Automating Functional Tests Using Selenium

Antawan Holmes and Marc Kellogg
Digital Focus
antawan.holmes@digitalfocus.com, marc.kellogg@digitalfocus.com

Abstract

Ever in search of a silver bullet for automated functional testing for Web Applications, many folks have turned to Selenium. Selenium is an open-source project for in-browser testing, originally developed by ThoughtWorks and now boasting an active community of developers and users. One of Selenium's stated goals is to become the de facto open-source replacement for proprietary tools such as WinRunner. Of particular interest to the agile community is that it offers the possibility of test-first design of web applications, red-green signals for customer acceptance tests, and an automated regression test bed for the web tier.

This experience report describes the standard environment for testing with Selenium, as well as modifications we performed to incorporate our script pages into a wiki. It includes lessons we learned about continuous integration, script writing, and using the Selenium Recorder (renamed IDE). We also discuss how long it took to write and maintain the scripts in the iterative development environment, how close we came to covering all of the functional requirements with tests, how often the tests should be (and were) run, and whether additional automated functional testing below the GUI layer was still necessary and/or appropriate.

While no silver bullet, Selenium has become a valuable addition to our agile testing toolkit, and is used on the majority of our web application projects. It promises to become even more valuable as it gains widespread adoption and continues to be actively developed.

1. Introduction

Automating functional tests is one of the traditional problems facing software development projects. Whereas automated unit testing has achieved deep traction, in agile and non-agile projects alike, functional testing frequently is still done manually. In recent years, tools such as Fit and FitNesse [1] have helped make it easier to automate functional testing of software applications. However, web applications have long remained difficult to test because of multi-tiered architecture, multiple browsers, and web technologies such as JavaScript. Some teams have chosen expensive solutions such as Mercury Interactive's WinRunner or Rational Robot (with mixed results). Recently the open-source tool Selenium, originally developed by ThoughtWorks, has gained attention as a possible silver bullet for the problems of automated testing for Web Applications.

We have been using Selenium on five different web application projects for about a year. Our teams consisted of 4 to 12 developers and 1 to 2 business analysts, with the latter role responsible for writing our Selenium tests (along with other responsibilities). One of the projects used a .NET architecture, while the four others were J2EE applications.

1.1. Background

Since our teams practice an agile methodology, we are committed to writing acceptance tests before beginning development on stories. Passing these tests signals completion of a story, so we write functional tests that capture these acceptance criteria. For a web application, a functional test could be simply that a user manually navigates through the application to verify the application behaves as expected.

But since automating a test is the best way to make sure it is run often, we try to automate our functional acceptance tests whenever we can. And our holy grail is to be able to write the tests before development, so that the development team can have a runnable verification that the story is complete.

Our team came to Selenium after using several other tools to automate web testing. We tried the open-source tools Canoo WebTest and HttpUnit, but found them to be insufficient, as they could not handle most instances of in-page JavaScript. The JavaScript problem is solved by the proprietary tool QuickTest Professional (from Mercury Interactive), which offers

Proceedings of AGILE 2006 Conference (AGILE'06)
0-7695-2562-8/06 $20.00 © 2006 IEEE

record-and-play of test scripts and also runs tests directly in a browser. However, we found that most of our recorded scripts in QTPro would break after small page changes, so a significant amount of script maintenance was necessary. In addition, it was not obvious that we would be able to write QTPro scripts before development. As a result, we turned to Selenium – it showed the promise of easy-to-write scripts that were relatively easy to maintain and which could be written before the code.

1.2. Selenium

Selenium is a web testing tool which uses simple scripts to run tests directly within a browser. It uses JavaScript and iframes to embed the test automation engine into the browser. [2] This allows the same test scripts to be used to test multiple browsers on multiple platforms.

