CHAPTER 9

Testing a Site with ApacheBench, JMeter, and Selenium

Automated testing—the process of having a computer run a scripted set of commands and checking the results—has come a long way in recent years. In the 1980s, the widespread adoption of graphical user interfaces (GUI) put a damper on the promise of automated testing. It was difficult and expensive to create tests that could accurately and reliably navigate a GUI. Small changes in position or functionality of the screen elements would often break existing tests and substantial resources would be required to correct the tests so they worked under the new conditions. This made testing more expensive than it was worth. The growth of the Web and standardization of HTML-based web applications has breathed new life into the field of automated testing.

Not only is it much more cost-effective to create tests (because even GUI components are accessible as web page elements through code), but it has become essential with the growing complexity of web-based applications. At the lowest level, automated testing can verify the baseline functionality of a web site or web application. As a site changes, additional tests can be added to the group of tests used on the site.

A thorough set of tests can do an excellent job of peering into the dark corners of an application, ensuring that the more rarely used features are tested. Further, load testing can tell you the maximum number of concurrent users that can address your site without considerable access slowdown. Load testing can also provide early warning by showing you which parts of the site will break under a heavy load so the problems can be corrected before real users are subjected to them.

If you're an individual or member of a small team that maintains a Joomla web site, automated testing can be critical. You probably don't have time to check every facet of the application each time you make a change. Further, if you don't visit your site or parts of your site regularly, you can set up scheduled automated tests that can ensure that the site is functioning properly and hasn't faulted because of system failure or a hacker attack.

Until recently, automated testing tools have been extremely expensive, often running into the thousands or even tens of thousands of dollars. The efforts of the open source community have resulted in an increasing number of industrial-grade tools available for free.

In this chapter, you'll learn how to test your Joomla site using three important open source tools:

- *ApacheBench*: ApacheBench is a basic load-testing tool that you can use to quickly and easily to put a load on your server and examine the site performance at various levels.
- *Selenium*: One of the most popular testing tools, Selenium is a suite of tools that provides everything from browser-based GUI test creation (using Selenium IDE) to cross-browser distributed execution of tests on various browsers and operating systems.
- *Apache JMeter*: A professional-grade tool that you can use for most aspects of testing—including directly accessing your MySQL database connection and creating graph reports of increasing user loads on a target server.

When these three tools are used in conjunction, you can run almost every primary test against your Joomla site.

■ **Note** Selenium performs a type of testing known variously as "acceptance testing" or "black box testing" or "functional testing" on a web application. This type of testing simulates a real user or customer performing operations on an application and tests the application as a whole—including the user interface. Other types of testing such as "glass box testing" or "white box testing" are tests that are performed at a lower level on the code itself such as the unit testing. The Joomla framework itself is increasingly adding unit tests that you can run yourself with PHPUnit (see instructions at http://docs.joomla.org/Running_Automated_Tests_for_the_Joomla_CMS).

Testing Overview

Before you dive into the world of testing, it is useful to review a few concepts and some of the terminology used in the field. When listening to professional testers (or reading articles they've written), it can sometimes seem like they're speaking a different language. Because the results of testing are so fundamental to standard application use, however, a small amount of explanation can make most of the seemingly indecipherable text as plain as day.

To begin, the most basic question in testing is: what are we trying to test? Tests are broken down into a few basic areas. Although there is some overlap (and some tests cover multiple areas), the primary areas include functional testing, integration testing, regression testing, unit testing, and performance testing.

Functional Testing

As it sounds, functional testing tests the functions of a system. For example, when the New Task button is clicked, is a new task created by the software? When the print option is selected, does the page print? These high-level tests are relevant to Joomla installations in the areas of preventing broken links and making sure extensions are executing as they should.

Because functional testing is an attempt to simulate user interaction, most web-based functional testing needs to be executed within a browser. Many Joomla sites use JavaScript or Flash to add additional site functionality. You can only test portions of the site that use this technology if the code (JavaScript or Flash/ActionScript) is actually executed.

■ **Note** At a professional level, there are a growing number of so-called "headless browsers" that allow QA automation to run tests on JavaScript and other code without requiring the overhead of actually launching a graphic-intensive browser application. Google is using their V8 JavaScript engine (http://code.google.com/p/v8/) to crawl sites with this technology. This technology is evolving rapidly, so if you're interested, you should perform a web search for `headless browser` to find the most robust technology in this field.

Integration Testing

Integration testing assures that multiple components work together properly. Many of the problems with Joomla sites result from the failure of integration testing. When the new shopping cart was installed, was the site search component tested to make sure it could see items held in the product catalog? If a new template is set as the default, do all of the existing pages display properly?

The modular nature of Joomla makes it particularly susceptible to integration failure. Site administrators regularly add new templates, modules, and components. All major parts of the site should be subjected to integration testing when additional Joomla extensions have been added to make sure the new technology is working properly with the existing site technology.

Regression Testing

These tests make sure that the site hasn't "regressed" after the addition of new features. For example, did installing the new animated drop-down menu module break the JavaScript of the tooltip module that was working before? Regression testing seeks to make sure that new changes don't damage existing systems.

Regression testing is generally performed during stages of integration testing as the integration of new technology is the most common reason for previously working parts of the system to fail. This form of testing can be tedious because it requires you to constantly review the same subsystem again and again—subsystems that generally WON'T be broken by the new technology. For high-volume sites, however, this form of testing is perhaps the most important because of the ripple effects that can occur if one piece of code breaks another.

■ **Note** Regression problems can have a tremendous impact on revenue. On the site where I work (www.ehow.com), a developer accidentally pushed a new module to production that added an erroneous closing tag to the HTML code. This problem prevented a set of Google advertisements from appearing on the page. If regression testing had not caught the problem, the flaw would have been live on the site over the weekend and caused untold loss of revenue. For smaller sites, a similar problem might not even be caught until a monthly AdSense check arrives with a fraction of the normal payment.

Unit Testing

These are programmer-oriented tests that are written to test a very small part (or unit) of the programming code. While unit testing is very popular in some areas of the computer world (such as the Java world), Joomla developers haven't widely adopted the practices of unit testing. Increasingly, the developers of Joomla have been adding unit tests to the source code so they can test the core code before each release.

Unit testing has proven most effective in particular specialized areas such as frameworks (the Joomla platform, for example), driver creation, and certain automated processes. Most Joomla developers, even extension developers, will find the maintenance of tests prohibitively time-consuming. For this reason, unit testing won't be detailed in this book. If you want to learn unit testing, Joomla is a great place to start! You can access all of the Joomla unit tests on GitHub at https://github.com/joomla/joomla-platform/tree/staging/tests.

Performance Testing

You can test the performance of the site or application in many different ways and from many perspectives. Performance testing is both difficult and very important. This type of testing will show you what the user will experience in terms of load times and site speed. Performance is even becoming important for SEO reasons as a slow site will be spidered much less often by the search engine crawlers, which means that it will take more time to detect and index new content.

One of the most common forms of performance testing—load testing—will determine the performance of a site when many simultaneous users access it. However, this may not be as straightforward as it seems. If the performance is bad, these questions immediately pop up in an attempt to determine the bottleneck:

- Is the server itself slow (a slow processor) or is the server starved for resources (not enough RAM, slow drive performance,and so forth)?
- Is the Internet connection slow?
- Is the firewall creating a fail-down condition?

You must also examine a good performance test closely. It could mean the site is performing well, but it could alternatively mean that the test . . .

- . . . wasn't representative. For example, the performance test may have been run on a local network and is not a representation of the experience of a normal user accessing the site from across the country.
- . . . received a "false positive" where the test failed but appeared to return a good performance result. For example, if the test checks for server responsiveness, it might get the fastest response when the server can't find the requested web page (a 404 error). If this error response is not being checked, the overall performance figures will be contaminated.
- . . . didn't check for the proper things. For example, it is very common to use a JavaScript library such as SWFObject to load and play Adobe Flash content. However, most test software (including Apache AB and Apache JMeter) don't execute Flash code. This means that the Flash content is invisible to the standard load tests in these applications. Therefore, if your home page consists of 130K of HTML code, but a 3-megabyte Flash movie, the standard load tests will not reflect the slow loading experience of an average viewer accessing your site for the first time.
- . . . addressed overly optimal conditions. Most advanced web applications (including Joomla) provide some form of caching of pre-rendered content. The cached content also generally has an expiration date or time (often as short as 15 minutes). That means most load-testing tools, if the cache remains enabled, will not provide representative processor load or response speed. All of the simulated user requests for the same page will return the same cache file, which may not accurately simulate the average user who will request an uncached page. This is a very common difficulty when creating performance tests because caching occurs throughout the system—including temporary tables on the database server.

Examining performance testing results carefully ensures that the information on performance that is being sought is effectively checked in the test.

Understanding Testing Terminology

The terminology used by testers provides effective shorthand with which they can communicate common ideas that need to be covered very often. Some of the terms you are likely to come across include:

- *SUT*: The System Under Test is the system that is being tested. For a Joomla site, this generally refers to the actual server where the site is deployed.
- *AUT*: The Application Under Test is the application or site that is being tested. For Joomla users, the AUT will mean the Joomla application and extensions regardless of whether it is deployed on a staging or production server.
- *Test coverage*: The amount of a desired area that is accessible by a particular test or set of tests. For example, if a Selenium test for menus checks all of the top-level menus (of which there are 12), but doesn't test any of the left menus (of which there are 24), the test coverage for menus is said to be 33%—only 12 of the 36 menus are covered by testing. This doesn't mean that 33% of the tests pass or fail; it specifically refers to the amount of the particular area that are *covered* by tests. All of the 36 menus may work or all of the 36 menus could fail and the test coverage number would not change.
- *White box/Glass box testing*: Testing of a system by testers that know the code and how it is supposed to operate. For example, when a programmer performs white box testing, the inner workings of the routines are known to the tester. This form of testing has the advantage that the developer understands all of the features of the AUT and so can perform more in-depth testing of

particular functionality. The drawback is that the developer knows how it is "supposed" to work, meaning that a user who doesn't know the inner workings will often try something unplanned and therefore expose a break in the system that will be invisible to the white box tester.

- *Black box testing*: When the tester doesn't understand the inner workings of the SUT or AUT. The tester treats the system as a black box and expects when specific input is applied, the functionality or output will be as expected.
- *Test harness (or test fixture)*: The testing tools, testing data (input and output), and configuration are referred to collectively as the harness.

The more you work with testing systems, the more you will come across this terminology. Knowing these basics will help you to read the QA testing literature and have a conversation with a QA engineer with a basic shared understanding of the concepts to be discussed.

We've now covered enough generalities about testing. You're ready to get started. Most web testers begin testing a site with ApacheBench because it is very easy to use and provides some really useful statistics that show the basic performance of a SUT.

Using ApacheBench for Performance Testing

Installing the Apache web server will also install the excellent (although basic) load-testing tool known as ApacheBench (ab). It can be very useful in quick and simple load testing and creating a basic benchmark for system performance. This can be particularly helpful when you're trying to determine your production web server's connection performance.

On Windows, you'll likely find `ab` in your `bin\` path, like this:

```
C:\Program Files\Apache Software Foundation\Apache2.2\bin\ab.exe
```

On Linux, you'll find it in a path like this:

```
/usr/bin/ab
```

On Mac, your path might be:

```
/usr/sbin/ab
```

or

```
/applications/mamp/library/bin/ab
```

The `ab` program is run at the command line and includes the following command-line parameters:

-A auth-username:password

-c Set the number of concurrent users that the `ab` will simulate accessing the site

-C Add a cookie to the page request

-e Write output data to a comma-separated-value (CSV) file

-k Enable HTTP keep alive feature to perform multiple requests on the same HTTP session (default setting for keep alive is disabled)

-n Number of requests to attempt; this defaults to a single request that is not useful for any type of benchmarking

-p Send an HTTP POST command using the post-data found in the specified file

-q Quiet mode will suppress the progress count reporting

-t Specifies the time limit for benchmarking; the default is no limit

-w Output results as a HTML table

To test with 100 connections and 15 simulated simultaneous users, you can execute this command line:

```
ab -n 100 -c 15 http://www.example.com/
```

It will return statistics like this:

```
Server Software:        Apache/2.2.3
Server Hostname:        www.example.com
Server Port:            80

Document Path:          /
Document Length:        2018 bytes

Concurrency Level:      15
Time taken for tests:   5.565972 seconds
Complete requests:      100
Failed requests:        0
Write errors:           0
Total transferred:      232200 bytes
HTML transferred:       201800 bytes
Requests per second:    17.97 [#/sec] (mean)
Time per request:       834.896 [ms] (mean)
Time per request:       55.660 [ms] (mean, across all concurrent requests)
Transfer rate:          40.60 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        9   80 498.1      10    3919
Processing:    63  366 1088.9     70    5066
Waiting:       62  346 1089.9     68    5064
Total:         73  446 1179.2     80    5075
```

You can examine these statistics to determine the basic response speed of the web server. The most important basic number will be the *Requests per second* or RPS. This number tells you what number of concurrent requests the site can handle under the load you placed on it. In this case, the site can perform with almost 18 simultaneous requests or if 1,000 users accessed your site in the same minute, they would have a reasonably speedy page response. As a general guideline, 8 RPS is too slow, 12 RPS seems average for GoDaddy Joomla sites, popular news sites run at about 17 RPS, and sites with full caching solutions (such as Varnish or Akamai) can run as fast as 40 RPS.

