Assignment 1

For ACS-3909

Due: September 25, 2023

Information:

- Your submission must **clearly indicate your name** in each file (use a comment in .js files and in .html files)
- Your **submitted filenames** must clearly indicate to which assignment and exercise they belong. For example, if you solve Assignment 1, exercise 1.1 name your file: "A1_E1-1_app.js" or something similar.
- Keep an eye on Nexus for clarifications/corrections of this assignment!
- **No late submissions**

The exercises will ask you to make yourself familiar with functions, packages, libraries, etc. that we have not explicitly introduced in the lectures. This is by design. I am linking for each exercise resources that can help you in finding the information you might need to fulfill the exercise.

In some instances, I will disallow or restrict the modules you are allowed to use. Besides theses instances there are many ways to do things, so see the resources I give you as hints – if you find another (even better) way of doing things: props to you!

Exercise 1.1 Callbacks, Promises, Async/Await

Consider the following JS code that does not work as intended:

```
//The following code does not work. Make it work with callbacks first. Then with
// promises

function noAddGrunt(msg){
    setTimeout((msg) => {return msg+"*grunt*";}, 1000);
}

function noAddSmack(msg){
    setTimeout((msg) => {return("*smack*"+msg)}, 1000);
}

function noAddBreath(msg){
    setTimeout( (msg) => {return(msg+"*wheez*")}, 1000);
}

let msg = "telephone";
let modded = noAddGrunt(msg);
modded = noAddSmack(modded);
modded = noAddBreath(modded);
console.log(modded);  //expected outcome: *smack*telephone*grunt**wheez*
```

Subtask A: Write a JS-file that fixes the above code using callback-functions, so that it logs to console after 3 seconds:

```
*smack*telephone*grunt**wheez*
```

The calls must happen one after another, meaning that after 1 second there exists an object named *msg* with value *telephone*grunt**, after two seconds it becomes **smack*telephone*grunt** and so on…

Subtask B: Do the same as in Subtask A, but use Promises with *.then-chaining* instead of Callbacks

Subtask C: Do the same as in Subtask A, but use async/await-syntax instead of .then (you can still use *.then* for the last Promise, if you need to)

Exercise 1.2 HTML-Responses in Vanilla Node

Possibly useful resources:

- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for...in
- https://nodejs.org/api/fs.html#fs_fs_readdir_path_options_callback
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/forEach

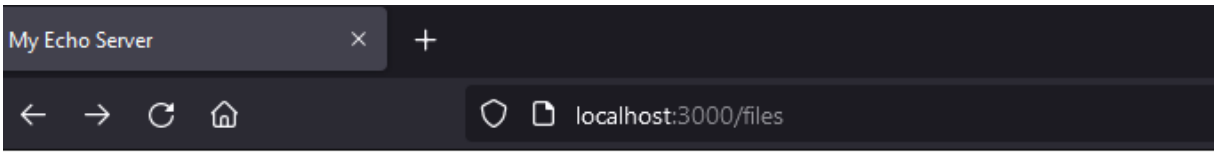
Restrictions: You must use a Vanilla-Node server (no Express)

Subtask A: Remember the Echo-Server we have created in Vanilla-Node.js. The response to the client was a in plain text. Create a Vanilla-Node.js echo-server (i.e., do not use Express) that responds in HTML instead. Your server must:

- Create an HTML response, which will be displayed as HTML in the browser
- The response must contain

    - The HTTP-method that was used by the client
    - The URL that was requested by the client
    - All HTTP Request-Headers that were sent by the client
- The response must be formatted, such that each request-header is a bullet point inside of an HTML -element

Here is an example, of how a valid solution would look like in the browser:



Received GET-request to /files with headers:

- host: localhost:3000
- user-agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:104.0) Gecko/20100101 Firefox/104.0
- accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8
- accept-language: en-CA,en-US;q=0.7,en;q=0.3
- accept-encoding: gzip, deflate, br
- dnt: 1
- connection: keep-alive
- cookie: Pycharm-a9d38409=2360ad2c-1a87-4ee6-8c50-d4aa0e033e17; Webstorm-ded6f4de=79509
- upgrade-insecure-requests: 1
- sec-fetch-dest: document
- sec-fetch-mode: navigate
- sec-fetch-site: none
- sec-fetch-user: ?1
- pragma: no-cache
- cache-control: no-cache

