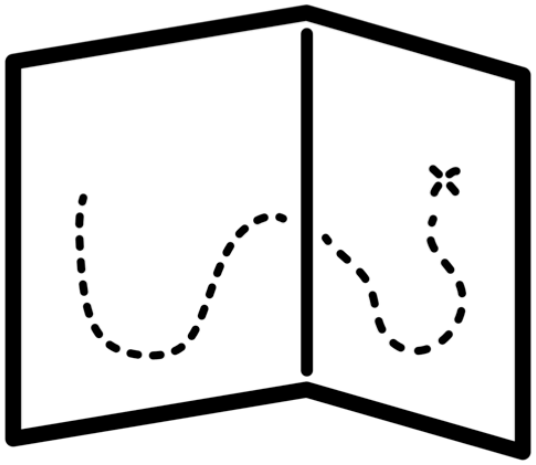
How to use Selenium, successfully



The Selenium Guidebook

Java Edition

by Dave Haeffner

Version 4.0.0

Preface

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

What you have before you is a distilled and actionable guide culled from my consulting practice and full time positions held doing Quality Assurance over the past 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 Ruby, 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 to run in JUnit, with Maven managing the third party dependencies.

All third-party libraries (a.k.a. "gems") are specified in a `Gemfile` and installed using Bundler with

`bundle install`.

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 16.

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

Feedback

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

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 Java with JUnit.

Chapter 4

A Programming Primer

This section will prime you with just enough programming concepts (and how they pertain to testing) 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.

Installing Third-Party Libraries

There are numerous third-party libraries available in Java that can help you quickly add functionality to your test code. To handle installing them (and their dependencies) you should use some form of dependency management.

The examples in this book use Maven to manage third-party libraries.

Choosing An IDE (Integrated Development Environment)

Java is a vast and picky language. In order to write test code quickly and effectively (and to avoid absolute frustration), you'll want to use an IDE.

The two most prominent options available are Eclipse and IntelliJ IDEA.

The examples in this book were written using IntelliJ IDEA Community Edition (but either IDE will work just fine).

Installation

Here are some installation instructions to help you get started quickly.

- Linux
- OSX
- Windows

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 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 to be an effective test automator right now.

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

- Object Structures (Variables, Methods, and Classes)
- Access Modifiers (public, protected, private)
- Object Types (Strings, Integers, Booleans, etc.)
- Actions (Assertions and Conditionals)
- Annotations
- Inheritance

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

Object Structures

Variables

Variables are objects where you can store and retrieve values. They are created and referenced by a name that:

- is case-sensitive
- must not be a keyword (or reserved word) in Java
- starts with a letter

If the variable name is one word, it should be all lowercase. If it's more than one word it should be CamelCase (e.g., `exampleVariable`).

You can store a value in a variable by using an equals sign (e.g., `=`). But you'll only be able to store a value of the type you specified when creating it.

```
String exampleVariable1 = "string value";
System.out.println(exampleVariable1.getClass());
// outputs: class java.lang.String

Integer exampleVariable2 = 42;
System.out.println(exampleVariable2.getClass());
// outputs: class java.lang.Integer
```

NOTE: In the code snippet above we're using `System.out.println();` to output a message. This is a common command that is useful for generating output to the console (a.k.a. terminal).

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

```
String 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. We define a method within a class (more on those next) by specifying a modifier (which we'll cover in `Access Modifiers`), a return type, and a name.

A return type is used to specify what type of an object you want to return after the method is executed. If you don't want to return anything, specify the return type as `void`.

Method naming follows similar conventions to variables. The main difference is that they tend to be a verb (since they denote some kind of an action to be performed). Also, the contents (e.g., the body) of the method are wrapped in opening and closing brackets (e.g., `{}`).

```
public void sayHello() {
  // your code goes here
}
```

Additionally, you can make a method accept an argument when calling it. This is done with a parameter.

```
public void sayHello(String message) {
  System.out.println(message);
}
```

We'll see methods put to use in numerous places in our test code. First and foremost each of our tests will use them when setting up and tearing down instances of Selenium.

```
public void setUp() {
    driver = new FirefoxDriver();
}

public void tearDown() {
    driver.quit();
}
```

Classes

Classes are a useful way to store the state and behavior of something complex for reuse. They are where variables and methods live. And they're defined with the word `class` followed by the name you wish to give it. Class names:

- must match the name of the file they're stored in
- should be CamelCase for multiple words (e.g., class ExampleClass)
- should be descriptive

To use a class you first have to define it. You then create an instance of it (a.k.a. instantiation). Once you have a class instance, you can access the methods within it to trigger an action.

The most common example of this in Selenium is when you want to represent a page in your application (a.k.a. a page object). In the page object class you store the elements from the page you want to use (e.g., state) and the actions you can perform with those elements (e.g., behavior).

