Moderne Echtzeitkommunikation mit HTML5 und WebSockets[1]

Prof. Dr. Stefan Zander

9. Oktober 2017

[1] Ausgewaehltes Kapitel im Rahmen der Lehrveranstaltung "Entwicklung Webbasierter Anwendungen"

Lernziele:

- *Kennenlernen der Grundtechniken moderner HTML5 Echtzeitanwendungen*
- *Aufbau des WebSocket Protokolls*
- *Entwicklung erster WebSocket-basierter Client-Server-Anwendungen*
- *Entwicklung einfacher WebSocket-Server mittels JavaScript und Node.js*

Contents

1 *Motivation*

HTTP is a *stateless* request-response protocol that requires the client to open a connection to a server in order to send and receive data. The connection is closed when the server delivered its response. If the client requires additional or updated information, it has to establish a new connection to the server again and issue a request.

HTTP is stateless

HTTP closes the connection after every response

This scenario has two main bottlenecks:

1. It does not allow the server to send notifications to the client (so-called *PUSH notifications*) without an a prior client-side request.
2. For every client's request, a new connection with the entire communication overhead has to be established in order to exchange data with a server.

The consequences of both bottlenecks are that HTTP is not suited for implementing the *Observer pattern* in Web Applications as well as the *Model-View-Controller* architecture, since a model is not able to notify view autonomously in case of any updates. More severely, HTTP is not suited for the realization of pure *real-time Web applications* in which events are sent between clients and servers upon their occurrence, ie., at the time when they are created or captured. These issues are addressed by the WebSocket protocol and its accompanying technology stack.

Per default, HTTP does not support the Observer pattern and a Model-View-Controller architecture

WebSockets create a bi-directional communication channel between a server and a client. This channel can be used by clients and servers likewise to send textual and/or binary data[2]. One of the big advantages of using WebSockets for real-time communication on the Web is its little overhead; the header of WebSocket messages only requires 2 to 6 bytes.

[2] Some websites try to circumvent ad blockers by sending advertisements through WebSocket connections.

The WebSocket standard consists of two main elements:

1. The **WebSocket Protocol**—standardized through the ITEF in RFC6455
2. The **JavaScript Webbrowser API**—standardized through the World Wide Web Consortium (W3C)

1.1 Polling and Long-Polling

Many of today's (near) real-time Web applications are realized trough the principles of *polling* or *long-polling*. During polling, the client issues request to the server according to a pre-determined frequency, e.g., every 2 seconds. Those requests are answered immediately by the server, even in cases when no new data are available . The following Javascript code excerpt demonstrates this principle.

Listing 1: A simple Javascript code that demonstrates the polling principle

```
function poll() {
  xhr.send();
  clearTimeout(timeoutId);
  timeoutId = setTimeout(poll(), 2000);
}
timeoutId = setTimeout(poll(), 2000);
```

The `send()` method is repeatedly executed through the `setTimeout()` method. By means of this principle, resources hosted on a server will be repeatedly requested and processed on the client side. In case there are no changes since the last issued request, the server answers with an empty response (see Figure 1).

The server sends an empty response in cases nothing has changed since the last request.

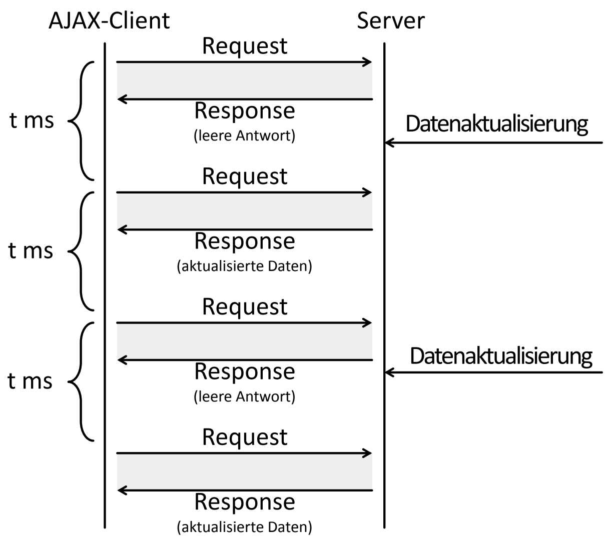

Figure 1: Principle of HTTP Polling (reprinted from Gorski et al. [2015])

Polling can be easily implemented in a browser-independent manner. However, due to many unnecessary requests and computational resources needed to process such requests, polling creates substantial overhead that is not to be underestimated. An important aspect in this context is the *temporal resolution* through with requests are issued. For a applications that are required to exhibit a near real-time behavior such as auction or chat systems, the temporal resolution needs to be of rather fine granularity. The requirements of other applications such as monitoring systems for plants might be satisfied with a larger temporal frequency.

Polling is browser-independent.

Short update frequencies cause substantial processing overhead in Polling.

