Get started with Selenium 2

End-to-end functional testing of your web applications in multiple browsers

Sebastiano Armeli-Battana March 06, 2012

Selenium is a well-known web application testing framework used for functional testing. The new version, Selenium 2, merges the best features of Selenium 1 and WebDriver (a parallel project to Selenium). In this article, learn how to make the easy transition from Selenium 1 to Selenium 2. Examples show how to use Selenium 2, how to test remotely, and how to migrate your written tests from Selenium 1 to Selenium 2.

Introduction

Selenium is a popular framework for testing the user interface (UI) of a web application. It is an extremely powerful tool for running end-to-end functional tests. You can write tests in several programming languages and Selenium executes them into one or multiple browsers.

Selenium, hereafter called Selenium 1, is not the only tool able to automate functional tests in a browser. WebDriver, created by Simon Stewart (from Google), is a project with a similar goal. To control the browser, WebDriver relies on independent clients using native support. WebDriver offers only Java bindings and does not support the same number of browsers as Selenium 1.

Selenium 1 + WebDriver = Selenium 2

Selenium 1 and WebDriver merged to produce a better product—Selenium 2, or Selenium WebDriver, which was released in 2011. Selenium 2 has the clean and object-oriented APIs from WebDriver and interacts with browsers in the best way possible for that browser. Selenium 2 does not use a JavaScript sandbox, and it supports a wide range of browsers and multiple language bindings. At the time of this writing, Selenium 2 provides drivers for:

- Mozilla Firefox
- Google Chrome
- Microsoft Internet Explorer
- Opera
- Apple iPhone
- Android browsers

With Selenium 2, you can write tests in Java, C#, Ruby, and Python. Selenium 2 also offers a headless driver based on HtmlUnit, which is a Java framework for testing web applications. HtmlUnit is really fast, but it is not realistic as a driver associated with a real browser.

At the moment, Selenium 2 is still under development as minor issues are being resolved. The current version is 2.9. Drivers for Safari and Blackberry should be integrated in the near future.

In this article, learn how to use Selenium 2 to test web applications. Examples show how to remotely implement tests. Learn also how to move your written tests from Selenium 1 to Selenium 2.

Download the source code used in this article.

Getting started with Selenium 2

In this section, learn how to use the Selenium 2 framework for a relatively simple test of a web application. Java is the language for the development environment. You need the selenium-java-<version>.jar containing the Java binding (see Related topics to download). In a Maven project, you just need to include the right dependency in your pom.xml, as shown in Listing 1.

Listing 1. Selenium-java dependency

```
<dependency>
    <groupId>org.seleniumhq.selenium</groupId>
    <artifactId>selenium-java</artifactId>
    <version>2.9.0</version>
</dependency>
```

Now you can start writing tests. The main component in the WebDriver API is the `WebDriver` interface. There is an implementation of this common interface for each browser available. For instance, the `FirefoxDriver` class will take care of controlling Mozilla Firefox. Listing 2 shows how to instantiate a specific implementation inside your test. You can use the test framework that best suits your needs, such as JUnit or TestNG.

Listing 2. `FirefoxDriver` instantiated

```
public class Selenium2Example1Test {
    @Test
    public void test() {
        // Instantiate a webDriver implementation
        WebDriver webdriver = new FirefoxDriver();
    }
}
```

To load the page to test, use the `get()` method. In Listing 3, the GitHub homepage (https://github.com) is loaded into the Firefox instance created previously.

Listing 3. Load the page under test

```
WebDriver webdriver = new FirefoxDriver();
webdriver.get(https://github.com);
```

You can now make some assertions on the page just loaded. Suppose you want to test that the page title is equal to `"GitHub - Social Coding"`, as in Listing 4. WebDriver offers the `getTitle()` method; you can leverage the chosen testing framework to make an assertion.

Listing 4. Assertion on the page title

```
Assert.assertEquals("GitHub - Social Coding", webdriver.getTitle());
```

After you finish the test, it's a good practice to kill the WebDriver instance using the `quit()` method, as shown in Listing 5.

Listing 5. Killing the WebDriver instance

```
webdriver.quit();
```

`FirefoxDriver` is just one of the WebDriver implementations available. You could execute the same test using the ChromeDriver to run the test inside Chrome. Listing 6 shows the full example using the ChromeDriver.

