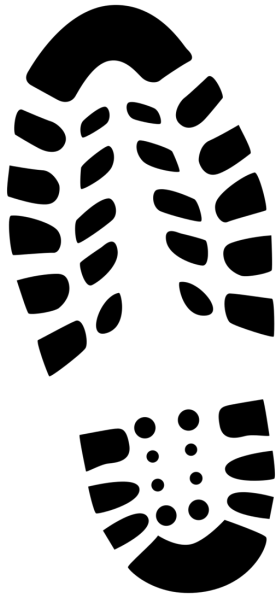
Selenium Bootcamp



Java Edition

by Dave Haeffner

Preface

Thanks for checking out my bootcamp on Getting Started with Selenium!

I'm Dave Haeffner, the writer of Elemental Selenium and The Selenium Guidebook. My goal for this book is to provide you with enough information to get started on the right foot with automated web testing, while also keeping things as actionable and succinct as possible.

Enjoy!

Cheers,
Dave H
@TourDeDave

Table of Contents

Chapter 1
The First Things You Need To Know

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

But before we dig in, there are a few things you'll want to know before you write your first test.

Define a Test Strategy

A great way to increase your chances of automated web testing success is to focus your efforts by mapping out a testing strategy. The best way to do that is to answer four questions:

1. How does your business make money (or generate value for the end-user)?
2. How do your users use your application?
3. What browsers are your users using?
4. What things have broken in the application before?

After answering these, you will have a good understanding of the functionality and browsers that matter most for the application you are testing. This will help you narrow down your initial efforts to the things that matter most.

From the answers you should be able to build a prioritized list (or backlog) of critical business functionality, a short list of the browsers to focus on, and include the risky parts of your application to watch out for. This prioritized list will help you make sure you're on the right track (e.g., focusing on things that matter for the business and its users).

Pick a Programming Language

In order to work well with Selenium, you need to choose a programming language to write your acceptance tests in. Conventional wisdom will tell you to choose the same language as what 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 development), 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?

Also, as you are considering which language to go with, consider what open source frameworks already exist for the languages you're eyeing. Going with one will save you a lot of time and give

you a host of functionality out of the box that you would otherwise have to build and maintain yourself -- and it's FREE.

You can see a list of available open source Selenium frameworks here.

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.

For this course we'll be using the Java programming language. If you need help installing Java, then check out one of the following links:

- Linux
- Mac
- Windows

Choose an Editor

In order to be productive when writing Java code, you will want to use an integrated development environment (IDE). Here are some of the more popular ones:

- Eclipse
- IntelliJ

Next, we'll be diving into how to decompose a web app and writing our first test.

Chapter 2
Writing Your First Selenium 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 (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).

```
<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.

```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() {
        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.

```
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.

```
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.

Chapter 3

How To Write Maintainable Tests

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 package called `pageobjects` in our `src/tests/java` directory. Then let's add a file to the `pageobjects` package called `Login.java`. When we're done our directory structure should look like this.

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

And here's the code that goes with it.

```java
// filename: pageobjects/Login.java

package pageobjects;

import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;

public class Login {

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

    public Login(WebDriver driver) {
        this.driver = driver;
        driver.get("http://the-internet.herokuapp.com/login");
    }

    public void with(String username, String password) {
        driver.findElement(usernameLocator).sendKeys(username);
        driver.findElement(passwordLocator).sendKeys(password);
        driver.findElement(loginFormLocator).submit();
    }

    public Boolean successMessagePresent() {
        return driver.findElement(successMessageLocator).isDisplayed();
    }

}
```

At the top of the file we specify the package where it lives and import the requisite classes from our libraries. We then declare the class (e.g., `public class Login`), specify our field variables (for the Selenium instance and the page's locators), and add three methods.

The first method (e.g., `public Login(WebDriver driver)`) is the constructor. It 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 here and store it in the `driver` field (so other methods can access it). Then the login page is visited (with `driver.get`).