Long-Polling is intended to reduce the additional overhead caused by empty responses send through polling. In Long-Polling, the server answers promptly *only* in case of changes; if no change takes place, the server waits a pre-defined amount of time before it sends an empty response. Iff a change is detected during that time, the server answers promptly (as described before). The connection to the client remains open during the time of waiting and will be closed upon sending an immediate response or an empty message. The client however, immediately issues a new request to the server in both cases. This procedure resembles a permanent connection between client and server. Figure 2 demonstrates this principle.

During Long-Polling the server answers promptly when detecting a change event.

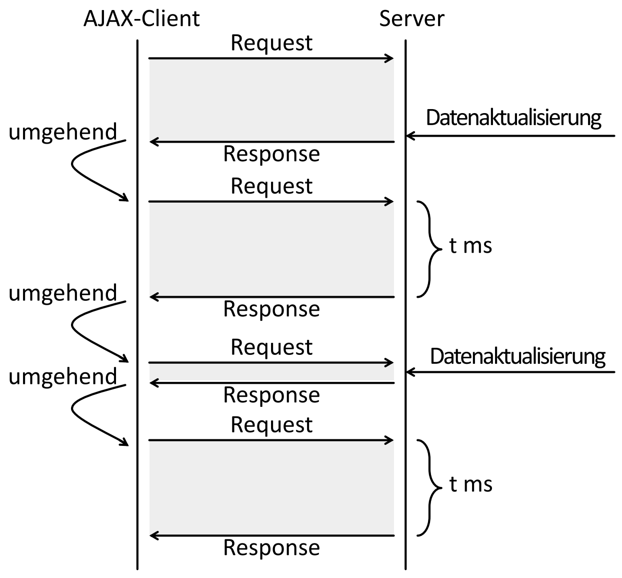

Figure 2: Principle of Long-Polling (reprinted from Gorski et al. [2015])

2 *Establishing a WebSocket Connection*

This section details all the steps involved in the set-up a WebSocket connection. The set-up is triggered by a HTTP request-response communication between a client and a server.

Upon establishing a WebSocket connection, the client sends an *opening handshake*[3] on the basis of HTTP in order to open a bi-directional *WebSocket Channel* through which messages between a client and the server are exchanged. The client therefore sends a special HTTP GET request as depicted subsequently.

[3] The opening handshake is also called "<WebSocket Handshake">

```
GET /chat HTTP/1.1
Host: domain.net
Connection: Upgrade
upgrade: websocket
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Origin: http://domain.net
Sec-WebSocket-Version: 13
Sec-WebSocket-Protocol: chat, superchat
Sec-WebSocket-Extensions: x-webkit-deflate-stream
[...]
```

- "<`GET /chat`"> addresses the WebSockets endpoint; a WebSocket server can offer multiple endpoints.
- "<`Connection: Upgrade`"> requests the server to switch from HTTP to another protocol, signaled through the "<`upgrade: websocket`"> header.
- "<`Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==`"> is a Base64-coded string that contains a randomly created number; this number is used by the client to test whether the server is able to process WebSocket requests, i.e., whether it supports the WebSockets Protocol.
- "<`Origin: http://domain.net`"> signals the server the domain from which the request has initially been sent. Upon this information, the server can decide whether to accept an connection request or not.
- "<`Sec-WebSocket-Version: 13`"> indicates the WebSocket Protocol version used for the communication channel.
- "<`Sec-WebSocket-Protocol: chat, superchat`"> allows the client to signal the server the subprotocols it is able to process through the WebSocket channel. The same principle also applies to the extension header "<`Sec-WebSocket-Extensions:`">.

Response sent by the server

The client can send multiple extensions whereas the server has the privilege to answer only to those it supports. The subsequently given HTTP excerpt displays a response sent by the server upon accepting a request:

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
Sec-WebSocket-Protocol: chat
```

Upon accepting the client's request, the server creates a new value for the header "<`Sec-WebSocket-Accept`"> by appending its / a Globally Unique Identifier (GUID)

```
258EAFA5-E914-47DA-95CA-C5AB0DC85B11
```

to the client's handshake request "<`Sec-WebSocket-Key`"> header value ("<`dGhlIHNhbXBsZSBub25jZQ==`">).

The merged text string

