

A Look at Selenium

by Grig Gheorghiu

Selenium is a functional and acceptance testing tool for Web applications. The Selenium project originated within ThoughtWorks and is being actively developed there with assistance from the open source community.

Before describing some unique features of Selenium, I'll borrow Bret Pettichord's terminology to say that there are two main classes of Web application testing tools:

- Tools that simulate browsers ("Web protocol drivers") by implementing the HTTP request/response protocol and by parsing the resulting HTML. Examples of such tools are HttpUnit (Java), WWW::mechanize (Perl), and mechanize and webunit (Python).
- Tools that automate browsers ("Web browser drivers") by driving them, for example, via COM calls in the case of Internet Explorer. Examples of such tools are Watir (Ruby), Samie (Perl), Pamie (Python), and JSSh (Mozilla extension).

The main difference between Selenium and other HTTP test tools is that Selenium uses an actual browser driven via JavaScript to play back testing scripts. This means that Selenium has some unique features not available in any other Web test tool:

- It can test client-side functionality implemented in JavaScript.
- It can be used cross-platform and cross-browser, which makes it a perfect tool for browser compatibility testing.

Currently, Selenium supports Firefox/Mozilla on Windows, Linux, and Mac OS X; Internet Explorer on Windows; Konqueror on Linux; and Safari (experimental support) on Mac OS X.

In a nutshell, this is how Selenium is used. The Selenium framework is deployed on the same server that is running the application under test. Acceptance tests are

described in HTML tables, similar to tests written in FIT/FitNesse. They contain commands such as **click** and **type** and assertions such as **verifyValue**. This mini-language is called Selense. Selenium runs the acceptance tests via an actual browser that is driven by a JavaScript engine called the BrowserBot. As a tester, you point your browser to a URL, such as <http://www.yourwebsite.com/TestRunner.html>; at this point, the BrowserBot engine is downloaded in your browser and is ready to process your test cases represented as HTML test tables. The BrowserBot translates the tests written in Selense into JavaScript commands that it sends to the browser. The browser executes the JavaScript commands and runs the application under test in a separate HTML frame. The results of the test execution are highlighted in the test table in green or red, signifying the pass or fail of each test.

Creating and Running HTML-based Tests in Selenium

At the time of this writing, the latest Selenium version is 0.3. It can be downloaded from <http://selenium.thoughtworks.com/download.html>. To install Selenium, unzip the downloaded selenium-0.3.0.zip file, which will create a directory named selenium-0.3.0. Then copy the selenium subdirectory to a location that can be served via your Web server.

To quickly start experimenting with Selenium, I copied the selenium directory to the DocumentRoot directory of my Apache server and then pointed my browser to <http://www.example.com/selenium/TestRunner.html>. (Example.com is a fictitious domain name.) At that point, I could see the default test suite that ships with Selenium, which tests Selenium itself by exercising test commands and assertions. (See Figure 1.)

To run the currently selected test, click the "Selected Test" button in the upper right Control Panel frame. The Selenium test runner will go through each row of the test table, run the command, retrieve the result, compare it with the expected value, and color the row green or red, depending on whether the actual result matched the expected result. To run all the tests in the test suite, click the "All Tests" button in the Control Panel.

I used a Selenium test table to test an

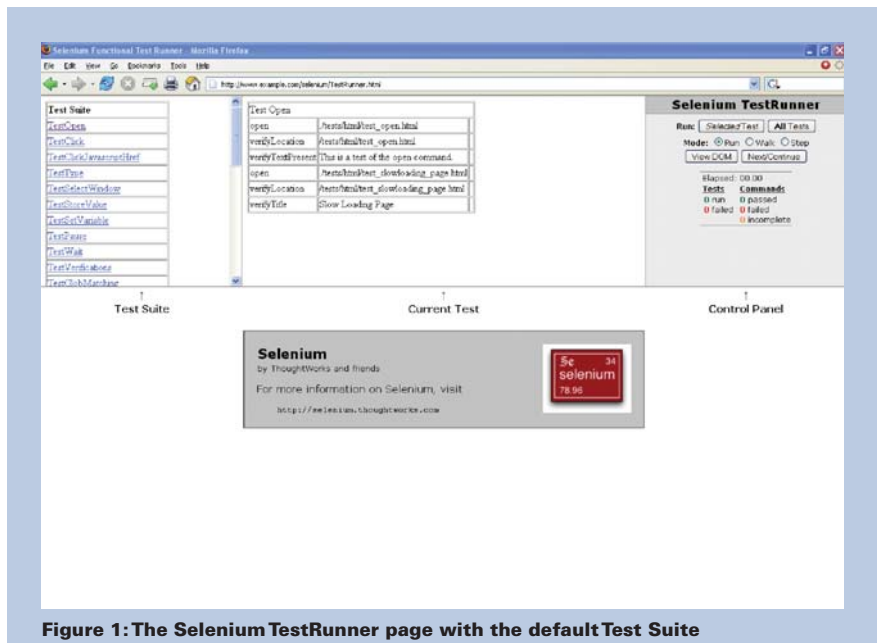


Figure 1: The Selenium TestRunner page with the default Test Suite

out-of-the-box installation of Plone, which is a portal/content management system Web application based on Zope/Python. (See the StickyNotes to view the table.) TestNewUser exercises and verifies the process of a new user joining the Plone site. Similar test tables can be created for any Web application that provides a new user registration process. Specifically, the following functionality is tested in the TestNewUser table:

