Automatic regression tests with Selenium

REMOTELY CONTROLLED BROWSER

www.sxc.hu

Testing complex web applications does not necessarily mean investing in expensive, proprietary tools such as Test Director or Silk Performer. Selenium is for free; it can remotely control any major browser, and it is programmable in Perl. **BY MICHAEL SCHILLI**

The only way to say for sure if a web application will work after modifying the code is to try all the functions in a browser. This means opening every single page, pressing every single button, and filling out every single text box. And you have to go through this multiple times to check out every possible success or error scenario.

The people from quality assurance would probably not look forward to the prospect of performing these monotonous tests manually. At the same time, this approach leaves you open to human error. We need an automatic test.

Testing Javascript

Simple web applications can be tested with screen scrapers such as the *WWW::Mechanize* CPAN module, but the tool trips up over Javascript. Although there are plenty of Javascript implementations with matching Perl APIs, a satisfactory solution for the interaction between the Javascript engine and the browser DOM (Document Object Model) has not been forthcoming thus far. This field is particularly complex, as Javascript manipulates the HTML in loaded pages, triggers repeating actions, opens new windows, or retrieves data from servers, and injects it into the page. In addition to this, there are pitfalls in the ways various browsers implement the DOM.

Now Thoughtworks has released an open source project titled Selenium [2] that gives programmers a simple solution to the problem. It feeds Javascript code to the browser, thus giving programmers the ability to remotely control the browser and automate tests. To illustrate how to automate tests with Selenium, Figure 1 shows a simple web form that accepts user comments. Users supply their email addresses along with the comment to allow the webmaster to respond.

To prevent the email field from remaining blank or containing the wrong input, the page's HTML contains a Javascript program that checks the input when the user presses the *Send* button. If the script discovers an error, instead of sending the user's comment to the server, the browser pops up a warning dialog. If the input is a syntactically valid email address, the data gets transmitted to the *ACTION* URL, which

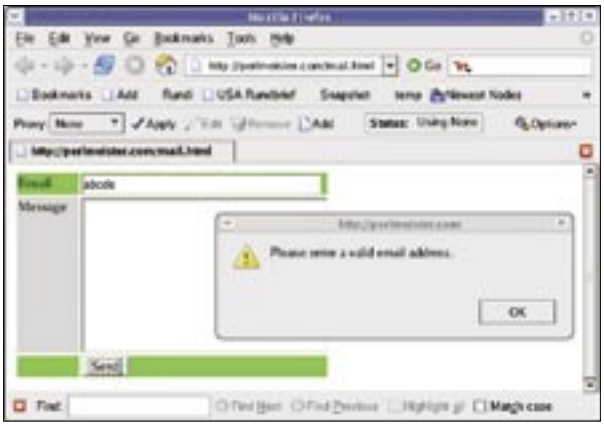

Figure 1: The web form we will be testing accepts an email address and checks the input for validity.

is the CGI script */cgi/feedback.cgi* on the web server. It sends back a page containing a message of "Thanks for your feedback!" to indicate success. We want the automated test to verify both the error and the success case; both the warning dialog, and the success message need to be caught and compared to a canned version to ensure the web application is working properly.

Listing 1 shows a test script, *emailcheck*, which opens up a connection to the Selenium server. The *start()* command tells the server to open a Firefox browser in a separate session. *open()* then tells the server to load the HTML page shown in Figure 1 from the web server; this is the page with the Javascript code for checking the email address.

Figure 2: Use HTML with JavaScript to check if a typed email address can be accepted.

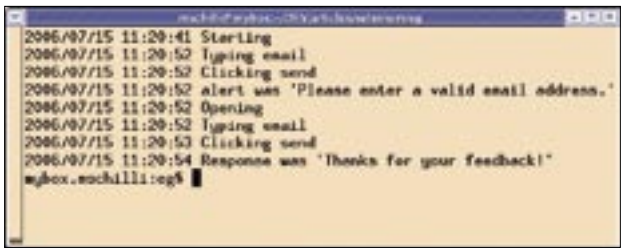

Figure 3: The emailcheck test script uses Selenium to control a Firefox browser session. It simulates the input of an invalid and a valid email address and outputs both the content of the resulting error dialog and the successful server response.

