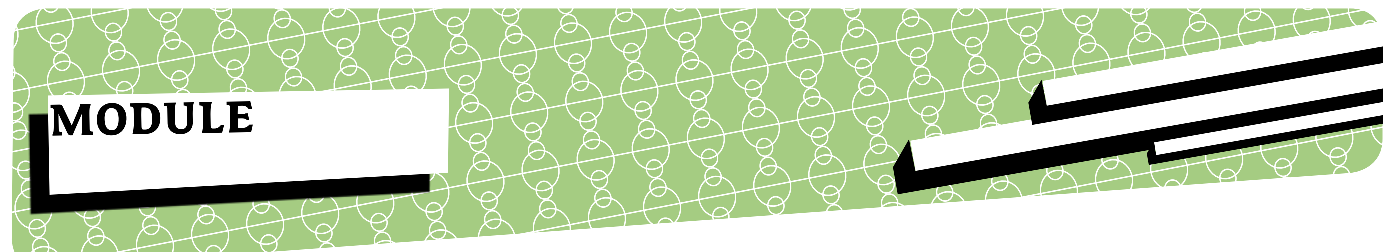

Renée Bäcker

Websockets mit Mojolicious

Websockets sind ein Hypethema im Bereich Webanwendungen. Mit Websockets können "Echtzeitanwendungen" umgesetzt werden. Ein typisches Beispiel ist der Chat. Wir wollen aber eine kleine Monitoringseite für einen Server schreiben.

Aber was sind Websockets überhaupt?

Websockets ist ein Protokoll, das entworfen wurde, um eine bi-direktionale Verbindung zwischen einer Webanwendung und einem Client zu ermöglichen. Das Protokoll basiert auf TCP und wurde in HTML5 integriert. Mittlerweile unterstützen auch alle modernen Browser Websockets. Möchte man aber auch ältere Browser bedienen, so muss man sich um ein Fallback kümmern. Die Fallbacks sollen hier aber kein Thema sein, nur ein kleiner Tipp dazu: Mit socket.io kann man sich der umständlichen Arbeit mit verschiedenen Fallbacks entledigen. Socket.io erkennt automatisch, welche Alternative verwendet werden kann, wenn Websockets nicht möglich sind.

Möchte man in einer Webanwendung dem Client immer wieder aktuelle Daten übermitteln, muss man mit bisherigen Standardmitteln entweder mit *Polling* oder *Long Polling* arbeiten. Dabei hält der Server eine Verbindung sehr lange offen wenn keine Daten vorliegen - anstatt einfach eine leere Antwort zu schicken. Wenn dann Daten für den Client vorliegen, schickt der Server sie und schließt damit den HTTP/S-Request ab.

Mit dem *Long Polling* kann man vermeiden, dass der Client in sehr kurzen Abständen beim Server nachfragen muss ob irgendwelche Daten vorliegen. Jede Anfrage bringt einen gewissen Overhead mit, weil die Verbindungen aufgebaut werden und etliche Informationen mitgeschickt werden müssen.

Mit Websockets wird die Verbindung einmal aufgebaut und bleibt dann bestehen. So können sich Client und Server gegenseitig zu jeder Zeit Nachrichten zuschicken. Websockets funktionieren auch über Domaingrenzen hinweg - im Gegensatz zu AJAX-Requests.

Technisch gesehen in ein Websocket-Request nichts anderes als ein aufgewerteter *GET*-Request. Der Ablauf einer Websocket-Verbindung ist in Abbildung 1 dargestellt.

Werfen wir mal einen Blick auf die Header, die beim Verbindungsaufbau mitgeschickt werden. Im Folgenden ist die Anfrage des Clients zu sehen:

```
User-Agent: Mozilla/5.0 [...] Firefox/29.0
Upgrade:               websocket
Sec-WebSocket-Version: 13
Sec-WebSocket-Key: ZUu04F2K1Nt5ScTibzVdyg==
Pragma:                no-cache
Origin:                http://localhost:3000
Host:                 localhost:3000
Connection:            keep-alive, Upgrade
Cache-Control:             no-cache
Accept-Language:       de,en-US;q=0.7,en;q=0.3
Accept-Encoding:       gzip, deflate
Accept:                text/html
```

