Selenium 2.0 WebDriver

Selenium WebDriver was developed to better support dynamic web pages where elements of a page may change without having to reload the page. WebDriver’s goal is to supply a well-designed object-oriented API that provides improved support for modern advanced web-app testing problems.

Setting Up a Selenium-WebDriver Project in Java

The easiest way to set up a Selenium 2.0 Java project is to use Maven. Maven will download the Java bindings (the Selenium 2.0 Java client library) and all its dependencies, and will create the project for you, using a maven pom.xml (project configuration) file. Once you’ve done this, you can import the maven project into Eclipse.

First, create a folder to contain your Selenium project files. Then, to use Maven, you need a pom.xml file. This can be created with a text editor. Your pom.xml file will look something like this. Create this file in the folder you created for your project.

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>MySel20Proj</groupId>

<artifactId>MySel20Proj</artifactId>

<version>1.0</version>

<dependencies>

<dependency>

<groupId>org.seleniumhq.selenium</groupId>

<artifactId>selenium-server</artifactId>

<version>3.0.1</version>

</dependency>

</dependencies>

</project>

Be sure you specify the most current version. Check the Maven download page for the current release and edit the above dependency accordingly.

Now, from a command-line, CD into the project directory and run maven as follows:

mvn clean install

This will download Selenium and all its dependencies and will add them to the project.

Finally, import the project into your preferred development environment.

Selenium-WebDriver API Commands and Operations  
Fetching a Page

Ex.: driver.get(“http://www.google.com”);

Locating of Elements (WebElements)

By ID  
This is the most efficient and preferred way to locate an element. Common pitfalls that UI developers make is having non-unique IDs on a page or auto-generating the ID, both should be avoided. A class on an html element is more appropriate than an auto-generated ID.

Example of how to find an element that looks like this:

<div id=“coolestWidgetEver”>...</div>

WebElement element = driver.findElement(By.id(“coolestWidgetEver”));

By Class Name

“Class” in this case refers to the attribute on the DOM element. Often in practical use there are many DOM elements with the same class name, thus finding multiple elements becomes the more practical option over finding the first element.

Example of how to find an element that looks like this:

<div class=“vegetable”><span>Carrot</span></div><div class=“vegetable”><span>Celery</span></div>

List<WebElement> veggies = driver.findElements(By.className(“vegetable”));

By Tag Name

The DOM Tag Name of the element.

Example of how to find an element that looks like this:

<iframe src=“...”></iframe>

WebElement frame = driver.findElement(By.tagName(“iframe”));

By Name

Find the input element with matching name attribute.

Example of how to find an element that looks like this:

<input name=“tofu” type=“text”/>

WebElement tofu = driver.findElement(By.name(“tofu”));

By Link Text

Find the link element with matching visible text.

Example of how to find an element that looks like this:

<a href=“http://www.google.com/search?q=tofu”>tofu</a>>

WebElement tofu = driver.findElement(By.linkText(“tofu”));

By Partial Link Text

Find the link element with partial matching visible text.

Example of how to find an element that looks like this:

<a href=“http://www.google.com/search?q=tofu”>search for tofu</a>>

WebElement tofu = driver.findElement(By.partialLinkText(“tofu”));

By CSS

Like the name implies it is a locator strategy by css. Native browser support is used by default, so please refer to w3c css selectors for a list of generally available css selectors. If a browser does not have native support for css queries, then Sizzle is used.

Beware that not all browsers were created equal, some css the might work in one version may not work in another.

Example of how to find the tofu below:

<div id=“food”><span class=“soy”>soybeans</span><span class=“soy processed”>tofu</span></div>

WebElement tofu = driver.findElement(By.cssSelector(“#food span.soy.processed”));

By XPath

At a high level, WebDriver uses a browser’s native Xpath capabilities wherever possible.

|  |  |  |  |
| --- | --- | --- | --- |
| Driver | Tag and Attribute Name | Attribute Values | Native Xpath Support |
| HtmlUnit Driver | Lower-cased | As they appear in the HTML | Yes |
| Internet Explorer Driver | Lower-cased | As they appear in the HTML | No |
| Firefox Driver | Case insensitive | As they appear in the HTML | Yes |

This is a little abstract, so for the following piece of HTML:

<input type=“text” name=“example” />

<INPUT type=“text” name=“other” />

List<WebElement> inputs = driver.findElements(By.xpath(“//input”));

The following number of matches will be found

|  |  |  |  |
| --- | --- | --- | --- |
| XPath expression | HtmlUnit Driver | Firefox Driver | Internet Explorer Driver |
| //input | 1 (“example”) | 2 | 2 |
| //INPUT | 0 | 2 | 0 |

Sometimes HTML elements do not need attributes to be explicitly declared because they will default to known values. For example, the “input” tag does not require the “type” attribute because it defaults to “text”. The rule of thumb when using xpath in WebDriver is that you **should not** expect to be able to match against these implicit attributes.