The second method (e.g., `public void with(String username, String password)`) is the core functionality of the login page. It's responsible for filling in the login form and submitting it. By accepting strings parameters for the username and password we're able to make the functionality here dynamic and reusable for additional tests.

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

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

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

package tests;

import org.junit.Test;
import org.junit.Before;
import org.junit.After;
import static org.junit.Assert.*;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.firefox.FirefoxDriver;
import pageobjects.Login;

public class TestLogin {

    private WebDriver driver;
    private Login login;

    @Before
    public void setUp() {
        driver = new FirefoxDriver();
        login = new Login(driver);
    }

    @Test
    public void succeeded() {
        login.with("tomsmith", "SuperSecretPassword!");
        assertTrue("success message not present",
                login.successMessagePresent());
    }

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

Since the page object lives in another package, we need to import it (e.g., `import pageobjects.Login;`).

Then it's a simple matter of specifying a field for it (e.g., `private Login login`), creating an instance of it in our `setUp` method (passing the `driver` object to it as an argument), and

updating the test with the new actions.

Now the test is more concise and readable. If you save this and run it (e.g., `mvn clean test` 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 (remember: controlled chaos) and able 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:

```
<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.

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

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

```
//filename: pageobjects/Login.java
...
    By successMessageLocator = By.cssSelector(".flash.success");
    By failureMessageLocator = By.cssSelector(".flash.error");
...
    public Boolean successMessagePresent() {
        return driver.findElement(successMessageLocator).isDisplayed();
    }

    public Boolean failureMessagePresent() {
        return driver.findElement(failureMessageLocator).isDisplayed();
    }
}
```

Now we're ready to add another test to check for a failure condition.

```
//filename: tests/TestLogin.java
...
@Test
    public void failed() {
        login.with("tomsmith", "bad password");
        assertTrue("failure message wasn't present after providing bogus credentials",
                login.failureMessagePresent());
    }
...
```

If we save these changes and run our tests (`mvn clean test`) we will see two browser windows open (one after the other) testing for successful and failure login scenarios.

Why Asserting False Won't Work (yet)

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

```
@Test
    public void failed() {
        login.with("tomsmith", "bad password");
        assertFalse("success message was present after providing bogus credentials",
                login.successMessagePresent());
    }
```

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

```
org.openqa.selenium.NoSuchElementException: Unable to locate element: {"method":"css
selector","selector":".flash.error"}
Command duration or timeout: 123 milliseconds
For documentation on this error, please visit: http://seleniumhq.org/exceptions/
no_such_element.html
Build info: version: '2.43.1', revision: '5163bceef1bc36d43f3dc0b83c88998168a363a0',
time: '2014-09-10 09:43:55'
System info: host: 'asdf', ip: '192.168.1.112', os.name: 'Mac OS X', os.arch: 'x86_64',
os.version: '10.10.1', java.version: '1.8.0_25'
Driver info: org.openqa.selenium.firefox.FirefoxDriver
Capabilities [{applicationCacheEnabled=true, rotatable=false, handlesAlerts=true,
databaseEnabled=true, version=34.0.5, platform=MAC, nativeEvents=false, acceptSslCerts=
true, webStorageEnabled=true, locationContextEnabled=true, browserName=firefox,
takesScreenshot=true, javascriptEnabled=true, cssSelectorsEnabled=true}]
Session ID: b6648aef-5be5-e542-add1-265ed2a35a65
...
```

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 is more concise.

Part 3: Confirm We're In The Right Place

Before we can call our page object finished, there's one more addition we should make. We'll want to add an assertion to make sure that Selenium is in the right place before proceeding. This will help add some resiliency to our test.

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

