

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

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

What Selenium Is and Is Not Good At

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

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

Selenium Highlights

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

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

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

Chapter 2

Defining A Test Strategy

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

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

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

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

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

What To Do With The Answers

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

Question 1: Money/Value

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

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

Question 2: Usage Data

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

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

Question 3: Browsers

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

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

Question 4: Risky Bits

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

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

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

Now You Are Ready

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

Chapter 3

Picking A Language

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

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

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

What To Do With The Answer

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

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

Some Additional Food For Thought

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

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

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

Outro

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

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

Chapter 4
A Programming Primer

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

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

Installation

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

If you're running on a Mac and you want to use Homebrew, then be sure to check out this write-up from Treehouse.

Installing Third-Party Libraries

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

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

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

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

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

Interactive Prompt

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

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

```
>
```

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

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

Choosing A Text Editor

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

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

It's important to pick an editor that works for you and makes you productive. So if you're new to programming and text editors then it's probably best to go with something more intuitive like Sublime Text or WebStorm. 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, Methods, and Classes)
- Scope
- Types of Objects (Strings, Integers, Data Structures, Booleans, etc.)
- Actions (Assertions and Conditionals)
- Inheritance
- Promises

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

Object Structures

Variables

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

A variable name:

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

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

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

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

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

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

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

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

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

Methods

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

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

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

example_method();
```

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

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

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

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

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

Classes

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

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

Class names:

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

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

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

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

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

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

function LoginPage(driver) {
}

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

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

Scope

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

Local Variables

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

Class Variables

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

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

Global Variables

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

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

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

Environment Variables

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

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

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

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

Types of Objects

Strings

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

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

Numbers

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

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

Collections

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

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

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

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

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

Booleans

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

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

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

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

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

Actions

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

Assertions

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

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

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

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

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

Conditionals

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

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

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

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

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

More on that in chapters 12 and 13.

Inheritance

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

Inheritance in JavaScript can be established by:

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

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

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

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

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

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

Promises

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

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

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

Additional Resources

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

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

Chapter 5

Anatomy Of A Good Acceptance Test

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

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

Atomic & Autonomous Tests

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

Grouping Tests

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

Being Descriptive

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

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

Test Runners

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

Version Control

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

- Git
- Mercurial
- Subversion

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

Chapter 6
Writing Your First Test

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

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

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

Selenium is able to find and interact with elements on a page by way of various locator strategies. The list includes (sorted alphabetically):

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

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

How To Find Locators

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

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

How To Find Quality Elements

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

Ripe candidates for this are `id` and `class` attributes. Whereas text (e.g., the text of a link) is less ideal since it is more apt to change. This may not hold true for when you make assertions, but it's a good goal to strive for.

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

When you can't find any unique elements have a conversation with your development team letting them know what you are trying to accomplish. It's typically a trivial thing for them to add helpful semantic markup to a page to make it more testable. This is especially true when they know the use case you're trying to automate. The alternative can be a lengthy and painful process which will probably yield working test code but it will be brittle and hard to maintain.

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

An Example

Part 1: Find The Elements And Write The Test

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

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

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

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

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

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

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

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

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

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

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

});
```

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

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

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

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

Part 2: Figure Out What To Assert

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

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

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

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

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

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

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

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

A Quick Primer on CSS Selectors

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

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

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

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

Part 3: Write The Assertion And Verify It

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

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

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

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

Just To Make Sure

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

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

If it fails then we can feel reasonably confident that the test is doing what we expect and we can change the assertion back to normal before committing our code.

This trick will save you more trouble that you know. Practice it often.