Using JavaScript

You can execute arbitrary javascript to find an element and as long as you return a DOM Element, it will be automatically converted to a WebElement object.

Simple example on a page that has jQuery loaded:

WebElement element = (WebElement) ((JavascriptExecutor)driver).executescript(“return $(‘.tofu’)[0]”);

Finding all the input element for every label on a page:

List<WebElement> labels = driver.findElements(By.tagName(“label”));

List<WebElement> inputs = (List<WebElement>) ((JavascriptExecutor)driver).executeScript(

“var labels = arguments[0], inputs = []; for (var i=0; i < labels.length; i++){“ +

“inputs.push(documents.getElementById(labels[i].getAttribute(‘for’))); } return inputs;”, labels);

Getting text values

People often wish to retrieve the innerText value contained within an element. This returns a single string value. Note that this will only return the visible text displayed on the page.

WebElement element = driver.findElement(By.id(“elementID”));

element.getText();

User Input – Filling In Forms

We’ve already seen how to enter text into a textarea or text field, but what about the other elements? You can “toggle” the state of checkboxes, and you can use “click” to set something like an OPTION tag selected. Dealing with SELECT tags isn’t too bad:

WebElement select = driver.findElement(By.tagName(“select”));

List<WebElement> allOptions = select.findElements(By.tagName(“option”));

for (WebElement option : allOptions) {

System.out.println(String.format(“Value is: %s”, option.getAttribute(“value”)));

option.click();

}

This will find the first “SELECT” element on the page, and cycle through each of its OPTIONs in turn, printing out their values, and selecting each in turn. As you will notice, this isn’t the most efficient way of dealing with SELECT elements. WebDriver’s support classes include one called “Select”, which provides useful methods for interacting with these.

Select select = new Select(driver.findElement(By.tagName(“select”)));

select.deselectAll();

select.selectByVisibleText(“Edam”);

This will deselect all OPTIONs from the first SELECT on the page, and then select the OPTION with the displayed text of “Edam”.

Once you’ve finished filling out the form, you probably want to submit it. One way to do this would be to find the “submit” button and click it:

driver.findElement(By.id(“submit”)).click();

Alternatively, WebDriver has the convenience method “submit” on every element. If you call this on an element within a form, WebDriver will walk up the DOM until it finds the enclosing form and then calls submit on that. If the element isn’t in a form, then the NoSuchElementException will be thrown:

element.submit();

Moving Between Windows and Frames

Some web applications have many frames or multiple windows. WebDriver supports moving between named windows using the “switchTo” method:

driver.switchTo().window(“windowName”);

All calls to driver will now be interpreted as being directed to the particular window. But how do you know the window’s name? Take a look at the javascript or link that opened it:

<a href=“somewhere.html” target=“windowName”>Click here to open a new window</a>

Alternatively, you can pass a “window handle” to the “switchTo().window()” method. Knowing this, it’s possible to iterate over every open window like so:

for (String handle : driver.getWindowHandles()) {

driver.switchTo().windows(handle);

}

You can also switch from frame to frame (or into iframes):

driver.switchTo().frame(“frameName”);

Popup Dialogs

Starting with Selenium 2.0 beta 1, there is built in support for handling popup dialog boxes. After you’ve triggered an action that opens a popup, you can access the alert with the following:

Alert alert = driver.switchTo().alert();

This will return the currently open alert object. With this object you can now accept, dismiss, read its contents or even type into a prompt. This interface works equally well on alerts, confirms, and prompts. Refer to the JavaDocs or RubyDocs for more information.

Navigation: History and Location

Earlier, we covered navigating to a page using the “get” command (driver.get(“[http://www.example.com](http://www.example.com/)”). As you’ve seen, WebDriver has a number of smaller, task-focused interfaces, and navigation is a useful task. Because loading a page is such a fundamental requirement, the method to do this lives on the main WebDriver interface, but it’s simply a synonym to:

driver.navigate().to(“[http://www.example.com](http://www.example.com/)”);

To reiterate: “navigate().to()” and “get()” do exactly the same thing. One’s just a lot easier to type than the other!

The “navigate” interface also exposes the ability to move backwards and forwards in your browser’s history:

driver.navigate().forward();

driver.navigate().back();

Please be aware that this functionality depends entirely on the underlying browser. It’s just possible that something unexpected may happen when you call these methods if you’re used to the behavior of one browser over another.