A Script Types and Clicks

The *type()* method simulates a user typing text, and expects the name of the <*INPUT*> field in a HTML form, along with the input text as parameters. In the first test case (Line 22 ff.), *WWW::Selenium* writes a string of *abcde* to the form's email field, identified by the *name* attribute *from* (Figure 2). The *click()* method is then called with the name of the submit button as an argument (*send*) to simulate a click on the button.

As the address string does not contain an @ or a dot, the input can't be a valid email address – this causes the web page to display the warning dialog. *get_alert()* catches the dialog in Line 30, and returns the error message it finds displayed. For test purposes, *emailcheck* logs the text string it found. Figure 3 shows the debug output from the script.

In the second test case (see Line 33), the simulator writes a string of *abcde@foo.com* to the email field: this is a syntactically valid address. After the *click()*, the simulator waits for up to 50 seconds (50000 milliseconds) for the server's response page to load (*wait_for_page_to_load()*). *get_body_text()* then reads the result, and outputs the string. Finally, *stop()* closes the browser opened previously by *start()*.

Tons of Tests

As Selenium scripts are mainly used for regression tests, the CPAN module *Test::WWW::Selenium* adds the *Test Anything Protocol* (TAP, see [4]) to the Selenium client. Scripts that create output in this format can be combined using modules such as *Test::Harness* to steadily expanding regression test suites that are easy to run, and return neatly formatted results.

Test::WWW::Selenium extends the *WWW::Selenium* class, adding a number of commands which in turn output other Selenium commands, and check if they

Listing 1: emailcheck

```
01 #!/usr/bin/perl -w
02 use WWW::Selenium;
03 use Log::Log4perl qw(:easy);
04 Log::Log4perl->easy_init(
05  $DEBUG);
06
07 my $url =
08   "http://perlmeister.com";
09
10 my $sel = WWW::Selenium->new(
11  host    => "localhost",
12  port    => 4444,
13  browser => "*firefox "
14    . "$ENV{FIREFOX_HOME}" .
15      "/firefox-bin",
16  browser_url => $url,
17 );
18
19 DEBUG "Starting";
20 $sel->start();
21
22 $sel->open(
23  "$url/test/mail.html");
24 DEBUG "Typing email";
25 $sel->type("from", 'abcde');
26
27 DEBUG "Clicking send";
28 $sel->click("send");
29 my $alert =
30   $sel->get_alert();
31 DEBUG "alert was '$alert'";
32
33 DEBUG "Opening";
34 $sel->open(
35  "$url/test/mail.html");
36
37 DEBUG "Typing email";
38 $sel->type("from",
39  'abcde@foo.com');
40
41 DEBUG "Clicking send";
42 $sel->click("send");
43 $sel->wait_for_page_to_load(
44  50000);
45
46 my $body =
47   $sel->get_body_text();
48 DEBUG "Response was '$body'";
49
50 $sel->stop();
```

trigger the expected response. For example, *open_ok()* is used to load a website instead of *open()*. According to TAP, if the test is successful, a response of *ok 1* is returned; an error returns *not ok 1*.

With *WWW::Selenium* methods such as *get_body_text()* or *get_title()*, which retrieve page details, you just remove the *get_*, and append the test method name from the *Test::More* collection. For example, *title_is($title)* checks if the page title returned by *get_title()* matches the string stored in *$title*. And *body_text_like ($regex)*, which derives from *get_body_ text()*, prints the success message, if the returned HTML matches the regular expression stored in *$regex*.

Googling Zombies

The *gtest* test script (Listing 2) launches Firefox, surfs to Google, enters a string of *schilli* in the search field, and presses the *Google Search* button. The test cases check if the search was successful, and if the match list contains a string of *perl-meister*. To tell the *click()* method which button to press, the method normally expects the button's *name* property. Alternatively, you can specify the click target via Selenium's *Element Locators*. Given that Google's search button displays "Google Search", here's how to point Selenium to an input form field with this *VALUE* attribute:

```
//input[@value="Google Search"]
```

The *WWW::Selenium* POD documentation has a comprehensive list of methods for controlling the Selenium server, and thus the browser, via the Perl client. Selenium supports the whole palette of browser gimmicks, from opening multiple popup windows, through triggering Javascript events, to placing the text cursor in form elements.