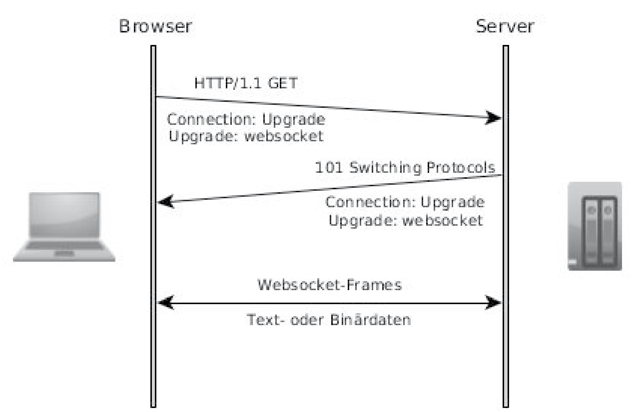

Abbildung 1: Websocket-Verbindung

Ein paar Begriffserklärungen hierzu:

• Origin
Feld muss mitgeschickt werden, Server kann damit Cross-Domain Anfragen handeln

• Sec-WebSocket-Key
zufälliger 16-Byte-Wert (Base64 kodiert)

• Sec-WebSocket-Protocol
optional, Unterstützung von Subprotokollen. Mittlerweile gibt es eine ganze Reihe von möglichen Subprotokollen, z.B. *WAMP* (*http://wamp.ws*) oder *AMQPWSB10* (Websocket Transport für AMQP 1.0).

• Sec-WebSocket-Extension
optional, Protokoll Erweiterung von Client unterstützt.

Was weiterhin in den Headern auffällt sind zwei Felder:

```
Upgrade:      websocket
Connection:   keep-alive, Upgrade
```

Wie oben schon kurz erwähnt wird die Websocketverbindung mit einem GET-Request aufgebaut. Das *Upgrade*-Feld wurde mit HTTP/1.1 eingeführt. Ein weiteres Einsatzgebiet für das Feld ist die Verwendung von *Transport Layer Security* (TLS). Der Client beginnt also mit einer normalen Klartext-Anfrage, die später auf eine neuere HTTP-Version oder ein anderes Protokoll umgestellt wird.

Prinzipiell können mehrere mögliche Protokolle angegeben werden:

```
Upgrade:  HTTP/2.0, SHTTP/1.3, IRC/6.9
```

Der Server kann dann prüfen welche Protokolle er versteht und kann dann umswitchen (wie, werden wir bei den Headern der Antwort sehen).

Diese Upgrade-Angabe gilt aber immer nur für die aktuelle Anfrage, deswegen muss das Schlüsselwort *Upgrade* im *Connection*-Feld auftauchen.

Die Header der Antwort sind in Listing 1 zu sehen.

Und der Status-Code ist `HTTP/1.1 101 Switching Protocols`.

Auch hier erst ein paar Begriffserklärungen:

• Sec-WebSocket-Accept
- Sec-WebSocket-Key verknüpft mit einer GUID verknüpft werden -> SHA1-Hash -> Base64
- Stellt sicher (Client), dass der Server die Anfrage verstanden hat.
- Ein falscher Wert wird als serverseitige Ablehnung verstanden

• Sec-WebSocket-Protocol
optional. Max. 1 Eintrag aus der Auswahl vom Client

• Sec-WebSocket-Extension
optional

Ist die Verbindung aufgebaut, kann man mit den Websockets arbeiten. Diese haben ein zwei Attribute, die Informationen zu den Websockets beinhalten. Das ist zum einen das Attribut `readyState`, das Informationen zum Status der Verbindung gibt. Das zweite ist ein Attribut, auf das nur lesend zugegriffen werden kann.

Folgende Werte sind für das Attribut erlaubt:

0: Verbindung wurde noch nicht aufgebaut
1: Verbindung aufgebaut, Kommunikation möglich
2: Befindet sich im *closing handshake*
3: Verbindung geschlossen oder nicht möglich

Da in der Regel Timeouts für die Verbindungen existieren, sollte man immer wieder den Status der Verbindung prüfen und ggf. neu mit dem Server verbinden.

```
Upgrade:              websocket
Server:               Mojolicious (Perl)
Sec-WebSocket-Accept: /3TMv7qMvMVspKwbO9nq2cYDiCk=
Date:                 Thu, 05 Jun 2014 22:46:55 GMT
Content-Length:       0
Connection:           Upgrade
```

Listing 1

Da die Kommunikation asynchron ist, muss man mit Events arbeiten um auf Nachrichten etc. reagieren zu können. Websockets kennen vier solcher Events:

- open

Mit einem Callback auf `Socket.onopen` kann man darauf reagieren wenn die Verbindung aufgebaut wurde.

