Building a screen scraper to outsmart JavaScript

Every Little Bit

If an API for web information is not available, scripters often use Perl as a crowbar for screen scraping. In this article, we show how to overcome obstacles posed by JavaScript.

By Michael Schilli

No fewer than three venerable Linksys routers push Ethernet packets around in the Perlmeister Labs. The Tomato firmware [1] has provided many years of reliable service without a single glitch (Figure 1).

Because the Tomato admin website supports all kinds of useful settings and displays informative status data, I thought it would be a good idea to write a screen scraper that regularly retrieved the data, dumped it into a database on my home computer, and alerted me in case of suspicious events.

Oh, No, Not JavaScript!

Oh, yes! When I attempted to grab the Basic Auth protected page by doing this:

```
wget http://root:passwort@192.162.0.1
```

I quickly discovered that Tomato uses JavaScript to update the fields in the display and that a simple web scraper like the Perl WWW::Mechanize module just downloads the JavaScript code instead of the uptime data I was looking for.

To talk the page into displaying the data correctly, you need a JavaScript engine: The engine interprets the code and updates the DOM (Document Object Model) of the page being displayed in the browser in line with the instructions it finds. Basic screen scrapers will not do this; they act like browsers with JavaScript disabled and will not serve up the goodies.

The Seventh Wonder of the World

The CPAN WWW::Scripter module handles the Herculean task of implementing the required browser actions in Perl. In combination with the WWW::Scripter::Plugin::Ajax plugin, the HTML::DOM module providing the DOM interface, and the Pure Perl ECMAScript (JavaScript) Engine JE, it gives me all the functions I need.

When you think about how many DOM-specific browser differences exist just between Internet Explorer and Firefox, you can imagine how much work went into these modules. Also, the DOM module acts like another browser; differences between its implementation and the desktop browser used are inevitable otherwise. Another approach to implementing a JavaScript-controlled script client would be to use a browser remote control such as Selenium [2].

Listing 1 [3] starts by loading WWW::Scripter and then pulls in the Ajax plugin, which is available separately on CPAN, with the `use_plugin()` method. The class is derived from WWW::Mechanize, and thus from LWP::UserAgent, so it supports the `get` method for grabbing websites. Because the router prompts you for a `root` password on HTTP access, the script inherits and uses the `credentials()` method to provide the password.

Outsourcing the Password

To avoid hard-coding the password into the script, the `slurp` function slurps it from `pw.txt` in the current directory. The file is a one-liner with the password and should be protected against unauthorized read or write access. Of course, this approach isn't perfect from a security point of view, but you have to hide the key under the mat somewhere if you want the script to run automatically without the user typing the password every time.

Engines Running

When `get` picks up the page from the router's web interface, it initially doesn't contain any data – but just the embedded JavaScript code. The `wait_for_timers()` call then starts the JavaScript engine and lets it work with the page content. This would now block until the last

MIKE SCHILLI

Mike Schilli works as a software engineer with Yahoo! in Sunnyvale, California. He can be contacted at *mschilli@perlmeister.com*. Mike's homepage can be found at *http://perlmeister.com*.

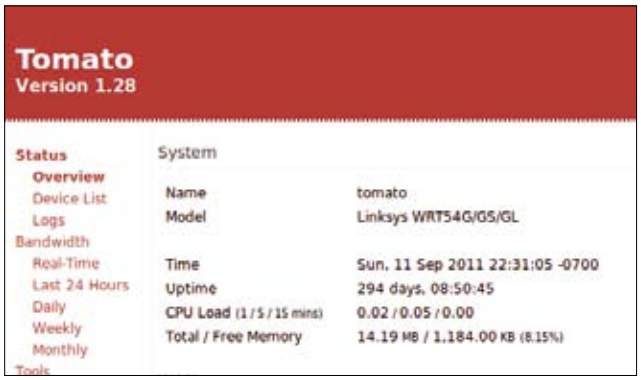

Figure 1: Tomato's overview page lists the router uptime in days and hours, among other information.

JavaScript timer in the code stopped running, which would take ages for many websites. The `max_wait` parameter thus tells the script not to wait for more than one second. This delay is sufficient for the router page to fill its dynamic fields. The call to `content()` then returns the updated HTML with the data.

Picking HTML Goodies

As you can see in Figure 3, the uptime value is surrounded by a mess of HTML tags, and you could extract it with regular expressions or with an HTML parser. Listing 1 uses the HTML::TreeBuilder::XPath XPath parser from CPAN, which is probably the most convenient method.