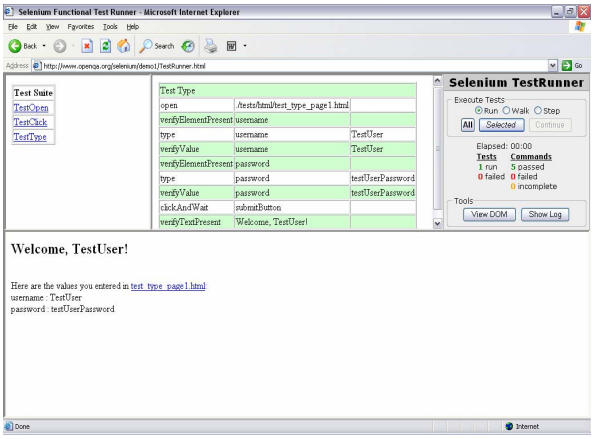

Figure 1: Selenium inside Internet Explorer

Selenium gives the user a standard set of commands such as **open** (a URL), **click** (on an element), or **type** (into an input box); it also provides a set of verification commands to allow the user to specify expected values or behavior. The tests are written as HTML tables and run directly in the browser, with passing tests turning green and failing tests turning red as the user watches the tests run.

Because Selenium is JavaScript-based and runs directly in the browser (the user can see the test running), it overcomes some of the problems encountered by users of HttpUnit or Canoo WebTest, particularly problems related to testing JavaScript functionality.

Two additional useful tools are also available for use with Selenium:

1. the Selenium IDE (originally called the Recorder), which allows users to navigate their applications in Firefox and record their actions, forming tests; and,
2. a "Remote Control" server which allows users to write tests directly within the programming language of their choice -- thus enabling conditional logic within tests, try/catch blocks, and other powerful functionality available only in programming languages.

1.3. Getting started

Setting up Selenium is easy, although there is a catch. The basic installation of Selenium must be hosted by the same web server as the Application Under Test (AUT). This restriction is due to the fact that JavaScript has built-in security against cross-site scripting. [3] After installation, the user simply needs to begin writing and running tests.

2. Writing good tests

Selenium tests are not difficult to write. Because Selenium allows the identification of elements using the browser's DOM object, the test can be written using specific identifiers of the necessary element, such as name, id, or xpath:

Table 1: Input data for a Selenium Test

type	name=theField	Text to submit
clickAndWait	id=SubmitButton	
assertText	xpath=//h1/span	Success!

This test might be written for an HTML page as simple as:

```
<html>
<input name="theField">
<input id="SubmitButton" type="submit">
<h1><span>Success!</span></h1>
</html>
```

When the test is run, each command is highlighted as it executed, and the assert steps turn red or green to indicate success or failure. After the whole test is complete, the test is marked as red or green in the Suite.

2.1. Keeping tests self-contained

Selenium naturally supports a Suite of tests, and tests for a certain story or iteration can be grouped together in a Suite and run sequentially. But for

flexibility and maintainability, we strove to keep our tests as independent and self-contained as possible. This allowed us to move tests around and delete them, as well as refactor tests mercilessly. We frequently included one test inside many others to reduce code duplication, and we used SetUp and TearDown tests throughout our suites in order to get the application to the desired state for testing one piece of functionality. In fact, we found that our test-writing style became:

1. Write the tests before development;
2. After development, get them to green; then,
3. Refactor as mercilessly as possible.

By refactoring our tests, we reduce duplication of test code and increase the maintainability of the test suite.

2.2. Writing our own extensions

Some would argue that the top benefit of an open source project is that it can be licensed for free; cost is undoubtedly a top reason, but it is the availability of source code that makes open source what it is. [4] As an open source project, Selenium allows end users to write and share extensions or other code modifications, even ones that are project-specific.

We took advantage of this freedom by writing some custom functions to make some of our tests easier to write. We created a special Selenium command that selected a value from a JavaScript-based tree after storing the parent element into a reusable variable. Selenium is built to handle such user extensions, as it merely requires adding the JavaScript code to one Selenium file.

Using such custom functions also allowed us to have more readable tests.

3. Putting Selenium tests into the wiki

After writing enough tests to have a successful implementation for our projects, we found that there were a few recurring pain points:

1. Keeping the tests organized was a nuisance;
2. Writing the tests in HTML was unnatural; and,
3. Using variables across multiple tests was tricky. We realized early on that using variables would prevent us from having to change every test when a field ID would change, but it was tricky enough that we weren't doing it.