Keep in mind this is the RPS of the page itself—none of the sub-files such as images, CSS, JavaScript, Flash,and so forth that are on the page were loaded in these requests. That means you have a baseline of what the server can do (including executing the Joomla PHP code), but it doesn't reflect what a full page load would look like.

You also want to look closely at the maximum connection times. If you run the test once and the maximum times are high, and then run the same test again and the numbers are much lower, this probably means a cache is being used in the second run. For Joomla testing with ApacheBench, it is a good idea to run a set of tests with caching turned off so you can get a worse-case scenario of your web site response.

■ **Tip** If you want a more in-depth explanation of examining the various statistics output by Apache AB, I've posted an article on my web site (www.joomlajumpstart.com/general-joomla/13-joomla-resources/63-testing-joomla-with-apache-ab) that examines the numbers in detail and how they relate to Joomla testing.

Using ApacheBench for performance testing isn't something that you generally perform too often on a Joomla site unless you have a very high traffic growth. You generally want to perform such analysis when you first launch your site so you can correct any clear problems with the server or server setup. Thereafter, periodically running Apache AB on your site will give you an idea if something is going wrong.

■ **Note** For testing of page load speed, Joomla administrators would benefit most by optimizing the template or module-loading aspects of the page. There are two popular browser plug-ins: YSlow from Yahoo (http://developer.yahoo.com/yslow/), which is available for most browsers, and Page Speed from Google (https://developers.google.com/speed/), which is available for Firefox and Chrome. These tools analyze a page load and tell you not only what is making page load slow from the front-end perspective, but they each make suggestions for page optimization.

Introducing the Selenium Suite

Selenium is actually a number of different programs that work in concert to provide various aspects of automated web testing across many platforms. At the time of this writing, there were four primary tools in the Selenium suite:

- *Selenium IDE*: A browser plug-in that's used to record and edit test scripts. Use it to group sets of tests into a test suite. Selenium IDE can also replay scripts and save scripts in a variety of language formats (such as Java, and so on) for execution on Selenium Server. At the time of this writing, the IDE is only available for the Firefox browser.
- *Selenium Core*: A PHP/JavaScript web application that can run individual test scripts and test suites in any browser (including Google Chrome, Internet Explorer 6 and above, Firefox 2 and above, Safari 2 and above, Opera 2 and above, and others). Because of security restrictions on web servers, the Selenium Core must execute on the web server of the site being tested.
- *Selenium Server (until recently known as Remote Control (RC))*: A Java-based application server that can open and close browser windows and execute scripts automatically. This allows test suites to be fully automated and tested across a variety of browsers with no required user intervention.
- *Selenium Grid*: A controlling application for Selenium Server that allows coordinated parallel control of multiple distributed instances of RC. You can distribute test execution across multiple machines to scale up the testing process.

Although you can deploy the full Selenium suite (http://seleniumhq.org/download/) at even large organizations, most Joomla webmasters are likely to mostly use the IDE and Core for browser compatibility testing. Organizations using Joomla will gain the most benefit from the regular use of the RC and Grid applications.

Selenium IDE

Selenium IDE is a browser-based plug-in that, in its simplest form, acts as a macro recorder for web sites (you'll learn how to set it up in a later section). It can record the actions of a user as a web site is browsed. Each step of the interaction with the web site (such as clicking a button or entering text into a field) is stored as an entry in the Selenium test. In Figure 9-1, you can see the Selenium IDE that has stored the procedure for a simple search of the Google web site.

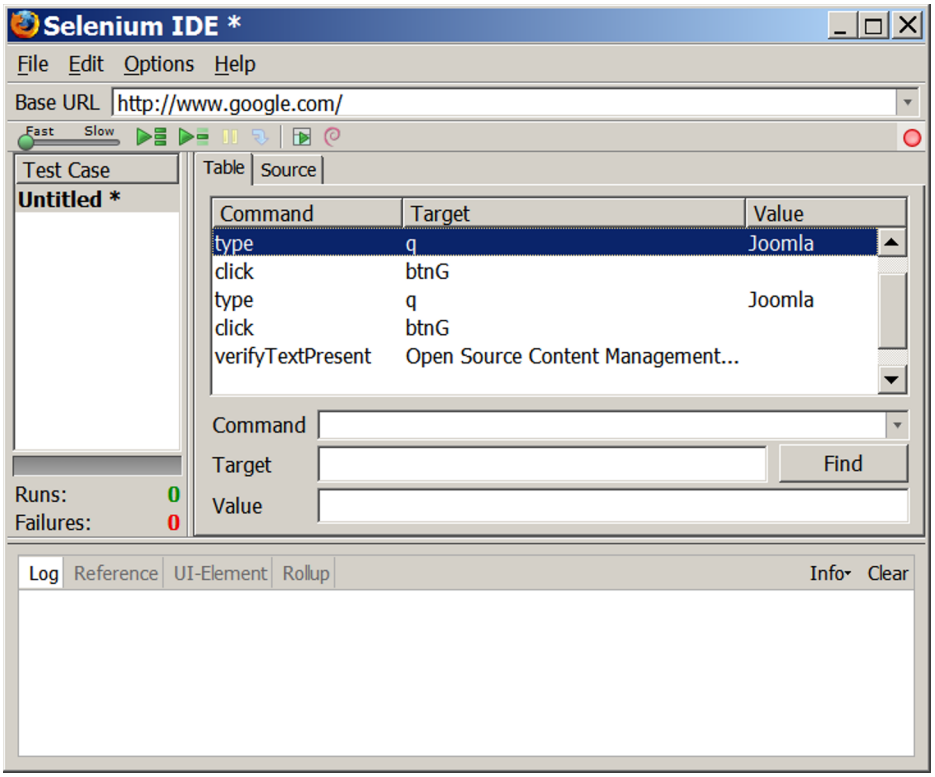

Figure 9-1. *The Selenium IDE records each event that occurs in the browser as a separate line entry*

For simple tests, you can simply save the recording and later manually run it to re-test the operation of a part of the web site. Selenium, however, is much more powerful than a simple macro recorder. It lets you:

- Test elements on the web page to make sure the events are causing the expected results. For example, a login test might enter a username and password and Selenium can check text on the web page to make certain the login was successful.
- Embed JavaScript in tests to allow the execution to dynamically supply or store conditions. For example, the Joomla login system will not allow duplicate usernames so a static script will fail the second time it's executed. Using JavaScript, the new username can be randomly generated, allowing the script to execute properly each time.

- Set up test suites (groups of tests) that will be run sequentially so you can test the entire site or multiple conditions in one run. The tests are also timed, so the execution speed and performance for multiple executions can be determined.
- Automate the execution of tests through a variety of languages such as Java, Python, and PHP so you can run the tests automatically on a regular basis with the results reported via email.

By default, tests are stored in a standard formatted HTML table for easy editing, modification, combination, and splitting. You can easily convert tests to any of the other languages that Selenium supports. Additionally, the conversion process is a straightforward set of JavaScript functions so you can write a converter for nearly any language in a small amount of time.

Selenium Core

The Selenium Core has a fairly rudimentary interface since it is designed specifically to execute tests that have been recorded on the Selenium IDE (which has limited browser support) on nearly every popular browser.
In Figure 9-2, you can see that the primary screen duplicates a number of the buttons and the general display of the main IDE interface. It can run one test, all tests, push the test run, and report the log and error findings.

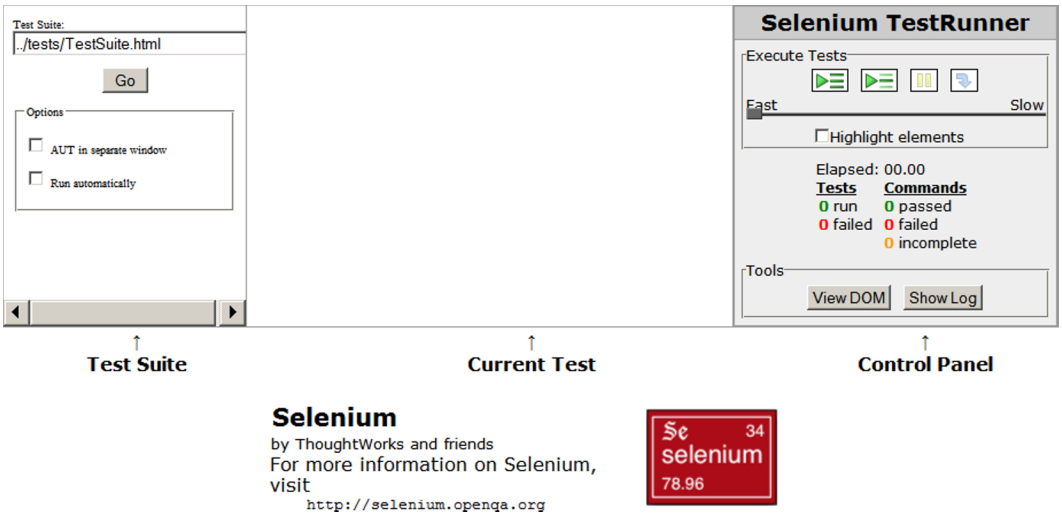

Figure 9-2. *The Core interface resembles a boiled-down version of the IDE execution interface*

The real power of the Selenium Core lies in its ability to execute on any browser. While the IDE can only run in Firefox, you can literally run the Core in any browser that can access it and execute JavaScript. This means that you can check even mobile browsers such as Mobile Internet Explorer and iPhone's Safari for site compatibility.

Therefore, Selenium Core can replay the test scripts in nearly all browser types including Internet Explorer 6 and above, Mozilla Firefox, Safari, Opera, and others. If the browser supports JavaScript and access to the Document Object Model (DOM), it can very likely run the Selenium suite.

■ **Tip** There is a fair chance that you will have to make some modifications to your tests to allow them to run properly on the different browsers. For example, Internet Explorer seems to render linefeeds differently than Firefox (even on the same Windows machine), so a `verifyText` command that passes on a multi-paragraph block of text in one browser may fail on another. Typically, slight modifications to the test will make it accurately determine a pass/fail.

Unlike the IDE, the Core must be located in a browsable path on the web server that hosts the web site. Due to security restrictions, this also means that the tests run on the Core can only address pages on that server.

Selenium Server (Formerly Selenium Remote Control or RC)

The Selenium Server system is a Java-based application server that can execute various browsers and run the Selenium tests within each browser instance. Additionally, RC can execute tests scripted in a variety of languages, which means you can use the programming system that you are most comfortable using (PHP, Python, Java, and so forth) to execute tests. The real power in this lies in the ability to make programmed tests that include loop logic and use `if...then` data execution control.

The RC system is the part of the Selenium suite that lends itself best to automated execution of the test suites. This can be useful to system administrators not only to ensure that the site is functioning properly at all times, but provides an excellent notification service in case the site is attacked and penetrated by a hacker. Even a basic title assert and a content check will warn you that your site was defaced. Comprehensive content checking can help detect subtle modifications.

Selenium Grid

The Selenium Grid allows you to distribute execution of Selenium Server tests across multiple computers. This means that you can execute a set of tests that would normally take a long time to execute simultaneously in multiple places, allowing magnitudes of timesavings.

Because of the resource requirements for setting up a grid (multiple computers with enough RAM to run scores of browser instances), most Joomla sites won't need to deploy a Selenium Grid for testing. However, if your site were to achieve the traffic loads that the central Joomla site experiences, the Grid should certainly be investigated.