- message

Mit `onmessage` kann man mit den Nachrichten arbeiten.

- error

Sollen Fehler behandelt werden, muss man auf mit einem Callback für `Socket.onerror` darauf reagieren.

- close

Wird die Verbindung geschlossen, wird das `close`-Event gefeuert. Dafür kann ein Callback für `Socket.onclose` definiert werden.

Ansonsten gibt es noch zwei Methoden: Mit `send` werden Nachrichten an den Server geschickt und mit `close` die Verbindung beendet.

Auf Clientseite muss alles mit JavaScript gemacht werden, auf der Serverseite gibt es viele Möglichkeiten. In diesem Artikel werden die Websockets mit Mojolicious behandelt. In Mojolicious ist ein Handling von Websockets nativ enthalten. Man braucht keine zusätzlichen Module oder Plugins.

Unsere Beispielanwendung soll nur relativ klein sein. Auf einem Server soll eine kleine Mojolicious-Anwendung sein, weswegen wir `Mojolicious::Lite` einsetzen. Der Code der Aktionen funktioniert genauso aber auch in einer größeren Mojolicious-Anwendung.

Besucht der Benutzer die Startseite der Anwendung bekommt er gleich ein paar Informationen aufgelistet. Um die Beispielcodes klein zu halten, wird hier nur eine reine Textdarstellung genommen und nicht irgendwelche schönen Grafiken.

Beginnen wir also mit einem `Mojolicious::Lite`-Gerüst:

```
#!/usr/bin/perl

use Mojolicious::Lite;

# more code comes here

app->start;
```

Als nächstes eine einfache Seite, auf der die Infos ausgegeben werden. Dazu brauchen wir eine Route und ein Template. Auf ein Layout verzichten wir hier der Einfachheit halber.

Folgender `__DATA__`-Bereich muss dann hinzugefügt werden:

```
__DATA__

@@monitor.html.ep
<!DOCTYPE html>
<html>
 <body>
   <table>
     <tr>
       <td>CPU-Load:</td>
       <td id="load"></td>
     </tr>
     <tr>
       <td>Uptime:</td>
       <td id="up"></td>
     </tr>
     <tr>
       <td>Freeram:</td>
       <td id="free"></td>
     </tr>
     <tr>
       <td>Totalram:</td>
       <td id="total"></td>
     </tr>
   </table>
   <input type="text" name="info_input"
          id="info_input" /><br />
   <div id="info"></div>
 </body>
</html>
```

Und wir brauchen noch eine Route.

```
get '/' => sub {
   shift->render( 'monitor' );
};
```

Soweit, so gut. In Abbildung 2 sieht man wie die Seite dann aussieht.

Bis jetzt ist noch nichts mit Websockets im Spiel. Das werden wir erst nach und nach einführen. Wie bereits erwähnt ist auf Clientseite JavaScript notwendig. Für die angenehmere Arbeit - wenn auch hier nicht unbedingt notwendig - verwenden wir hier jQuery. Außerdem brauchen wir noch Java

```
<head>
  <script src="//ajax.googleapis.com/ajax/libs/jquery/1.9.1/jquery.min.js"
     type="text/javascript"></script>
  <script src="/foo.js"></script>
</head>
```

Listing 2

vaScript, das die Websocket-Verbindung initiiert und die Nachrichten vom Server auswertet.

Das Template bekommt also noch einen `head`-Bereich (Listing 2)

Das *foo.js* ist unser eigenes JavaScript (Listing 3). Der Code kommt auch in den `__DATA__`-Teil mit einem vorangestellten `@@foo.js`. Hinweis: Sollen "statische" Inhalte mit Mojolicious::Lite aus dem `__DATA__`-Bereich ausgeliefert werden, dürfen die nur einen Dateisuffix (z.B. *.js*) haben. Man kann also nicht

```
@@jquery.min.js
...
```

machen, da das von Mojolicious::Lite als Template interpretiert werden würde.