Figure 4 shows a Firefox window controlled by Selenium. At the top of the window you can see the control elements Selenium passed in; the website to be tested is in the lower half of the window. The zombie browser is using default settings, this explains the empty customizable toolbox, and the search window is set to the default of Google, whereas my browser uses my employer's search engine, of course (Figure 1). Selenium creates a Firefox profile of its own for its tests, and ignores the user preferences in the browser.

Figure 4: For its tests, Selenium sets up its own browser profile. It then tells the browser to surf to a page, while showing the remote control features in the top half of the window.

On closer inspection of the browser URL box, you might notice that something unusual is going on. The URL shown here, *http://www.google.com/selenium-server/...*, points to a Selenium server, which is not running on the search giant's site. Figure 5 explains the mystery: Selenium puts a proxy between the browser and the web server, which manipulates all outgoing requests.

Two in One

The Selenium server fulfills two functions at the same time. If a test script requests an Internet page, it talks to the Selenium remote control, which passes the request in to the browser. But the Selenium server sets the browser up to use a proxy, rather than communicate directly with the Internet. This proxy is – you guessed it – no one else but the Selenium server itself. And this allows Selenium to intercept the data the browser sends out to or receives from the Internet, and to manipulate the traffic between the two as needed.

If the Selenium remote control asks the browser to request an URL, it attaches some control parameters and a session ID to the request. The browser then sends out a request for the specified URL via the proxy. This way, Selenium gains control again, strips the parameters intended for itself, contacts the web server specified in the request, and retrieves the response. Before sending it back to the browser, the proxy adds Javascript code to the response that trigger remote control actions in the browser, to type input into entry fields, for example.

As an interesting implementation detail, the Selenium server does not di-

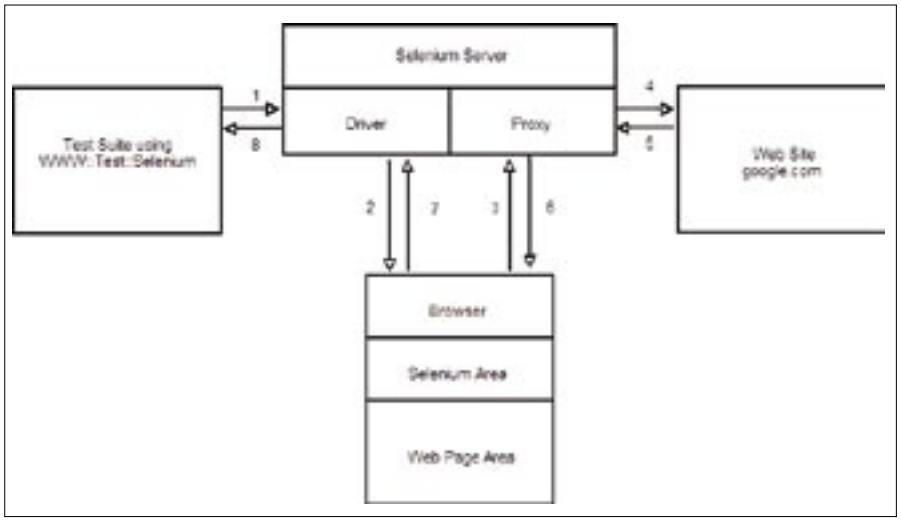

Figure 5: The test script talks to the Selenium server, which in turn remote-controls to the browser. Selenium's proxy server manipulates the browser's communication with the website.

rectly remote-control the browser. Instead, the browser continually asks for new instructions due to the Javascript code fed to it by the proxy. Effectively, the injected Javascript allows similar manipulations as a Firefox plugin would, however, the Selenium method has the advantage of being able to control browsers by various vendors.

The Selenium remote control is written in Java and available from [2] under an Apache license. The Zip archive has both the Java source code, and the *.jar* archive. To start the Janus-headed server, enter the following command:

```
LD_LIBRARY_PATH=/path/to/
firefox-1.5 java -jar
selenium-server.jar
```

Make sure that you have the dynamic libraries for your Firefox installation, such as *libmozjs.so*, in your Firefox directory. If you installed the browser in the standard path, there is no need to specify the *LD_LIBRARY_PATH*. This also applies to the *new* constructor of the *WWW::Selenium* class in your test scripts. Instead of specifying the long-winded installation path, you can simply use "**firefox*" as the *browser* parameter, if the binary is in your search path. In this case, you can omit setting the *FIREFOX_HOME* environment variable variable before running the test scripts.

