**Lab: Server**

The purpose of this lab is to study the concepts discussed during [INF203 on Web server technologies](http://perso.telecom-paris.fr/dufourd/cours/inf203/).

Rather than install a turnkey web server (such as Apache), set up (e.g. modify httpd.conf configuration files) and add specific behaviors in languages like PHP, Java, C / C ++ …, this lab proposes to create a server from scratch (or almost). You will program in JavaScript, using NodeJS and its support for HTTP / HTTPS protocols.

**To do**

For this lab, you will need to upload the different JavaScript files you created. Everything will have to be zipped and dropped below.

To program, use the “strict” mode of JavaScript, the indentation and comments.

***Thank you for using zip (and not tar and gzip or bzip2). Zip all files directly into a single zip, do not zip the folder.***

***No spaces or accented characters or special characters in the name of the zip file.***

***This lab is to be done alone***.

Please respect the file names and IDs we ask you to use, otherwise automatic grading will not work and you will not have the grade matching your work.

**Use local URLs in your files**, so your production works regardless of the URL of the server, and so that it also works on the grading machine.

In all your functions that respond to HTTP requests, put a try {} catch {} to catch all the exceptions, and show the error messages.

To test each url you respond to, start your server locally and send the url to your running server, either from a navigator or with the command line tool curl. With option -i, curl shows you the HTTP headers before the content. curl seems available by default on Windows (in Powershell) and MacOS and Linux.

**Use of NodeJS modules**

NodeJS is based on the concept of **modules**. A module is a block of JavaScript code that you can load and use in your code, that is to say as a library. To load a module, NodeJS uses the import directive. It is used like this:

import {func1,func2...} from "module";

There are many modules available for NodeJS. Some are installed automatically with NodeJS (see the documentation [Here](https://nodejs.org/api/)). The fs module, management of the “File System”, is used to read / write files (see the documentation of this module [here](https://nodejs.org/api/fs.html)). The module http manages receiving and sending HTTP messages (see the documentation of this module [here](https://nodejs.org/api/http.html)).

NodeJS provides the command line tool npm to load more modules. To download and install a module, use the following command line:

> npm init //only do this line the first time

> npm install module\_name --save

npm init creates a package.json. You can use the default response to most questions.

The --save option adds a dependency to the package.json file which is required for the grader to know that it needs to load this extra module.

If you have a problem with npm, for example it does not exist on your system, look [there for instructions](https://labgrader.r2.enst.fr/files/server/serverLab.html#npm)

You can verify that a module is installed by checking that there is a subdirectory with the module name in the node\_modules folder. You can then use require.

For example, to install the http module which is a simple http server, use:

> npm install http

And then, in the JavaScript code, write:

import {createServer} from "http";

**Additional useful information**

Additional useful information can be found in: [Labs Overview](https://perso.telecom-paristech.fr/dufourd/cours/inf203/labs-overview.html)

**Simple Web Server**

**Question 1:** Here is a minimal server, to be put in a file server.mjs:

"use strict";

import {createServer} from "http";

// request processing

function webserver( request, response ) {

response.setHeader("Content-Type", "text/html; charset=utf-8");

response.end("<!doctype html><html><body>Server works!</body></html>");

}

// server object creation

const server = createServer(webserver);

// server listens

server.listen(8000, (err) => {});

This server returns an HTML string confirming that the server works, e.g. when you access the root of the server, that is, “http://localhost:8000/”. The method webserver is a callback function that takes 2 parameters: a request object containing information about the query (URL, methods, headers …) and a response object that contains methods for create and send the response, especially the response.end method.

Test this server.

Then modify the server to listen to a port number given on the command line instead of 8000.

The command line to start the server on port 8000 will then be: node server.mjs 8000

To get the command line arguments, use the array process.argv

In the rest of the text, the port 8000 is used in the URLs given as example, but the actual port depends on the number given on the command line starting the server.

**Question 2:** Modify the previous server so that receiving the URL “http://localhost:8000/kill” will stop the server.

The root url will still trigger the same response as in Question 1a. Receiving the kill url will trigger:

* sending the header that says content will be in HTML
* sending the HTML text response that says “The server will stop now.”
* calling process.exit(0)

**Question 3:** Using the fs module, modify the web server so that it serves the files present in the server directory and all subdirectories. Use the same server.mjs file.

The access URL is http://localhost:8000/root/<path> where <path> is the path of the file that should be served. To take examples, the url http://localhost:8000/root/foo.bar refers to a foo.bar file that is in the current folder where node.js was run, beside the server.mjs; the url http://localhost:8000/root/foo/bar.txt refers to a bar.txt file that is in the subfolder foo of the current folder.

The fs module proposes different methods to check the existence a file, or read the contents of a file.

Make sure the server will respond with the code 404 if the file does not exist.

Check that the server is not serving files in parent directories, ie using http://localhost:8000/root/../file.txt.

Make sure to answer with the good MIME type for the most common files (HTML, CSS, JS, PNG, …).

You will control your results with the network inspector of your Web browser.

Note: Serving the files in the server folder means than when you access http://localhost:8000/root/foo/tutu/titi.html in your browser, then the server does this in the “callback” function:

* the server receives the browser request
* the server retrieves from the URL the name of the requested file: foo/tutu/titi.html
* the server checks if the folders and the file exist (using the fs module), and if not, sends an error code and an error message to the browser
* the server reads the file (using the fs module)
* the server places the contents of the file in the response
* the server sends the response to the browser

**Question 4:** Modify the previous web server so that it processes GET requests for the address http://localhost:8000/hello?visiteur=xxxx where xxxx is a string of characters. Continue to use the server.mjs file. The server will respond **with HTML code** saying hello xxxx. It should be OK for xxxx to contain accents, spaces … You can use the unescape method of the module querystring.

Test your server by writing an HTML page (named exercise1c.html) containing a form that issues these queries: an HTML form uses the \<form\> element, \<input\> and a \<button\> for submit. The form will use the GET method and its action will be hello.

Make sure it works with names containing spaces, accented/special characters …

This is where you begin to experience the “client-server” paradigm: the input is made in the browser on the client machine, in HTML; there could be an integrity check in the client before sending, then the data is sent to the server (as part of the URL), then the server processes the request within your code that handles the GET on hello.

Check for XSS vulnerabilities (injection of HTML code, injection of JavaScript code) but do not fix them. Note: If you made a server response in plain text you’ll see that the vulnerabilities do not show. Correct the server response to be in HTML.

**Question 5:** Modify the previous server to process requests to http://localhost:8000/bonsoir?user=xxxx. The server will save in memory all received user values. The server will respond, at each new request, with an HTML response of the type:

"bonsoir xxxx, the following users have already visited this page: yyyy, zzzz, yyyy, bbbb" Check the XSS vulnerability. Vulnerability is more serious in this case. Correct this vulnerability, especially for user=<b>Toto</b> and user=<script>alert('hello');</script> by detecting < and > and replacing them with \_.

**Question 6:** Modify the previous server to process requests to http://localhost:8000/clear by clearing the memory of past bonsoir requests. After a clear, a request to http://localhost:8000/bonsoir?user=xxxx will produce the answer: "bonsoir xxxx, the following users have already visited this page: "