Web server sockets

Socket To Me

HTML5 adds "WebSockets," allowing web clients to establish permanent connections to web servers. A sample Perl web application reveals in a browser in real time which pages users are visiting on a busy web server. *By Mike Schilli*

Instead of the usual request/response game run by the standard HTTP protocol, the WebSocket API included in the HTML5 standard gives standard browsers the option of communicating bidirectionally over persistent connections with the web server.

Once a regular HTML page has been received by the browser, embedded JavaScript can then open a WebSocket connection to a special URL, using the new "ws" scheme, as in `ws://server/path`. The WebSocket opener defines a callback that gets triggered immediately once the WebSocket server sends a message via the newly opened persistent connection. In other words, the browser application immediately responds to server signals without needing to poll the server at regular intervals, enabling web browsers to engage in all kinds of real-time applications, from online games to streaming logfiles.

Can My Browser Do It?

Not all web browsers support the new WebSocket standard yet. To see if a particular browser has the protocol implemented and activated, test to see if the `window.WebSocket` DOM element exists within a snippet of JavaScript. Or, even without writing any code, use the WebSocket.org website [2], which offers a tool that displays the browser's capabilities. Green indicates that your browser is ready and a simple data echoing application lets you type in characters that get sent to the server via the socket and are then played back by the server and redisplayed in your browser. For example, Figures 1 and 2 show Firefox 4 with the WebSocket API disabled, and then enabled. See the "Vulnerabilities" section below to switch between the modes.

Currently, Firefox 4 and Google Chrome at least have limited support for the protocol; if you want a complete implementation based on the latest standard, you have to install Firefox 6 (Aurora).

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: WebSocket.org confirming that Firefox 4 can't communicate via WebSockets without `network.websocket.enabled` set to `true`.

Figure 2: If the user enables the configuration explained in the "Vulnerabilities" section, Firefox 4 uses the WebSocket API draft 76 legacy protocol.

WebSockets Tested

Figure 3 shows the WebSocket test application `cntdwn-random` of Listing 1 in action. The browser receives descending numeric values from the server at random intervals. The server starts the counter at `10`, sends it to the rendered browser page via a WebSocket, and then goes to sleep for a random fraction of a second before entering the next round. When the countdown has reached `0`, the server sends a string that reads "BOOM!" and terminates the WebSocket communication. The browser displays the incoming server messages asynchronously and in real time. They are pushed down by the server and displayed in the web page's HTML immediately via a JavaScript callback function, triggered immediately when a server message arrives, without the client having to actively poll the server.

Figure 3: The WebSocket test script performs a countdown with values trickling down in random intervals.

To implement the test server, Listing 1 turns to CPAN's Mojolicious::Lite frame-

LISTING 1: cntdwn-random

```
01 #!/usr/local/bin/perl -w
02 ############################
03 # cntdown-random
04 # Mike Schilli, 2011
05 # (m@perlmeister.com)
06 ############################
07 use strict;
08 use Mojolicious::Lite;
09 use Mojo::IOLoop;
10
11 my $listen =
12   "http://localhost:8083";
13 @ARGV = (
14  qw(daemon --listen), $listen
15 );
16
17 my $loop =
18   Mojo::IOLoop->singleton();
19
20 ############################
21 websocket "/myws" => sub {
22 ############################
23  my ($self) = @_;
24
25  my $timer_cb;
26  my $counter = 10;
27
28  $timer_cb = sub {
29   $self->send_message(
30    "$counter");
31   if ($counter-- > 0) {
32    $loop->timer(rand(1),
33     $timer_cb);
34   } else {
35    $self->send_message(
36     "BOOM!");
37   }
38  };
39
40  $timer_cb->();
41 };
42
43 ############################
44 get '/' => sub {
45 ############################
46  my ($self) = @_;
47
48  (my $ws_url = $listen) =~
49    s/^http/ws/;
50  $ws_url .= "/myws";
51  $self->{stash}->{ws_url} =
52    $ws_url;
53 } => 'index';
54
55 app->start();
56
57 __DATA__
58 @@ index.html.ep
59 % layout 'default';
60 Random counter:
61 <font size=+2 id="counter">
62 </font>
63
64 @@ layouts/default.html.ep
65 <!doctype html><html>
66 <head>
67 <title>Random Countdown
68 </title>
69 <script
70  type="text/javascript">
71   var socket = new WebSocket(
72           "<%== $ws_url %>" );
73  socket.onmessage =
74    function (msg) {
75      document.getElementById(
76        "counter").innerHTML =
77        msg.data;
78    };
79 </script>
80 </head>
81 <body> <%== content %>
82 </body>
83 </html>
```

work introduced in the last issue, which enables experimenting programmers to put together a ready-to-run web application in just a couple of minutes. Besides normal web protocols like HTTP, it also supports WebSockets and uses closures to keep the status of each WebSocket client in memory. As defined in line 12, it opens a wireframe web server on port 8083 on the localhost, to which the browser in Figure 3 has been pointed.