```
dGhlIHNhbXBsZSBub25jZQ==258EAFA5-E914-47DA-95CA-C5AB0DC85B11
```

will then be converted into a hash value using the SHA-1[4] cryptographic hash function and transformed into a Base64-coded text string. This text string "<`s3pPLMBiTxaQ9kYGzzhZRbK+xOo=`"> consequently represents the value of the `Sec-WebSocket-Accept` header.

[4] `https://en.wikipedia.org/wiki/SHA-1`

With the `Sec-WebSocket-Accept` header, the client is able to extract its initially sent `Sec-WebSocket-Key` header value in order to verify the server's integrity, i.e., to check wether the response originally came from the intended server.

3 *WebSocket Frames*

In the WebSocket Protocol, data is transmitted as a *sequence of frames*. A Websocket Frame consists of two parts:

1. A **header** that contains control data
2. A **payload** containing user data

A simplified illustration of a WebSocket Frame according to the RFC 6455 Specification is depicted in Figure 3.

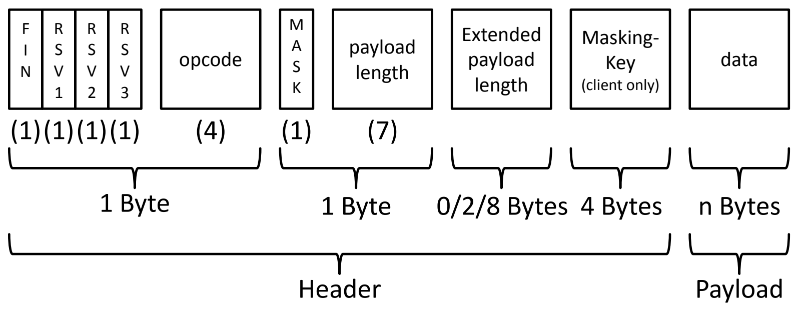

Figure 3: A simplified representation of a WebSocket Frame according to the RFC 6455 Specification (re-printed from [Gorski et al., 2015])

From this picture, it is immediately obvious how lean the WebSocket Protocol is; its header is only a fraction of the size an HTTP header requires. A header sent from the server to the client only requires 2 Bytes in case of default size specifications. Frames sent from the client to the server need to be masked with a 4 Byte key;

the minimum size of such a frame's header is thus 6 Byte. When the header contains additional user data, e.g., in case of extended payload length information, the header grows by 2 or 8 Byte to a maximal length of 14 Bytes.

The base framing protocol defines a frame type with an `opcode`, a `payload length`, and designated locations for `Extension data` and `Application data`, which together define the `Payload data`. Certain bits and opcodes are reserved for future expansion of the protocol.

A data frame MAY be transmitted by either the client or the server at any time after opening handshake completion and before that endpoint has sent a `Close` frame.

3.1 *Structure*

The WebSocket Specification defines the base framing protocol using the *Augmented Backus-Naur-Form (ABNF)*[5]. The structure of the WebSocket Protocol header is depicted below followed by a brief description of its main constituents.

[5] More details about the ABNF are given in the RFC5234: `https://tools.ietf.org/html/rfc5234`

```
  0                   1                   2                   3
  0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
 +-+-+-+-+-------+-+-------------+-------------------------------+
 |F|R|R|R| opcode|M| Payload len |    Extended payload length    |
 |I|S|S|S|  (4)  |A|     (7)     |             (16/64)           |
 |N|V|V|V|       |S|             |   (if payload len==126/127)   |
 | |1|2|3|       |K|             |                               |
 +-+-+-+-+-------+-+-------------+ - - - - - - - - - - - - - - - +
 |     Extended payload length continued, if payload len == 127  |
 + - - - - - - - - - - - - - - - +-------------------------------+
 |                               |Masking-key, if MASK set to 1  |
 +-------------------------------+-------------------------------+
 | Masking-key (continued)       |          Payload Data         |
 +-------------------------------- - - - - - - - - - - - - - - - +
 :                     Payload Data continued ...                :
 + - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - +
 |                     Payload Data continued ...                |
 +---------------------------------------------------------------+
```

Figure 4: Structure and constituents of the WebSocket Protocol header

`FIN: 1 bit`

Indicates that this is the final fragment in a message. The first fragment MAY also be the final fragment. If the `FIN` bit is 0, the server will keep listening for more parts of the message; otherwise, the server considers the message delivered.

`RSV1, RSV2, RSV3:  1 bit each`

MUST be 0 unless an extension is negotiated that defines meanings for non-zero values. If a nonzero value is received and none of the negotiated extensions defines the meaning of such a nonzero value, the receiving endpoint MUST Fail the WebSocket Connection.

`Opcode: 4 bits`

Defines the interpretation of the `Payload data`. If an unknown opcode is received, the receiving endpoint MUST Fail the WebSocket Connection. The following values are defined.

- `%x0` denotes a continuation frame
- `%x1` denotes a text frame
- `%x2` denotes a binary frame
- `%x3-7` are reserved for further non-control frames
- `%x8` denotes a connection close
- `%x9` denotes a ping
- `%xA` denotes a pong
- `%xB-F` are reserved for further control frames

`Mask: 1 bit`

Defines whether the `Payload data` is masked. If set to 1, a masking key is present in masking-key, and this is used to unmask the `Payload data`. All frames sent from client to server have this bit set to 1.

`Payload length: 7 bits, 7+16 bits, or 7+64 bits`

The length of the `Payload data`, in bytes: if 0-125, that is the payload length. If 126, the following 2 bytes interpreted as a 16-bit unsigned integer are the payload length. If 127, the following 8 bytes interpreted as a 64-bit unsigned integer (the most significant bit MUST be 0) are the payload length. Multibyte length quantities are expressed in network byte order. Note that in all cases, the minimal number of bytes MUST be used to encode the length, for example, the length of a 124-byte-long string can't be encoded as the sequence 126, 0, 124. The payload length is the length of the `Extension data` + the length of the `Application data`. The length of the `Extension data` may be zero, in which case the payload length is the length of the `Application data`.

To read the payload data, you must know when to stop reading. That's why the payload length is important to know. Unfortunately, this is somewhat complicated. To read it, follow these steps:

1. Read bits 9-15 (inclusive) and interpret that as an unsigned integer. If it's 125 or less, then that's the length; you're done. If it's 126, go to step 2. If it's 127, go to step 3.
2. Read the next 16 bits and interpret those as an unsigned integer. You're done.
3. Read the next 64 bits and interpret those as an unsigned integer (The most significant bit MUST be 0). You're done.

Source: `https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API/Writing_WebSocket_servers`

`Masking-key: 0 or 4 bytes`

All frames sent from the client to the server are masked by a 32-bit value that is contained within the frame. This field is present if the mask bit is set to 1 and is absent if the mask bit is set to 0.

`Payload data: (x+y) bytes`

The `Payload data` is defined as `Extension data` concatenated with `Application data`.

`Extension data: x bytes`

The `Extension data` is 0 bytes unless an extension has been negotiated. Any extension MUST specify the length of the `Extension data`, or how that length may be calculated, and how the extension use MUST be negotiated during the opening handshake. If present, the `Extension data` is included in the total payload length.

`Application data:  y bytes`

> Arbitrary `Application data`, taking up the remainder of the frame after any `Extension data`. The length of the `Application data` is equal to the payload length minus the length of the `Extension data`.

3.2 *Fragmentation*

The WebSocket Specification allows a fragmented transmission of user data over a WebSocket Channel, i.e., data might be sent "piece-by-piece" over the communication channel. This feature is useful in scenarios where the data to be transmitted are not completely available or buffered at the time a request was issued. Setting the `FIN`-bit to `0` signals the communication partner that further frames are expected to arrive unless a request or response is complete.

3.3 *Masking*

To avoid confusing network intermediaries (such as intercepting proxies) and for security reasons[6], a client MUST *mask* all frames that it sends to the server. The server, on the contrary, MUST close the connection to the client upon receiving a frame that is not masked. The client MUST do the same in cases when it receives masked frames from a server. More details about the masking semantics are expounded in Section 5.1 of the WebSocket Specification[7].

[6] See Section 10.3 of the WebSocket specification under `https://tools.ietf.org/html/rfc6455#section-10.3`

[7] See `https://tools.ietf.org/html/rfc6455#section-5`

Both the client or the server can choose to send a message at any time — that is one of the main beneficial features of WebSockets. However, extracting information from these frames of data requires some additional work. Although all frames follow the same specific format, data going from the client to the server is masked using *XOR encryption*[8] (with a 32-bit key). Section 5.3 of the WebSocket Specification describes this in detail.

[8] `https://en.wikipedia.org/wiki/XOR_cipher`

For every new frame that is to be transmitted as part of a communication, the client MUST generate a unique 32-Bit random number, that acts as *Masking Key*. The masking key is then inserted as value in the `Masking-key` header field. It will then be used to mask the payload data by applying it periodically to its bit values using an *XOR* operator. The following algorithm demonstrates this principle.

The Masking Key of a data frame sent by the client is a randomly created unique 32-Bit number.

Payload data will be masked with the Masking Key using an XOR operator.