Listing 6. ChromeDriver sample

```
public class Selenium2Example2Test {
    @Test
    public void test() {
        System.setProperty("webdriver.chrome.driver",
"src/main/resources/drivers/chrome/chromedriver-mac");

        // Instantiate a webDriver implementation
        WebDriver webdriver = new ChromeDriver();

        webdriver.get(https://github.com);

        Assert.assertEquals("GitHub - Social Coding", webdriver.getTitle());
    }
}
```

Before instantiating the ChromeDriver, you need to set the `"webdriver.chrome.driver"` system property. This property points to the location of the ChromeDriver file for your OS (see Related topics to download). The example in Listing 6 uses the version for the Mac; versions for Windows and Linux are also available.

To execute the same test inside Internet Explorer, you need to use an instance of the `InterentExplorerDriver` class, as shown in Listing 7.

Listing 7. `InternetExplorerDriver` instantiation

```
WebDriver webdriver = new InternetExplorerDriver();
```

When using `InterenetExplorerDriver`, you might encounter a security issue saying: "Protected Mode must be set to the same value (enabled or disabled) for all zones." To overcome the problem, you can set a specific capability, as shown in Listing 8.

Listing 8. Security capability set for Internet Explorer

```
DesiredCapabilities capability=DesiredCapabilities.internetExplorer();
capability.setCapability(
              InternetExplorerDriver.INTRODUCE_FLAKINESS_BY_
IGNORING_SECURITY_DOMAINS, true);
WebDriver webdriver = new InternetExplorerDriver(capability);
```

To execute a test inside Opera, you need to instantiate the `OperaDriver` class, which was developed directly by Opera. Remember to include the JAR containing the driver inside your project. If you're using Maven, you just need to add the dependency in Listing 9.

Listing 9. `OperaDriver` dependency

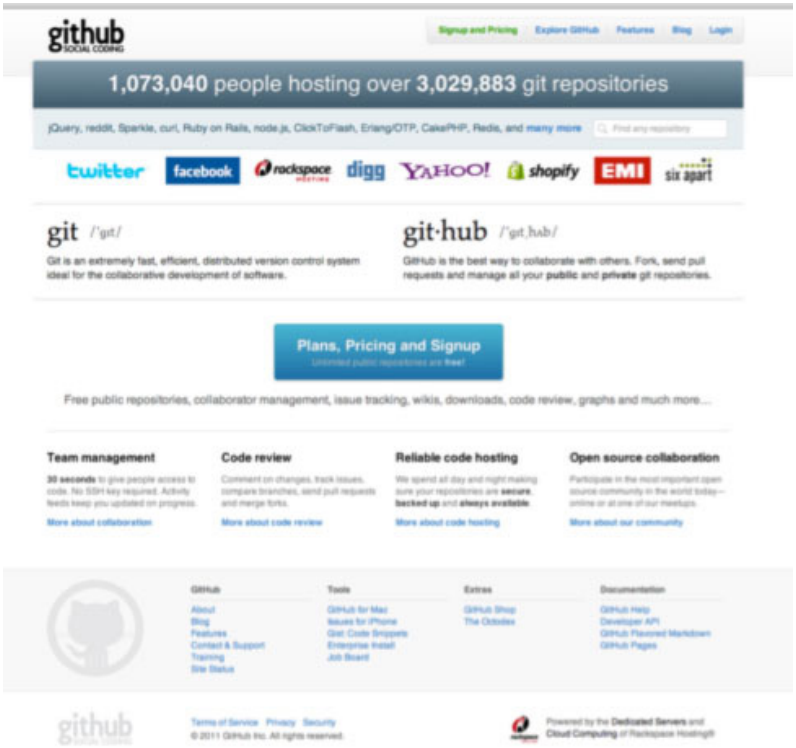
```
<dependency>
    <groupId>com.opera</groupId>
    <artifactId>operadriver</artifactId>
    <version>0.7.3</version>
</dependency>
```

Additional configurations are required to run the tests inside an iPhone or Android browser emulator.