```
// code in page object class
public class Login {

    private WebDriver driver;
    By usernameLocator  = By.id("username");
    By passwordLocator  = By.id("password");
    By loginFormLocator = By.id("login");

    public void with(String username, String password) {
// ...
// code in test that uses the page object
Login login = new Login
login.with("username", "password"):
```

Access Modifiers

When specifying an object (e.g., a variable, method, or class) you can apply a modifier. This modifier denotes what else can access the object. This is also known as "scope".

For classes you can apply `public` or nothing (a.k.a. package-private). `public` makes the class visible to all other classes. Specifying nothing makes it visible to just classes within the same package. A package is a way to group related classes together under a simple name.

For members of a class (e.g., variables and methods) you can use `public`, nothing, `private`, and `protected`.

- `public` and nothing behave the same as with classes
- `private` makes it so the member can only be accessed from within the class it was specified
- `protected` makes it so the member can be accessed by other classes in the same package (just like with nothing) and by a subclass of the member's class in another package

The best thing to do is to follow a "need-to-know" principle for your objects. Start with a `private` scope and only elevate it when appropriate (e.g., from `private` to `protected`, from `protected` to `public`, etc.).

In our Selenium tests, we'll end up with various modifiers for our objects.

```
// When creating a test class it needs to be public for JUnit to use it
public class TestLogin {

    // Our Selenium object should only be accessed from within the same class
    private WebDriver driver;
```

Types

Objects can be of various types, and when declaring a method we need to specify what type it will return. If it returns nothing, we specify `void`. But if it returns something (e.g., a Boolean) then we need to specify that.

The two most common types we'll see initially in our tests are Strings and Booleans. Strings are a series of alpha-numeric characters stored in double-quotes. Booleans are a `true` or `false` value.

A common example of specifying a return type in our test code is when we use Selenium to see if something is displayed on a page.

```
public Boolean successMessagePresent() {
    return isDisplayed(successMessageLocator);
}
```

After specifying the return type when declaring the method, we use the `return` keyword in the method to return the final value.

Actions

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

Assertions

An assertion is a function that allows us to test assumptions about our application.

For instance, in our test we could be testing the login functionality of our application. After logging in, we could check to see if something specific is displayed on the page (e.g., a sign out button, a success notification, etc.). This display check would return a boolean, and we would use it to assert that it is what we expected.

```
// method that looks to see if a success message is displayed after logging in
    public Boolean successMessagePresent() {
        return isDisplayed(successMessageLocator);
    }

// assertion in our test to see if the value returned is the value expected
    assertTrue("success message not present", login.successMessagePresent());
```

Conditionals

In addition to assertions, we can also leverage booleans in conditionals. Conditionals (a.k.a. control flow statements) are a way to break up the flow of code so that only certain chunks of it are executed based on predefined criteria. The most common control flow statements we'll use are `if`, `else if`, and `else`.

The most common use of this will be in how we configure Selenium to run a different browser.

```
if (browser.equals("firefox")) {
        driver = new FirefoxDriver();
    } else if (browser.equals("chrome")) {
        System.setProperty("webdriver.chrome.driver",
                System.getProperty("user.dir") + "/vendor/chromedriver");
        driver = new ChromeDriver();
    }
```

NOTE: The commands **`System.setProperty`** and **`System.getProperty`** are used to set and retrieve runtime properties in Java. This is a handy function to know since it comes built into Java and it enables us to easily create and retrieve values when running our tests.

Annotations

Annotations are a form of metadata. They are used by various libraries to enable additional functionality.

The most common use of annotations in Selenium is when specifying different types of methods (e.g., a setup method, a teardown method, a test method, etc.) to be run at different times in our test execution.

```
// methods in a test file
    @Before
    public void setUp() {
    // this method will run before each test
// ...
    @Test
    public void descriptiveTestName() {
    // this method is a test
// ...
    @After
    public void tearDown() {
    // this method will run after each test
```

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 readily available to all child classes.

Inheritance is done when declaring a child class by:

- providing the class name
- using the `extends` keyword
- providing the name of the parent class

```
public class Parent {
  static String hairColor = "brown";
}

public class Child extends Parent {
  public static void main(String[] args) {
    System.out.println(hairColor);
  }
}

// running the Child class outputs "brown"
```

We'll see this in our tests when writing all of the common Selenium actions we intend to use into methods within a parent class (a.k.a. a base page object or facade layer). Inheriting this class will allow us to call these methods in our child classes (e.g., page objects). More on this in Chapter 9.

Additional Resources

Here are some additional resources that can help you continue your Java learning journey.

- Learn Java Online
- Oracle Java Tutorials
- tutorialspoint
- Java Tutorial for Complete Beginners (video course on Udemy)
- Java In a Nutshell
- Java For Testers

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 copy (e.g., text, or the text of a link) is less ideal since it is more apt to change. This may not hold true for when you make assertions, but it's a good goal to strive for.

If the elements you are attempting to work with don't have unique `id` or `class` attributes directly on them, look at the element that houses them (a.k.a. the parent element). Oftentimes the parent element has a unique element that you can use to start with and drill down into 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 generally not a hard thing for them to add helpful, semantic markup to make test automation easier. This is especially true when they know the use case you're trying to automate. The alternative can be a lengthy, painful process which will probably yield working test code -- but it will be brittle and hard to maintain test code.

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 package called `tests` in our `src/tests/java` directory. Then let's add a test file to the package called `TestLogin.java`. We'll also need to 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 48 and newer. See Chapter 12 for more detail about browser drivers.

NOTE: There is a different geckodriver for each major operating system. Be sure to download the one that is appropriate for your machine. This example was built to run on OSX.

When we're done our directory structure should look like this.