- Navigate to the main Plone Web page.
- Click the “New User?” button.
- Fill in the registration form with a random user name, a password, and an email address.
- Submit the form and verify that the user is registered.
- Click the “log in” button.
- Verify that the user is logged in.

The actions are fairly self-explanatory and consist of things such as navigating to HTML pages, clicking links and submit buttons, entering text, etc. The verification commands check the actual values of the different HTML elements under test against expected values. In general, HTML elements on the Web pages under test can be referred to in Selenium commands via “element locators,” which can be one of the following:

- Identifiers: the `id` or `name` attribute of the element
- DOM traversal syntax: `document.forms[‘myForm’].myDropdown`
- XPath syntax: `//img[@alt=‘The image alt text’]`

(See the StickyNotes for more information on the syntax of Selenium commands.) To run the TestNewUser test with Selenium, start by creating and then saving the test table in its own HTML file called TestNewUser.html in the tests subdirectory of the Selenium directory. Then create a custom test suite, as an

HTML file, that contains a table like this:

Custom Test Suite
TestNewUser

Save this file as CustomTestSuite.html. In the table above, TestNewUser is a link to the actual TestNewUser.html test file. To have Selenium run your custom test suite, pass a “test” parameter to TestRunner.html by opening a URL: `http://www.example.com/selenium/TestRunner.html?test=../tests/CustomTestSuite.html`.

When you run the TestNewUser test in Selenium (see Figure 2), the bottom frame shows the embedded browser executing the JavaScript commands represented by the Selense commands in the test table. The mid-upper frame shows all the assertion rows colored in green because in this case all the tests passed. The Control Panel in the upper right frame shows a summary of Total/Pass/Fail counts for the current test.

My experience with Selenium has been very positive. But, there are some caveats that are being addressed by the developers:

- HTML frames are not supported (although unofficial patches are available on the mailing lists).
- No import/export from spreadsheets or other formats into HTML tables is

available. (Some tools for handling CSV have been mentioned on the mailing lists.)

Using the Stand-alone Selenium Server

Deploying Selenium on the server hosting the application under test usually is not a problem because testers will primarily need to test the Web application that they own. However, this can prove to be a problem when trying to learn and experiment with the tool. Fortunately, a stand-alone implementation of Selenium is available that provides a work-around for this limitation. Windows XP users can download selenium-server-0.3.0-win-setup.exe installer from the Selenium download page at `http://selenium.thoughtworks.com/download.html`. To experiment with the server, run the provided Perl, Python, and Ruby scripts, which test the Google search functionality.

If you are on a different platform such as Linux or Mac OS X, you still can run the stand-alone server version of Selenium. The XML-RPC server is written in Python and is based on the Twisted framework. (See the StickyNotes for more details on the stand-alone Twisted implementation.)

Driving Selenium Programmatically

The stand-alone Selenium server is one example of an implementation of Selenium that can be “driven” programmatically.

