**Lab: Discovery of WebSockets**

The purpose of this lab is to study the concepts discussed during [INF203 on Web technologies](http://perso.enst.fr/dufourd/cours/inf203/) and in particular JavaScript. The goal of this lab is to experiment with WebSockets, technology allowing messaging between a Web page and its server.

The steps of this lab are:

1. create a simple server
2. add websockets listening
3. create an HTML page that connects to the websockets server
4. upgrade the messages from text to JSON on both sides
5. add sending a plain message from page to server
6. add sending a request from page to server
7. add processing of the request in the server and response
8. add taking the server response into account in the page
9. repeat steps 6-7-8 for a more complex request

**Exercise 0**

In a file called serverws.mjs, create a simple server using express. You can use elements from the server2 lab, simplified to just deal with serving files from a folder.

Start and test this server.

Then add the WebSockets functionality with:

import {server as WebSocketServer} from 'websocket';

const app = express();

const server = createServer(app);

...

const wsServer = new WebSocketServer({httpServer: server});

wsServer.on('request', processRequest);

function processRequest(request) {

const connection = request.accept(null, request.origin);

connection.on('message', (o) => connectionOnMessage.call(connection, o));

connection.on('close', () => connectionOnClose.call(connection));

connection.on('error', (o) => connectionOnError.call(connection, o));

}

The first line imports the WebSocketServer function. Next two lines are known stuff from the server2 lab. Line 5 creates the WebSocket server as an extension of the HTTP server. Line 6 says: whenever something happens with WebSockets, call the function processRequest.

The WebSockets server sits there and listens. When someone connects, the WebSockets server calls processRequest. processRequest just opens and configures a connection object.

Within a connection, three things can happen: a message, an error, the end of the connection (=close). You need to program the three functions that will deal with these three types of events.

Other events than those three exist, but we are not using them in this lab.

For a first version, program the three connectionOn\* functions with just console.log.

You can run this server, but nothing will happen until you create a page that connects.

**Exercise 1**

Create an HTML page called test.html. It should contain just a div with id="content" and a script element pointing to file test.js.

Create the test.js file with:

const connection = new WebSocket("ws://"+location.host);

const contentDiv = document.getElementById("content");

The first line connects to the same server from which the HTML comes. The second line creates an object to display responses.

In the navigator, the WebSockets events are: onopen, onclose, onmessage, onerror.

So you should program four functions in this way:

connection.onopen = function () {...};

Initially, just use console.log to display what happens. Or you could change the textContent of the div, as you wish.

Place the HTML and JS files in the files folder of the server. Start the server. Open your browser and get the HTML file.

To be checked:

* the server has served the HTML and JS to the browser
* the HTML has loaded the JS
* the JS has tried to connect to the server
* the server has received the request and opened a connection

Experiment with this setup:

* upon creating the connection, the server could send a message; check the reception in the page.
* after one second, the page could send a message; check the reception on the server

**Exercise 2**

At this moment, the messages to and from the server are plain text.

We are going to send back and forth more complex messages, so it is good to “upgrade” the messages to JSON.

When page or server receive a message, they should try to JSON.parse it. If it does not work, then just display the message.

Test this by adding a button in the HTML, which calls a function send. This function sends the string '{"type": "message", "content": "whatever"}' to the server.

On the server, the received string is processed with JSON.parse, and since the type is "message", it is just displayed with console.log.

**Exercise 3**

It is time to add a complete request with a meaningful response: display the files present on the server.

Add a button in the HTML, calling a function sending '{"type": "ls"}' on the websocket connection.

Add in the server the processing: if type === "ls", then look for the list of files in the files folder, construct a response and send it to the page in another message of type "ls" too, with another property called "response" which contains an array of file names.

Then in the JS file of the client side, when you receive a message of type "ls", from the response property, you can construct, in the div, a list of a elements pointing to the files.

**Exercise 4**

It is time to add a more complex request: display the content from a URL.

Add a text input in the HTML, with an ID attribute so that you can fetch the value typed by the user. Add a button in the HTML, calling a function with the intent to display the content of the URL typed in the text input. We assume the URL points to an HTML resource. The type of the message the client is sending in this case is “geturl”.

Add in the server the processing: if type === "geturl", then:

* get the URL from the url property of the received message
* fetch the HTML pointed at by the URL << problem 1
* create and send a message back to the page.

Then in the JS file, when the message type is “geturl”, you get the text of the HTML to display. << problem 2

Problem 1: there are multiple solutions to get the content of a URL from inside a node.js program. One way is to use the request function from the http module. Another way I have also tried is to use the get function of the axios module.

Problem 2: there are multiple solutions to display HTML contained in a string. If it was an HTML fragment we could put it directly in the innerHTML property of the div. Here, it will be a complete HTML with header and body, so I have put it in the property srcdoc of an iframe element, child of the div.

**Conclusion**

In this lab, you have coded:

* a simple WebSocket server
* a simple HTML page connecting with WebSocket
* a few messages exchanged between the two: this is an alternative to AJAX
* a request for content from inside node.js: this can be used as a basis for crawling, for proxy functionality, etc.

**Annex - Using Node JS**

JavaScript is a “complete” programming language. You can program in JavaScript everything you could do with any other programming language (Python, Java, C, …).

[NodeJS](https://nodejs.org/en/) is a runtime environment of JavaScript code. NodeJS uses the same JavaScript engine as Google Chromium. It can be used on the server side to generate web pages. It can also be used on the command line (like python for example) to run a program. That’s what we will do.

On machines in the TPT lab rooms, you can use it as follows: the > symbol at the beginning of the line refers to the NodeJS prompt, i.e. where NodeJS waits until you input code and press Enter, and the $ symbol refers to the Unix command prompt; the other lines are the results of the execution.

$ nodejs

> console.log ('Hello World');

Hello World

undefined

>

The first line displays the desired result. The second displays the return value of the console.log function which is undefined.

This way of using NodeJS is very convenient for testing short programs, one line. This becomes more complicated for several lines. For this purpose, NodeJS accepts the name of a file as parameter. For example, if the hello.js file contains:

console.log ('Hello World');

The use of this file is simply:

$ nodejs hello.js

Hello World

$

NodeJS also offers the possibility to debug your code, set breakpoints, inspect variables and stack, etc. To use the debugger, run node with the command:

$ nodejs debug hello.js

The debug mode documentation is located [here](https://nodejs.org/api/debugger.html).

At the top of each file, put "use strict"; as a first line

If you have messages like “SyntaxError: Block-scoped declarations (let, const, function, class) not yet supported outside strict mode” then the “use strict”; at the top of your file is missing.