Web Testing Made Easy

Marc Guillemot
Independent Consultant
Bachstrasse 24
53 332 Bornheim, Germany

mg@internetzky.de

Dierk König
Canoo Engineering AG
Kirschgartenstrasse 7
4051 Basel, Switzerland
+41 (0)61 228 94 44
dierk.koenig@canoo.com

Abstract

In this paper we describe WebTest, an Open Source tool for automated testing of web applications. In particular we will show how to quickly create tests that shine with excellent maintainability and runtime performance as well as perfect integration in the application development cycle.

Categories and Subject Descriptors
D.2.5 [**Software engineering**]: Testing and Debugging – *Testing tools*
D.2.9 [**Software engineering**]: Management – *Software quality assurance*

General Terms Documentation, Reliability, Verification.

Keywords Automatic acceptance test, web application, test driven development, change control

1. Introduction

The rise of Extreme Programming (XP) and Test Driven Development (TDD) have popularized automated testing practices. JUnit and its derivates have become a standard for unit tests making them an integral part of the program sources. Automated end-to-end tests are crucial as well but can easily become cumbersome when done with inappropriate tools. WebTest [1] is an open source project born in 2001 in a large bank after the collapse of different test automation projects. Created and further developed by people writing and maintaining large test suites, it focuses on productivity features needed in the everyday life of automated testers.

2. FIRST TEST

2.1 Overview

WebTest tests are commonly expressed as Ant [2] scripts in XML format. Each action, here called a step, has its own tag that looks like <invoke .../>, <verifyTitle .../>. WebTest provides over 100 different steps which support not only testing HTML pages (including JavaScript – even AJAX – and applets) but also XML content (including Web Services), PDF and Excel spreadsheets and emails. Combined with Ant's built-in capabilities, this allows thorough testing of all kinds of web applications. Additionally WebTest is easily extensible for any developer to facilitate testing of application specific issues.

Running WebTest tests causes XML reports to be generated. These can be further transformed into nice HTML documents showing all the details of the test execution including each page visited. Excellent reporting is one of WebTest's highly-regarded strengths.

2.2 Installation

Provided you have Java installed, installation is as simple as downloading the latest version and unpacking it. WebTest has practiced continuous integration since its origin[1] and each successful change performed by a committer generates a new version. As a consequence major releases are mostly artificial and the latest build is always the best choice. WebTest is written in Java and therefore works on almost any machine, from the developer's workstation to the integration build server.

2.3 Writing a Test

It's a good practice to separate test suite definition from the tests for simplicity. A minimal test script could look like:

```
<project name="example" default="test">
<target name="test">
 <webtest name="test OOPSLA site">
  <steps>
  <invoke url="http://www.oopsla.org/2006"/>
  <verifyTitle text="OOPSLA 2006"/>
  <setInputField name="searchword"
                          value="WebTest"/>
  <clickButton label="Search"/>
  <verifyText text="Total \d+ results found"
                            regex="true"/>
  </steps>
 </webtest>
</target>
</project>
```

2.4 Running the Test

In order to execute the tests, simply call the command script that comes with the WebTest distribution. Noteworthy is that the execution doesn't require a graphical user interface. This allows multiple tests to be run simultaneously or allows to continue working while the tests are running. Test execution results are saved in an XML file that can be simply transformed for visualization. Figure 1 shows an extract of the overview generated with the standard layout.

[1] it was perhaps the first Open Source project to use public continuous integration to build itself

OOPSLA'06 October 22-26, 2006, Portland, Oregon, USA.
ACM 1-59593-491-X/06/0010.

No	Result	Name	Steps
1	✓	test OOPSLA site	5 / 5

Figure 1. Results overview.

Contrary to JUnit tests, it is usually not enough just to see a green or a red check mark as result of a test since more information is needed. The report contains the details of the execution too: they show the exact execution scenario and each page received from the server allowing the tester to reproduce the test manually when needed (Figure 2).

	Step	Parameter
✓	**invoke** Resulting page	**url** http://www.oopsla.org/2006
✓	**verifyTitle**	**text** OOPSLA 2006

Figure 2. Test execution details (extract).