The path expression starting in line 34 first descends to the "body" tag in the hierarchy of the HTML document and – thanks to the double slash – then plumbs the depths for the tags specified to the right.

A TR tag with the `id` attribute of "uptime" is the target here; it includes a TD tag with a `class` attribute of "content" as shown in Figure 3. The `findvalue` extracts the hidden text, and the script just needs to output the value on its standard output.

The Tomato admin page with the current bandwidth usage statistics poses a more dynamic problem. The chart shown in Figure 4 is available in the `/bwm-realtime.asp` path of the router URL. The chart, which is created by JavaScript, shows the fluctuations during the last 24 hours. The table below shows the current values (kilobits per second) for received data (RX) and transmitted data (TX). Besides the first value for the current measurement, Tomato lists peaks, mean values (Avg), and the total number of bits transferred since starting the measurement, which starts when the page has been loaded.

The first time you load the page, all values are zero; but the table starts to fill up with interesting data in just a couple of seconds. That's why Listing 2 waits for no less than five rounds in the `rounds()` function call in line 27, in which it runs the `check_timers()` function to tell the scripts to run the timers in the JavaScript code. It then waits for one second in line 28. After completing the mandatory rounds, line 42 calls the callback passed into the `rounds()` function; this is an empty shell for the trial rounds in line 27, but the `extract_bandwidth` function is defined in line 46 for the real operations as of line 28.

The script uses the CPAN HTML::TableExtract module as the parser for the information hidden in the HTML tables. In line 52 its `parse()` method accepts the page's HTML code, which has been modified by JavaScript, and creates a syntax tree from it. The `first_table_found()` method then searches for the first HTML table that coincidentally contains the required data from Tomato's bandwidth page.

During the test phase, when I didn't know which table contained what information, the `tables()` method in the same module helped me find out by returning all the tables it found as objects. `coords()` tells you the position and nesting details in the hierarchy, and `rows()` outputs the content page by page. The man-page explains how this module works in great detail.

Tomato displays the values in Kbps (kilobits), but also adds a value for KBps

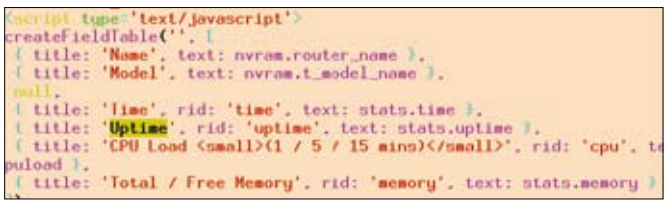

Figure 2: Simply extracting the website's HTML source will not give you the uptime.

Figure 3: The JavaScript engine has filled out the uptime data after running the JavaScript engine via WWW::Scripter.

LISTING 1: tomato-overview

```
01 #!/usr/local/bin/perl -w
02 ############################
03 # tomato-overview - Scrape
04 #   simple JavaScript-enabled
05 #   page
06 # Mike Schilli, 2011
07 # (m@perlmeister.com)
08 ############################
09 use strict;
10 use WWW::Scripter;
11 use Sysadm::Install qw(:all);
12 use HTML::TreeBuilder::XPath;
13
14 my $w = new WWW::Scripter;
15 $w->use_plugin('Ajax');
16
17 my $pw = slurp "pw.txt";
18 chomp $pw;
19 $w->credentials("root", $pw);
20 $w->get(
21  'http://192.168.0.1');
22
23 $w->wait_for_timers(
24  max_wait => 1);
25
26 my $tree =
27   HTML::TreeBuilder::XPath
28   ->new;
29
30 $tree->parse($w->content());
31
32 my $uptime =
33   $tree->findvalue(
34     '/html/body//' .
35     'tr[@id="uptime"]/' .
36     'td[@class="content"]'
37   );
38
39 print "uptime: $uptime\n";
```

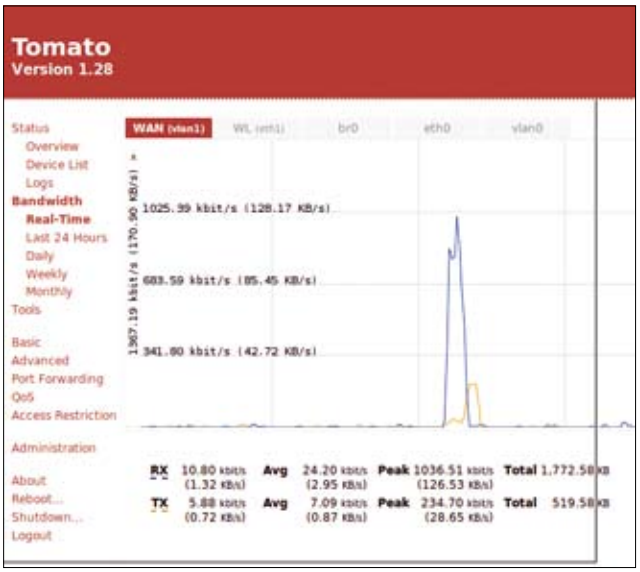

Figure 4: The Tomato router regularly updates the current bandwidth statistics using JavaScript.