In der Variablen `ws` wird die Websocket-Verbindung gehalten. Wenn das Dokument fertig geladen ist, wird mit `ws = new WebSocket( ws_url );` diese Verbindung aufgebaut. Danach kommen die Eventhandler. Im Falle eines Fehlers soll einfach eine Meldung auf der (JavaScript-)Konsole ausgegeben werden. Wenn die Verbindung aufgebaut ist, soll ebenfalls eine Meldung in der Konsole auftauchen.

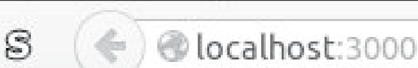

Abbildung 2: Eine einfache Seite für die Monitoringausgabe

Der wichtigste Handler ist hier der `onmessage`-Handler. Hier steht das drin, was gemacht werden soll wenn eine Nachricht vom Server kommt. Hier kommt einfach eine JSON-Struktur (siehe Listing 4) an, wovon die ganzen Informationen einfach in der HTML-Tabelle angezeigt werden soll.

Dann gibt es auf der Seite noch ein Eingabefeld. Sobald dort etwas eingegeben wird, wird die Nachricht an den Server geschickt (über die Methode `send` in dem `bind`). Der Server schickt diese Meldung an alle Clients. Das kann dann dazu verwendet werden, den Kollegen mitzuteilen, dass man neue Software auf dem Server installiert.

Kommen wir zum interessanteren Teil - der serverseitige Code. Wir brauchen jetzt eine Route, das von dem `ws = new WebSocket( ws_url );` angesteuert wird. Als Schlüsselwort steht hier `websocket` zur Verfügung.

```
var ws;

$(document).ready(function() {
   ws = new WebSocket( 'ws://server:3000/' );

   ws.onerror = function() {
       console.log( "an error occured" );
   };

   ws.onopen = function() {
       console.log(
           "ws connection established" );
   };

   ws.onmessage = function(msg) {
       var res = JSON.parse( msg.data );

       if ( res["info"] ) {
         $('#info').text( res["info"] );
         return;
       }

       $('#load').text( res["info"] );
       $('#up').text( res["info"] );
       $('#free').text( res["title"] );
       $('#total').text( res["title"] );
   };
});

$('#info_input').bind( 'change', function() {
  ws.send( $('#info_input').val() );
});
```

Listing 3

```
websocket '/' => sub {
  my $self = shift;
};
```

Auf eines ist hier zu achten: Diese Route muss vor dem `get '/' => sub {}` kommen. Da der Verbindungsaufbau für Websockets nur ein *GET*-Request ist, würde bei der falschen Reihenfolge eben diese `get`-Route zuschlagen.

In der Aktion an sich müssen wir zuerst einen Stream erzeugen, weil die Verbindung ja offen bleiben muss. Mojo kommt mit einer eigenen Eventloop, die wir hierfür verwenden können:

```
Mojo::IOLoop->stream(
    $self->tx->connection
)->timeout( 300 )
```

Für die aktuelle Verbindung (`$self->tx->connection`) wird dieser Stream erzeugt. Damit dieser nicht gleich wieder geschlossen wird, setzen wir ein Timeout von 300 Sekunden.

Da die Nachrichten aus dem Eingabefeld an alle Clients geschickt werden sollen, müssen wir uns die Clients merken. Dazu brauchen wir einen "globalen" Hash. Hier `%clients`.

```
my %client;
websocket '/' => sub {
  my $self = shift;
};
```

Für das Merken der Clients verwenden wir folgenden Code:

```
my $id = $self->tx =~ /(0x\w+)/ && $1;

$client{$id} = $self;
```

Als nächstes müssen wir die Nachrichten entgegennehmen und an alle Clients schicken.

```
$self->on( message => sub {
    my ($ws, $msg) = @_;

    for my $client_id ( keys %client ) {
        my $json = Mojo::JSON->encode({
           info => $msg,
        });

        $self->app->log->debug( $json );
        $client{$client_id}->send( $json );
    }
});
```

Wenn die Clients die Verbindung schließen, sollte diese aus der Liste der Clients gelöscht werden:

```
$self->on( finish => sub {
    delete $client{$id};
});
```

Das hier das Event nicht *close* heißt, wie es bei den Websockets genannt wird, liegt daran, dass hier die Mojo-eigene Eventloop verwendet wird. Dort heißt das Event *finish*.