Using the Selenium IDE on a Joomla Site

Joomla sites are perfect for testing using Selenium. Because they have a standardized presentation through the Joomla menuing system, you can create general tests that are used in a variety of tests in the suite.

For example, if you wanted to test all of the registration features of the Joomla site, you could make a test that performs and verifies a test login. You may use one script to test to make sure the desired modules are on particular pages. You could use another script to log out the test user. You could use this script to bracket any of the registration execution of the site. A third script may check the metadata information in the page headers to make sure appropriate description and OpenGraph Facebook tags are present.

What makes Selenium so compelling for Joomla site administrators is the ease with which it can be set up, configured, and used. From the moment you begin to download the browser plug-in to executing your first test should be under 10 minutes.

Installing Selenium IDE on Firefox

At the time of this writing, the Selenium recorder, or Selenium IDE, is only available for Firefox. Although the recorder is limited to a particular type of browser, you can play back and test the recorded tests on nearly any browser (using Selenium Core), so tests can confirm basic browser compatibility across a wide range of browsers.

In this section, you'll learn how to install Selenium on the Firefox browser. Installing Selenium is as easy as installing any other Firefox add-on. It is probably easiest to go directly to the Selenium site to get the latest version.

Alternately, you can use the Firefox Add-ons manager by selecting the Tools ➤ Add-ons menu that will display the Add-ons window. Click on the `Get Extensions` link at the bottom-right corner that will take you to the Firefox add-ons page at this URL (for the English-language Firefox site):

```
https://addons.mozilla.org/en-US/firefox/
```

In the search box, type "Selenium" and you should see the Selenium IDE entry. Click on the `Add to Firefox` button and the Selenium IDE will download and install. You will need to restart Firefox to enable that add-on. Optionally, you can go directly to the Selenium site (`http://seleniumhq.org/projects/ide/plugins.html`), download the XPI file, and double-click the file to install it on your browser.

Once it is installed, Selenium IDE should appear as an option in the Tools menu in Firefox. Selecting that menu option will display the Selenium window where you can open, create, and execute tests.

Recording Tests

Recording a test is a simple process with Selenium's IDE. You might start out recording an example of a popular web site that you know doesn't heavily use JavaScript (which can potentially get in the way of Selenium). In this example, we'll start with Google.

Navigate to the web site you want to test and copy the URL out of the browser address bar. Select the Tools ➤ Selenium IDE option to display the Selenium window and paste the URL into the Base URL box at the top of the window. Your window should match the one shown in Figure 9-3.

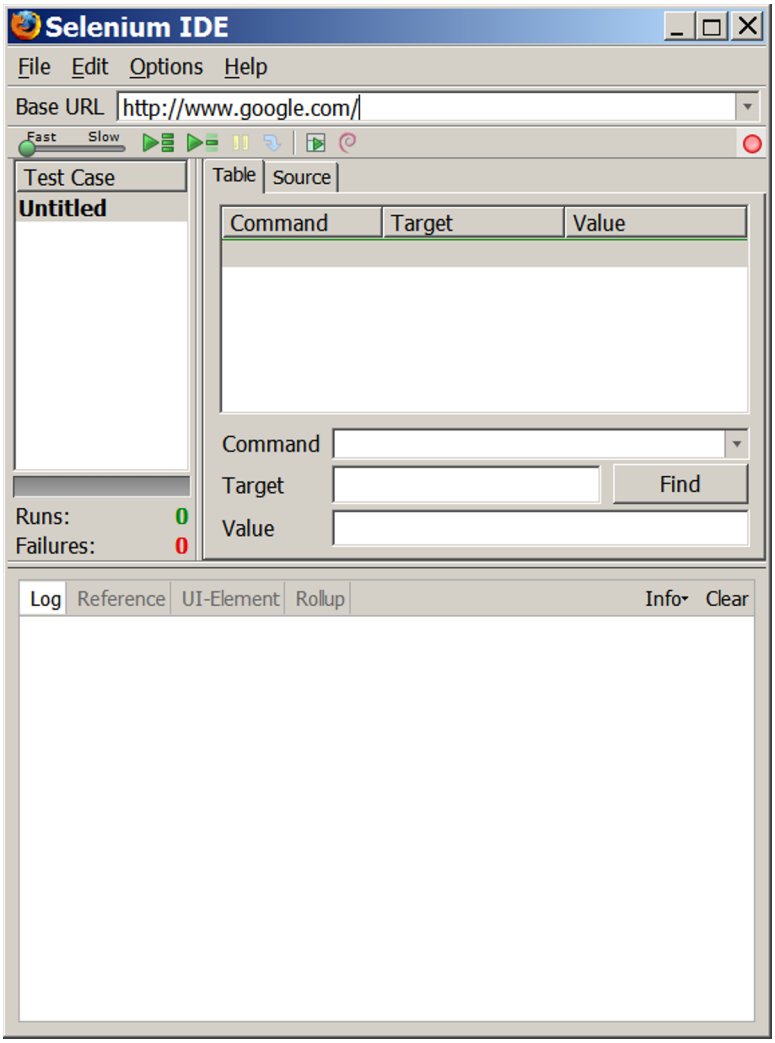

Figure 9-3. *The Base URL should match the URL of the site you want to record*

Click on the red record button in the right corner if it isn't already recording. You can mouse over the button and a tool tip will appear stating `Now Recording` if it is recording. Any actions related to the web browsing will now be recorded as line items in the IDE.

Let's start with something very simple. Search for `Joomla` on Google. Once the search page appears, you should see the first entry is the Joomla.org site.

There will be a site description, such as `Open Source Content Management System. [Open Source, GPL]` directly below the link to the site. Move the mouse pointer to the beginning of this text, click and hold down the mouse button, select the text of the description, and release the mouse button. Move the pointer back over the selected text and click the right mouse button (or, on the Macintosh, hold down the Ctrl key and click). This will display the context menu that has some Selenium-added options.

As you can see in Figure 9-4, there are a number of options at the bottom of the menu related to Selenium commands. You will see that one option states verifyTextPresent followed by the text you selected. Click on this option to add the text verification to the script.

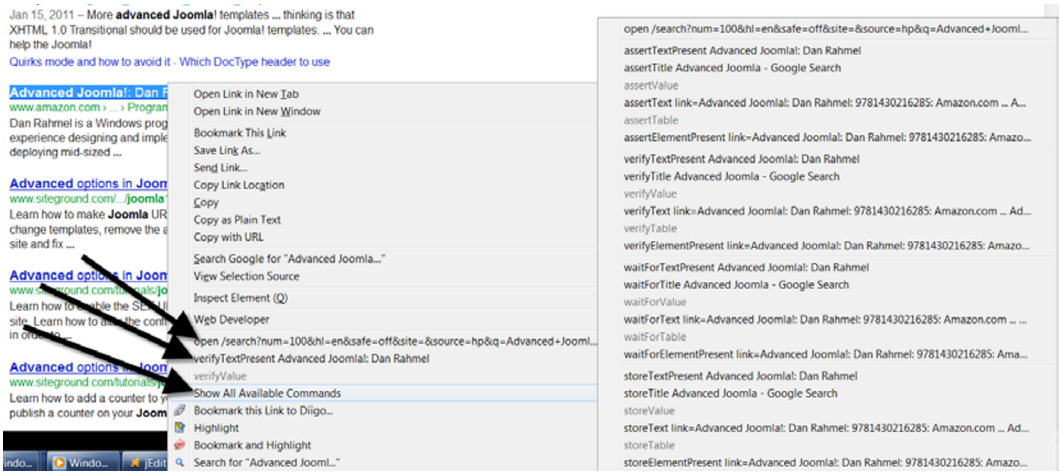

Figure 9-4. *The context menu in Firefox has additional options created by the Selenium add-on*

When you return to the Selenium window, you should see all of the line items recording the various execution items. Click on the red record button to stop the recording. On the toolbar, you will see two play buttons: on the left for `Play entire test suite` and on the right for `Play current test case`. Click on the `Play current test case` and watch the browser window (see Figure 9-5).

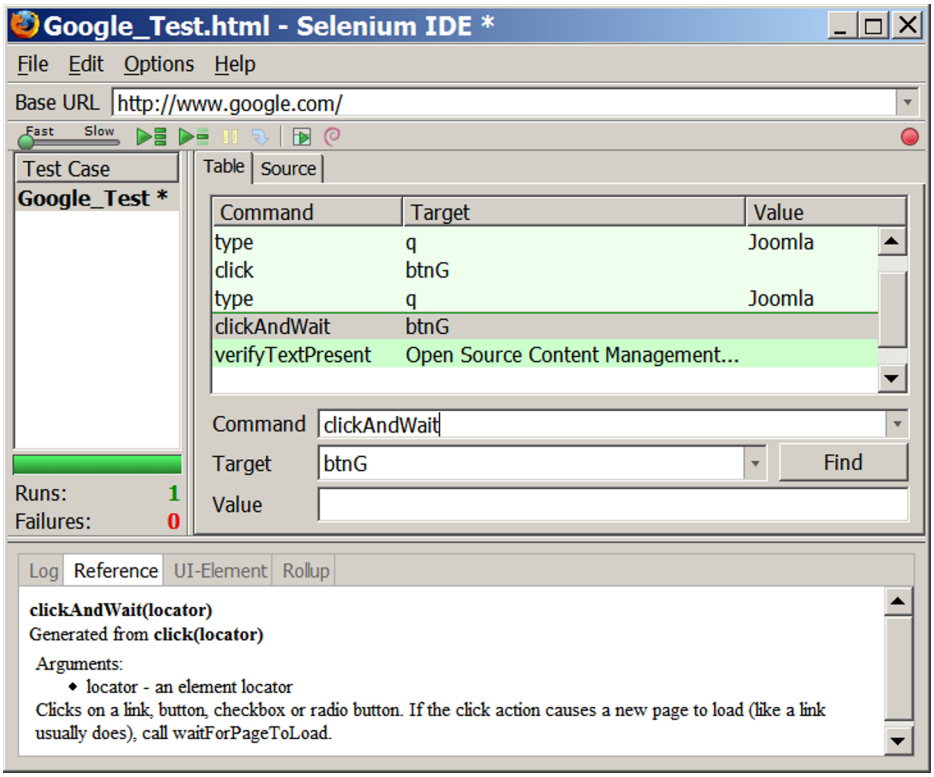

Figure 9-5. *Click on the Play current test case option to execute the current test*

You should see Selenium address the main page of the IDE, enter the search text, click the button, and attempt to verify the text on the page. If everything ran properly, the log pane at the bottom of the window should have displayed all of the execution steps and all of the line items at the top should now have a lime green background color.

■ **Note** You may have to clear the tab in Firefox before clicking the play button if you already have the search page open.

If the final text verification failed, you will see red error text in the log pane that reads `[error] false` and the verify item will have a red background. If you did get this error, you may be puzzled because the text seems to be on the browser screen.

The error was likely caused by a timing issue with the execution of the script. You may have noticed that the script executed very quickly. In the next section, we'll make a modification to the script to make this a non-issue. For now, let's do something a little different.

In the top-left corner of the window, you may have noticed a slider with a green oval that is labeled Fast at one end and Slow at the other. This slider determines the speed of the execution of the test. By default, it is set to the fastest speed.

Click and drag the slider all the way to the right so it is set to execute the slowest. Then click the `Play current test case` to run the test again. Did it work correctly that time with all entries green and no errors? The problem earlier was caused by the test executing too quickly so the text verification was attempted before the page loaded. Instead of slowing down the script, however, we can make Selenium wait until the page returns.

Modifying the Script

You will find that you will often need to modify a script after the initial recording so it will execute and check for the items on the page that you want. This is especially true when you have Ajax implemented on the site. Scripts for executing Ajax will typically need pause commands added to the execution to allow the Ajax requests time to return data from the server.

In the last script, when you clicked on the Google Search button, the IDE most likely recorded a `click` command. When you attempted to re-execute the script, the search didn't have enough time to return the page and load into the browser, causing the subsequent test failure. You can make a simple modification to the script to make it pause execution until the page has returned to the browser.

In the Selenium IDE window, the list of executed commands shows the sequence of execution. Directly above the text verification, you should see a row that has the value `click` in the Command column and `btnG` in the Target column. This is the command you want to change. If you change the command from `click` to `clickAndWait`, script execution will wait until the page has loaded completely before continuing the execution sequence.

Click on the line with the `click` command and the three text boxes below the command list should display the values of that entry. Change the command to read `clickAndWait`, as you can see in Figure 9-6 and set the execution speed slider back to the fastest position.