```
byte[] maskingKey; // 4 byte randomly created Masking Key
byte[] payloadData; // data to be transmitted to the server
byte[] maskedData; // masked data that are to be created

for(int i=0; i<payloadData.length; i++)
  maskedData[i] = payloadData[i] ^ maskingKey[i%4];
```

The payload's and masking key bytes are bitwise connected using the XOR operator. Since the XOR operation is self-inverse such as

```
payloadData[i] ^ maskingKey[i%4] ^ maskingKey[i%4] = payloadData[i]
```

the server is able to recreate, i.e., to unmask the payload data sent by the client. xxx contains a good example that demonstrates this principle.

3.4 *Frametypes*

As mentioned previously, the WebSocket Protocol is able to sent both text-based and binary payload data frames. In the following, we give a brief overview of the different frame types defined in the WebSocket Specification.

- **Textdata Frames**
  Textdata frames are configured using the `0x1 Opcode` header value. The payload data of such frames is usually encoded using the UTF-8 character encoding.

- **Binarydata Frames**
  Binarydata frames are initiated with the `0x2 Opcode`. By setting the `Extended Payload Length` header, an endpoint is able to transfer payload data in exabyte sizes. Additionally, a fragmented transmission can also be initiated.

- **Control Frames**
  Control frames are divided into a *Ping-Frame* that is sent in order to check whether an established WebSocket connection is still active or to measure the latency between a client and a server. If an endpoint receives a Ping-Frame it MUST answer with a *Pong-Frame*. The third control frame type is the *Close-Frame*, which is used to close an active WebSocket connection. If a server or client receives a Close-Frame, it MUST answer with another Close-Frame in order to initiate a *closing handshake*. The closing handshake is complete when both endpoints have sent and received a Close-Frame.

4 *Protocol Analysis Tools*

To be added....

5 *Creating a WebSocket Client*

The WebSocket technology is a fixed element in the HTML5 family of standards; as such WebSockets are supported by all major desktop and mobile browsers. Information whether a specific browser version incorporates WebSocket support can be obtained from the website

`http://www.caniuse.com/` .

In order to find out whether a browser supports the WebSockets technology and implements both protocol and API, its `window` object MUST contain a `WebSocket` element. This element is missing in cases when a browser lacks native WebSockets support. WebSocket browser support can be checked *programmatically* with the following minimal JavaScript code.

Listing 2: A minimal website with JavaScript embedded to test WebSocket browser support

