Wrocław University of Science and Technology

Politechnika Wrocławska

Advanced Web Technologies

Lab

Topic: Web Socket

Developed by: Piotr Jozwiak

Date: March 2023

Number of hours: 2 hours

Table of Contents

Entry

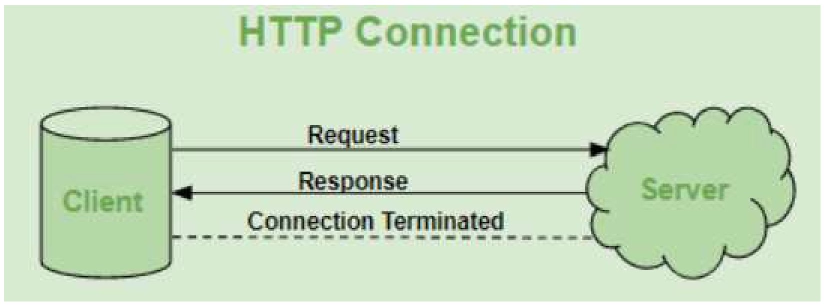
Before we discuss what WebSocket itself is, let's recall how the standard HTTP protocol works.

HTTP and its limitations

HTTP is a one-way protocol where the client sends a request and the server sends a response. For example, when a user sends an HTTP/HTTPS request to the server, upon receiving the request, the server sends a response to the client. Each request is associated with a corresponding response, after sending the response, the connection is closed, each HTTP or HTTPS request establishes a new connection to the server each time, and after receiving the response, the connection is terminated.

HTTP is a stateless protocol that runs on top of TCP, which is a connection-oriented protocol, it guarantees delivery of data packets using three-way handshaking methods and retransmission of lost packets. In principle, HTTP can run on any connection-oriented protocol, such as TCP or SCTP. When a client sends an HTTP request to the server, a TCP connection is opened between the client and the server, and when a response is received, the TCP connection is terminated. Each HTTP request opens a separate TCP connection to the server, e.g. if the client sends 10 requests to the server, 10 separate TCP connections will be opened and closed after receiving a response.

Summarizing the above, it is clear how the standard HTTP protocol is based on the Client-Server architecture. The client (browser) always initiates communication, and basically this communication is a simple conversation like asking a question and receiving a concise answer to it from the server right after the question is asked. The diagram of this communication is shown in the figure below:



Please pay special attention to the fact that after sending a response from the Server, the server closes the connection. If the client (browser) is interested in something additional, he must start communication again. In the above communication scheme there is no possibility to:

- Asking questions and waiting for an answer as the Server will have knowledge
- Allowing the Server to initiate communication with the Client to inform the Client of an event.

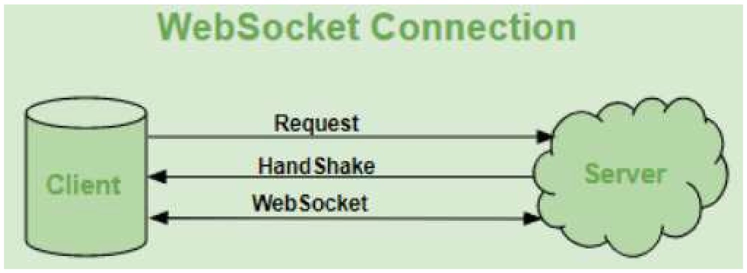
Let's seea case where the Customer orders a request via the website, which will take a long time on the server side. In order for the browser to monitor the status of the order, it must, for example, ask the Server about the current status every 1 second. This is similar to a situation where the user himself refreshes the page with the F5 button every 1 second to check if the task is ready. It is easy to find weaknesses in this behavior, called HTTP Pooling, in the form of sending a large amount queries that do not change anything in our situation except consumption and waste of resources in the form of the number of connections made to the server as well as computing or memory resources.

The solution to this problem may be the use of a mechanism called HTTP Long Polling. Long Polling is a near-real-time data access pattern. The client initiates a TCP connection (usually an HTTP request) with a maximum duration (e.g. 20 seconds). If the server has data to return, it returns it immediately, usually in batches up to a certain limit. If not, the server pauses the request thread until data becomes available, then returns the data to the client. In principle, this solution boils down to extending the request timeout to e.g. 20 seconds. This solution has its disadvantages, because during this time the thread serving this request is running on the server all the time. Only the network layer has been optimized, in which we do not need to establish many connections. In addition, in the next step, our communication is basically one-way. After establishing a connection by sending a request, the server has some time (eg 20 seconds) to prepare and send a response. During this time, the communication channel does not allow you to send another query from the client. So this type of solution is a half way.