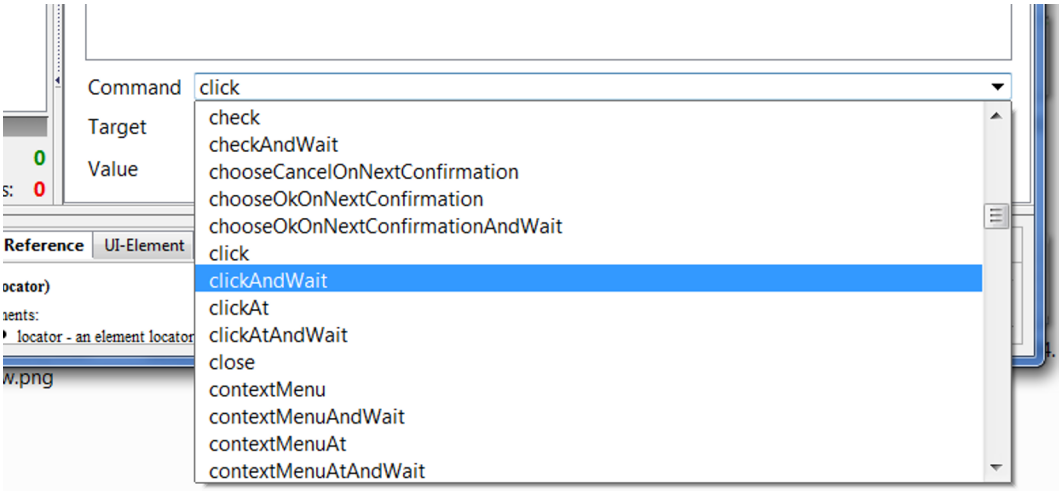

Figure 9-6. *Change the click command to the clickAndWait command for that entry*

Execute the test case again. This time, the test should have executed very quickly and yet the text still verified properly. This simple example of a test script modification is very common.

Once the script is working properly, save it by using the File ➤ Save Test Case option in the Selenium IDE. For a filename, type `Google_Test1.html`. The test code is actually stored in HTML table format. You can edit the code with any standard text editor.

Examining the Script Code

It can be useful to modify the Selenium code directly, especially when you begin putting together groups of tests known as test suites. If you open the file that you just saved using a text editor such as Notepad, your script might look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head profile="http://selenium-ide.openqa.org/profiles/test-case">
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<link rel="selenium.base" href="http://www.google.com/" />
<title>Google_Test</title>
</head>
<body>
<table cellpadding="1" cellspacing="1" border="1">
<thead>
<tr><td rowspan="1" colspan="3">Google_Test</td></tr>
</thead><tbody>
<tr>
        <td>open</td>
        <td>/</td>
        <td></td>
</tr>
```

```
<tr>
        <td>type</td>
        <td>q</td>
        <td>Joomla</td>
</tr>
<tr>
        <td>click</td>
        <td>btnG</td>
        <td></td>
</tr>
<tr>
        <td>type</td>
        <td>q</td>
        <td>Joomla</td>
</tr>
<tr>
        <td>clickAndWait</td>
        <td>btnG</td>
        <td></td>
</tr>
<tr>
        <td>verifyTextPresent</td>
        <td> Joomla! - the dynamic portal engine and content management system </td>
        <td></td>
</tr>

</tbody></table>
</body>
</html>
```

If you're used to reading HTML code, you can see that this is a very simple file. You can edit it manually if you ever need to make changes when the Selenium IDE is not installed on a machine.

Creating and Running a Test Suite

The test we created was very basic and was meant to get you started. Now that you have a basic understanding of the Selenium execution, you can begin to make it address your Joomla site. The following demonstration takes you through the process of creating a test suite that will allow you to test your entire Joomla site with each test execution to focus on a particular area.

In these examples, I'll be running a copy of Joomla out of the folder `http://localhost/joomlaadv` that has a generic site setup. We'll create a test suite that tests a menu that is only available to registered users. To make this possible, we'll record several different test scripts:

- *joomla_login*: This script will perform the login of the registered user to the system.
- *joomla_logout*: This script will log out of the system.
- *joomla_menu1*: For a logged-in user, this will ensure that the menu is available and accessible.
- *joomla_nomenu1*: After logout, this test will make sure the menu isn't present.

A suite of tests, as you'll soon see, can execute multiple test scripts sequentially. By breaking down the execution pieces into separate scripts, you can re-use the scripts such as the login and logout scripts across multiple situations. For example, you can use the same login script to run one registered user script and then, later in the script execution, you can use the same login to test another registered user feature.

1. Open the Selenium IDE and select the File ➤ New Test Case option. To the left of the area that shows the Table and Source tabs is a list of current tests in the active test suite. By default, you can't see this area, but if you move your mouse over to the left side of the Table tab, you'll see it change into the resize area pointer.
2. Click and drag the left side of the Table tab to the right side of the window until the test suite list appears. At the top of the column, you should see the text `Test Case` and the entry for the new case you just created should read `Untitled`.
3. Right-click on the case and select the Properties option. A dialog will appear allowing you to rename the selected script, so enter `joomla_login` and click the OK button.
4. Follow the same process you did last time. Copy the URL for the site into the Base URL text box.
5. Click the red button to begin recording.
6. Type your username and password into the login box in Joomla and click the Login button. When the login is successful, select the greeting text (such as "Hi, Charley") and insert a verify text test into the Selenium test using the context menu.

■ **Tip** If you have typing completion active on your browser (that will auto-complete a text field if you have entered it previously), it is a good idea to deactivate it before you begin recording a test. Generally, Selenium can't record text selected from an auto-complete add-on.

---

7. Return to the Selenium IDE and click on the red button to stop recording. In the browser window, click the Logout button so you can test the script. Play the script once to make sure it works and make any necessary adjustments.
8. Using the File ➤ Save Test Case option, save the test as `joomla_login.html`. After it has saved, select New Test Case and name it `joomla_logout`. Click the record button, return to the browser window (you should be logged in from the test run of the login script), click the Logout button, and do a verify text on the vanilla login greeting that should now display.
9. Save this test as `joomla_logout.html`. You now have the first of a series of tests that you can reuse for all your site testing.
10. Create another Test Case that checks for a menu that should only be available to a registered user. Use a verifyTextPresent to confirm that the menu text is on the page. Save this test as `joomla_menu1.html`.

For the `joomla_nomenu.html`, begin a new test. Make sure you're logged out. This time your test will verify that there is NOT text on the page. To create this type of test, it is usually easiest to begin by creating a basic test that verifies text that will be on all pages. Once the fundamental two or three commands have been established in the Selenium test, click the red record button to stop recording.

1. In the Selenium IDE, in the column labeled Test Cases that shows all of the cases in the current Test Suite, select the `joomla_menu1.html` test. The commands for that test should be displayed in the list box.
2. Right-click on the verifyTextPresent command and select the Copy option.
3. Reselect the `joomla_nomenu` test.
4. Right-click on the empty list item below the last command and select the Paste option.
5. Click on the item you just pasted into the test and the command text box should be filled (with `verifyTextPresent`) and the Target box should fill with the menu text.
6. Just to the right of the command text box, you'll see a drop-down arrow. Click on the arrow and you should see a list of all of the available Selenium test commands. Scroll down until you see the `verifyTextNotPresent` command and select that. Now the item in the script will confirm that the menu text is NOT on the page.
7. Make sure you are logged out and run the test. It should pass. Log in and try it again. The test should fail. You now should have all four test cases that you can use in the first small test suite of the site.
8. Save the suite of four tests by selecting the File ➤ Save Test Suite option and name the file `TestSuite1.html`. You need to rearrange the tests in the proper order and tests. Some versions of Selenium IDE allow you to drag and drop to rearrange the tests. If you can't rearrange your tests from within the Selenium IDE, you'll need to open the suite file in a text editor.

Like the formats of the tests, the execution of test suites are arranged as rows in a standard HTML table. You will need to arrange the tests so that they execute in this order: `joomla_login.html`, `joomla_menu1.html`, `joomla_logout.html`, and `joomla_nomenu.html`.

When you open the suite again in Selenium IDE, you should be able to execute the full set of tests to confirm proper functioning of a menu for both registered and unregistered conditions. You should begin to understand how you can create an entire battery of tests that will examine all of the menus and options of your Joomla site. It should also be apparent how you can reuse the login and logout scripts for many other tests.

Although static scripts such as this are very powerful, Selenium tests perform much more complicated tasks. For one thing, you can embed JavaScript in a test to make the test perform on-the-fly calculations. You might use JavaScript to create unique fields for forms such as the registration system that may require them.

Embedding JavaScript for a Dynamic Registration Test

In Joomla, each user name must be unique, which makes it difficult to create a static test that checks the registration system. With such a static test, the tester would need to change the username for the test each time before the test is run. However, what if you could use JavaScript to randomize the username each time? That would avoid the duplicate problem.

To create such a dynamic test, first record a standard static test where you enter all of the fields for registration. It would be useful if you enter some common field such as a name that always begins with Test so these users will be easy to locate and delete from the Joomla Administrator interface.

Make sure you perform a verifyTextPresent at the end of the test that confirms the registration has been successful. Once you have finished the last step, open the Selenium IDE and click the red button to stop recording.

In some versions of Joomla, the JavaScript for the e-mail validation makes the e-mail text boxes invisible to the Selenium IDE recorder. In this case, you will need to manually add these fields. On the version of Joomla at the time of this writing, the e-mail field names had element IDs of `jform_email1` and `jform_email2`. With your recording, right-click on the final command (which should be `clickAndWait`) and select the Insert New Command option from

the drop-down menu. Mirroring the commands above it, set the command to *type*, the target to `id=jform_email1`, and the value to whatever e-mail address you're using for testing. Do the same for the second e-mail field, changing the target to `id=jform_email2`. Now the test should execute properly.

If you run the test again, you should receive an error because the same username was attempted again. The page should show an error like this:

```
This username you entered is not available. Please pick another username.
```

Locate the line in the script with a command of *type* that enters the `username` field. Click on the command to bring up the variables for it in the text boxes at the bottom of the screen. The Value text box should display the entry that you made for the username. Replace that text with the following value:

```
javascript{"t"+Math.ceil(Math.random()*10000)+"test"}
```

This test inserts a random number between 0 and 10,000 into the `username` field, making it unlikely to generate a duplicate. If you execute the test again, it will fail at a different point. In this case, it will say the e-mail address is already used. Because the same value must be placed in both the e-mail address and the e-mail confirmation field, we can't just use a random value each time—we'll have to randomize and then store it.

Right-click on the row where you enter the username and select the Insert New Command option. Set the command to *store*, set the value to *myrand*, and enter the following in the target field:

```
javascript{"t"+Math.ceil(Math.random()*10000)+"test"}
```

After that is executed, the random text will be stored in the *myrand* variable. Modify the value of the username row to `${myrand}` and the value of both of the e-mail fields to `${myrand}@sogetthis.com` so they hold the same value.

■ **Tip** If you've never heard of the Mailinator site (`www.mailinator.com`), you've been missing out. Mailinator is a free service that accepts e-mail from any source and lets you view it. There is no blocking. E-mail is stored in memory, so it only stays for around 48 hours. This service is fantastic for testing because you have unlimited e-mail addresses and the mail doesn't hang around clogging up an e-mail box. Mailinator has many alternate domain addresses and one of them is `sogetthis.com` as used in the preceding example.

The test should now operate properly each time you run it! If you save the test to HTML and open it in a text editor, the body of the code should look something like this:

```
<table cellpadding="1" cellspacing="1" border="1">
<thead>
<tr><td rowspan="1" colspan="3">joomla_registration</td></tr>
</thead><tbody>
<tr>
        <td>open</td>
        <td>/joomlaadv/</td>
        <td></td>
</tr>
<tr>
    <td>type</td>
        <td>id=jform_name</td>
        <td>lover</td>
</tr>
```

```
<tr>
        <td>store</td>
        <td>javascript{Math.ceil(Math.random()*10000)}</td>
        <td>myrand</td>
</tr>
<tr>
        <td>type</td>
        <td>id=jform_username</td>
        <td>${myrand}</td>
</tr>
<tr>
        <td>type</td>
        <td>id=jform_password1</td>
        <td>lover1</td>
</tr>
<tr>
        <td>type</td>
        <td>id=jform_password2</td>
        <td>lover1</td>
</tr>
<tr>
        <td>type</td>
        <td>id=jform_email1</td>
        <td>${myrand}@sogetthis.com</td>
</tr>
<tr>
        <td>type</td>
        <td>id=jform_email2</td>
        <td>${myrand}@sogetthis.com</td>
</tr>
<tr>
        <td>clickAndWait</td>
        <td>css=button.validate</td>
        <td></td>
</tr>