```
<!doctype html>
<html>
<head>
  <title>Websocket Browser Support</title>
  <meta charset="utf-8" />
</head>
<body>
  <div id="message">
  </div>
  <script type="text/javascript">
    var parent = document.getElementById("message");
    var node = document.createTextNode("");
    parent.appendChild(node);
    if ("WebSocket" in window) {
      node.textContent = "Prima! WebSockets werden unterstuetzt.";
    } else {
      node.textContent = "Schade, WebSockets werden nicht von
    Ihrem Browser unterstuetzt.";
    }
  </script>
</body>
</html>
```

5.1 *Namingscheme*

A WebSocket server can be accessed using the following naming scheme[9]:

```
"ws:" "//" host [ ":" port ] path [ "?" query ]
```

[9] The Augmented Backus-Naur-Form as defined in RFC 3986 is used as naming scheme notation.

Optional elements are enclosed in brackets (‘[’ and ‘]’); Strings remain “as is”; and normal terms are replaced with concrete values. As a consequence, a WebSocket URL differs from a usual Web URL only by the protocol element.

The RFC 6455 Specification also defines a second WebSocket URL protocol type "<`wss`"> for secure WebSocket connections—comparable to "<`https:`">.

5.2 *Creating a WebSocket Connection Object*

For creating a WebSocket connection object, the W3C defines a standard JavaScript WebSocket API. A connection to a WebSocket server using the endpoint URL

```
ws://echo.websocket.org/
```

can be created in JavaScript using the following line of code

```
var ws = new WebSocket('ws://echo.websocket.org');
```

The WebSocket constructor takes as first argument a string that represents the URL of the WebSocket endpoint; specific subprotocols such as SOAP etc. can be passed as second optional parameter to the WebSocket constructor.

5.3 WebSocket States

The successful creation of a WebSocket object implies a successful handshake between the client and server; the handshake is automatically conducted during the instantiation of the object. The WebSocket Specification defines *four states* for the WebSocket object that represent the current status:

```
CONNECTING --> readyState: 0
      OPEN --> readyState: 1
   CLOSING --> readyState: 2
    CLOSED --> readyState: 3
```

Figure 5 represents the different states of the WebSocket object as well as transitions between them:

After the creation of the WebSocket object, it remains in the state `CONNECTING` for a brief moment; the `readyState` attribute hence carries the value `0`. During the `CONNECTING` phase, the WebSocket object tries to establish a connection to the server and initiate the handshake. If the handshake failed, the object transitions to state `CLOSED`; the `readyState` attribute carries the value `3`. In case of successful handshake, the objects transitions to state `OPEN` with `readyState` value of `1`. The WebSocket object is able to send and receive frames in this state. This state lasts until either one of the partners initiates a closing handshake or in case an error occurs.

5.4 Eventhandler

Event handler in general are used to react to state transitions. When the state of the WebSocket object changes, the corresponding event is created. This event is then propagated to the respective handler and will be processed by it. I.e., the handler defines the behavior that will be executed when the corresponding event occurs, i.e., when the WebSocket objects transitions to a specific state. The WebSocket Standard defines the following event handlers and corresponding states:

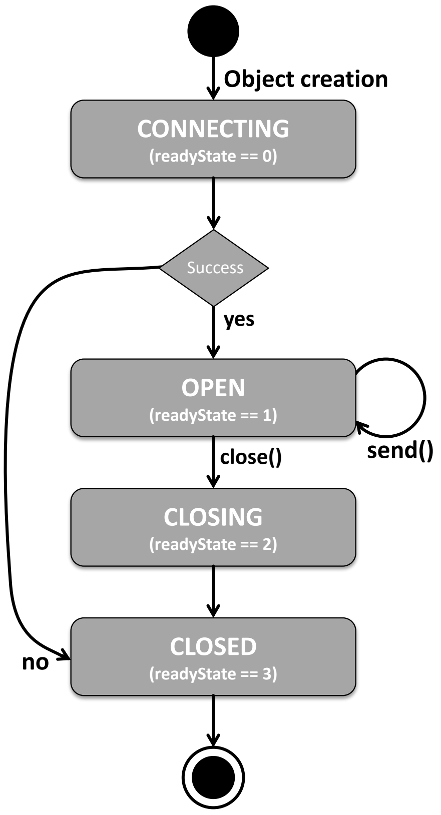

Figure 5: The different states a WebSocket object can be in together with transitions. (re-printed from Gorski et al. [2015])