Cookies

Before we leave these next steps, you may be interested in understanding how to use cookies. First of all, you need to be on the domain that the cookie will be valid for. If you are trying to preset cookies before you start interacting with a site and your homepage is large / takes a while to load an alternative is to find a smaller page on the site (typically the 404 page is small, e.g., <http://example.com/some404page>).

// Go to the correct domain

driver.get(“[http://www.example.com](http://www.example.com/)”);

// Now set the cookie. This one’s valid for the entire domain

Cookie cookie = new Cookie(“key”, “value”);

driver.manage().addCookie(cookie);

// And now output all the available cookies for the current URL

Set<Cookie> allCookies = driver.manage().getCookies();

for (Cookie loadedCookie : allCookies) {

System.out.println(String.format(“%s -> %s”, loadedCookie.getName(), loadedCookie.getValue()));

}

// You can delete cookies in 3 ways

// By name

driver.manage().deleteCookieNamed(“CookieName”)’

// By Cookie

driver.manage().deleteCookie(loadedCookie);

// Or all of them

driver.manage().deleteAllCookies();

Changing the User Agent

This is easy with the Firefox Driver:

FirefoxProfile profile = new FirefoxProfile();

profile.addAdditionalPreference(“general.useragent.override”, “some UA string”);

WebDriver driver = new FirefoxDriver(profile);

Drag and Drop

Here’s an example of using the Actions class to perform a drag and drop. Native events are required to be enabled.

WebElement element = driver.findElement(By.name(“source”));

WebElement target = driver.findElement(By.name(“target”));

(new Actions(driver)).dragAndDrop(element, target).perform();

Driver Specifics and Tradeoffs

Selenium-WebDriver’s Drivers

WebDriver is the name of the key interface against which tests should be written, but there are several implementations. These include:

HtmlUnit Driver

This is currently the fastest and most lightweight implementation of WebDriver. As the name suggests, this is based on HtmlUnit. HtmlUnit is a java based implementation of a WebBrowser without a GUI. For any language binding (other than java) the Selenium Server is required to use this driver.

Usage

WebDriver driver = new HtmlUnitDriver();

Pros

- Fastest implementation of WebDriver

- A pure Java solution and so it is platform independent

- Supports JavaScript

Cons

- Emulates other browsers’ JavaScript behavior (see below)

JavaScript in the HtmlUnit Driver

None of the popular browsers uses the JavaScript engine used by HtmlUnit (Rhino). If you test JavaScript using HtmlUnit the results may differ significantly from those browsers.

When we say “JavaScript” we actually mean “JavaScript and the DOM”. Although the DOM is defined by the W3C each browser has its own quirks and differences in their implementation of the DOM and in how JavaScript interacts with it. HtmlUnit has an impressively complete implementation of the DOM and has good support for using JavaScript, but it is no different from any other brower: it has its own quirks and differences from both the W3C standard and the DOM implementations of the major browsers, despite its ability to mimic other browsers.

With WebDriver, we had to make a choice; do we enable HtmlUnit’s JavaScript capabilities and run the risk of teams running into problems that only manifest themselves there, or do we leave JavaScript disabled, knowing that there are more and more sites that rely on JavaScript? We took the conservative approach, and by default have disabled support when we use HtmlUnit. With each release of both WebDriver and HtmlUnit, we reassess this decision: we hope to enable JavaScript by default on the HtmlUnit at some point.

Enabling JavaScript

If you can’t wait, enabling JavaScript support is very easy:

HtmlUnitDriver driver = new HtmlUnitDriver(true);

This will cause the HtmlUnit Driver to emulate Firefix 3.6’s JavaScript handling by default.

Firefox Driver

Controls the Firefox browser using a Firefox plugin. The Firefox Profile that is used is stripped down from what is installed on the machine to only include the Selenium WebDriver.xpi (plugin). A few settings are also changed by default (see the source to see which ones). Firefox Driver is capable of being run and is tested on Windows, Mac, Linux. Currently on versions 3.6, 10, latest – 1, latest

Usage

WebDriver driver = new FirefoxDriver();

Pros

- Runs in a real browser and supports JavaScript

- Faster than the Internet Explorer Driver

Cons

- Slower than the HtmlUnit Driver

Modifying the Firefox Profile

Suppose that you wanted to modify the user agent string (as above), but you’ve got a tricked out Firefox profile that contains dozens of useful extensions. There are two ways to obtain this profile. Assuming that the profile has been created using Firefox’s profile manager (firefox -ProfileManager):

ProfilesIni allProfiles = new ProfilesIni();

FirefoxProfile profile = allProfiles.getProfile(“WebDriver”);

profile.setPreferences(“foo.bar”, 23);

WebDriver driver = new FirefoxDriver(profile);