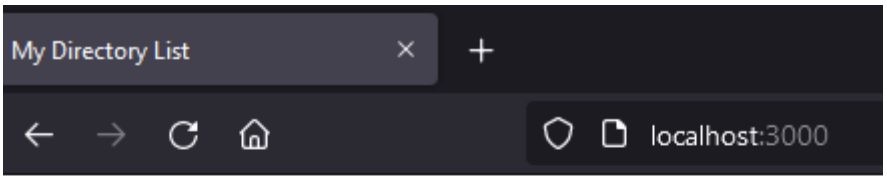
Subtask B: Still using only Vanilla-Node (no Express) create a server that lists all the filenames and subdirectories of a directory (called "./public" relative to your server-file). As in Subtask A the response must be in HTML. Each filename must be a clickable link, i.e., use <a>-elements for these. Each folder should be marked as a folder instead and not be clickable. Clicking a link to a file must open that file in the browser. Assume you have the following files in your "./public" directory:

- At least one JPG-image
- At least one HTML-file
- At least one JS-file
- At least one subdirectory

Your server must:

- **Not** hard-code the filenames. Meaning: If the files in the folder change, after refreshing the browser that change should be reflected
- Respond with HTML that contains a -element forming a list of filenames and directories
- Respond with a 404-code and a message that a specific file could not have been found **or** a specific filename-extension was not successfully matched to a MIME-type, if a request to a non-existent file was made.

Here is an example of what a valid response could look like:



- deer.jpg
- index.html
- main.js
- subdir (Folder)
- uploadForm.html

Exercise 1.3

Resources:

- https://nodejs.org/api/modules.html#the-module-object
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes/constructor

Restrictions: This exercise requires you to write **two separate** JS-files. One contains your Vanilla-Node server, the other contains your routes, as follows:

See the following code, which is a Vanilla-Node server using two externally defined routes through the lines:

```
echoRoute.route(req, res);
htmlRoute.route(req, res);
```

Server.js:

```
// Resources:
// Destructuring assignment: https://developer.mozilla.org/en-
US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment

const http = require("http");
const {echoRoute, htmlRoute} = require("./3_RouteHandlerRouter");

const hostname = '127.0.0.1';
const port = 3000;


const server = http.createServer( (req, res) => {
    echoRoute.route(req, res);
    htmlRoute.route(req, res);
    res.end("Default response");
});

server.listen(port, hostname, () => {
    console.log(`Server running at http://${hostname}:${port}`);
});
```

Your task is to code the respective second JS-file[1]. This second file must contain:

- A class called *RouteEntry* with a constructor that takes a *url* and *callback* as arguments. Further it must have a method called *route* that applies the *callback* only if the URL requested by the client matches the *url* provided in the constructor. The *route* method takes a *req* and *res* object as arguments.
- Outside of the *RouteEntry* class: Two distinct functions that can be used as callbacks in RouteEntries. Be creative. They can do anything, as long as they give a response to the client.
- the creation of two RouteEntries that use the two distinct functions from above and mount them to two distinct URL-paths.
- the export of the two RouteEntries, so that they can be required by the Vanilla-Node server. In the above example the two RouteEntries are called *echoRoute* and *htmlRoute*

Exercise 1.4

Resources:

- https://expressjs.com/en/4x/api.html#req
- https://expressjs.com/en/4x/api.html#app.use
- https://nodejs.org/api/fs.html check promises for appendFile, stat, rm, rename
- https://nodejs.org/api/fs.html#class-fsstats
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Date

Restrictions: For this exercise you have to use an Express-server.

Write middleware[2] that being used by your express-server does the following:

- Before any other processing the request is being logged to console, specifically log
  - the HTTP method
  - the requested url
  - if any query-parameters had been submitted also log these
  - the IP from which the request came.
- Also, before any other processing the request is also logged to a file called *requests.log* on your file system. New requests are appended to the end of the file. In addition:
  - Every request received is logged together with its data and time it arrived.
  - Once the file *requests.log* is larger than 5 KB it is renamed to *old_requests.log* (replacing any previous versions of a file of that name) and a new file called *requests.log* is started. This is called a “Rotating file handler” and allows us to keep the total amount of logged queries to stay below 10 KB in total.

Hint: To test your logging you should consider implementing at least a couple of routes for your server and also access it through your browser by your actual (subnet-)IP address, instead of “localhost” or “127.0.0.1”.