The Selenium server listens on port 4444. The *WWW::Selenium* constructor expects this information in the *port* parameter, which the scripts set to 4444.

Concept "Same Origin"

The security restrictions imposed on Javascript are a hindrance. Based on the "same origin" principle, the Javascript code in one domain can't manipulate the content coming from a website in another domain. This is why the *WWW::Selenium* class constructor uses the *browser_url* parameter to store the root URL of the website. Tests can only be performed on the specified domain. Scripts to test interaction between multiple domains are not possible, although an upcoming Selenium release envisages dynamic modification of the root URL.

The project also offers a Firefox plugin titled Selenium IDE. When launched via the *Tools* menu, the plugin opens a dialog (Figure 6), and then goes straight to record mode to log the actions performed within the browser using Selenium's own language, Selenese. Selenium IDE can then be used later to reproduce the logged steps, thus giving users the ability to define test suites. You can extract the logged data and use it for writing regression tests in Perl.

Foreign Territory

A Perl script that uses Selenium can remotely control browsers on other computers, thus giving developers the ability to test the compatibility of their web applications. To remotely control Microsoft's Internet Explorer on Windows, just copy the *.jar* file for Selenium Remote Control to a Windows system and type:

```
java -jar selenium-server.jar
```

This assumes you have a Java environment on the Windows system. Type *ipconfig* at the command line of the Windows computer to discover its IP address. To run the test suite on the controlling machine against Windows Internet Explorer, pass the IP into the *WWW::Selenium* constructor:

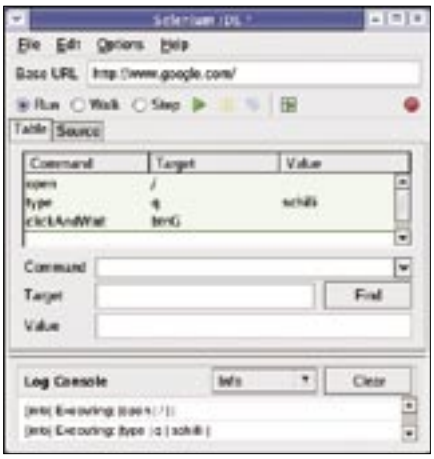

Figure 6: The Selenium IDE Firefox plugin logs and extracts browser actions.

```
host => "192.168.0.70",
port => 4444,
browser => "*iexplore",
```

Selenium supports any popular browser on Linux, Windows, and Mac OS X. The Java remote control server will run on any platform with Java support. The test scripts will go on running on the original Linux system. Now stop making up excuses for not testing your Web applications thoroughly and cross-platform! ■

Listing 2: gtest

```
01 #!/usr/bin/perl -w
02 use Test::WWW::Selenium;
03 use Test::More tests => 4;
04
05 my $url =
06   "http://www.google.com";
07
08 my $sel =
09   Test::WWW::Selenium->new(
10  host    => "localhost",
11  port    => 4444,
12  browser => "*firefox "
13    . "$ENV{FIREFOX_HOME}" .
14      "/firefox-bin",
15  browser_url => $url,
16   );
17
18 $sel->open_ok($url);
19
20 $sel->type_ok("q", "schilli",
21  "Type query");
22
23 $sel->click_ok(
24  '//input[@value=' .
25  '"Google Search"]',
26  "Clicking Search"
27 );
28 $sel->wait_for_page_to_load(
29  5000);
30
31 $sel->body_text_like(
32  qr/perlmeister/,
33  "perlmeister found"
34 );
```

INFO

[1] Listings for this article: *http://www.linux-magazine.com/Magazine/Downloads/72/Perl*

[2] Selenium: *http://www.openqa.org/selenium*

[3] Selenium IDE Firefox plugin: *http://www.openqa.org/selenium-ide*

[4] Regression tests with Perl: Michael Schilli, "Testing Tools", Linux Magazine 12/2005, *http://www.linux-magazine.com/issue/61/Perl_Regression_Tests.pdf*