Testing with Selenium 2

With Selenium 2, you can build tests that are more complex than those in the previous section. In this section, you'll test that the top navigation in the GitHub homepage has five list items: Signup and Pricing, Explore GitHub, Features, Blog, and Login. Figure 1 shows the Github homepage.

Figure 1. Github homepage



Review the HTML code behind the top navigation—shown in Listing 10.

Listing 10. HTML code behind the top navigation

```
<html>
<head>
    ...
    </head>
    <body class="logged_out env-production">
        <div id="main">
            <div id="header" class="true">
...
                <div class="topsearch">
                        <ul class="nav logged_out">
                            <li class="pricing">
<a href="https://github.com/plans">Signup and Pricing</a>
</li>
                            <li class="explore">
<a href="https://github.com/explore">Explore GitHub</a>
</li>
                          <li class="features">
<a href="https://github.com/features">Features</a>
</li>
                            <li class="blog">
<a href="https://github.com/blog">Blog</a>
</li>
                         <li class="login">
<a href="https://github.com/login">Login</a>
</li>
                       </ul>
</div>
...
            </div>
            ...
        </div>
       ...
    </body>
</html>
```

You can leverage the WebDriver API to retrieve, from inside the HTML code, the elements you need to test. The `findElement()` and `findElements()` methods return an instance or a list of instances, of the common interface `WebElement`. The `WebElement` interface, in a clean and object-oriented manner, is used for all of the elements in a page. There are different strategies used in the API to locate a UI element. The strategies are represented by the different types of parameters passed into the `findElement()` and `findElements()` methods. Listing 11 shows the different strategies adopted through the different methods applied to the abstract class `By`.

Listing 11. Using the `findElement()` method

```
WebElement element1 = webdriver.findElement(By.id("header"));
WebElement element2 = webdriver.findElement(By.name("name"));
WebElement element3 = webdriver.findElement(By.tagName("a"));
WebElement element4 = webdriver.findElement(By.xpath("//a[@title='logo']"));
WebElement element5 = webdriver.findElement(By.cssSelector(".feautures"));
WebElement element6 = webdriver.findElement(By.linkText("Blog"));
WebElement element7 = webdriver.findElement(By.partialLinkText("Ruby"));
WebElement element8 = webdriver.findElement(By.className("login"));
```

Using one of the strategies in Listing 11, you can start writing the test to retrieve the first elements: the `LI` tags inside the `UL` tag with the `nav` class. Listing 12 uses Xpath (`By.xpath()`).

Listing 12. Xpath

```
List<WebElement> webElements = webdriver.findElements(By
            .xpath("//ul[@class='nav logged_out']/li"));
```

Listing 13 uses CSS selectors (`By.cssSelector()`) to retrieve the `LI` tags.

Listing 13. CSS selector

```
List<WebElement> webElements = webdriver.findElements(By
            .cssSelector("ul.nav li"));
```

At this point, you can make the first assertion on the number of items retrieved, as shown in Listing 14.

Listing 14. Assertion on the number of items

```
Assert.assertEquals(5, webElements.size());
```

The previous step verified that the number of `LI` tags is equal to five.

The next step is to retrieve each anchor (`A` tag) inside each `LI` tag. Listing 15 shows how to get the anchor inside the first `LI`. The tagName (`By.tagName()`) strategy is used for this case.

Listing 15. Retrieving `A` anchor inside the first `LI tag`

```
WebElement anchor1 = webElements.get(0).findElement(By.tagName("a"));
```

In a similar fashion, you can collect all five anchors, as shown in Listing 16.

Listing 16. Retrieving all the anchors inside `LI` tag

```
WebElement anchor1 = webElements.get(0).findElement(By.tagName("a"));
WebElement anchor2 = webElements.get(1).findElement(By.tagName("a"));
WebElement anchor3 = webElements.get(2).findElement(By.tagName("a"));
WebElement anchor4 = webElements.get(3).findElement(By.tagName("a"));
WebElement anchor5 = webElements.get(4).findElement(By.tagName("a"));
```

At this stage, you can verify if the text inside the anchors corresponds to the expected string. To retrieve the text inside a tag, WebDriver provides the `getText()` method. Listing 17 shows the full test method, with the assertions at the bottom of the test.