Bleibt noch, den Clients regelmäßig die Monitoringinformationen zu schicken. Das soll in dieser Anwendung alle drei Sekunden passieren. Hier erzeugen wir mit `Mojo::IOLoop` einen Timer. Hier wird die Nachricht aber nicht an alle Clients geschickt sondern immer nur an den Client der die Verbindung aufgebaut hat.

```
my $scheduler =
 Mojo::IOLoop->recurring( 3 => sub {
   my $si    = sysinfo;
    my $json = Mojo::JSON->encode({
      uptime    => $si->{uptime},
      load      => $si->{load1},
       freeram   => $si->{freeram},
       totalram => $si->{totalram},
    });

    $self->app->log->debug(
       "Sending data: ".$json);
    $self->send( $json );
});
```

Was noch fehlt: Bis jetzt wird die Verbindung mit dem Timeout unterbrochen. Der Client sollte also regelmäßig etwas über die Leitung schicken, damit die Verbindung erhalten bleibt. Im Code, der zum Download bereitsteht, ist diese Änderung bereits enthalten.

Zum guten Schluss sollen die Websocketverbindungen getestet werden, um sicherzustellen, dass der Server auf bestimmte Nachrichten richtig reagiert und auch regelmäßig Daten schickt. Was hierbei nicht getestet wird ist das JavaScript.

Für das Testen von Websockets, stellt Mojolicious auch einige Methoden bereit - in `Test::Mojo`.

Als erstes sollte man testen, ob eine Verbindung aufgebaut werden kann und ob der richtige Status-Code zurückgeliefert wird. Wir erinnern uns, dass bei einem erfolgreichen Verbindungsaufbau der Status `101` zurückgeliefert wird, weil ein Upgrade des Protokolls vorgenommen wird.

```
use Test::More;
use Test::Mojo;

do './server';

my $t  = Test::Mojo->new();
my $ws = $t->websocket_ok( '/' );

$ws->status_is( 101 );

done_testing();
```

Das `do './server'` ist notwendig weil `Mojolicious::Lite` verwendet wird. Ohne diesen Befehl weiß `Test::Mojo` nicht, welche Anwendung gestartet werden soll. Bei Mojolicious-Anwendungen kann man `new` den Anwendungsnamen übergeben.

Der nächste Schritt ist es, das Schicken der Nachrichten an den Server zu testen und zu prüfen ob der Server auch die richtige Nachricht zurückschickt.

```
$ws->send_ok( 'tester2' );
$ws->message_ok();
$ws->message_is( '{"info":"tester2"}');
```

Mit `send_ok` wird geprüft, ob das Senden an sich fehlerfrei funktioniert hat, während `message_ok` einfach nur prüft, ob eine Nachricht vom Server ankam. Ob diese Nachricht auch richtig aussieht, kann man mit `message_is` prüfen. Weiß man nicht genau wie die Nachricht aussieht, kann man mit `message_like` arbeiten. Wie bei `Test::More` gibt es auch noch die umgekehrten Funktionen. In diesem Fall `message_isnt` und `message_unlike`.

Abschließend wird mit

```
$ws->finish_ok;
```

getestet ob die Verbindung ordnungsgemäß geschlossen werden kann.

Jetzt geht es noch darum, die Monitoringnachrichten zu testen.

```
my $monitoring = $t->websocket_ok('/');
sleep 4;
$monitoring->message_ok;
$monitoring->json_message_has('/uptime');
$monitoring->json_message_has('/load');
$monitoring->json_message_has('/freeram');
$monitoring->json_message_has('/totalram');
$monitoring->finish_ok;
```

Da die Monitoringinformationen nur alle drei Sekunden wird erstmal vier Sekunden gewartet. Mit `json_message_has` kann festgestellt werden, ob eine JSON-Struktur ein bestimmtes Element hat. Da wir JSON verwenden, können wir diese Methode verwenden. Die JSON-Nachricht des Servers wird automatisch in eine Perl-Datenstruktur umgewandelt. Als Parameter kann man einen XPath-angelehnten Selector übergeben.