```
pom.xml
src
   test
       java
           tests
               TestLogin.java
vendor
    geckodriver
```

And here is the test file populated with our Selenium commands and locators.

```java
//filename: tests/TestLogin.java

package tests;

import org.junit.Test;
import org.junit.Before;
import org.junit.After;
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.firefox.FirefoxDriver;

public class TestLogin {

    private WebDriver driver;

    @Before
    public void setUp() {
        System.setProperty("webdriver.gecko.driver",
                System.getProperty("user.dir") + "/vendor/geckodriver");
        driver = new FirefoxDriver();
    }

    @Test
    public void succeeded() {
        driver.get("http://the-internet.herokuapp.com/login");
        driver.findElement(By.id("username")).sendKeys("tomsmith");
        driver.findElement(By.id("password")).sendKeys("SuperSecretPassword!");
        driver.findElement(By.cssSelector("button")).click();
    }

    @After
    public void tearDown() {
        driver.quit();
    }
}
```

After importing the requisite classes for JUnit and Selenium we create a class (e.g., `public class TestLogin` and declare a field variable to store and reference an instance of Selenium WebDriver (e.g., `private WebDriver driver;`).

We then add setup and teardown methods annotated with `@Before` and `@After`. In them we're creating an instance of Selenium (storing it in `driver`) and closing it (e.g., `driver.quit();`). Thanks to the `@Before` annotation, the `public void setUp()` method will load before the test and the `@After` annotation will make the `public void tearDown()` method load after the test. This abstraction enables us to write our test with a focus on the behavior we want to exercise in the browser, rather than clutter it up with setup and teardown details.

In order for the instantiation of Selenium to work with Firefox, we need to specify the path to the `geckodriver` we downloaded into the `vendor` directory. We're able to do this by specifying a system property (e.g., `System.setProperty("webdriver.gecko.driver")`) and providing the full path to the file which we find by using the project directory path and appending `/vendor/geckodriver` to it.

NOTE: If you are testing on an older version of Firefox (e.g., 47 or earlier) then you don't need to download **`geckodriver`**. You will be able to use the legacy FirefoxDriver implementation. To do that you just need to disable Marionette (the new Firefox WebDriver implementation that geckodriver connects to) which would look like this:

```
System.setProperty("webdriver.firefox.marionette", "false");
```

Our test is a method as well (`public void succeeded()`). JUnit knows this is a test because of the `@Test` annotation. In this test we're visiting the login page by it's URL (with `driver.get();`), finding the input fields by their ID (with `driver.findElement(By.id())`), sending them text (with `.sendKeys();`), and submitting the form by clicking the submit button (with `By.cssSelector("button")).click();`).

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

Part 2: Figure Out What To Assert

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

```
<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 assertions in this markup. There's the flash message class (most appealing), the logout button (appealing), or the copy from the `h2` or the flash message (least appealing).

Since the flash message class name is descriptive, denotes 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 spaces (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 to use it.

```
//filename: tests/TestLogin.java

package tests;

import static org.junit.Assert.*;
// ...
    @Test
    public void succeeded() {
        driver.get("http://the-internet.herokuapp.com/login");
        driver.findElement(By.id("username")).sendKeys("tomsmith");
        driver.findElement(By.id("password")).sendKeys("SuperSecretPassword!");
        driver.findElement(By.cssSelector("button")).click();
        assertTrue("success message not present",
                driver.findElement(By.cssSelector(".flash.success")).isDisplayed());
    }
// ...
```

First, we had to import the JUnit assertion class. By importing it as `static` we're able to reference the assertion methods directly (without having to prepend `Assert.`). Next we add an assertion to the end of our test.

With `assertTrue` we are checking for a `true` (Boolean) response. If one is not received, a failure will be raised and the text we provided (e.g., `"success message not present"`) will be in displayed in the failure output. With Selenium we are seeing if the success message is displayed (with `.isDisplayed()`). This Selenium command returns a Boolean. So if the element is visible in the browser, `true` will be returned, and our test will pass.

```
//filename: tests/TestLogin.java
// ...
    @Test
    public void succeeded() {
        driver.get("http://the-internet.herokuapp.com/login");
        driver.findElement(By.id("username")).sendKeys("tomsmith");
        driver.findElement(By.id("password")).sendKeys("SuperSecretPassword!");
        driver.findElement(By.cssSelector("button")).click();
        assertTrue("success message not present",
                driver.findElement(By.cssSelector(".flash.success")).isDisplayed());
    }
```

When we save this and run it (`mvn clean test` 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

force a failure and run it again. A simple fudging of the locator will suffice.

```
assertTrue("success message not present",
                driver.findElement(By.cssSelector(".flash.successasdf")).isDisplayed
());
```

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

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

Want the rest of the book?

Buy your copy HERE