```
// filename: pagesobjects/Login.java
...
import static org.junit.Assert.assertTrue;

public class Login {
...
    public Login(WebDriver driver) {
        this.driver = driver;
        driver.get("http://the-internet.herokuapp.com/login");
        assertTrue("The login form is not present",
                driver.findElement(loginFormLocator).isDisplayed());
    }
...
```

After importing the assertion library we put it to use in our constructor (after the Selenium command that visits the login page). With it we're checking to see that the login form is displayed. If it is, the tests using this page object will proceed. If not, the test will fail and provide an output message stating that the login form wasn't present.

Now when we save everything and run our tests (e.g., `mvn clean test` from the command-line), they will run just like before. But now we can rest assured that the tests will only proceed if the login form is present.

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/experience. The example demonstrated above is a simple approach. Here are some additional resources to consider as your testing practice grows:

- https://code.google.com/p/selenium/wiki/PageObjects
- https://code.google.com/p/selenium/wiki/PageFactory (a Page Object generator/helper built into Selenium)
- https://github.com/yandex-qatools/htmlelements (a simple Page Object framework by Yandex)

Now that you understand how to write maintainable tests with page objects, our next installment will dive into writing resilient tests.

Chapter 4
Writing Resilient Test Code

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

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

But there is a simple approach that makes up the bedrock of reliable and resilient Selenium tests -- and that's how you wait and interact with elements. Gone are the days of waiting for the page to finish loading, or hard-coding sleeps, or doing a blanket wait time (a.k.a. an implicit wait). Nay. Now are the wonder years of waiting for an expected outcome to occur for a specified amount of time. If the outcome occurs before the amount of time specified, then the test will proceed. Otherwise, it will wait the full amount of time specified.

We accomplish this through the use of explicit waits.

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 which 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 `DynamicLoading.java` in the `pageobjects` package.

```
pom.xml
src
   test
       java
           pageobjects
              Base.java
              DynamicLoading.java
              Login.java
           tests
               TestLogin.java
```

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

```
// filename: pageobjects/DynamicLoading.java

package pageobjects;

import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;

public class DynamicLoading extends Base {

    By startButton = By.cssSelector("#start button");
    By finishText  = By.id("finish");

    public DynamicLoading(WebDriver driver) {
        super(driver);
    }

    public void loadExample(String exampleNumber) {
        visit("http://the-internet.herokuapp.com/dynamic_loading/" + exampleNumber);
        click(startButton);
    }

    public Boolean finishTextPresent() {
        return waitForIsDisplayed(finishText, 10);
    }

}
```

Since there are two examples to choose from we created the method `loadExample` which accepts a String of the example number we want to visit as an argument. And similar to our login page object, we have a display check for the finish text (e.g., `finishTextPresent()`). This check is slightly different though. Aside from the different name, it has a second argument (an integer value of `10`). This second argument is how we'll specify how long we'd like Selenium to wait for an element to be displayed before giving up.

Let's update our base page object to enable explicit waits, adding this new `waitForIsDisplayed` method to use them.

```
// filename: pageobjects/Base.java
...
    public Boolean waitForIsDisplayed(By locator, Integer... timeout) {
        try {
            waitFor(ExpectedConditions.visibilityOfElementLocated(locator),
                    (timeout.length > 0 ? timeout[0] : null));
        } catch (org.openqa.selenium.TimeoutException exception) {
            return false;
        }
        return true;
    }

    private void waitFor(ExpectedCondition<WebElement> condition, Integer timeout) {
        timeout = timeout != null ? timeout : 5;
        WebDriverWait wait = new WebDriverWait(driver, timeout);
        wait.until(condition);
    }