```
  Event Handler          Event
---------------------------------
  onopen        -->      open
  onerror       -->      error
  onmessage     -->      message
  onclose       -->      close
```

Usually, event handler are represented as functions, sometimes anonymous functions, that are assigned to specific handler attributes (properties) of the WebSocket object. A handler is a function that will be executed when the event the handler is registered for occurs. Since JavaScript treats functions as objects, handler functions can be assigned to the event handler properties (`onopen`, `onerror`, etc.) of the WebSocket object. Not every event handler need to be connected with a handler function[10]; it is absolutely ok to define handler functions exclusively for the events of interest. Following a good programming style, however, handler functions should be defined for all possible events that can occur in order to achieve robustness and minimize the risk of unwanted and unintended side effects.

[10] For instance, a sensor that distributes sensed values (so-called *obserables*) unidirectionally via the WebSocket protocol might only define handler functions for opening or closing a connection but not for receiving information from a server.

5.5 JavaScript Client API

The following JavaScript program[11] represents the set of handlers, a WebSocket client needs to implement in order to establish a WebSocket connection to an endpoint. The code fragment shows how to

[11] Please note that checks whether a browser supports the WebSocket protocol have been omitted due to readability and comprehensibility reasons.

initiate a connection to a WebSocket server, how to send messages to the server, and how to print out received messages to the JavaScript console. It includes handler functions for all major events.

Listing 3: A simple JavaScript client for accessing an echo WebSocket server

```
 1 var ws = new WebSocket('ws://echo.websocket.org/');
 2 
 3 ws.onopen = function() {
 4   console.log('WebSocket-Verbindung aufgebaut.');
 5   ws.send('Hallo WebSocket Endpoint!');
 6   console.log('Uebertragene Nachricht: Hallo WebSocket Endpoint!')
      ;
 7 };
 8 
 9 ws.onmessage = function(message) {
10   console.log('Der Server sagt: ' + message.data);
11   ws.close();
12 };
13 
14 ws.onclose = function(event) {
15   console.log('Der WebSocket wurde geschlossen oder konnte nicht
      aufgebaut werden.');
16 };
17 
18 ws.onerror = function(event) {
19   console.log('Mit dem WebSocket ist etwas schiefgelaufen!');
20   console.log('Fehlermeldung: ' + event.reason + '(' + event.code
      + ')');
21 };
```

Due to the asynchronous character of JavaScript, the script code is not blocked when the client tries to establish a connection to the echo server. After the processing of line 1 (instantiation of the WebSocket object), the runtime environment continues executing the script and does not wait until the handshake is complete. Technically, the instantiation is asynchronously executed in another internally managed thread. Upon executing the script, for anonymous handler functions are assigned to their respective event handler objects. The event handler function for the `onopen` event notifies the user in case the handshake is complete by printing a message to the JavaScript console. The `ws` object then transitions into the `OPEN` state and creates a WebSocket text frame, which is to be sent to the server using the `ws.send()` method. When the client receives a response, which is also a text frame, the browser calls the `ws.onmessage()` handler function that prints the message to the JavaScript console and closes the connection.

Listing 3 demonstrates how handler functions can be defined for events defined by the WebSocket Specification. In order to achieve a greater degree of flexibility, event handlers can also be defined using the `addEventListener()` method alternatively, which allows to define *multiple handlers* for a single event. Listing 4 illustrates this principle for the `open` event:

```
ws.addEventListener("open", function() {
  console.log('Die WebSocket-Verbindung wurde aufgebaut.');
  ws.send('Hallo WebSocket Endpoint!');
  console.log('Uebertragene Nachricht: Hallo WebSocket Endpoint!')
    ;
};
```

Listing 4: Using the method `addEventListener()` to register multiple event handlers for a single event.

6 *Creating a WebSocket Server in Node.js*

This section describes the creation of an echo WebSocket server using *Node.js*[12] and the *WebSocket.io*[13] module. However, a WebSocket implementation is available for literally all major server-side frameworks and languages. This module represents a WebSocket implementation for Node.js and offers an API to utilize its functionality. More information about how to install Node.js for your target platform and operating system respectively can be obtained from the official Node.js website as well as from the Node.js section in this lecture notes. Please bear in mind that the Node.js section is *work in progress* and only provides information on a rather generic and basic level.

[12] `https://nodejs.org/en/download/`

[13] `https://www.npmjs.com/package/websocket.io`

In order to create a simple echo server using WebSockets.io and Node.js, a few initial steps need to be conducted. As a first step, a project directory has to be created, e.g., using the following terminal commands:

```
mkdir echo_websocket_io
cd echo_websocket_io
```

As a next step, the WebSocket.io module needs to be downloaded and installed using Node.js' module manager `npm` by entering the following command in the project folder:

```
npm install websocket.io
```

Now, we can create a file (e.g., `echoServer.js`) hosting the server code; please note that the file must have the suffix `.js`. Listing xxx expounds the code for an elementary echo server.

The `WebSocket.io` module is imported in line 1, together with the basis modules `fs` for reading the `index.html` file from the file system and `http` for creating a http server. The lines 5 to 18 hold the code for creating an elementary Web server that listens for incoming http requests and answers all requests by delivering the `index.html` file. This file serves as the client and contains the client code as described in Section 5.5. It is responsible for establishing a connection to the server. The WebServer is instantiated by combining the