(kilobytes). Because they differ only by a constant factor, the regex in the map instruction in line 62 filters the latter value by truncating after the first whitespace character. The first column in @cols is thus either "RX" or "TX" followed by the current transfer rate. The fourth and sixth columns (array indexes 3 and 5 in the script) contain the values for the average bandwidth and the peak values. Line

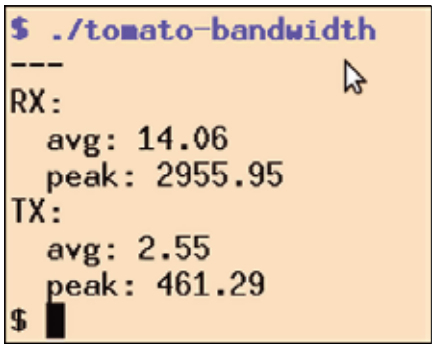

Figure 5: WWW::Scripter slurping information from a JavaScript-based website.

64 pushes them as hashes with the keys "avg" and "peak" into the "RX" or "TX" keys of another hash.

To allow for easy machine-processing of the script output, line 70 uses the `Dump` method provided by the YAML module loaded in the program header to print the resulting hash. Figure 5 shows the results at the command line. A post-processing script can use the same module's `Load` method to load the hash into memory and work with it.

LISTING 2: tomato-bandwidth

```
01 #!/usr/local/bin/perl -w
02 ##############################
03 # tomato-bandwith - Java-
04 # Script-enabled screen
05 # scraper
06 # Mike Schilli, 2011
07 # (m@perlmeister.com)
08 ##############################
09 use strict;
10 use Sysadm::Install qw(:all);
11 use WWW::Scripter;
12 use HTML::TableExtract;
13 use YAML qw(Dump);
14
15 my $w = new WWW::Scripter;
16 $w->use_plugin('Ajax');
17
18 my $pw = slurp "pw.txt";
19 chomp $pw;
20
21 $w->credentials("root", $pw);
22 $w->get(
23   'http://192.168.0.1/' .
24   'bwm-realtime.asp'
25 );
26
27 rounds($w, 5, sub { });
28 rounds($w, 1,
29  \&extract_bandwidth);
30
31 ##############################
32 sub rounds {
33 ##############################
34  my ($w, $rounds, $callback)
35    = @_;
36
37  for (1 .. $rounds) {
38   $w->check_timers();
39   sleep(1);
40  }
41
42  $callback->($w->content);
43 }
44
45 ##############################
46 sub extract_bandwidth {
47 ##############################
48  my ($html) = @_;
49
50  my $te =
51    HTML::TableExtract->new();
52  $te->parse($html);
53
54  my $ts =
55    $te->first_table_found();
56
57  my %bw = ();
58
59  foreach my $row ($ts->rows)
60  {
61   my @cols =
62     map { /(\S+)/ } @$row;
63
64   $bw{ $cols[0] } = {
65    avg  => $cols[3],
66    peak => $cols[5],
67   };
68  }
69
70  print Dump(\%bw);
71 }
```

Installation

Besides the WWW::Mechanize web scraper, the dynamic grabber scripts also need the CPAN WWW::Scripter and WWW::Scripter::Plugin::Ajax modules, which you can install with a CPAN shell in Ubuntu. The XPath parser used in Listing 1 (HTML::TreeBuilder::XPath) and the table parser used in Listing 2 (HTML::TableExtract) are also available from CPAN.

Of course, you could use SSH for safer communication with the Tomato router, and you might even flash a new image that provides a web API. But the tricks shown here serve as a perfect example for advanced screen scraping and can be applied to virtually any interesting page on the web.

Scraper hackers just have to make sure that vacuuming off the data doesn't contravene the provider's TOS (Terms of Service), because many providers prohibit scraping. ■■■

INFO

[1] Tomato: *http://www.polarcloud.com/tomato*
[2] Selenium: *http://seleniumhq.org/*
[3] Listings for this article: *http://www.linux-magazine.com/Resources/Article-Code*