}
```

Selenium comes with a wait function which we wrap in a private method (e.g., `private void waitFor`) for reuse in this class.

This method accepts two arguments -- the condition we want to wait for (e.g., `ExpectedCondition<WebElement>`) and the amount of time we want Selenium to keep checking for (e.g., `Integer timeout`). If a `null` value is passed as an argument for the timeout, then the wait time is set to `5` seconds. This is handled by a ternary operator (e.g., `timeout = timeout != null ? timeout : 5;`).

The `waitForIsDisplayed` method has two parameters -- one for a locator, another for the timeout. Inside the method we call `waitFor` and send it an `ExpectedCondition` to check for the visibility of an element (e.g., `.visibilityOfElementLocated(locator)`). This is similar to our previous display check, but it uses a different Selenium API function that will work with the explicit waits function. You can see a full list of Selenium's `ExpectedConditions` here. Unfortunately, this function doesn't return a Boolean, so we provide one . If the condition is not met by Selenium in the amount of time provided, it will throw a timeout exception. When that happens, we catch it and return `false`. Otherwise, we return `true`.

It's worth noting that the second parameter is optional when calling `waitForIsDisplayed` (e.g., `Integer... timeout`). If a timeout value is specified, it will get passed to the `waitFor` method . `If nothing is specified,` null `will get passed instead. This gives us the freedom to call this method in our page objects without specifying a timeout (e.g.,` waitForIsDisplayed(locator) `) or with a timeout (e.g.,` waitForIsDisplayed(locator, 20)`).

More On Explicit Waits

In our page object when we're using `waitForIsDisplayed(finishText, 10)` we are telling Selenium to check if the finish text is visible on the page repeatedly. It will keep checking until either the element is displayed or reaches ten seconds -- whichever comes first.

It's important to set a reasonably sized default timeout for the explicit wait method. But you want to be careful not to make it too high. Otherwise you can run into similar timing issues you get from an implicit wait. But set it too low and your tests will be brittle, forcing you to run down trivial and transient issues.

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 this one 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 static sleep would).

If you're thinking about mixing explicit waits with an implicit wait -- don't. If you use both together, you're going to run into issues later on due to inconsistent implementations of implicit wait across local and remote browser drivers. Long story short, you'll see inconsistent and odd test behavior. You can read more about the specifics here.

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 `TestDynamicLoading.java` in the `tests` package.

```
pom.xml
src
   test
       java
           pageobjects
              Base.java
              DynamicLoading.java
              Login.java
           tests
               TestDynamicLoading.java
               TestLogin.java
```

The contents of this test file are similar to `TestLogin` with regards to the imported classes and the `setUp` / `tearDown` methods.

```
// filename: tests/TestDynamicLoading.java

package tests;

import org.junit.Test;
import org.junit.Before;
import org.junit.After;
import static org.junit.Assert.*;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.firefox.FirefoxDriver;
import pageobjects.DynamicLoading;

public class TestDynamicLoading {

    private WebDriver driver;
    private DynamicLoading dynamicLoading;

    @Before
    public void setUp() {
        driver = new FirefoxDriver();
        dynamicLoading = new DynamicLoading(driver);
    }

    @Test
    public void hiddenElementLoads() {
        dynamicLoading.loadExample("1");
        assertTrue("finish text didn't display after loading",
                dynamicLoading.finishTextPresent());
    }

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

In our test (e.g., `public void hiddenElementLoads()`) 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 rendered.

When we save this and run it (`mvn clean test -Dtest=TestDynamicLoading` from the command-line) it will run, wait for the loading bar to complete, and pass.

Part 3: Update Page Object And Add A New Test

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

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

Here's the markup for it.

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

  <br>

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

  <br>
</div>
```

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

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

Let's add a second test to `TestDynamicLoading.java` called `elementAppears()` that will load this second example and perform the same check as we did for the previous test.

```
// filename: tests/TestDynamicLoading.java
...
    @Test
    public void hiddenElementLoads() {
        dynamicLoading.loadExample("1");
        assertTrue("finish text didn't display after loading",
                dynamicLoading.finishTextPresent());
    }