3. SUSTAINABLE TESTS

Automated tests make most sense when they can be reused through multiple releases of an application to ensure that no regressions occur. To that end it is necessary to be able to adapt the tests to the changes without writing them from scratch again (typical strategy for capture / replay tools). The maintenance effort rises with the amount of changes. It can only be kept manageable by following good engineering practices. We will see here some practices that help increasing tests life expectancy.

3.1 Test Accurately

Web pages change over time and page tests need to cope with these changes. Some changes like minor layout adaptations may be insignificant and should go unnoticed through the test suite.

On the other hand, all significant deviations from the specified behavior must certainly break the test run. Consequently, the art of writing high quality, meaningful, and sustainable tests means specifying test cases such that they cover the guaranteed behavior but leave the most possible leeway for insignificant changes within the application under test. Typical means of achieving this are by using regular and XPath expressions within the tests. For instance using the verifyXPath step with a path like /html/body/div/div[2]/table/tbody/tr[3]/td[1] is more likely to fail due to insignificant changes in page layout than something like //td[@id='total'].

3.2 Write Testable HTML

A frequent cause of problem lies in incorrect HTML code. HtmlUnit [3], the underlying "browser" used by WebTest, is not as forgiving as popular browsers that accept really badly formed HTML content. What may appear as a weakness is often a strength as it helps to discover and fix problems early. Bad HTML is likely to cause problems later, when displayed by some browsers or by accessibility tools like voice browsers and screen readers.

Even in well-formed HTML code, it may be difficult to precisely specify the content part that is used by the test. As seen in the previous example it is profitable to put information in the HTML code (here an id for a td) to ease test formulation. This requires close interaction and collaboration between testers and developers as advised by agile development practices.

3.3 Organize

Test automation is a software development activity and its rules apply. Particularly, it is important to take particular care when choosing names, to avoid duplication, and to use parameters instead of hard-coded values where appropriate. WebTest makes it possible to follow these rules by extracting shared test sequences into groups, xml entities, and macros.

3.4 Profit From Ant

Modern Java IDEs like Eclipse have built-in support for Ant scripts, and thus WebTest, for editing and executing. This allows for instance to easily start tests from within the IDE, to jump to entities or macro definitions.[2]

Being Ant-based allows WebTest to smoothly integrate in continuous integration tools like Cruise Control. This is very important as tests become more valuable the more frequently they are run.

3.5 Going Further

WebTest is built upon well known open source libraries (HtmlUnit, commons-httpclient, rhino, jaxen, and many others) and allows the tester to use them transparently through its steps. Nevertheless it is sometimes necessary to gain finer control over these underlying layers. A specific script step allows to perform small interactions using any Bean Scripting Framework (BSF) [4] compliant language. Custom steps can be registered to perform very specific tasks what is mostly done in Java.

Among the script languages, Groovy [5] is particularly interesting. In addition to the usual integration options, Groovy offers the slick alternative of specifying WebTest tests in terms of object-oriented scripts leveraging all language features while directly using all steps provided by WebTest. This is the kind of functional tests that are automatically generated by Grails [6].

4. CONCLUSION

WebTest is a mature solution to intensively test web applications that fits particularly well in the application development process. Its declarative approach makes the tests highly readable allowing to read them as a specification. WebTest is regularly improved thanks to the contributions of testers that have to cope daily with tests. These improvements make testing easier and more powerful without to loose backward compatibility thanks to WebTest excellent self test coverage.

References

[1] Canoo WebTest, http://webtest.canoo.com

[2] Apache Ant, http://ant.apache.org

[3] HtmlUnit, http://htmlunit.sourceforge.net

[4] Bean Scripting Framework, http://jakarta.apache.org/bsf/

[5] Groovy, http://groovy.codehaus.org

[6] Grails Functional testing, http://grails.codehaus.org/Functional+Testing

[2] a child project of WebTest named WebTestClipse has been recently started and should bring additional dedicated functionalities in Eclipse