We decided that we could address all of these issues by leveraging the power of our project wiki. A wiki is a collaborative web environment where any user can change the pages. It is our company's standard project center – where we capture all of our meeting notes, requirements, stories, and tests. A wiki page can be changed by any user and is open to the whole team, including the stakeholders.

By including our Selenium tests in our wiki, we gave them the highest possible profile (everyone can see them, track them, and run them) and also were able to integrate them with our other automated functional tests – a byproduct of using FitNesse as our wiki.

3.1. FitNesse

FitNesse is an automated test tool, wiki, and web server all rolled into one application. Its stated goal is to enable customers, testers, and programmers to collaboratively **learn what their software should do**, and to automatically compare that to **what it actually does do.** [5] Because any user can modify a page and FitNesse keeps past versions of pages, all of the members of the team, including the customers, use it as a communication tool and everyone is an owner of the content.

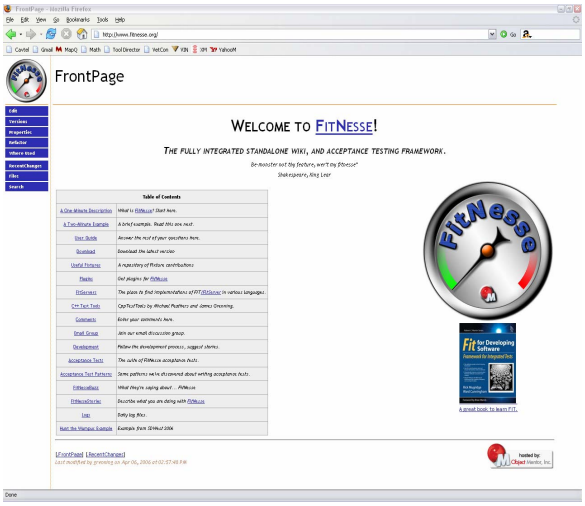

Figure 2: FitNesse Wiki

3.2. Integrating Selenium and FitNesse

Selenium's built-in functionality is to read the tests from a designated URL, so we pointed it to our wiki suite. There were a couple of technical challenges, such as authenticating with the wiki, but in the end we were able to run our wiki suite and tests from Selenium's TestRunner.

Because the wiki was centrally located, a developer or tester was also able to run their own instance of Selenium and the application, using the central test bed, and thus run the Selenium tests against their own code at any time.

4. Continuous integration

After every developer checks in code, we run all of our unit tests and integration tests to make sure that the check-in did not break the build or the tests. Because it's important not to break the passing acceptance tests as well, we wanted to add our Selenium tests to the continuous integration build.

For our project's integration builds, we used CruiseControl. By setting up a deployment of our application and Selenium on our integration machines, we were able to include Selenium tests into integration and nightly builds.

4.1. CruiseControl

CruiseControl is a framework a for continuous build process. It incorporates Ant and source control tools to allow immediate feedback on the build and test status of a project's checked-in code. [6]

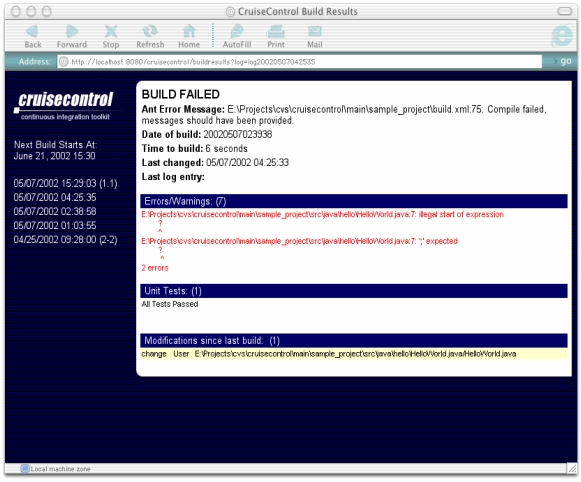

Figure 3: A CruiseControl Build Result

Because our Selenium suites consisted of many tests and involved many application-level transactions, the entire suite tended to take a very long time to run after several iterations (for one project, after 12 iterations – 24 weeks – the test suite took four hours to run). As a result, it was impractical to include the Selenium tests as part of our continuous integration builds. However, we do run the tests as part of a nightly build process, and we even run the tests against instrumented code in order to report on test coverage using Cobertura.