    @Test
    public void elementAppears() {
        dynamicLoading.loadExample("2");
        assertTrue("finish text didn't render after loading",
                dynamicLoading.finishTextPresent());
    }
...
```

When we run both tests (`mvn clean test -Dtests=TestDynamicLoading` from the command-line) we will see that the same approach will work for both cases.

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 against various browsers.

It's simple enough to write your tests locally against Firefox and assume you're all set. Once you start to run things against other browsers, you may be in for a rude awakening. The first thing you're likely to run into is the speed of execution. A lot of your tests will start to fail when you point them at either Chrome or Internet Explorer, and likely for different reasons.

In my experience, Chrome execution is very fast, so you will see some odd timeout failures. This is an indicator that you need to add explicit waits to parts of your page objects that don't already have them. And the inverse is true when running things against older version of Internet Explorer (e.g., IE 8). 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 and find the failures. Take each failed test, adjust your code as needed, and run it against the browsers you care about. Repeat until you make a pass all the way through each of the failed tests. Then run a batch of all your tests to see where they fall down. Repeat until everything's green.

Once you're on the other side of these issues, the amount of effort you need to put into it should diminish dramatically.

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.

Next, in the final installment of this course, we'll take an in-depth look at packaging up your tests and running them against different browser and operating system combinations.

Chapter 5

Packaging 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.

Global Setup & Teardown

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

To do that, we'll want to create a base test. So let's create a new file called `Base.java` in the `tests` package.

```
pom.xml
src
   test
       java
           pageobjects
              DynamicLoading.java
              Login.java
           tests
               Base.java
               TestDynamicLoading.java
               TestLogin.java
```

And here are the contents of the file.

```java
// filename: tests/Base.java

package tests;

import org.junit.Rule;
import org.junit.rules.ExternalResource;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.firefox.FirefoxDriver;

public class Base {

    protected WebDriver driver;

    @Rule
    public ExternalResource resource = new ExternalResource() {

        @Override
        protected void before() throws Throwable {
            driver = new FirefoxDriver();
        }

        @Override
        protected void after() {
            driver.quit();
        }

    };

}
```

After importing a few necessary classes we specify the `Base` class and wire up some methods that will take care of setting up and tearing down Selenium before and after every test.

It's worth noting that we could have used methods with `@Before` and `@After` annotations just like before. But if we did that we'd be giving up the ability to use these annotations in our tests (which we'll want to use for page object instantiation).

To preserve this functionality we're using JUnit's `ExternalResource` Rule. This rule has `before` and `after` methods that execute prior to methods annotated with `@Before` and `@After`. For more info on JUnit's Rules, read this. Now let's update our tests to establish inheritance with this base test class, remove the Selenium setup/teardown actions, and remove the unnecessary import statements. When we're done our test files should look like this:

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

import org.junit.Test;
import org.junit.Before;
import static org.junit.Assert.*;
import pageobjects.Login;

public class TestLogin extends Base {

    private Login login;

    @Before
    public void setUp() {
        login = new Login(driver);
    }

    @Test
    public void succeeded() {
        login.with("tomsmith", "SuperSecretPassword!");
        assertTrue("success message not present",
                login.successMessagePresent());
    }

    @Test
    public void failed() {
        login.with("tomsmith", "bad password");
        assertTrue("failure message wasn't present after providing bogus credentials",
                login.failureMessagePresent());
        assertFalse("success message was present after providing bogus credentials",
                login.successMessagePresent());
    }

}
```

```
// filename: tests/TestDynamicLoading.java

package tests;

import org.junit.Test;
import org.junit.Before;
import static org.junit.Assert.*;
import pageobjects.DynamicLoading;

public class TestDynamicLoading extends Base {

    private DynamicLoading dynamicLoading;

    @Before
    public void setUp() {
        dynamicLoading = new DynamicLoading(driver);
    }

    @Test
    public void hiddenElementLoads() {
        dynamicLoading.loadExample("1");
        assertTrue("finish text didn't display after loading",
                dynamicLoading.finishTextPresent());
    }

    @Test
    public void elementAppears() {
        dynamicLoading.loadExample("2");
        assertTrue("finish text didn't render after loading",
                dynamicLoading.finishTextPresent());
    }

}
```

Running Tests On Any Browser

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

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 to scale and 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.

An Example

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 than we have been. Let's start by creating a file for these various configuration values.