</tbody></table>
</body>
</html>
```

This is a simple example of some of the capabilities that are possible with live JavaScript execution. In addition to generating test fields, the embedded JavaScript can use regular expressions to process strings and use the DOM to navigate the page elements and actively examine or manipulate the current page. You can even use the Selenium `store` commands to write values into JavaScript variables.

■ **Tip** It would be preferable if you wouldn't send hundreds of junk e-mails to some miscellaneous e-mail domain. If you have your own domain, I would suggest you use that domain name for your test e-mail account.

Refining the Selenium Tests

The Selenium IDE has many useful features that help you build proper tests. You can set breakpoints and use a variety of other commands within your script for proper execution.

You can right-click on any line in the Selenium IDE and set a breakpoint at that location. When you next execute the test, the execution will halt at that point so you can manually step through the code. You can also press the X key after selecting any line in the test to execute that individual command.

Please review the manual for Selenium commands that can be useful in creating tests. Some of the most helpful that I use frequently include:

- *pause*: Halts test execution for a specified number of seconds. This is useful if the page has to perform some operation such as an Ajax request or JavaScript event that will take time before the page will be ready for testing.
- *deleteCookie*: Can delete a cookie so programs that insert a cookie in the browser can be effectively tested.
- *assertElementPresent*: Check for the presence of an image or other page element. This is very useful if you need to check a non-text item such as a Flash object.
- *handleAlert*: Intercepts a JavaScript alert dialog and can report the contents. This is very useful for performing failure testing on form fields.

You can find a reference of all of the available Selenium commands as part of the Selenium Core installation. In the Selenium Core folder, you'll find the reference file here:

```
<selenium installation root path>/SeleniumCore/reference.html
```

Other Selenium IDE Options

In this chapter, we've only scratched the surface of the power available through the IDE. Take a look at some of the options available for special configuration under the Options ➤ Options menu item. The Options dialog has a number of check box options including:

- *Remember base URL*: This is useful in some circumstances and trouble in others. If you have many stage and production sites, you might not want to record the base URL so the tests can be used on any site.
- *Record assertTitle automatically*: I always use this option so that there is a baseline check on every page the test script will access.
- *Record absolute URL*: Stores the absolute URL (instead of the default relative URL) for each location in the test.
- *Enable UI-Element*: Allows the UI-Element location code to execute for abstraction of page element names for more durable tests.

You might notice that there is an option to configure the location of the directories that hold these Selenium extensions. There are a number of extensions available that you can download from the Selenium web site that add better locator functionality to find elements on a page, supplement the test logging function, and other enhancements.

Working Through Selenium's Limitations

There are a number of actions in Selenium that, when the test is replayed, break because they weren't recorded by the application properly. The back arrow, for example, often causes a problem. Selenium allows you to manually edit tests and direct action input so there is nearly always a way to work around the broken test. The best method is to try and find an alternate method of doing the same operation.

For example, if you need to return to the home page of a site, and recording the back button doesn't work properly, usually the top banner on a web site provides a link to the home page and you can use that instead.

Selenium Extensions

Selenium features an extension plug-in system, so if you want to add a missing feature and you know JavaScript, you can easily add this feature to the system. One popular extension is the UI-Element that allows mapping between a list of custom-defined reference names and elements on the HTML page.

For example, if the page contained references to a text input field that held the first name of the user that was named `id66182`, you can create a UI-Reference that calls that element `txtFirstName` so the tests you create would be easier to read and understand.

Using UI-Reference can also be helpful for pages that change often—particularly during a development cycle. You can create a single reference that all of your tests can use. When the elements of the page change, a single update to the reference file will allow the tests to work properly again.

This extension is a useful one to study because it contains many of the common operations that are needed in a file, including the ability to read a formatted data file. The reference file is stored in JSON (JavaScript Object Notation) format, which is a very lightweight but powerful way to organize hierarchical data. The extension is comprised of a single JavaScript file named *ui-element.js*.

There are a number of additional commands that you can add to the Selenium system, including:

- Recording image titles
- Assertions for right-click context menus
- Check for an Ajax-built table
- Verify editable or non-editable field
- if . . . then test control
- while looping in tests
- Locators used to locate page elements for testing

All of these extensions are programmed in JavaScript, so you can also use them as a foundation for developing your own custom functionality. You can access these plug-ins on the Selenium site (`http://seleniumhq.org/projects/ide/plugins.html`) with instructions for installation and execution.

Adding Language Formats

Selenium can output tests in many language formats. Although by default, tests are stored as HTML (as you have seen), there are a number of other languages that the Selenium IDE can export including PHP, JUnit, Ruby RSpec, and others. To add additional language support, you can either download additional JavaScript files (.js extension) or even create your own. You will have to open the .js file in a text editor, select all of the code, and select the Edit ➤ Copy option to save the text to the clipboard.

Select the Options ➤ Options menu item and click the Add button on the Formats tab. Paste the code into the body text box. There is a default template there by default, so you'll likely want to select that existing test and paste over it. Set the format name in the top text box and click the OK button.

You should immediately see the format added to the listbox on the Formats tab. It will also be available in the Format submenu of the Options menu.

Setting up Selenium for Flash Recordings

If you have Flash on your site, you can even test that Flash with Selenium, although you have to adjust the Selenium system to record events not just on DOM elements, but also general clicks. Because Flash is not actually part of the DOM, you will need to record click events that will then be replayed by the Selenium script.

You can add the following JavaScript code to the page that contains Flash in order to record the clicks for inside the Flash stage for testing:

```
Recorder.removeEventHandler('click');
Recorder.addEventHandler('clickAt', 'click', function(event) {
var x = event.clientX - editor.seleniumAPI.Selenium.prototype.getElementPositionLeft(event.target);
var y = event.clientY - editor.seleniumAPI.Selenium.prototype.getElementPositionTop(event.target);
this.record('clickAt', this.findLocator(event.target), x + ',' + y);
}, { capture: true });
```

Modifying and Customizing the Selenium IDE

The Selenium IDE is written in JavaScript, so if you see a limitation in the Selenium system or want to add a feature that cannot be effectively created with an extension, you can directly access the Selenium code. The Selenium plug-in is located in this directory on a Windows system:

```
C:\Documents and Settings\<USER_NAME>\Application Data\
  Mozilla\Firefox\Profiles\<???>.default\extensions\
  {???}\chrome\Selenium-ide.jar
```

Replace the ??? with the directories you find there. A .jar file is simply a zip-compressed file with a different extension. Most zip-compatible systems will recognize the .jar extension and natively uncompress the files there. If not, simply change the extension to .zip and uncompress the archive. You can then examine the JavaScript files contained in the archives and make any desired changes.

Using Selenium Core

The tests that you have executed in the Selenium IDE have been limited to execution on the Firefox browser. With Selenium Core, however, you can execute the Selenium tests in nearly any browser. You can download the Selenium Core here:

http://seleniumhq.org/projects/core/

To install it, simply uncompress the files into a directory on your web server and copy your saved Selenium IDE tests there. If you install the Core on localhost, you can access it at a URL like the following:

http://localhost/SeleniumCore/core/TestRunner.html

Enter the URL of your test suite in the text box, as shown in Figure 9-7. Once you click the Go button, the test suite will load. You can then execute individual tests or the entire suite.

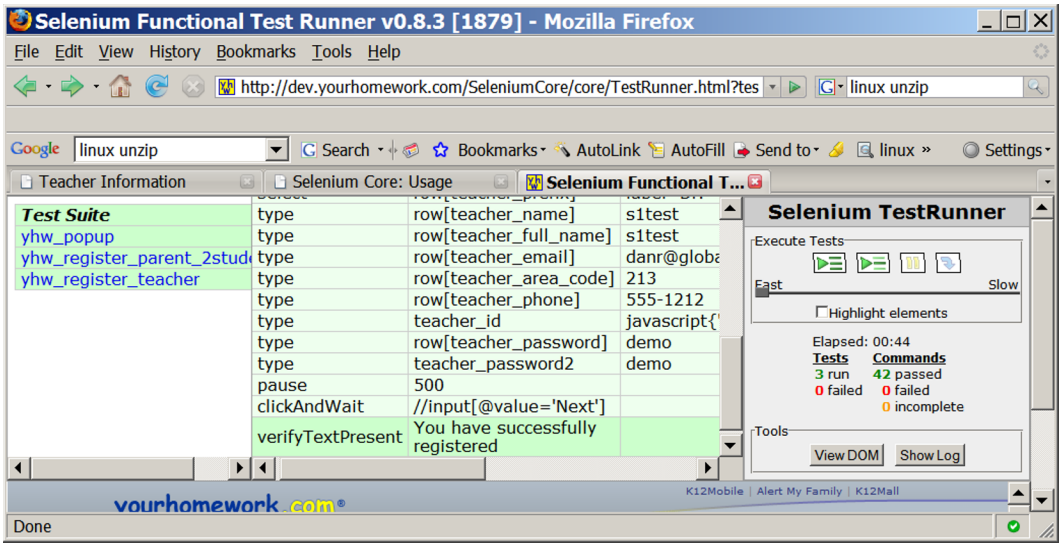

Figure 9-7. *Enter the URL of your test suite HTML and click the Go button to execute the tests*

You should be able to execute the tests on any browser that can access the web server and can execute JavaScript. Note that tests that perform on one browser may not perform the same on another browser or platform. You may have to slightly modify your tests to get them to execute properly. This is particularly apparent with Internet Explorer when verifyTextPresent values have line feeds as they are handled differently between the browsers.

■ **Note** You may need to change the header of a script for it to run properly in Core. The IDE recordings generally use the XML header to signify that they conform to the XML standard (are well-formed HTML). You may have to remove the XML header line from your test files to get them to execute properly in a browser when executing the script on Selenium Core. They sometimes cause a syntax error when the Selenium core reads them for execution. That means that the files will begin with an XML declaration (e.g. `<?xml version="1.0" encoding="UTF-8"?>`). At the time of this writing, the Core would fault when it encountered this header generating a syntax error. If your system generates such an error when you execute your IDE tests on Selenium core, try removing this first line and your test will most likely execute properly.

Using Apache JMeter

Apache JMeter has so many features that an in-depth examination of the program is beyond the scope of this book. However, even using a small portion of JMeter's features can generate critical information useful to any Joomla administrator. Additionally, if you create an enterprise-level Joomla site, there are better than average chances that JMeter will be your choice of testing tool.

JMeter provides the following testing features that are relevant to Joomla users:

- GUI interface for easily creating functionality tests
- Performance testing capabilities for many server types, including HTTP, HTTPS, SOAP, Database (including MySQL), LDAP, POP3 e-mail, and IMAP e-mail
- Multi-platform Java implementation that will run on Windows, MacOS, and Linux

- Multithreaded execution for effectively simulating a large number of simultaneous users
- Built-in proxy server to capture a web browsing session live (like Selenium IDE)
- Load statistics, including line/point graphs for visualizing performance results

JMeter is an application written in Java that you can run as either a standalone GUI application, as shown in Figure 9-8, or directly through the command line.

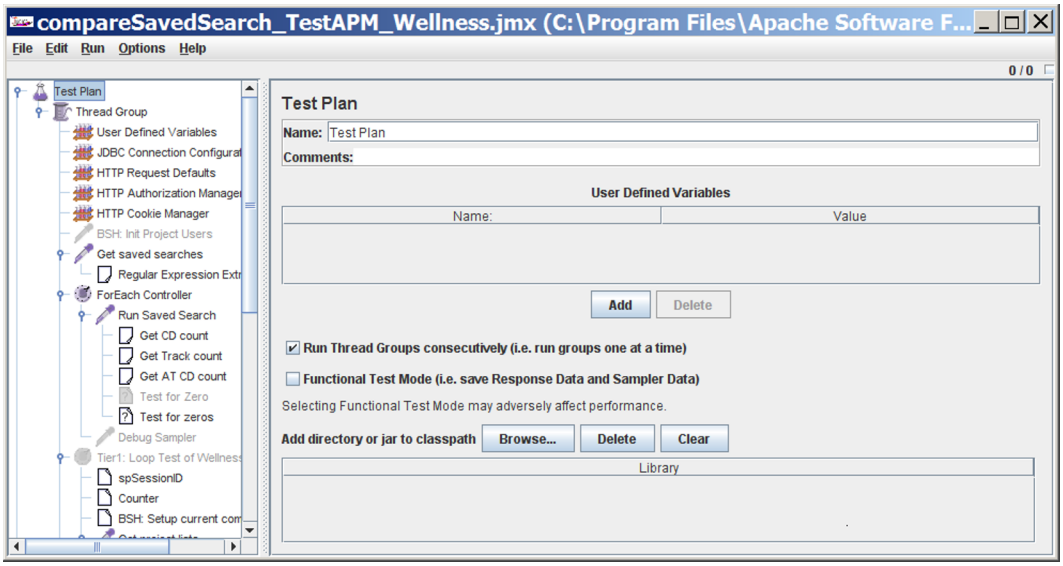

Figure 9-8. *The JMeter interface makes maintaining tests extremely easy*

Perhaps that best way to get a basic understanding of JMeter is by creating a quick load test and executing it. After you have created a foundation test plan (which you will likely use as a template for other test construction), you'll see how easily you can use the proxy server functionality included in JMeter to capture a live browser session.

You can download JMeter from the Apache Jakarta project web site (`http://jakarta.apache.org/jmeter/`) and simply expand the archive—no installation is required. JMeter does require a current version of Java that you can download from the Sun web site (`www.java.com/en/download/`).

You can execute JMeter on Windows by double-clicking the batch file that you'll find in the bin\ folder (`jmeter.bat`) or double-clicking on the jar itself (`ApacheJMeter.jar`). From the Linux or Mac GUIs, you can double-click on the jar, or from the command line you can use the bash script instead (`jmeter.sh`).

Creating a Simple Load Test

When JMeter is first launched, it displays a window split into two vertical panes. The left pane contains a basic hierarchical tree list that holds all of the nodes of the currently loaded test plan. Selecting a node in the left pane will populate the right pane with parameters and controls reflecting the selected item.

Every new JMeter test begins with a least one node titled *Test Plan*. All other test nodes are organized as children of this header node. It contains the overall configuration of the plan.

Let's begin creating our load test. Click on the *Test Plan* node in the left pane and the properties of the plan will be shown as in Figure 9-9. In the right pane that shows the properties of the node, change the name in the text box labeled `Name:` to read *Joomla Load Balance*. In the `Comments:` field, enter text such as *This test will load test the Joomla server*.

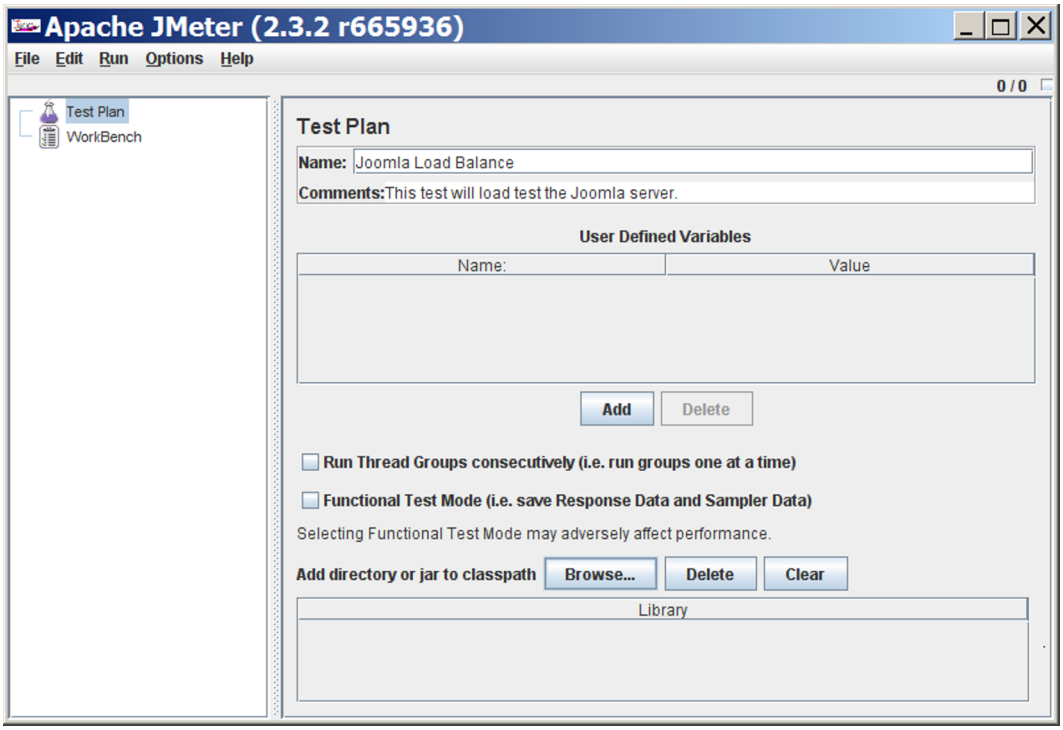

Figure 9-9. *The two panes of the JMeter interface with the left pane displaying a tree of all the nodes of a test plan and the right pane showing the properties of the selected item*

Most of the options that are used to add nodes to the test plan are accessed by right-clicking on one of the existing nodes. This will display a context menu of items relating to that node.

Right-click on the node that was previously labeled Test Plan (it will have updated to reflect the new Name property you entered previously), select the Add submenu, and then select the Thread Group option as shown in Figure 9-10. You will see a thread group node added as a child node to the test plan.

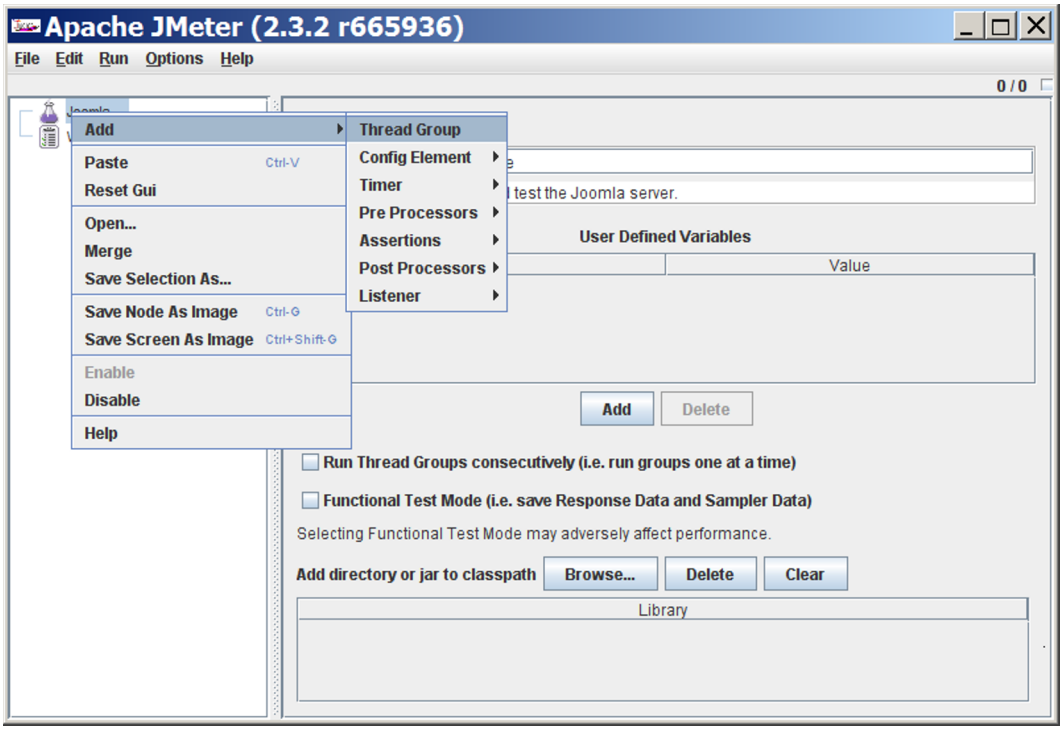

Figure 9-10. *Add a new Thread Group by right-clicking on the first node and selecting the Thread Group option*

All tests must be added to a thread group, so each test plan must have a minimum of one thread group. Some of the options that you will see with the current thread group selected include:

- *Action to be taken after a Sampler error*: You can halt all further tests when one error occurs, but usually the default setting of continue is best.
- *Number of Threads (users)*: This property determines the number of simulated concurrent users. This is the main variable that you will set to perform load testing. However, for the first test runs and setup of your test, it is best to leave this set to the default of 1 so you can complete repeated test executions quickly.
- *Ramp-Up Period (in seconds)*: This is the number of seconds to wait between the creation of one thread (user) and the next. Setting this option to zero will make the program attempt to start all threads at the same time. This is useful for simulating a burst access condition. Often you will want to increase this setting so you can easily see the graphic visualization of increasing load as new users are added to the request pool.
- *Loop count*: The count determines the number of loops that will be executed of the current thread group. Because a thread group can contain multiple page (or other) requests to simulate a session, you can use this to simulate multiple sessions for each user. Therefore, if the Number of Threads property was set to 10, the Loop count set to 5, and the number of page requests in the Thread Group is 6, then the complete test run would request 300 pages (10 x 5 x 6).

Leave all of these settings at their defaults for the moment.

Add User-Defined Variables

User-defined variables allow you to set variables in one place that you can use with other parts of the JMeter test. In this example, we're going to create three variables for items that you will likely want to change for various testing scenarios:

- *Log file*: The location of the log file where the data from the test will be recorded as an XML file
- *Log summary file*: The location of the log summary file where the summary of the test execution will be recorded in a comma-delimited CSV file