4.2. Cobertura

Cobertura is a code coverage tool for Java projects. It calculates the percentage of code accessed by tests. [7]

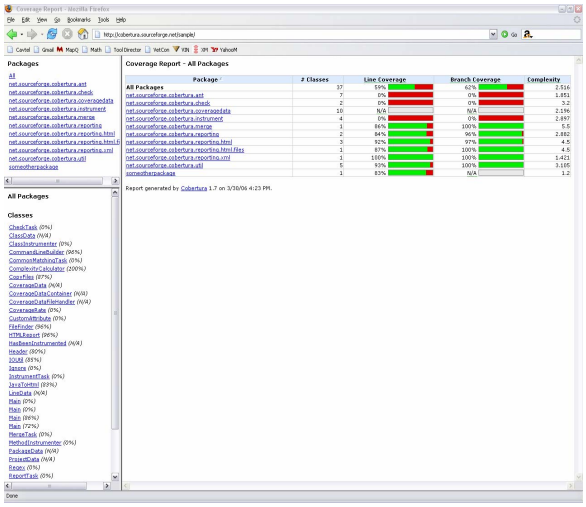

Figure 4: A sample Cobertura report

By including Selenium tests in the coverage analysis for our projects, we had a very good sense of which code in our projects was not being exercised by the application's various tests. For some of our projects, developers were very interested in writing a Selenium test in order to exercise code that they didn't feel was appropriate to test with JUnit tests. We ran the Selenium tests against the nightly build, and included the results in the coverage reports, tracking the coverage analysis every morning.

By incorporating Selenium into our nightly builds, with coverage included, they were a natural topic for our stand-up meetings every day, and everyone on the team saw value in them. On the other hand, because our Selenium tests were *only* running nightly against the newest builds, they didn't provide the immediate feedback that they otherwise might have. Below, we discuss some ways that we'd like to leverage their value even more.

5. Lessons learned

5.1. Selenium is open-source

We began using Selenium at version 0.5. Even today, it is only at a 0.6 release, and is under active development. Thus, certain stability and completeness issues were to be expected. One of our team members became an active contributor to the user forums, and steady monitoring of the user and developer forums for patches and usable extensions paid off. Our recommendation is to remember that Selenium isn't (or

at least doesn't have to be) considered a black box. By understanding, modifying, and extending its functionality, we were able to make our tests more readable and to handle multiple-step actions with one command.

5.2. Value to the team and the end users

The greatest value to our teams was that Selenium tests build up a regression test suite for the web layer of the application through the iterations of the agile process. This allowed developers to refactor the front-end code without fear of breaking the previously passing acceptance criteria for the application.

While our automated acceptance tests carried great value, there were instances where a failing Selenium test gave a false alarm. One of the most fundamental problems of GUI testing is that the team can make a change that keeps the application completely correct (even "the same" to the end user), but break an automated test. A simple example of such a situation is renaming a button.

Because of this inherent fragility to front-end testing, some team members tended to dismiss a breaking Selenium test as "the test's fault." This assessment was correct enough times to make these tests less valuable as immediate feedback than, say, a unit test. However, any given test did tend to stabilize after an iteration, i.e. two weeks. The result was that everyone did value the tests and the feedback, but we would try to make sure that a failed test really was "the code's fault" before our 10:00 AM standup meeting.

In addition to having value throughout the development lifecycle, Selenium tests were extremely valuable artifacts to hand off to the client at the end of iterations. Because our wiki directly linked stories with Automated Acceptance tests, and because it was very easy to demonstrate the Selenium tests really were testing the application (by running and watching!), the client was able to track the Selenium tests throughout the project and see that they were all green at the end of the iteration. And the fact that the tests carried so much regression value gave the whole team comfort that the rapidly-changing application was still stable.

5.3. Putting tests into suites

As mentioned above, our whole Suite of Selenium Tests took, in some cases, many hours to run. We found that keeping all of the tests in one Suite was the easiest way to manage the tests, but that frequently we wanted to group the tests in other ways (by story, by iteration, by functionality set). FitNesse supports Suites of Suites, but it was not clear how to get Selenium to do the same.