```
var ws = require(’websocket.io’),
    fs = require(’fs’),
    http = require(’http’);

// create WebServer and deliver ’index.html’ upon request
var httpServer = http.createServer(function (request, response) {
    fs.readFile(__dirname+"/index.html", function(error, data) {
        if (error) {
            response.writeHead(500);
            return response.end("Fehler beim Laden der Datei \"index.html\"");
        }
        else {
            console.log("index.html was requested");
            response.writeHead(200);
            response.end(data);
        }
    });
});

//combine WebServer with WebSocket-Server
var wsServer = ws.attach(httpServer);

wsServer.on(’connection’, function(client) {
    client.on(’message’, function(message) {
        console.log("[WebSocket-Server] Sending message: ’" + message + "’");
        client.send(message);
    });
});

httpServer.listen(4000);
console.log("Der Echo-Server laeuft auf dem Port: " + httpServer.address().port);
```

Listing 5: An elementary WebSocket echo server for Node.js using the `WebSocket.io` module

WebSocket.io module with the Web server (see line 21). Line 23-28 specifies the behavior of the WebSocket server: the server answers all requests by sending the echo message back to the client. Line 30 specifies the port (4000) through which the server can be requested.

The echo server can now be started by opening the project folder and entering the following command in the console:

```
node echoServer.js
```

In case everything was implemented correctly, the following message will be printed to the console:

```
Der Echo-Server laeuft auf dem Port: 4000
```

The index.html can now be requested by opening a WebSocket-enabled browser and entering the following URL in the browser's address field:

```
http://localhost:4000
```

The index.html file is depicted below:

Listing 6: HTML part of the WebSocket echo client

```html
<!doctype html>
<html>
<head>
    <title>Simple WS Echo Client</title>
    <meta charset="utf-8" />
</head>
<body>
    <header>
        <h2>WebSocket Client for Echo Server</h2>
    </header>
    <div>
        <label>Message:</label>
        <input type="text" id="message" placeholder="enter echo
    string here..."/>
        <input type="button" id="send" value="Send" onclick="
    sendMessage()" />
        <input type="button" id="close" value="Close Connection"
    onclick="closeConnection()" />
    </div>
    <div id="log_container">
        <p/>Log: <br />
        <textarea cols="80" rows="25" id="log"></textarea>
    </div>

    <!-- hier stehen die WebSocket aufrufe -->
    <script> ... </script>
</body>
</html>
```

Listing 7: JavaScript part of the WebSocket echo client

```
<script type="text/javascript">
    var parent = document.getElementById("log");
    var node = document.createTextNode("");
    parent.appendChild(node);
    if ("WebSocket" in window) {
        node.textContent = "Prima! WebSockets werden unterstuetzt.";
    } else {
        node.textContent = "Schade, WebSockets werden nicht von Ihrem Browser unterstuetzt";
    }

    var ws = new WebSocket("ws://" + window.location.host);

    ws.onopen = function() {
        parent.textContent = parent.textContent + "\nWebSocket-Verbindung aufgebaut mit " + ws.url; };

    ws.onmessage = function(message) {
        var data = message.data;
        parent.textContent = parent.textContent + "\n" + "Echo-Server antwortet: " + data.toString(); };

    ws.onclose = function(event) {
        if (this.readyState == 2) {
            parent.textContent = parent.textContent + "\n" + "Schliesse Verbindung zum Server...";
            parent.textContent = parent.textContent + "\n" + "Die Verbindung durchlaeuft den Closing Handshake...";
        }
        else if (this.readyState == 3) {
            parent.textContent = parent.textContent + "\n" + "Verbindung geschlossen!";
        }
        else {
            parent.textContent = parent.textContent + "\n" + "Unbekannter ReadyState: " + this.readyState;
        }
    };

    ws.onerror = function(event) {
        var reason = event.reason;
        var code = event.code;
        parent.textContent = parent.textContent + "\n" + "Ein Fehler ist aufgetreten: " + reason + " - " + code; }

    function sendMessage() {
        "use strict";
        var message = document.getElementById("message").value;
        parent.textContent = parent.textContent + "\n" + "Sending Message to Echo-Server: " + message;
        ws.send(message); }