Listing 17. Complete test

```
@Test
public void test() {
WebDriver webdriver = new FirefoxDriver();
webdriver.get("https://github.com");
List<WebElement> webElements = webdriver.findElements(By
               .xpath("//ul[@class='nav logged_out']/li"));
Assert.assertEquals(5, webElements.size());

    // Retrieve the anchors
    WebElement anchor1 = webElements.get(0).findElement(By.tagName("a"));
    WebElement anchor2 = webElements.get(1).findElement(By.tagName("a"));
```

```
    WebElement anchor3 = webElements.get(2).findElement(By.tagName("a"));
    WebElement anchor4 = webElements.get(3).findElement(By.tagName("a"));
    WebElement anchor5 = webElements.get(4).findElement(By.tagName("a"));

// Assertions
    Assert.assertEquals("Signup and Pricing", anchor1.getText());
    Assert.assertEquals("Explore GitHub", anchor2.getText());
    Assert.assertEquals("Features", anchor3.getText());
    Assert.assertEquals("Blog", anchor4.getText());
    Assert.assertEquals("Login", anchor5.getText());

    webdriver.quit();

}
```

After launching this test, a new Firefox window will open and stay open until all of the assertions are executed.

Testing remotely with Selenium Grid 2

You can either locally or remotely run tests in Selenium 2. To run remotely, the tests need to use a specific implementation of the `WebDriver` interface called `RemoteWebDriver`. You can specify the browser to be run using the `DesiredCapabilities` class. Listing 18 shows an example.

Listing 18. `RemoteWebDriver` and `DesiredCapabilities` classes

```
DesiredCapabilities capabilities = new DesiredCapabilities();
capabilities.setBrowserName("firefox");
capabilities.setVersion("7");
capabilities.setPlatform("MAC");
WebDriver webdriver = new RemoteWebDriver(capabilities);
```

With the `DesiredCapabilities` class, you can specify the browser name, the platform, and browser version. You can also specify many other capabilities the browser can support.

If you want to remotely execute structured tests and run multiple browsers (and possibly various virtual machines), Selenium Grid offers a good solution.

Selenium Grid 2 provides an infrastructure where each node representing a different browser registers itself against a hub. Singular tests will invoke a hub, which is in charge of dispatching each request to the right browser. Hub and nodes can run on different virtual machines.

To test remotely, you'll need to download selenium-server-standalone-<version>.jar on each machine you're going to use. To install a hub on a machine, go into the folder where you downloaded the required JAR and launch the command in Listing 19.

Listing 19. Starting the hub

```
java -jar selenium-server-standalone-2.9.0.jar ?role hub
```

You can access the Selenium Grid 2 console at http://localhost:4444/grid/console, where all the nodes available will be listed. To register a node, simply launch a command, as shown in Listing 20.

Listing 20. Node registered against the hub

```
java -jar selenium-server-standalone-2.9.0.jar
-role webdriver ?hub http://localhost:4444/grid/register -port 5556
```

The command in Listing 20 registers, by default, seven browsers: five Firefox instances, one Chrome instance, and one Internet Explorer instance. You can target a specific browser on a specific port, as demonstrated in Listing 21.

Listing 21. Firefox 7 instance registered on the hub

```
java -jar selenium-server-standalone-2.9.0.jar -role webdriver
-hub http://localhost:4444/grid/register -port 5556 -browser
browserName=chrome,version=14,platform=MAC
```

After registering a few browsers, the Selenium Grid 2 console should look like Figure 2.

Figure 2. Selenium Grid 2 console view

To use the grid, you need to specify inside your test cases the URL of the hub and the browser you want to control. Listing 22 shows where the constructor of the `RemoteWebDriver` class accepts the URL of the hub and an instance of `DesiredCapabilities` defining the specific browser.

Listing 22. `RemoteWebDriver` instantiation

```
DesiredCapabilities capability = new DesiredCapabilities();
capability.setBrowserName("chrome");
capability.setVersion("14");
capability.setPlatform(Platform.MAC);
WebDriver webdriver = new RemoteWebDriver(new URL("http://localhost:4444/wd/hub"),
 capability);
```

In this case, the hub will launch the node associated to Chrome, Version 14 (previously registered in Listing 21).