Another feature we found desirable is the ability to "tag" a test with multiple tags (e.g. "Iteration 3", "User Story A", "Functionality Set 7") and then be able to easily see a Suite made up of tests with a given tag. In general, members of our team did not run single tests in an ad hoc manner, but instead waited for the report from the nightly build. However, if we were able to more easily identify the appropriate tests, a developer could run just a small subset of them and get feedback more quickly than waiting for the nightly build.

5.4. Test first?

Because Selenium tests are easy to write, a tester or analyst can write the shell of a Selenium test very quickly without knowing what the implementation will be. Although we hoped that we could code to these tests, the test would seldom turn green after development. The reasons for this were usually minor: a field wasn't naturally identified by the name the tester chose, or the test command used needed to be **clickAndWait** instead of just **click**, etc. As a result, we did not usually require that the developer code to the test, and our process of writing the test before development (for its specification value), but getting the test green immediately **after** development, emerged.

5.5. IDs, xpath and performance

The first project we used Selenium for was an ASP.NET project, which automatically assigns IDs for every element on the page. However, when we switched to JSP-based applications, the IDs were only present when the code specified it.

Selenium supports several different techniques for identifying page elements, including names, IDs, xpath and more. For the most part it was not difficult to find a unique identification for an element, but occasionally parsing the HTML was tedious. Some tools we found valuable for this purpose were the DOM inspector, XPath checker, and the Selenium Recorder. The Selenium IDE is even better at this task. However, frequently these tools do not find the "easiest" representation of an element (for example, it might find an xpath expression for an element instead of an id). This would frequently contribute to longer-running tests, which was one of the most significant problems we encountered using Selenium. In general, using xpath expressions tends to make tests take much longer to run, especially in Internet Explorer. So we used IDs

or names where they existed, and went out of our way to add them into our JSP code when possible.

Due to the processing and memory needs of browsers running Selenium, as our suites grew larger and contained more tests, we needed to have a more robust environment. For example, one suite running on a P2 – 600 MHz machine with 512MB of RAM took over 11 hours to run. Upgrading our test environment to a P4 - 3 GHz machine with 1GB RAM took the time required down under 3 hours.

5.6. Ability to test all the requirements

We have used a number of testing tools in the past and Selenium is among the best, if not the best, at being able to perform every browser action that a user can perform, including such events as onMouseOver and onKeyPress. In addition, because Selenium allows users to write their own extensions, it is easy to create custom actions that do sophisticated manipulations. There are some JavaScript-restricted actions, such as downloading or uploading files, that are not supported, but even these actions have some workarounds for testing in some browsers.

Selenium doesn't perform database tests or other back-end tests, so we do need to do such tests using FitNesse or other techniques. One recent addition to the Selenium world, Selenium Remote Control, addresses some of these issues by allowing the user to write Selenium tests in other programming languages, thus leveraging the power of Selenium within more traditional automated test beds.

6. Summary

While Selenium is no silver bullet for the problems facing the web application tester, it handles many of the problems very well and doesn't add significant new ones. The active and growing community of users and developers indicates that it is filling a need for a variety of different user types, and widespread adoption seems imminent. Selenium is certainly worth evaluating for anyone looking to add a powerful web testing tool to their toolkit.

7. Acknowledgements

We would like to thank Jeff Patton for his valuable feedback and support in reviewing this paper throughout its evolution. Additional thanks to Jeff Nielsen for his encouragement and advice.

8. References

[1] FitNesse (Web Site: http://www.fitnesse.org)
[2] Selenium, (Web Site: http://www.openqa.org/selenium)
[3] Selenium Frequently Asked Questions, (Web Site: http://wiki.openqa.org/display/SEL/FAQ)
[4] David DeWolf, private correspondence.
[5] FitNesse One Minute Description, (Web Site: http://www.fitnesse.org/FitNesse.OneMinuteDescription)
[6] CruiseControl Home Page, (Web Site: http://cruisecontrol.sourceforge.net)
[7] Cobertura, (Web Site: http://cobertura.sourceforge.net)