Let's create a file called Config.java in the tests package.

```
pom.xml
src
   test
       java
           pageobjects
              DynamicLoading.java
              Login.java
           tests
               Base.java
               Config.java
               TestDynamicLoading.java
               TestLogin.java
```

In it we'll create an interface and specify field variables for the different configuration values we need.

```
// filename: tests/Config.java

package tests;

public interface Config {
    final String baseUrl        = System.getProperty("baseUrl",
"http://the-internet.herokuapp.com");
    final String browser        = System.getProperty("browser", "firefox");
    final String host           = System.getProperty("host", "localhost");
    final String browserVersion = System.getProperty("browserVersion", "33");
    final String platform       = System.getProperty("platform", "Windows XP");
    final String sauceUser      = "your-sauce-username";
    final String sauceKey       = "your-sauce-access-key";
}
```

`host` enables us to specify whether our tests run locally or on Sauce Labs. If we don't specify a value at runtime, then the tests will execute locally.

With `browser`, `browserVeresion`, and `platform` we can specify which browser and operating system combination we want our tests to run on. You can see a full list of Sauce's available platform options here. They also have a handy configuration generator (which will tell you what values to plug into your test) here. We've made so if no values are provided at run time, they will default to `firefox` version `33` running on `Windows XP`. `sauceUser` is your Sauce username, and `sauceKey` is your Sauce Access Key (which can be found on your account dashboard). An alternative to hard-coding your credentials is to store them in environment variables and retrieve them.

```
final String sauceUser      = System.getenv("SAUCE_USERNAME");
final String sauceKey       = System.getenv("SAUCE_ACCESS_KEY");
```

Do whichever you're more comfortable and familiar with.

Now we can update our base test class to work with Selenium Remote (which is how we'll be able to connect to Sauce Labs).

```
// filename: tests/Base.java
...
        @Override
        protected void before() throws Throwable {
            if (host.equals("saucelabs")) {
                DesiredCapabilities capabilities = new DesiredCapabilities();
                capabilities.setCapability("browserName", browser);
                capabilities.setCapability("version", browserVersion);
                capabilities.setCapability("platform", platform);
                String sauceUrl = String.format(
"http://%s:%s@ondemand.saucelabs.com:80/wd/hub",
                        sauceUser, sauceKey);
                driver = new RemoteWebDriver(new URL(sauceUrl), capabilities);
            } else if (host.equals("localhost")) {
                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();
                }
            }
        }
...
```

In our `before` method we've added a new conditional flow (e.g., `if` / `else if`) to check the `host` variable.

We start by checking to see if it's set to `"saucelabs"`. If it is we create a `DesiredCapabilities` object, populate it (with `browser`, `browserVersion`, and `platform` values), and connect to Sauce Labs using Selenium Remote (passing in the `DesiredCapabilities` object). This will return a Selenium WebDriver object that we can use just like when running our tests locally.

If the `host` variable is set to `"localhost"` then our tests will run locally just like before.

If we save everything and run our tests in Sauce Labs (e.g., `mvn clean test -Dhost=saucelabs`) then on the account dashboard we'll see our tests running in Firefox 33 on Windows XP.

And if we wanted to run our tests on different browser and operating system combinations, here are what some of the commands would look like:

```
mvn clean test -Dhost=saucelabs -Dbrowser="internet explorer" -DbrowserVersion=8
mvn clean test -Dhost=saucelabs -Dbrowser="internet explorer" -DbrowserVersion=10 -
Dplatform="Windows 8.1"
mvn clean test -Dhost=saucelabs -Dbrowser=firefox -DbrowserVersion=26 -Dplatform=
"Windows 7"
mvn clean test -Dhost=saucelabs -Dbrowser=safari -DbrowserVersion=8 -Dplatform="OS X
10.10"
mvn clean test -Dhost=saucelabs -Dbrowser=chrome -DbrowserVersion=40 -Dplatform="OS X
10.8"
```

Notice the properties with quotations (e.g., `"internet explorer"` and `"OS X 10.10"`). When dealing with more than one word in a runtime property we need to make sure to surround them in double-quotes (or else our test code won't compile).

That wraps up The Selenium Bootcamp.

There's a whole lot more to think about when it comes to using Selenium successfully. If you're interested in learning what it takes and how to do it, then grab a copy of The Selenium Guidebook today!

Cheers,
Dave H
@TourDeDave