Mojolicious excels in setting up web servers that respond to predefined URL paths. The module Mojo::IOLoop used in line 9 adds an event loop with a timer-controlled hook framework. Playing nicely with the event-based web server, it allows an application to perform random tasks within an active Mojolicious process from time to time.

Instead of a `new()` constructor, line 18 uses the `singleton()` method, which returns another reference to an event loop if the loop was defined previously. This is important because multiple, different event loops would cancel out their predecessors.

Perpetuum Mobile

In the test script, the callback stored in `$timer_cb` in line 28 defines a function that uses the `send_message()` Mojo method to send the global countdown value across the WebSocket to the browser and then decrements the value by one. After completing its task, line 32 calls the Mojo::IOLoop module's `timer()` method to schedule the next time the callback will be called.

The first argument to `timer()` is `rand(1)`, which returns a floating-point value between 0 and 1, defining the fraction of a second after which the next call will occur. The second argument, `$timer_cb` holds a reference to the callback function itself, causing a self-perpetuating loop calling the callback in irregular intervals. To get the ball rolling initially, line 40 issues the first call to the callback.

The `websocket` command in line 21 defines the jump target on the server for incoming WebSocket requests, in contrast to `get` in line 44, which responds to GET requests for the root path `/`. The GET request handler code in lines 46-53 converts the given `http://` URL into a `ws://` URL for WebSocket requests by means of a regular expression and adds the `/myws` path at the end. It then stuffs it into the HTML layout engine's stash under the key `ws_url`. To render the corresponding HTML with the embedded WebSocket URL, line 53 specifies `'index'` to point to the `@@ index.html.ep` template defined in line 58 of the trailing `__DATA__` area.

The HTML template contains some text (`Random Counter`) and a font element with an ID of `counter`. Although this will possibly irritate CSS purists, all we really want to do here is define some kind of HTML element with a known ID that the JavaScript code can then extract from the DOM and update its content with the current counter value.

JavaScript to Enterprise: Come in, Please!

The `'default'` layout reference in line 59 refers to the layout defined in lines 64 on, which creates an orderly HTML document from previously defined HTML snippets and adds JavaScript code for WebSocket communication. Line 71 creates a new `WebSocket` object using the *ws://* URL created previously. Once the WebSocket has successfully contacted the server and completed the necessary handshake (Figure 4), the message sent by the server to the browser via the WebSocket later on creates a JavaScript event by the name of `socket.onmessage`; its callback function is set in line 73.

The `data` attribute of the message object received by the callback contains the text string that `send_message()` wrapped up for it server side. As you would expect, this is the current counter value from the countdown, and line 75 only needs to look for the DOM element with the `counter` ID and set its `innerHTML` attribute to the counter value to display the value. That's all there is to it. If the user now directs their browser to the *http://localhost:8032* URL set in line 12, the countdown starts to run.

Mike: The last line of "JavaScript to Enterprise" specifies localhost:8032; line 12 specifies localhost:8083. Which number is correct?? -rls

Surfer Kibitzer

As a practical application, Listing 2 shoulder surfs active users of a website and shows the pages in a browser window, along with the URL and the displaying host, that surfers visit in real time (Figure 5). The website owner needs to install the script on the web server, start the Mojolicious server there, and then point his browser to the Mojolicious URL. The browser then starts updating its display with the web pages currently being served. How does this work?

The `tail()` function starting in line 91 of Listing 2 checks every second to see if the web server's `access.log` file contains new data. After opening the access log with Perl's `sysopen` command in O_NONBLOCK mode, subsequent calls to `sys-`

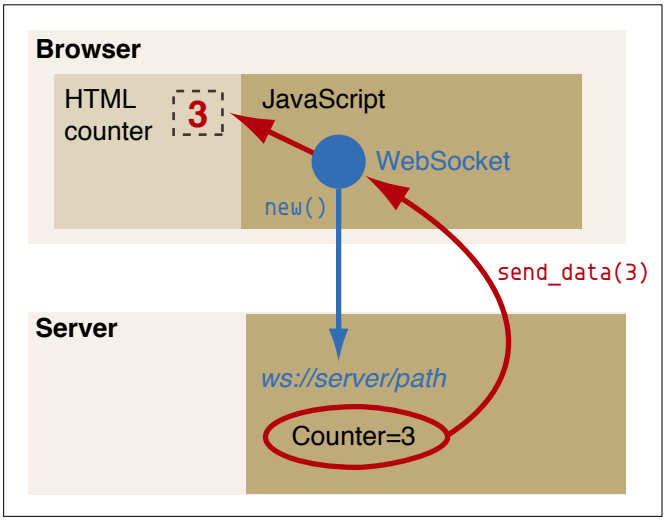

Figure 4: After the client has opened a WebSocket to the server, the server can asynchronously send data to the client.