[1] in the above example it is called *3_RouteHandlerRouter* you can call it whatever you like, though, as long as it works with your server.js

[2] Although we ask for two distinct functionalities you can structure your code in 1, 2, or more middleware-blocks, i.e., calls to *app.use*

Exercise 1.5

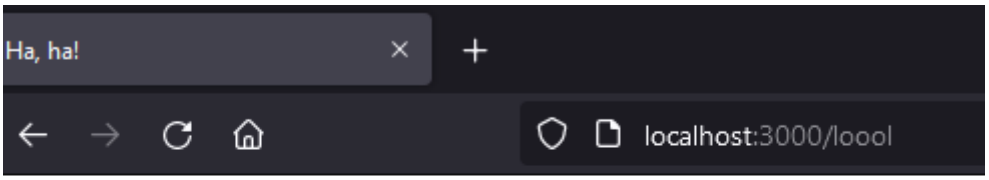
Resources:

- Advanced routing guide: https://expressjs.com/en/guide/routing.html

Restrictions: For this exercise you have to use an Express-server.

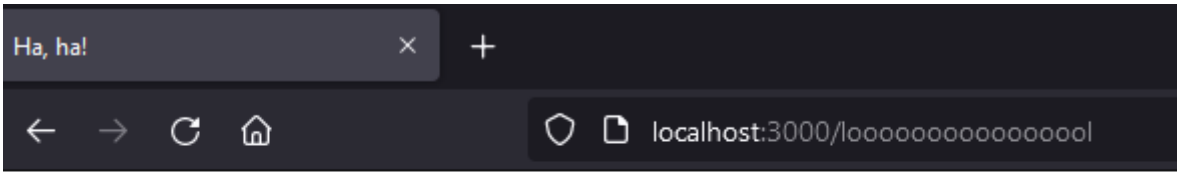
Create a server that accepts the URL */lol* with **any amount** of "o" between the two "l", e.g. */loool, /lool,…*
The response of your server to these urls must be a smiley face (you can use :D or an actual image) which is displayed larger or smaller inside the browser depending on the number of "o" in the url.
See these two screenshots as example:

A good chuckle:



:D

Snorting out a beverage:



:D

Exercise 1.6

Resources:

- https://expressjs.com/en/4x/api.html#res.sendFile
- https://expressjs.com/en/4x/api.html#res.download

Restrictions: For this exercise you must use an Express-server. You **cannot** use Express' static middleware or any other third-party middleware.

Create a server that answers to calls to any url of the form */public/somefile.ext* (where *somefile.ext* is a filename put in by the user, for example the regularly used "deer.jpg") by either:

- If the requested url is for a jpg: Send an image to the browser from a folder "./public"
- In all other cases: Triggers a download at the browser so that the requested file from the "./public"-folder can be saved to the user's system.

You are asked to do this in two different ways. First: Deliberately **not** use the *root* option of *res.sendFile* and *res.download*. Second: By using the *root* option. For both cases try to download the actual JS-file that contains the code of your server by moving outside of the "./public" folder!

Provide the URL-string that makes this possible (when not using the *root*-option in your code) as a comment.

Hint: To trick your server to go outside the "./public" folder you need to lookup how some special characters are URL-encoded, particularly useful will be the encodings for the dot "." and the slash "/".

Exercise 1.7

Resources:

- https://www.npmjs.com/package/sanitize-filename
- Find fs/promise.readFile and fs/writeFile in: https://nodejs.org/api/fs.html
- https://nodejs.org/api/fs.html#fsaccesspath-mode-callback
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/JSON/parse

Restrictions: None

In this exercise we combine a lot of the things of the previous exercises to code your very own file-server. If you are confident in having a good firewall at your router to protect you from outside requests, you might actually use this in your home-network to quickly share files between your devices (phones, tablets, laptop, PC,…)

Create a server that does the following things:

- On the url */files* it lists the contents of some directory on the server's filesystem, say "./public". The listed content is linked to the respective files (see next bullet point)
- On the url */files/somefilename*, where *somefilename* is put in by the user in the address-line of the browser: A download is triggered by the browser, so that you can save the file to the client's file system.
- On the url */upload/form*: With a GET-Request a form is loaded that allows the user to select a file on their system to be uploaded to the server's "./public" directory inside of a POST-request.

- Further your server keeps a download-counter for each file that has been downloaded. This download-counter must be saved and updated to disk on the server-side, meaning even if we restart the server the information on how often a file has been downloaded is not reset.