Selenium Bootcamp 2: Writing Your First Selenium Test

Selenium Bootcamp is a series of articles prepared by Selenium guru Dave Haeffner for Sauce Labs. Dave also authors the Elemental Selenium website, which includes tips for using Selenium, and where you can sign up for his weekly email on the topic of Selenium testing.

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 (or find the login form and submit 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).

```
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>
```

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 the parent element (`form`) does. So instead of clicking the submit button, we can find and submit the form.

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

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

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

```
//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() {
    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.id("login")).submit();
}

@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();`). Because of 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.

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 (with `.submit();`).

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: `html 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.id
("login")).submit(); 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.

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.

```
java 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.

Next up, we'll learn about writing maintainable test code.