Figure 5: Surfer Kibitzer showing which surfer is currently accessing which web page in the view window.

`read` will return new data but won't block if no new data is available.

If the script discovers any new GET requests, as shown in Figure 6, that point to one of the HTML pages on the website and do not originate with the script's own IP address, it bundles the requested URL along with the IP address of the requester (converted into a hostname) into a JSON construct and sends it to the browser's WebSocket in line 66.

On the JavaScript end, the `socket.onmessage` callback in line 150 unpacks the JSON format by enclosing it in parentheses and executing it with JavaScript's `eval()` function. To update the browser display, it then finds the HTML elements with the IDs `host`, `url`, and `pageview` in the HTML code (defined in lines 135-140) and packs the data extracted from JSON into the displayed fields.

Because the HTML `iframe` element in line 138 gets updated with a URL from the WebSocket, the spying browser will display the HTML of the URL extracted from the server's access log earlier. At the same time, the script updates the requested URL as a displayed text string and the resolved hostname of the requesting surfer at the top of the page.

This ultra-fast, asynchronous update, which is initialized by the server, thus lets you track who is watching what on your server in real time. To avoid stressing the script with large numbers of requests, line 71 limits the access log inspection to once every 5 seconds and just picks up the first line that matches the requirements in lines 48-55: only GET requests, to prevent unintended replays of data-modifying POSTs; only HTML pages (if you use a suffix other than `.html`, you need to modify this); and no requests from the spying browser's own IP address. The latter prevents sending the Mojolicious server into a spin because the spying browser's requests also show up in the web server's access log. Also, with the rate limiting explained previ-

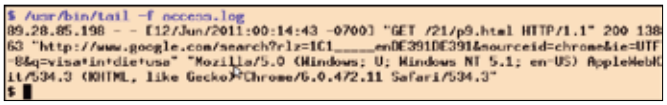

Figure 6: A request in the web server's access logfile.

Mike: Should "initialized by the server" be "initiated by the server"?? -rls

LISTING 2: apache-peek

```
001 #!/usr/local/bin/perl -w
002 ############################
003 # apache-peek
004 # Mike Schilli, 2011
005 # (m@perlmeister.com)
006 ############################
007 use strict;
008 use Mojolicious::Lite;
009 use ApacheLog::Parser
010   qw(parse_line_to_hash);
011 use Mojo::IOLoop;
012 use POSIX;
013 use Socket;
014 use JSON qw(encode_json);
015
016 my $listen =
017   "http://website.com:8083";
018 @ARGV = (
019  qw(daemon --listen), $listen
020 );
021
022 my $base_url =
023   "http://website.com";
024
025 my $file = "access.log";
026 sysopen my $fh, "$file",
027   O_NONBLOCK | O_RDONLY
028   or die $!;
029
030 my $loop =
031   Mojo::IOLoop->singleton();
032
033 ############################
034 websocket "/myws" => sub {
035 ############################
036  my ($self) = @_;
037
038  my $timer_cb;
039  $timer_cb = sub {
040   for
041     my $line (@{ tail($fh) })
042   {
043    my %fields =
044      parse_line_to_hash(
045     $line);
046
047    if (
048     $fields{request} eq "GET"
049     and $fields{file} =~
050     /\.html?$/
051     and
052
053     # skip our own requests
054     $fields{client} ne
055     $self->tx->remote_address
056      )
057    {
058     my $url = $base_url
059       . $fields{file};
060     my $data = {
061      url  => $url,
062      host => revlookup(
063       $fields{client}
064      ),
065     };
066     $self->send_message(
067      encode_json($data));
068     last;
069    }
070   }
071    $loop->timer(5, $timer_cb);
072   };
073   $timer_cb->();
074 };
075
076 ############################
077 get '/' => sub {
078 ############################
079   my ($self) = @_;
080
081   (my $ws_url = $listen) =~
082     s/http/ws/;
083   $ws_url .= "/myws";
084   $self->{stash}->{ws_url} =
085     $ws_url;
086 } => 'index';
087
088 app->start();
089
090 ############################
091 sub tail {
092 ############################
093   my ($fh) = @_;
094
095   my ($buf, $chunk, $result);
096
097   while ($result =
098    sysread $fh, $chunk, 1024)
099   {
100    $buf .= $chunk;
101   }
102
103   if ( defined $result
104    and defined $buf)
105   {
```

ously, the browser only updates the URLs it displays every five seconds, no matter how fast the client requests arrive.