Selenium Grid 2 is also backward compatible with Selenium 1. You can register Selenium 1 RC nodes (part of the Selenium 1 infrastructure) against the hub, as shown in Listing 23.

Listing 23. Selenium 1 RC node registration

```
java ?jar selenium-server-standalone-2.9.0.jar
-role rc ?hub http://localhost:4444/grid/register -port 5557
```

Migrating tests to Selenium 2 from Selenium 1

If you need to migrate written tests from Selenium 1 to Selenium 2, the transition is quite smooth. The Selenium 1 APIs are kept underneath the new APIs, making Selenium 2 completely backward compatible.

It's easy to convert your test from Selenium 1 into Selenium 2 thanks to the `WebDriverBackedSelenium` class. It takes an instance of a `WebDriver` and the URL under test as parameters, and it returns a Selenium instance. Listing 24 shows the same example as in Listing 16 but using the Selenium 1 API integrated in Selenium 2.

Listing 24. Selenium 1 integrated in Selenium 2

```
@Test
public void test() {
    String url = "https://github.com";
WebDriver webdriver = new FirefoxDriver();
webdriver.get(url);
Selenium selenium = new WebDriverBackedSelenium(webdriver, url);
    selenium.open(url);

// Get the number of LIs
    Number lis = selenium.getXpathCount("//ul[@class='nav logged_out']/li");

    Assert.assertEquals(5, lis.intValue());

    // Retrieve the text inside the anchors
    String anchor1Text = selenium.getText("//ul[@class='nav logged_out']/li[1]/a");
    String anchor2Text = selenium.getText("//ul[@class='nav logged_out']/li[2]/a");
    String anchor3Text = selenium.getText("//ul[@class='nav logged_out']/li[3]/a");
    String anchor4Text = selenium.getText("//ul[@class='nav logged_out']/li[4]/a");
    String anchor5Text = selenium.getText("//ul[@class='nav logged_out']/li[5]/a");

    Assert.assertEquals("Signup and Pricing", anchor1Text);
    Assert.assertEquals("Explore GitHub", anchor2Text);
    Assert.assertEquals("Features", anchor3Text);
    Assert.assertEquals("Blog", anchor4Text);
    Assert.assertEquals("Login", anchor5Text);

    webdriver.quit();
}
```

Selenium 2 has an increased focus on developers. It has a cleaner API than Selenium 1, as evidenced by the `getText()` and `getXpathCount()` method signatures. The Selenium 2 API is

also more object-oriented. For example, you are not allowed to deal with UI element objects, only strings.

Conclusion

Selenium 2 marks an evolution in automated testing inside a browser. It takes the best parts of Selenium 1 and WebDriver and provides tight integration with several browsers. The new API, which brilliantly suits developers' needs, offers an object-oriented approach and provides different patterns for use within your tests. Selenium 2 also:

- Overcomes limitations associated with the same origin policy.
- Offers better support for pop-ups.
- Efficiently controls native keyboard and mouse interactions.

The new version of the Selenium Grid, included in Selenium 2, makes it easy to remotely launch tests. Selenium 2 is backward compatible with Selenium 1, so upgrading to the new version is a snap. Selenium 2 can help ensure that your application works according to the requirements.

Downloadable resources

Description	Name	Size
Article source code	selenium2-gettingStarted	5,243KB

Related topics

- Selenium: Learn more if you're a developer of the Selenium browser automation framework.
- Selenium HQ: If you're developing with the framework, read more about how Selenium automates browsers.
- Selenium 2 documentation: Get more information about Selenium 2.0 and WebDriver.
- The history: Learn about the history of the Selenium project for test automation.
- Selenium wiki: Explore selenium information on a wide variety of topics.
- "Getting started with Selenium 2 and WebDriver" (QA Automation, August 2011): Get practical instructions for starting and stopping Selenium server and instantiating WebDriver.
- developerWorks on Twitter: Join today to follow developerWorks tweets.
- Selenium 2: Download the latest version.
- ChromeDriver file: Get the ChromeDriver for your OS.
- Download IBM product evaluation versions: Download or explore the online trials in the IBM SOA Sandbox and get your hands on application development tools and middleware products from DB2®, Lotus®, Rational®, Tivoli®, and WebSphere®.