WebSocket

With the above limitations in mind, if you want to build a real-time application without wasting too much resources, you should use the WebSocket functionality. The WebSocket protocol is, unlike HTTP, a two-way, full-duplex protocol in the Client-Server architecture. In order to indicate the willingness to use this protocol, we must indicate it in the request address in the form of the prefix ws:// or, for the version with encryption, wss://. This is equivalent to the http:// and https:// protocol selectors. The WebSocket connection itself is a stateful connection unlike HTTP. This means that the connection between the client and the server is maintained until one of the parties terminates it. The connection created in this way is used by both parties for two-way communication.

The connection itself is initially established as an HTTP connection. After the handshake, the client asks the Server to convert the connection to a WebSocket. If the Server is only able to do this, it replaces the HTTP connection with WS, so that further communication can flow freely between both parties. This is schematically presented in the figure below:



So when to use WebSocket:

1. Real-time web applications - are the basic case in which the use of WebSockets is fully justified.
2. Web gaming/video applications - are another example where the use of Web Sockets is required. Only here we have the possibility to maintain the necessary data stream all the time

for transferring fast-changing data, such as in games or video/audio streams, which cannot be implemented using the HTTP protocol.

3. Communicators (Chats) - this is another example in which WebSocket fits perfectly into the application. By its nature, instant messaging requires the maintenance of a two-way connection where either party can start broadcasting messages.

When not to use WebSockets? In principle, in every situation where the work of the application does not lose its functionality based on a standard HTTP connection. That is, in a situation where a simple request and immediate response completely meets the needs. Whenever requests are rarely sent, even if the answer is not available immediately, the next request will be made, e.g. in 5 minutes, without compromising the quality of the solution. In this case, with such long intervals between requests, the HTTP protocol seems to be sufficient.

In general, the differences between the two protocols are summarized in the table below:

WebSocket	HTTP
WebSocket is a two-way communication protocol that can transfer data from client to server or from server to client by reusing an established connection channel. The connection is kept alive until terminated by the client or server.	Protocol HTTP Is a one-way protocol that runs on top of the TCP protocol. The request always goes from the Client to the Server. The server must respond immediately to the request.
Used mainly for real-time applications, such as (trading, monitoring, notifications). Services use WebSockets to receive data on a single channel communication.	Traditional apps web, which arestateless (RESTful).
full-duplex	half-duplex

A simple WebSocket server based on NodeJS

Now that we've covered the theory behind WebSockets, let's take a look at the practical use of WebSockets. For this purpose, we will write a simple WS server using NodeJS. If you don't have NodeJS installed yet, I suggest using npm for that. After successfully installing NodeJS in the destination folder, let's initialize the Node project with the following command:

```
npm init
```

The above command will create a new project by asking for basic data. If you encounter this for the first time and want to know more details, I refer you to thishttps://nodesource.com/blog/an-absolute-beginners-guide-to-using-npm/ article. After proper initialization of the project, the main server file will be the index.js file. So, create such a file by entering the following content in it:

```
//importing http module
consthttp=require('http');
```

```
//importing ws module
constwebsocket=require('ws');

//creating a http server
constserver=http.createServer((req,res)=>{res.en
    d("I am connected");
});
//creating websocket server
constall=newwebsocket.server({server});

//calling a method 'on' which is available on websocket
objectall.he('headers', (headers,req)=>{
    //logging the
    headerconsole.log('WebSocket.on
    headers:\n');console.log(headers);
});

console.log('Listening onhttp://localhost:8000 ...');
//making it listen to port
8000server.listen(8000);
```

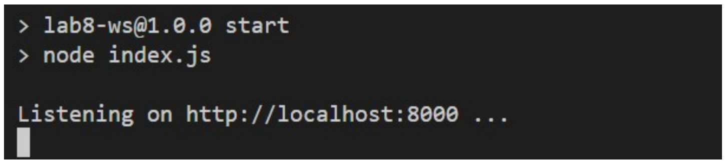
The example above won't run yet. Note that in the first two commands we import two modules: http and ws. The first one is already installed in the system along with NodeJS, but the second one must be installed by yourself. To do this, execute the command:

```
npm install ws
```

Now we can start our server with the command:

```
asl start
```

getting the following message in the console:



```
> lab8-ws@1.0.0 start
> node index.js

Listening on http://localhost:8000 ...
```

Our server should be listening at:http://localhost:8000. Connecting a browser to this address will not result in anything spectacular. The browser will make a traditional HTTP request. For our request to be a WebSocket request, we need to write a frontend page that will make such a connection. So create a client.html file with the following content:

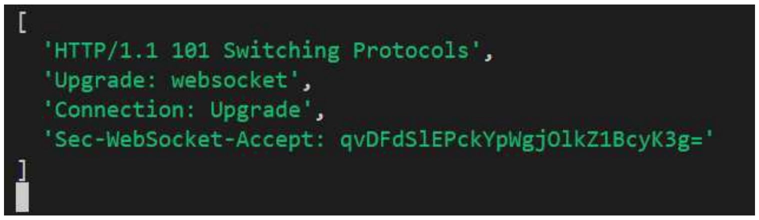
```
<html>

  <script>
```

```
    //calling the constructor which gives us the websocket object:
    wsleton=newWebSocket('ws://localhost:8000');
  </script>

  <bodysuit>
    <h1>This is a client page</h1>
  </bodysuit>
</html>
```

Nothing spectacular here at the moment, except that we're creating a WebSocket object that connects to our server. Making sure that the server is still running, run the client.html file in the browser and observe what appears on the NodeJS server console:



```
[
  'HTTP/1.1 101 Switching Protocols',
  'Upgrade: websocket',
  'Connection: Upgrade',
  'Sec-WebSocket-Accept: qvDFdSlEPckYpWgjOlkZ1BcyK3g='
]
```

Here we see the request header that was sent to the server. This is the header that asks to convert our connection in WebSocket. This header was displayed because in the server code we attached to the headers event. It's a lineall.he('headers', (headers,req)=>{.

Earlier, however, we created a constantallwhich is initialized with the objectnewwebsocket.server. This object provides us with the necessary API to create and manage a WebSocket connection. In this case, the objectallwill help us listenevents emitted when something happens. E.g. when a connection is established or terminated, etc.

Back to our headline. Pay attention to the first onea line containing the Switching Protocols command. Let's break down what happened:

- The first thing you will notice is that we have received status code 101. You may have seen codes 200, 201, 404 before. 101 is the HTTP status of 101 Switching Protocols. It says "Hey, I need an upgrade".
- The second line contains update information. Specifies that it wants to upgrade to WebSocket.
- This is what happens during a handshake. The browser uses the HTTP connection to establish the connection using HTTP/1.1 and then upgrades it to WebSocket.

The “headers” event is emitted before the response headers are written to the socket as part of the handshake. This allows you to check/modify the headers before sending them. In other words, it gives us the opportunity to decide whether to accept or reject the connection.

Let's add another event called connection and message, which is emitted after the client finishes establishing the connection. This is where we can send the first message to the customer. Sample code might look like this (we add to index.js):

```
//Event:
'connection'all.he('connection',
(on,req)=>{
    on.send('Welcome, your connection is ready');
    //receive the message from client on Event:
    'message'on.he('message', (msg,isBinary)=>{
        console.log('Received message from
        client:');constmessage=isBinary?msg:msg.();conso
        le.log(message);
    });
```

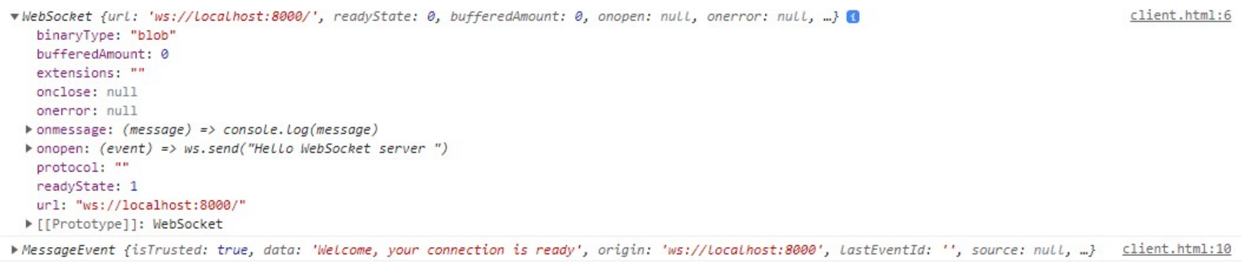
In addition, we also modify the client code (client.html file):

```
<script>
    //calling the constructor which gives us the websocket object:
    wsleton=newWebSocket('ws://localhost:8000');
    //logging the websocket property
    propertiesconsole.log(on);
    //sending a message when connection opens
    on.onopen= (event)=>on.send("Hello WebSocket server");
    //receiving the message from server
    on.onmessage= (message)=>console.log(message);
</script>
```

After restarting the server with the changes made and calling the client code, we get the following log on the server:

```
Received message from client:
Hello WebSocket server
```

On the browser side, however, we did not notice anything new. To see what has changed, we need to run the developer tools in the browser. In the case of Chrome, just press the F12 button. In the Console tab we get the following entries:



```
▼WebSocket {url: 'ws://localhost:8000/', readyState: 0, bufferedAmount: 0, onopen: null, onerror: null, …}      client.html:6
    binaryType: "blob"
    bufferedAmount: 0
    extensions: ""
    onclose: null
    onerror: null
  ▶onmessage: (message) => console.log(message)
  ▶onopen: (event) => ws.send("Hello WebSocket server ")
    protocol: ""
    readyState: 1
    url: "ws://localhost:8000/"
  ▶[[Prototype]]: WebSocket
▶MessageEvent {isTrusted: true, data: 'Welcome, your connection is ready', origin: 'ws://localhost:8000', lastEventId: '', source: null, …}   client.html:10
```

You can see here how we connect to the WebSocket server. Then we get a MessageEvent with a message from the server. Our connection is working fine.