    function closeConnection() {
        "use strict";
        ws.close();  }
</script>
```

7 *Digression: Node.js*

Node.js[14] (sometimes simply abbreviated as *Note*) is an open-source, cross-platform, runtime environment that allows developers to create all kinds of server-side tools and applications in JavaScript. The runtime is intended for usage apart of a browser context (i.e. running directly on a computer or server OS). As such, the environment omits browser-specific JavaScript APIs and adds support for more traditional OS APIs including HTTP and file system libraries.

[14] `https://nodejs.org/`

From a web server development perspective, Node.js offers a number of benefits, which are listed below:[15]

[15] This enumeration is taken from the Modzilla Web Development Portal: `https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/Introduction`

- Great performance! Node has been designed to optimize throughput and scalability in web applications and is a very good match for many common web-development problems (e.g. real-time web applications).
- Code is written in "plain old JavaScript", which means that less time is spent dealing with *context shift* between languages when both browser and web server code is written.
- JavaScript is a relatively new programming language and benefits from improvements in language design when compared to other traditional web-server languages (e.g. Python, PHP, etc.) Many other new and popular languages compile/convert into JavaScript so you can also use CoffeeScript, ClosureScript, Scala, LiveScript, etc.
- The node package manager (NPM) provides access to hundreds of thousands of reusable packages. It also has best-in-class dependency resolution and can also be used to automate most of the build toolchain.
- It is portable, with versions running on Microsoft Windows, Mac OS, Linux, Solaris, FreeBSD, OpenBSD, WebOS, and NonStop OS. Furthermore, it is well-supported by many web hosting providers, that often provide specific infrastructure and documentation for hosting Node sites.
- It has a very active third party ecosystem and developer community, with lots of people who are willing to help.

7.1 *A First minimal Web Server in Node.js*

For a first exercise, create a simple web server that is able to respond to any request using the Node HTTP package. The server will listen for any kind of HTTP request on the URL `http://127.0.0.1:8000/` and responds with a plain-text message `"<Hello World">` when a request is received.

```
//Load HTTP module
var http = require("http");

//Create HTTP server and listen on port 8000 for requests
http.createServer(function (request, response) {

   //Set the response HTTP header with HTTP status and Content
     type and we add a little bit more text to it
   response.writeHead(200, {'Content-Type': 'text/plain'});

   //Send the response body "Hello World"
   response.end('Hello World\n');
}).listen(8000);

// Print URL for accessing server
console.log('Server running at http://127.0.0.1:8000/')
```

Listing 8: A plain node.js HTTP server written in JavaScript

7.2 *Asynchrony*

JavaScript and Node.js make use of asynchronous APIs in which the API will start an operation and immediately return often before the operation is complete. Once the operation finishes, the API will use some mechanism to perform additional operations. For example, the code below will print out `"<Second, First">` because even though `setTimeout()` method is called first, and returns immediately, the operation does not complete for several seconds.

```
setTimeout(function() {
   console.log('First');
   }, 3000);
console.log('Second');
```

Using non-blocking asynchronous APIs is even more important on Node than in the browser, because Node is a *single threaded event-driven* execution environment. Single threaded means that all requests to the server are run on the same thread rather than being spawned off into separate processes as it is the case with e.g. XAMPP. This model is extremely efficient in terms of speed and server resources, but it does mean that if any function calls synchronous methods that take a long time to complete, they will block not just the current request, but every other request being handled by the web application.

There are a number of ways for an asynchronous API to notify an application that it has completed. The most common way is to register a *callback function* when an asynchronous API Call has to be invoked, that will be called back when the operation completes. This approach is used in the code excerpt above.

8 Digression: Web Server Programming

8.1 Introduction

Most large-scale websites use server-side code to dynamically display different data when needed, generally retrieved from databases stored on a server and sent to the client to be displayed via some code (e.g. HTML and JavaScript). Perhaps the most significant benefit of server-side code is that it allows you to tailor website content for individual users. Dynamic sites can highlight content that is more relevant based on user preferences and habits. It can also make sites easier to use by storing personal preferences and information — for example reusing stored credit card details to streamline subsequent payments. It can even allow interaction with users off the site, sending notifications and updates via email or through other channels. All of these capabilities enable much deeper engagement with users.

8.2 What is Server-side Website Programming?

Note: The text of the subsequently following paragraphs is based on `https://developer.mozilla.org/en-US/docs/Learn/Server-side/First_steps/Introduction`

Web browsers communicate with web servers using the HyperText Transport Protocol (HTTP). When you click a link on a web page, submit a form, or run a search, an HTTP request is sent from your browser to the target server. The request includes a URL identifying the affected resource, a method that defines the required action (for example to get, delete, or post the resource), and may include additional information encoded in URL parameters (the field-value pairs sent via a query string), as POST data (data sent by the HTTP POST method), or in associated cookies.

Web servers wait for client request messages, process them when they arrive, and reply to the web browser with an HTTP response message. The response contains a status line indicating whether or not the request succeeded (e.g. "<`HTTP/1.1 200 OK`"> for success). The body of a successful response to a request would contain the requested resource (e.g. a new HTML page, or an image, etc...), which could then be displayed by the web browser.

8.3 Static Websites

Static websites return the same hard-coded content from the server whenever a particular resource is requested. The diagram below shows a basic web server architecture for a static site. When a user wants to navigate to a page, the browser sends an HTTP "<`GET`"> request specifying its URL. The server retrieves the requested document from its file system and returns an HTTP response containing the document (i.e., the requested resource) and a success status (usually `200 OK`). If the file cannot be retrieved for some reason, an error status is returned (see client error responses and server error responses).

8.4 Dynamic Websites

Dynamic websites on the other hand generate content dynamically from e.g., databases whenever a resource is requested.

More content will follow...

References

P.L. Gorski, L.L. Iacono, and H.V. Nguyen. *WebSockets: Moderne HTML5-Echtzeitanwendungen entwickeln*. Hanser, 2015. ISBN 9783446443716. URL `https://books.google.de/books?id=rzS1rQEACAAJ`.