To analyze the web server's access logfile, the `parse_line_to_hash` function courtesy of the ApacheLog::Parser CPAN module called in line 44 parses each logfile line passed into it and converts it into a hash with the keys `client` (client IP address), `file` (file path requested in the URL), and so on. `revlookup()` called in line 62 and defined in lines 117-129 uses reverse DNS to convert an IP address into a hostname but keeps the original IP in case this fails.

Vulnerabilities

The WebSockets implementation in Firefox 4 and Google Chrome is based on Draft Version 76 of the protocol, which has a couple of vulnerabilities. Although these only occur in unencrypted communication and with poorly programmed web proxies, the end user would be exposed to attacks on the real Internet.

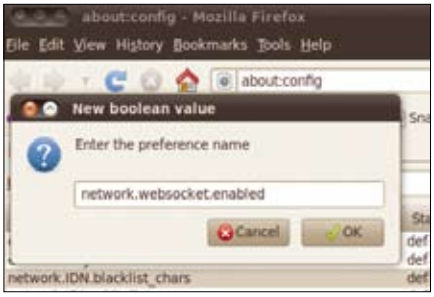

Figure 7: A new Firefox entry in about:config activates the WebSocket API.

The current version of Mojolicious from CPAN (1.42) thus only supports the modified version of the protocol based on the IETF 08 specification. Firefox 4 or Google Chrome don't support this, but Firefox 6 (Aurora) does. If you want to test this month's scripts with an older browser, download the older Mojolicious version 1.16 from CPAN; it was programmed with Draft 76 of the WebSocket protocol. Use on production systems poses a security risk.

Firefox 4 disables its own WebSockets by default because of the obsolete implementation, and you need to set the Boolean variables `network.websocket.enabled` and `network.websocket.override-security-block` in the about:config dialog to `true` to tell Firefox 4 to enable the feature (Figures 7 and 8).

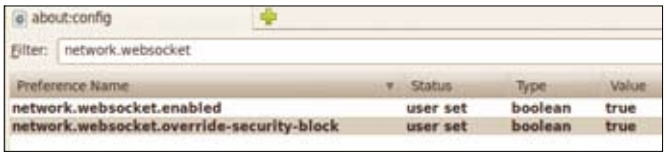

Figure 8: Users need to enable WebSockets explicitly in Firefox 4 because of security worries.

Installation

You can install the required CPAN modules, Mojolicious, ApacheLog::Parser, and JSON with a CPAN shell. The ApacheLog::Parser module might give you some grief because it is based on Time::Piece, whose test suite failed on my Ubuntu system. The reason for this is a year-old bug in interacting with Test::Harness, which doesn't impair the module's functionality but causes the CPAN shell tests to fail. `force install` ignores the failure and completes the installation.

WebSockets are still in their infancy, and it will take some time for all of today's browsers to implement the current version of the protocol. However, I can imagine many practical applications for browser applications communicating bidirectionally with the server without polling delays, especially in the gaming, chat, or video fields. ■■■

INFO

[1] Code for this article: *http://www.linux-magazine.com/Resources/Article-Code*

[2] WebSocket test page, *http://websocket.org/echo.html*

LISTING 2: apache-peek (continued)

```
106   chomp $buf;
107   my @lines =
108     map { s/\s+$//g; $_; }
109     split /\n/, $buf;
110   return \@lines;
111  }
112
113  return [];
114 }
115
116 ############################
117 sub revlookup {
118 ############################
119  my ($ip) = @_;
120
121  my $host = (
122   gethostbyaddr(
123    inet_aton($ip), AF_INET
124   )
125  )[0];
126  $host = $ip
127    unless defined $host;
128  return $host;
129 }
130
131 __DATA__
132 @@ index.html.ep
133 % layout 'default';
134
135 Host: <em id="host"></em>
136 URL: <em id="url"></em>
137
138 <iframe width=100%
139  height=800 src=""
140  id="pageview"></iframe>
141
142 @@ layouts/default.html.ep
143 <!doctype html><html>
144 <head><title>Apache Peek
145  </title>
146  <script
147   type="text/javascript">
148  var socket = new WebSocket(
149    "<%== $ws_url %>" );
150  socket.onmessage =
151    function (msg) {
152     var data = eval( "(" +
153            msg.data + ")" );
154     document.getElementById(
155          "host").innerHTML =
156       data.host;
157     document.getElementById(
158          "url").innerHTML =
159       data.url;
160     document.getElementById(
161    "pageview").setAttribute(
162      "src", data.url );
163    };
164    </script>
165  </head>
166  <body> <%== content %>
167  </body>
168 </html>
```