Alternatively, if the profile isn’t already registered with Firefox:

File profileDir = new File(“path/to/top/level/of/profile”);

FirefoxProfile profile = new FirefoxProfile(profileDir);

profile.addAdditionalPreferences(extraPrefs);

WebDriver driver = new FirefoxDriver(profile);

As we develop features in the Firefox Driver, we expose the ability to use them. For example, until we feel native events are stable on Firefox for Linux, they are disabled by default. To enable them:

FirefoxProfile profile = new FirefoxProfile();

profile.setEnableNativeEvents(true);

WebDriver driver = new FirefoxDriver(profile);

Internet Explorer Driver

The InternetExplorerDriver is a standalone server which implements WebDriver’s wire protocol. This driver has been tested with IE 7, 8, 9, 10, and 11 on appropriate combinations of Vista, Windows 7, Windows 8, and Windows 8.1. As of April 15, 2014, IE 6 is no longer supported.

The driver supports running 32-bit and 64-bit versions of the browser. The choice of how to determine which “bit-ness” to use in launching the browser depends on which version of the IEDriverServer.exe is launched. If the 32-bit version of IEDriverServer.exe is launched, the 32-bit version of IE will be launched. Similarly, if the 64-bit version of IEDriverServer.exe is launched, the 64-bit version of IE will be launched.

Usage

WebDriver driver = new InternetExplorerDriver();

Pros

- Runs in a real browser and supports JavaScript

Cons

- Obviously the InternetExplorerDriver will only work on Windows!

- Comparatively slow (though still pretty snappy!)

ChromeDriver

ChromeDriver is maintained / supported by the Chromium project itself. WebDriver works with Chrome through the chromedriver binary (found on the chromium project’s download page). You need to have both chromedriver and a version of chrome browser installed. chromedriver needs to be placed somewhere on your system’s path in order for WebDriver to automatically discover it. The Chrome browser itself is discovered by chromedriver in the default installation path. These both can be overridden by environment variables.

Usage

WebDriver driver = new Chromedriver();

Pros

- Runs in a real browser and supports JavaScript

- Because Chrome is a Webkit-based browser, the ChromeDriver may allow you to verify that your site works in Safari. Note that since Chrome uses its own V8 JavaScript engine rather than Safari’s Nitro engine, JavaScript execution may differ.

Cons

- Slower than the HtmlUnit Driver

Explicit and Implicit Waits

Waiting is having the automated task execution elapse a certain amount of time before continuing with the next step. You should choose to use Explicit Waits or Implicit Waits.

WARNING: Do not mix implicit and explicit waits. Doing so can cause unpredictable wait times. For example setting an implicit wait of 10 seconds and an explicit wait of 15 seconds, could cause a timeout to occur after 20 seconds.

Explicit Waits

An explicit wait is code you define to wait for a certain condition to occur before proceeding further in the code. The worst case of this is Thread.sleep(), which sets the condition to an exact time period to wait. There are some convenience methods provided that help you write code that will wait only as long as required. WebDriverWait in combination with ExpectedCondition is one way this can be accomplished.

WebDriver driver = new FirefoxDriver();

driver.get(“<http://somedomain/url_that_delays_loading>”);

WebElement myDynamicElement = (new WebDriverWait(driver, 10))

.until(ExpectedConditions.presenceOfElementLocated(By.id(“myDynamicElement”)));

This waits up to 10 seconds before throwing a TimeoutException or if it finds the element will return it in 0-10 seconds. WebDriverWait by default calls the ExpectedCondition every 500 milliseconds until it returns successfully. A successful return value for the ExpectedCondition function type is a Boolean value of true, or a non-null object.

This example is also functionally equivalent to the first Implicit Waits example.

Expected Conditions

There are some common conditions that are frequently encountered when automating web browsers. Listed below are a few examples for the usage of such conditions. The Java bindings include convenience methods so you don’t have to code an ExpectedCondition class yourself or create your own utility package for them.

- Element is Clickable – it is Displayed and Enabled.

WebDriverWait wait = new WebDriverWait(driver,10);

WebElement element = wait.until(ExpectedConditions.elementToBeClickable(By.id(“someid”)));

The ExpectedConditions package contains a set of predefined conditions to use with WebDriverWait.

Implicit Waits

An implicit wait is to tell WebDriver to poll the DOM for a certain amount of time when trying to find an element or elements if they are not immediately available. The default setting is 0. Once set, the implicit wait is set for the life of the WebDriver object instance.

WebDriver driver = new FirefoxDriver();

driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);

driver.get(“http://somedomain/url\_that\_delays\_loading”);

WebElement myDynamicElement = driver.findElement(By.id(“myDynamicElement”));