Of course, the above example is trivial. It only demonstrates a simple connection and sending a few messages. On stubbornness, a similar thing could be obtained with ordinary AJAX or

HTTP. The strength of this solution lies elsewhere. Having established a connection, nothing stands in the way of the server itself being able to initiate sending a message to the client at any time. However, I leave this part to you to do yourself.

Socket.IO library as an overlay on WebSocket

In the previous chapter, an example using the WebSocket API directly was discussed. However, we are not always forced to use a low-level API. Sometimes it is easier to use a library, which can speed up or facilitate many things. As an example, I would like to point to the Socket.IO library available here: https://socket.io/.

Socket.IO is a library that allows you to write real-time connections based on the HTTP protocol, providing two-way communication between the client and the server. Whenever possible, Socket.IO tries to use WebSocket for communication. However, if any of the parties is unable to provide such a connection, the library tries to provide a connection using Long Polling HTTP (usually AJAX).

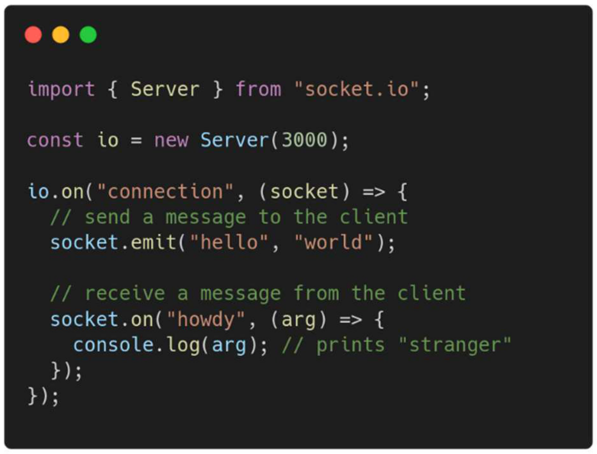
For a long time, Socket.IO officially provided only the implementation of the Node.js server and the JavaScript client. Recently, the developers of Socket.IO have also developed several other clients, in Java, C++ and Swift.

In addition to the officially supported implementations, there are many other community-maintained Socket.IO client and server implementations, covering programming languages and platforms such as Python, Golang, and .NET.

However, you have to bear in mind that the Socket.IO library is not as efficient as pure WebSocket. The WebSocket itself is also less memory consuming. This is because Socket.IO provides much more than pure WS.

Socket.IO provides features such as auto-reconnect, rooms and fallback for long polling. With a pure WebSocket, you don't use such features out of the box; you have to build all these capabilities yourself if they are relevant to your use case.

In many places, the reference to Socket.IO is very similar to pure WebSocket. Notice the example of Socket.IO code for server and client below:

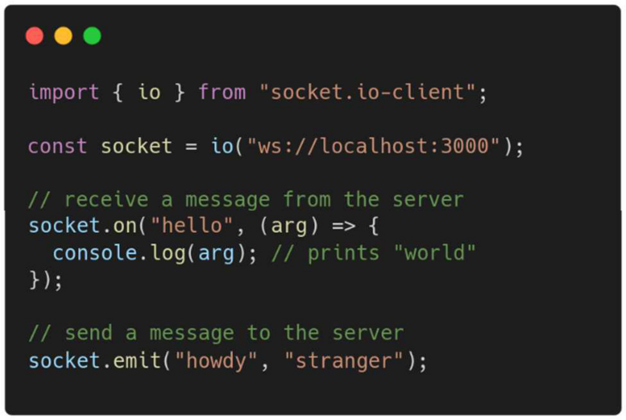


```javascript
import { Server } from "socket.io";

const io = new Server(3000);

io.on("connection", (socket) => {
  // send a message to the client
  socket.emit("hello", "world");

  // receive a message from the client
  socket.on("howdy", (arg) => {
    console.log(arg); // prints "stranger"
  });
});
```



```javascript
import { io } from "socket.io-client";

const socket = io("ws://localhost:3000");

// receive a message from the server
socket.on("hello", (arg) => {
  console.log(arg); // prints "world"
});

// send a message to the server
socket.emit("howdy", "stranger");
```

You can see how much working with Socket.IO resembles working with pure WebSocket. For more details, please read the Socket.IO documentation available here:https://socket.io/docs/v4/