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. The best way to accomplish this is through the use of **explicit waits**.

An Explicit Waits Primer

Explicit waits are applied to individual test actions. Each time you want to use one you specify an amount of time (in seconds) and the Selenium action action you want to accomplish.

Selenium will repeatedly try this action until either it can be accomplished, or the amount of time specified has been reached. If the latter occurs, a timeout exception will be thrown.

An Example

Let's step through an example that demonstrates this against a dynamic page on the-internet. The functionality is pretty simple -- there is a button. When you click it a loading bar appears for 5 seconds, after which it disappears and is replaced with the text `Hello World!`.
Part 1: Create A New Page Object

Here's the markup from the page.

```
<div class="example">
  <u>Dynamically Loaded Page Elements</u>
  <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
│           │   ├── DynamicLoading.java
│           │   └── Login.java
│           └── tests
│               └── TestLogin.java
```

Now let's populate it with the locators from the page and the Selenium actions we want to take.

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

package pageobjects;

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

public class DynamicLoading {

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

    public DynamicLoading(WebDriver driver) {
        this.driver = driver;
    }

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

    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 in that the function we're calling has a second argument (an integer value of `10`). This argument is how we'll specify how long we'd like Selenium to wait for an element to be displayed before giving up.
But the `waitForIsDisplayed` method doesn't exist yet, so let's create it and wire up our explicit wait code.

```
// filename: pageobjects/DynamicLoading.java
// ...
import org.openqa.selenium.support.ui.WebDriverWait;
import org.openqa.selenium.support.ui.ExpectedCondition;
import org.openqa.selenium.support.ui.ExpectedConditions;
// ...
    private Boolean waitForIsDisplayed(By locator, Integer timeout) {
        timeout = timeout != null ? timeout : 5;
        try {
            WebDriverWait wait = new WebDriverWait(driver, timeout);

wait.until(ExpectedConditions.visibilityOfElementLocated(locator));
        } catch (org.openqa.selenium.TimeoutException exception) {
            return false;
        }
        return true;
    }

}
```

This method accepts two arguments -- the locator we want to wait for (e.g.,`By locator`) and the amount of time we want Selenium to keep checking for (e.g., `Integer timeout`). And we check the timeout to see if one was provided. If it was then we use it, otherwise `5` seconds will be used (e.g.,`timeout = timeout != null ? timeout : 5;`).
Selenium has a wait class (e.g., `WebDriverWait`) that enables us to specify a condition to wait for (e.g., `ExpectedConditions.visibilityOfElementLocated`). This is similar to our previous display check used in the login page object, but has the added benefit of working with the explicit waits function. You can see a full list of Selenium's`ExpectedConditions` here.
Unfortunately, explicit waits don't return a Boolean, so we provide our own. 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`.

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 (e.g., like we did with the `5` seconds in `waitForIsDisplayed`). But you want to be careful not to make it too high. Otherwise you can run into similar timing issues you get from a setting an implicit wait

(which applies to all Selenium actions where an explicit wait has not been specified). 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. 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 across local and remote browser drivers. Long story short, you'll run into odd, transient 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
│           │   ├── 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 more 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">
  <u>Dynamically Loaded Page Elements</u>
  <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 surprise. 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, 9, etc.). 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.