- *Base path*: The base path of the web page where the test will be run; changing this parameter will let you run the test on various parts of your site

You can use variables for many more items than configuration parameters. They can specify information to be passed in a GET or posted with a POST command, added to strings, or included in almost any parameter in the system.

1. Right-click on the Thread Group node, select Add ➤ Config Element ➤ User Defined Variables. This will add the node as a child of the thread group. In the right pane of the window, click on the Add button at the bottom of the screen and you should see a line item added in the User Defined Variables list area.
2. Click on the box in the Name column of the list area and type `logfile` and you should see that text appear in the box. Hit the Tab key to move the focus to the value column and enter the path where you want the log file to be saved. As you can see in Figure 9-11, I've entered a path to save the log file at my drive root.

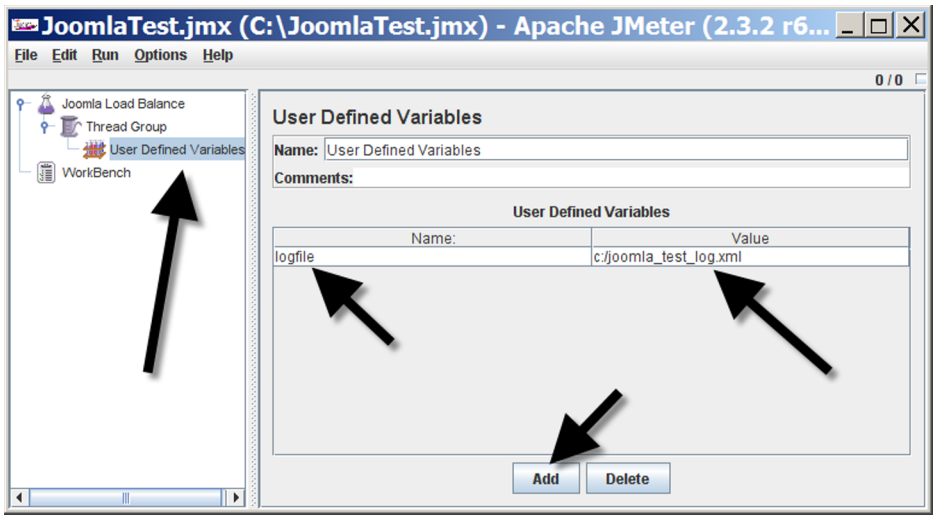

Figure 9-11. *Create a variable to store the log file name*

3. Create another variable and set the name to `logsummaryfile` and the value to the desired path such as `c:/joomla_test_summary.csv` to store the summary information. Add one more variable with a name `basepath` and the value to the base URL path of the page of your site to use. To simply test the front page of the site, you can set the value to `/` so the test will load the base page.

Access these variables in other parts of the system by encapsulating the variable name with the prefix `${` and the suffix `}`. We'll be using the base path in the next step.

■ **Tip** The User Defined Variables node is the best place to locate configuration data that may change from time to time. In the preceding example, the base path is held in this node so you can easily change it to test another copy of Joomla located at a different server path. Also, if you're going to give this test to someone else to run (such as QA department personnel), I would suggest that you always have this node as the first one in the test plan and then name it something like ___CONFIG VARIABLES___ so that someone who didn't author the test can easily spot it.

Add HTTP Request Defaults

Right-click on the Thread Group again, but this time select the Add ➤ Config Element ➤ HTTP Request Defaults option. In the Web Server frame, you'll see a text box to enter Server Name or IP. Enter the domain name that you want to test without the `http://` prefix. For example, you can enter "`www.example.com`" to test a site located there.

In the HTTP Request frame, put `${basepath}` in the Path text box. At the time the test is run, the value contained in the `basepath` variable will be substituted for this entry. Check the *Retrieve All Embedded Resources from HTML Files* box so that when the page is loaded, the extra files such as the images and the CSS files are loaded as well.

Add HTTP Authorization Manager

If you have an .htaccess setup with access limitations (only users with the proper passwords can access a particular page or pages), you will need to set up the HTTP Authorization Manager. This manager will hold URL routes and their proper username and password settings. You can set these once in a test and they are automatically applied when restricted pages are requested.

Right-click on the Thread Group again, but this time select the Add ➤ Config Element ➤ HTTP Authorization Manager option. By default, this list will be empty. Click on the Add button to create a row entry. In the first box, enter the entire base URL to the site (for example, `http://www.example.com`) that uses .htaccess access limiters. Enter the username and password of the login in their appropriate boxes. Finally, in the domain box, place a single forward slash (`/`) if the access permissions begin at the root folder.

Your setup should look like the one shown in Figure 9-12. If your test will access multiple hosts with .htaccess restrictions, add them here now. The access name and passwords in this node will be used automatically for all authentication challenges from the servers that the test accesses.

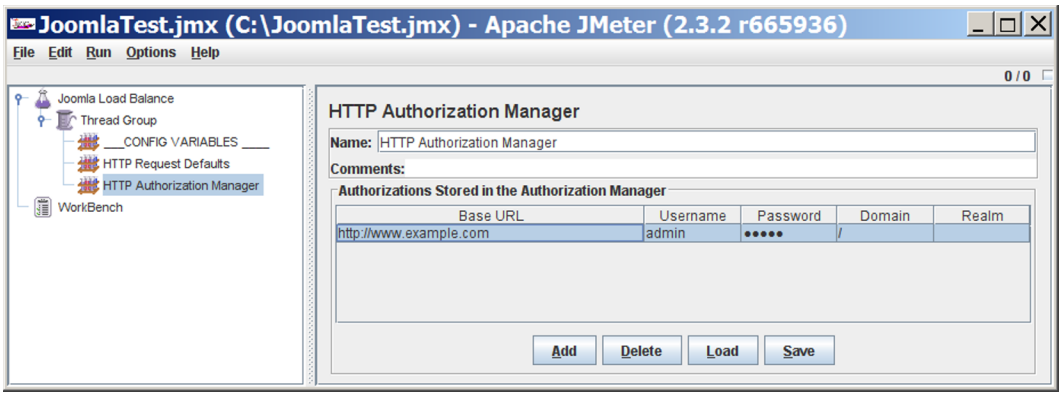

Figure 9-12. *Set up an entry for each area of the web site where an .htaccess username and password will be required*

Add HTTP Cookie Manager

If your session needs to simulate cookies (and most do), simply adding the HTTP Cookie Manager node to your test plan will automatically simulate cookies. Normally, adding this node is good enough for most tests. However, if you have some advanced tests that need to simulate pre-defined cookies, you can simulate explicit cookies and their values with this node.

Right-click on the Thread Group again, but this time select the Add ➤ Config Element ➤ HTTP Cookie Manager option. No configuration is required unless you want to manually specify various cookie variables that can be populated with constant values before the test is executed.

Add an HTTP Request Sampler

The Sampler nodes are perhaps the single most important part of a test because they perform the actual "sampling" of a System Under Test (SUT). A sampler sends a request (which may be an HTTP request, a database query, an LDAP lookup, or numerous other types) and stores the return reply. If a test is run that fails, JMeter makes it easy to examine the actual request that was sent as well as the data returned to the sampler. Having the request and the reply in one place can drastically shorten the time required to debug test problems.

Right-click on the Thread Group again, but this time select the Add ➤ Sampler ➤ HTTP Request option. The node added has a small eyedropper icon and is titled HTTP Request. Change the Name to Front Page. Set the path to a single forward slash (/) so the sampler will get the front page of the Joomla site. Your test plan should resemble the one shown in Figure 9-13.

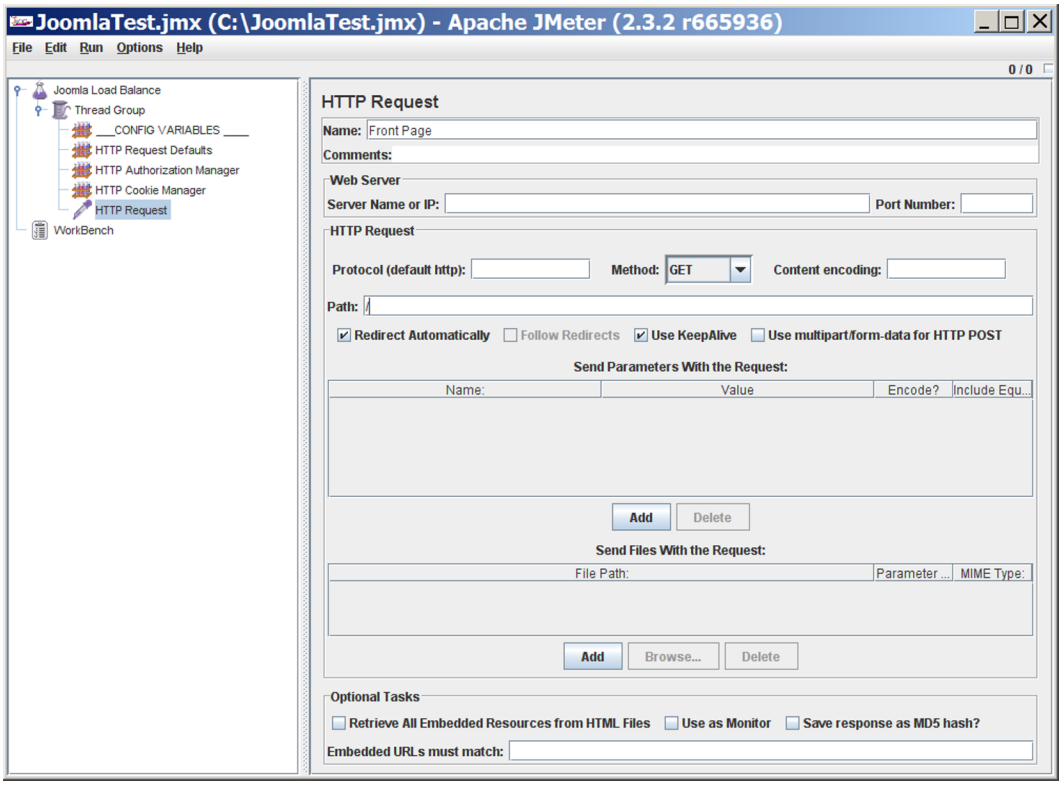

Figure 9-13. *Add an HTTP Request sampler to the current test*

Add a View Results Tree

All of the logic we've put into the test so far has been the setup of the actual test execution. However, there are other nodes that let you examine the results of the tests—either individually or as a summary. One results node type, the View Results Tree, will let you examine any single item executed in the test. This can be done hierarchically so that just as nodes in the test are organized with child nodes, so too the results are organized for easy navigation.

Right-click on the Thread Group again, but this time select the Add ➤ Listener ➤ View Results Tree option. You can put the path to a file name if you'd like to record all of the results of the test. By default, these are stored in an XML–formatted file, but other options including CSV format are available.

Add a Summary Report

The Summary Report works much like the View Results Tree, although instead of individual test results, it shows the results of test executions in aggregate. This report is particularly useful when performing load balancing as it allows a bird's-eye view of the execution.

Right-click on the Thread Group again, but this time select the Add ➤ Listener ➤ Summary Report option.

Add a Graph Results Node

Graphing results can help you spot trends in performance—especially peaks and valleys. It also helps you spot caching when the initial response time is very slow and when there is a precipitous drop in the response latency.

For the last node, right-click on the Thread Group again, but this time select the Add ➤ Listener ➤ Graph Results option. This will provide a graph or point chart of access speeds when the number of users is increased for load testing. It will allow you to visually examine the execution results.

Running the Test

Everything is ready for the first test, so let's run it! In the left pane, click on the View Results node. On the menu bar at the top of the window, select Run ➤ Start. Note the Stop menu option under the Run menu. This is very useful if you've started an elaborate load test and then realized there was some configuration error or other mass failure.

When this simple test completes, you should see the Front Page item in the results tree list, as shown in Figure 9-14. This list will show you every execution item of the test. In this case, we only have a single sampler that executed one time. If you configure the system for 5 users each with a loop count of 5 and added another sampler, you would see 50 items in this results tree (5 x 5 x 2).

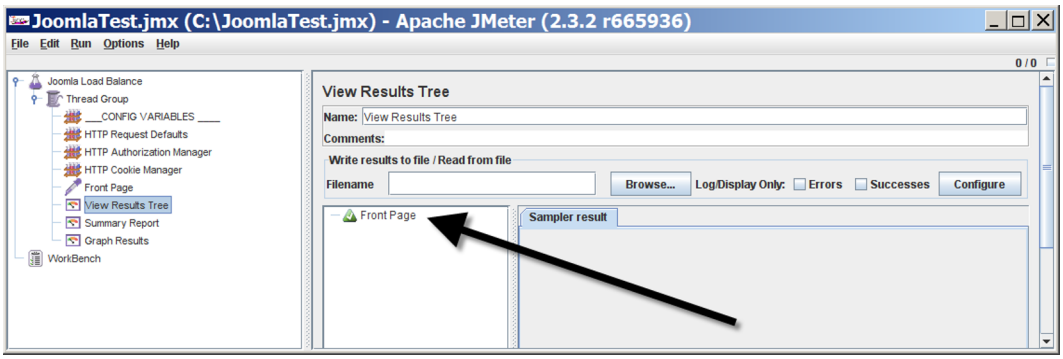

Figure 9-14. *The Front Page item will appear in the response list*

Click on the single result and you'll see the Sampler Request tab fill with information about the request execution. The text for the Sampler Request should look something like this:

```
Thread Name: Thread Group 1-1
Sample Start: 2009-07-26 17:18:56 AKDT
Load time: 299
Latency: 299
Size in bytes: 438
Sample Count: 1
Error Count: 0
Response code: 200
Response message: OK

Response headers:
HTTP/1.1 200 OK
Date: Mon, 27 Jul 2009 00:20:45 GMT
Server: Apache/2.2.3 (CentOS)
Last-Modified: Tue, 15 Nov 2005 13:24:10 GMT
ETag: "b80f4-1b6-80bfd280"
Accept-Ranges: bytes
Content-Length: 438
Connection: close
Content-Type: text/html; charset=UTF-8
```

From this information, you can see the load and latency times in milliseconds, the number of bytes returned to the request, the response code, and the information returned in the headers. If there was an error generated by the request (such as a 404 error), you can instantly see it here.

The Request tab is generally more useful if you have a failure. Click on the Request tab and you should see code like this:

```
GET http://www.example.com/
[no cookies]
Request Headers:
Connection: keep-alive
```

In this simple test, this information isn't very useful. However, in a more elaborate test, this will show the complete request string (that is, http://localhost/joomlaadv/index.php/article-anatomy/testarticle) so it can quickly lead you to the location of the error. If you use JMeter extensively, you will commonly adopt user-defined variables to build the request string dynamically. This will let you see any problems with the request URL.

Finally, click on the Response Data tab. This will display the actual data returned by the SUT and makes it easy to spot errors or problems. As you can see in Figure 9-15, the example request returned some basic HTML from the web page. If the sampler used had been a database sampler, the result set from the database query would be displayed in this pane.

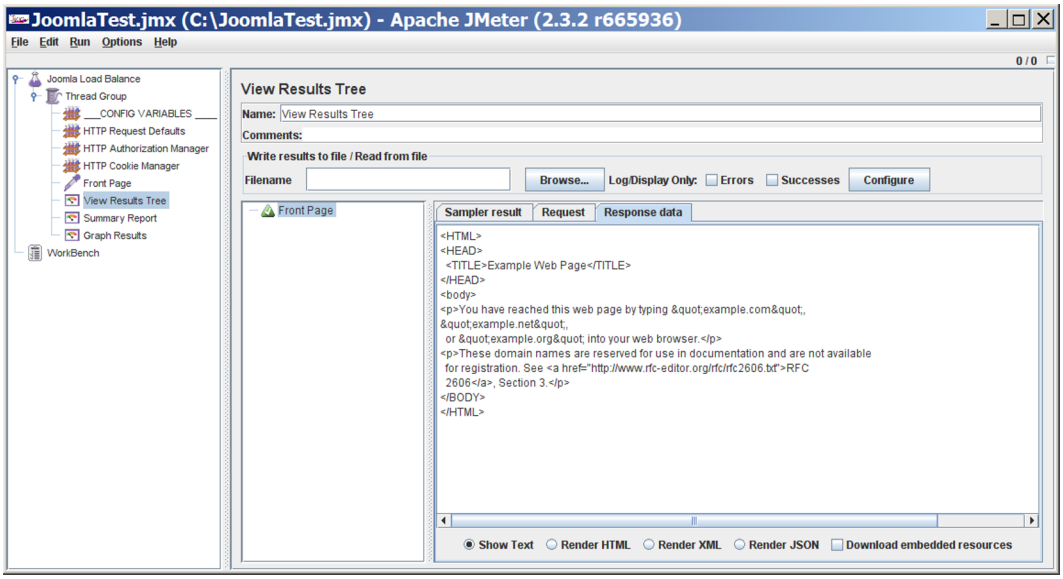

Figure 9-15. *The Response Data tab will show the actual data returned by the request*

You can probably already understand why this system is so powerful for testing. You can set up complete simulated sessions that request pages, check content, and report summary data. Let's scale up this simple test to see how a load-testing scenario for a web site might be accomplished.

Ramping Up the Users

To see a real load test, we need to increase the number of concurrent users accessing the site and also have them access it multiple times. By increasing the load, we should be able to see how the web server holds up (or bottlenecks) given a particular number of users.

Click on the Thread Group icon and increase the Number of Threads from 1 to 50. Set the loop count to 20 so each of the 50 simulated users will make the page request 20 times. This means a total of 1,000 page accesses.

Click on the Summary Report item and then use the Start option on the Run menu. In the top-right corner of the window, you should see a number that starts at 0/50 and increases until 50/50 is reached, as shown in Figure 9-16. These are the concurrent simulated users. JMeter will create a separate thread for each simulated user until all of them are created. When each simulated user has completed the specified number of loops, that thread will be killed and the number will decrease. When the number once again reaches 0/50, all the tests will be complete.

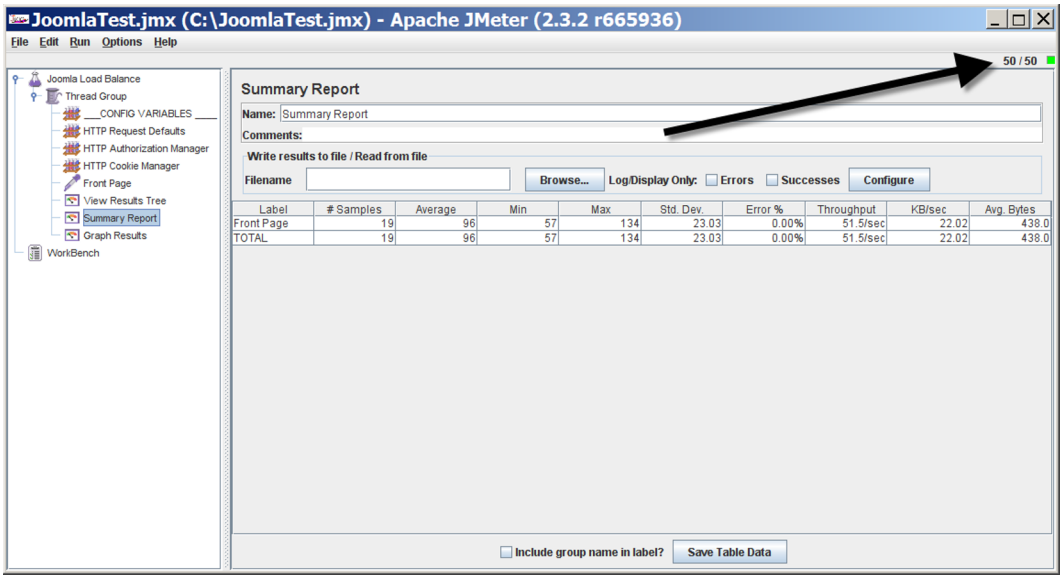

Figure 9-16. *The number of concurrent users requesting pages is shown in the top-right corner*

In the Summary Report table, you should see the # Samples column gradually increasing. The Error % column will show how many of the requests have failed. The Throughput column will tell you how many requests per second to which the server sent a response. The Avg. Bytes column is also useful because it shows the available page load for that URL in bytes.

To see how the server performed, click on the Graph Results node and you'll see a display like that shown in Figure 9-17. You can see each of the various factors such as average response time and throughput for the test. Note that the graph displays in milliseconds on the vertical access, so the higher the point or value that is displayed on the graph, the slower was the execution.

Figure 9-17. *The Graph Results node shows the execution results visually*

A single machine can often test more than a hundred simulated users. For a larger test, you can simply copy JMeter to another computer and run the test there at the same time.

Using the Proxy Server to Capture a Web Session

JMeter has a built-in proxy server that can sit between your browser and Internet connection and record all of the pages that the browser accesses. This allows you to create a list of sampler items quickly and simply.

1. Right-click on the Workbench icon in the left panel and select the Add ➤ Non-Test Elements ➤ HTTP Proxy Server option, as shown in Figure 9-18.

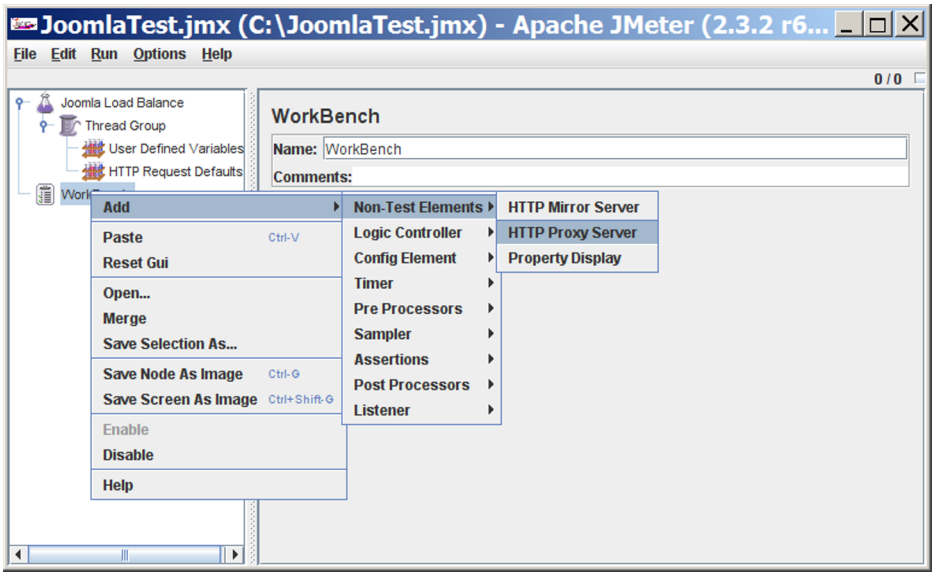

Figure 9-18. *Add an HTTP Proxy Server node to the WorkBench*

2. Set the port to 8080. Select the Destination where the new nodes with be recorded. In this case, select the Thread Group. In the Grouping drop-down menu, select the *Put each group in a new controller* option.
3. In the Content-type filter frame, enter `text/html` in the Include text box to include all HTML files.
4. In the Exclude text box, type `.*\.jpg .*\.png .*\.gif .*\.js .*\.css` to exclude image, CSS, and JavaScript files.
5. Click the Start button at the bottom of the window. Once the Proxy Server has started, the Stop button will be activated. Now you need to point your browser at the server to record your browsing activities.

After you've started the Proxy Server, you need to configure your browser to access the web through the JMeter proxy server. How this is done varies from browser to browser. Here are instructions for the popular Firefox and Internet Explorer browsers.

Configure Firefox to Use the Proxy Server

To configure the Firefox browser, select the Tools ➤ Options item to display the Preferences dialog box. Click on the Advanced icon and select the Network tab. In the Connection frame, click on the Settings button (see Figure 9-19).

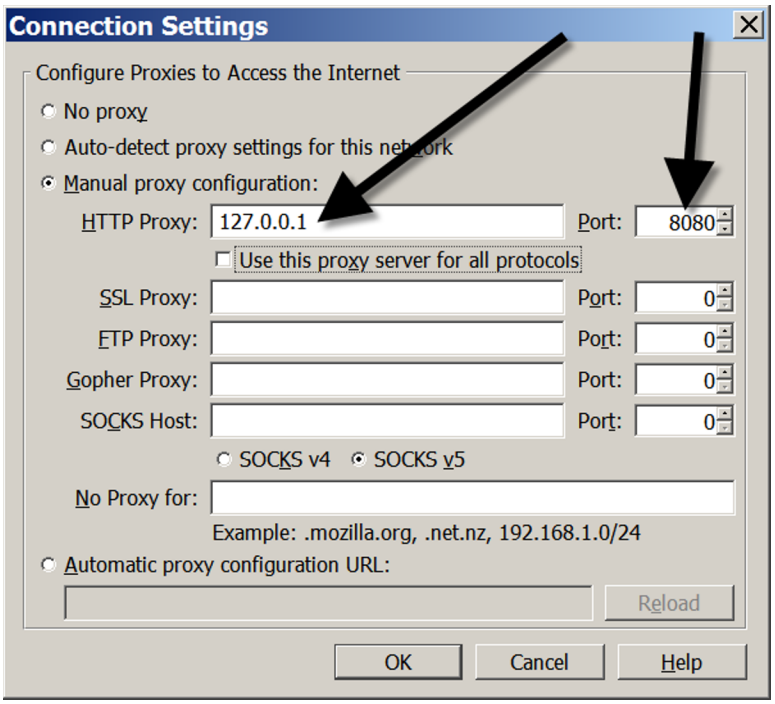

Figure 9-19. *Setup the Firefox browser to access the web site through the JMeter proxy server*

You need to set the proxy server settings to reference localhost (127.0.01) through port 8080, as shown in Figure 9-19. This will make the browser contact the JMeter proxy server that is running on localhost and use it to request all pages for browsing. With the browser accessing the JMeter proxy, it acts as a go-between for the Internet and can record all requests in the test.

Configure Internet Explorer to Use the Proxy Server

To configure the Internet Explorer browser, select the Tools ➤ Internet options item to display the dialog box.

On the Connections tab, click on the LAN settings button. The Local Area Network (LAN) Settings dialog box will display. There you can check the *Use a proxy server for your LAN* box and then enter the information for the JMeter proxy server. Typically this means a URL address of 127.0.0.1 (which is the same as localhost) and a port setting of 8080.

Editing a JMeter test

JMeter tests, even though the files have a .jmx extension, are actually XML files. Because they are plain text files, you can load them into a standard text editor. This provides a simple way to perform operations like search and replace. This capability is particularly useful with tests recorded using the Proxy Server. Often, the samplers recorded through this tool include full URL references, which make the tests less portable than if the URL was stored in the HTTP Defaults nodes.

Conclusion

In this chapter, you've learned how to use a variety of automated testing tools to profile and safeguard your Joomla site. You used Selenium to create fully automated acceptance tests for your Joomla site, ApacheBench to perform load testing, and JMeter to provide functional testing. The power of such automated testing tools will be apparent when you begin catching errors, page not founds, and other items when you make slight modifications to your site that—on the surface—should not affect other content.

Time spent creating tests pays great dividends in the long term. It takes some work to understand these tools and create tests that will look properly at your site. Once you do create these tests, however, they will serve you well and provide confidence that your site looks correct on a variety of browsers and over the long evolution of a web site. In the next chapter, we'll look at optimizing your development process with advanced development tools.