Continued on page 44

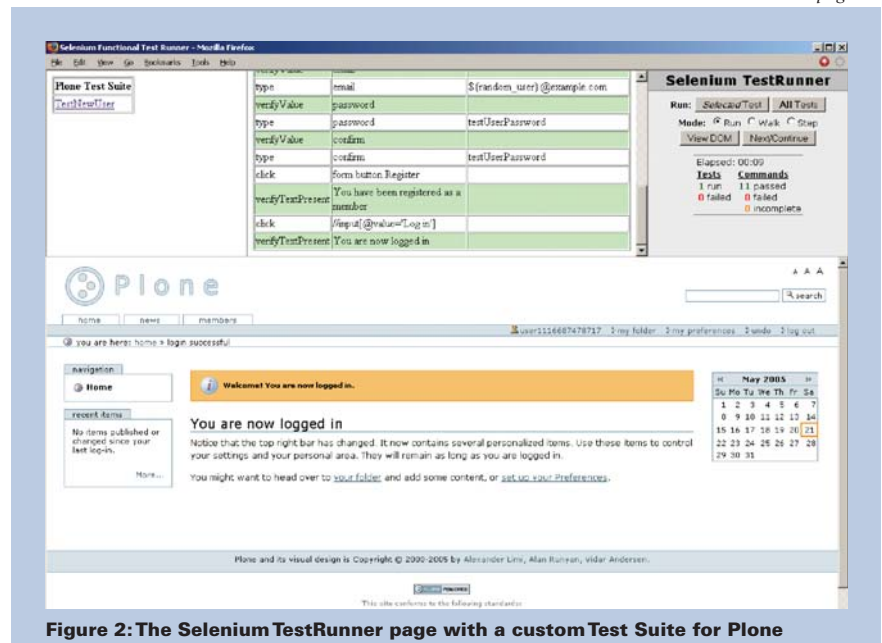


Figure 2: The Selenium TestRunner page with a custom Test Suite for Plone

The server exposes familiar Selenese commands (e.g., verifyTitle, verifyValue, type, etc.) as an API that can be used by test “toolsmiths” so they can apply their programming expertise to writing real programs rather than writing HTML tables. The test scripts communicate with the server via an XML-RPC connection, so they can be written in any language that provides an XML-RPC client library (Perl, Python, Ruby, Java, C#).

Apart from the stand-alone server, Selenium also offers Java, .NET, and Ruby language bindings that can be used to programmatically drive browsers via test scripts written in these languages. The main functionality provided by these bindings is the translation from their specific language into Selenese. The bindings implement the server process that exposes the Selenese API, as well as a queue mechanism used to communicate between the server process and the browser.

Conclusion

Development on Selenium is proceeding at a furious pace, and new features are added almost daily. Support is excellent; the Selenium developers are extremely responsive to all the queries addressed to the selenium-users and selenium-devel mailing lists. There is a budding community around Selenium, and I strongly encourage the programmatically inclined testers to consider joining the growing ranks of Selenium users and contributors. **{end}**

Grig Gheorghiu has worked for the past twelve years as a programmer, research lab manager, system/network/security architect, and most recently as a software test engineer. Grig is a member of the Agile Alliance and of the xpsocal user group. Contact him at grig@gheorghiu.net.

Sticky Notes

For more on the following topics, go to www.StickyMinds.com/bettersoftware

- Example of a Selenium test table
- More on the syntax of Selenium commands
- More on Twisted implementation
- Getting started with Selenium

If the concept of calling testing by another name appeals to you, there may be a way for you to take the sting out of the name in your workplace. I have compiled a table of alternative terms for testing. To come up with substitute names for testing, pick a word from each of the columns in Figure 1. For instance, Unified Acceptance Trials (A2, B7, C3). You can probably think of some additional combinations appropriate to your industry.

Now that there are possible alternatives to using the grating word “test,” we still have to deal with what to call the situation that exists when the expected result of running the test (experiment, study, examination, etc.) does not agree with the actual result. Common words currently used to describe this situation are “bug,” “defect,” “failure,” “fault,” or the somewhat less judgmental “incident.” However, it may be helpful to look at some additional alternatives. (See Figure 2.) These names also can be useful for annotating “defect” reports and findings using automated tracking tools. For instance, we could report a Potential Anomaly (A1, B1). For medical equipment a *Correctus minimus* may be appropriate, and for software used in biological applications, the *Hemiptera heteroptera* (Latin for “bug”) has a nice ring.

Perhaps you can add to the Don’t-Call-It-A-Bug table, as well.

With a little creativity, those of us who specialize in this craft with the negative connotation can have some fun coming up with more palatable terminology. In the not-too-distant future, we may hear a conversation as follows:

A	B
1. POTENTIAL	1. ANOMALY
2. SUSPECT	2. CORRECTNESS
3. TENTATIVE	3. BELIEVABILITY
4. PSEUDO	4. CERTAINTY
5. UNRESOLVED	5. CONVERGENCE
6. UNSTABLE	6. CORRELATION
7. IRREGULAR	7. CORRECTITUDE
8. ARBITRARY	8. CORRESPONDENCE
9. RANDOM	9. CENSURE
10. FUZZY	10. RESULT
11. BIASED	11. PRESENTATION

Figure 2: The Don’t-Call-It-A-Bug table

“The software was so good that the developers felt it to be without bugs and not necessary to test. We did, however, perform some Rapid Requirement Proofs and found a number of cases of Irregular Convergence and Biased Believability. These findings were handled by the developers as trivial enhancements, which have now been fully implemented, and we are ready to ship after performing the mandatory Independent Observational Scoring.”

Good luck. **{end}**

Gregory Pope has more than thirty years of experience applying common sense to developing software in the commercial and government sectors. Greg has held positions from programmer to CEO. He has been an invited keynote speaker for numerous international symposiums (STAR, Software Testing Automation, Quality Week, ITEA-DoD) and remains active in presenting papers and articles. Currently Greg works for the University of California at the Lawrence Livermore National Laboratory where he is Software Quality Engineering Group Leader and works with code teams doing advanced simulation on large parallel super computers.

Want searchable access to every article ever published in **Better Software**?

The **StickyMinds.com PowerPass** gets you there.

Visit StickyMinds.com/powerpass to learn more.