Week 5: NodeJS and Client-Server

1 Instructions

1. During this exercise, document your progress using screenshots (of the entire screen) to demonstrate that you have indeed completed all the required tasks.
2. Save all the screenshots in the GitHub project, in a dedicated folder called "proof," and present them chronologically in the README file. See the forum to understand what present in README means.
3. There's no need to make hundreds of screenshots, but please take enough to convince the person checking your submission that you completed the exercise as instructed.
4. Your README file must also include a link to the repository.
5. When you are ready to submit, download the repository from GitHub (the entire repository) and submit it. Your submission MUST NOT contain links to other locations where the files are (like Google Drive or a link to the GitHub repository instead of the actual files). You are allowed to submit a zip file.
6. The Moodle has a hard limit on the upload size, and will not allow you to upload files beyond 5MB. There is nothing I can do about it.

2 Setting up a basic NodeJS project

1. Install NodeJS.
2. Choose an empty folder that will contain the project and open the folder's path in a terminal
3. Create an empty project using the following command: npm init -y
   (a) Make sure that your terminal is open on the relevant folder
   (b) This command will just create a package.json file, with default settings
4. Create a file called app.js and add the following code:

```
console.log('Hello world');
```

5. Execute using the command: node app.js
   (a) Make sure you see the correct output
6. You can now essentially run any relevant JavaScript code.

3 Working with the HTTP module

1. Change app.js to contain the following code:

```
const http = require("http")

const server = http.createServer((request,response) => {
    response.end("Hello world")
})

server.listen(80)
```

2. Adjust the port number if needed, execute and demonstrate using a browser the code works.
3. Alternatively, we can load a module using a different syntax. An equivalent code will be:

```
import { createServer } from "http"

const server = createServer((request,response) => {
    response.end("Hello world")
})

server.listen(80)
```

4. Use this code instead and observe the error
5. To fix it, we need to add the following line into package.json, somewhere *inside* the curly bracekts:

```
"type": "module",
```

6. Make sure the code works now (use a browser)

4 Working with the request and the response

1. Replace the code in app.js with:

```
import { createServer } from "http"

createServer((request,response) => {
    response.statusCode = 404
    response.end("Hello world")
}).listen(80)
```

2. The only new line here is the one that changes the status code, and sends an error code (404 Not found) instead of the default (200 OK)

3. Execute and compare the difference with and without this line using the 'Developer toolbar'

4. Replace the code with the following:

```
import { createServer } from "http"

createServer((request,response) => {
    if (request.url == '/foo')
        response.write('<html><body><b>foo</b></body></html>')
    else if (request.url == '/bar')
        response.write('<html><body><u>bar</u></body></html>')
    else
        response.statusCode = 404

    response.end()
}).listen(80)
```

5. Make sure that you understand the code, and execute the code.

   (a) Specifically, notice how different url parameters cause different responses.

5 Using Express

Express is a module that simplifies the API and allows us to create sophisticated HTTP servers with ease.

1. Replace the code with the following:

```
import express from 'express'
const app = express()

app.get('/', (req, res) => {
    res.send('<html><body><h1>GET</h1></body></html>')
})

app.post('/', (req, res) => {
```

```
    res.send('<html><body><h1>POST</h1></body></html>')
})

app.listen(80)
```

2. Notice:
   (a) The use of express instead of plain http module
   (b) How we can easily create dedicated functions to handle different HTTP methods and different url paths
   (c) Modules, like express, need to be installed using the command: npm i express
   (d) Remember, put node_modules in .gitignore.
3. Create an html file with a form tag that submits the form to the server using GET.
4. Create another html file with a form tag that submits the form to the server using POST.
5. Execute and see that you get different responses.
6. Replace the code with the following:

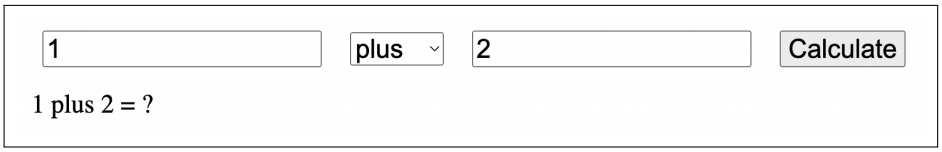
```
import express from 'express'
const app = express()

app.use(express.static('public'))

app.get('/', (req, res) => {
    res.send('<html><body><h1>GET</h1></body></html>')
})

app.listen(80)
```

7. Create a folder named 'public'
8. Put there an html file that contains an image tag with a src attribute to an image file in the same folder, and put the relevant image in the public folder
9. Execute and observe that when you go to: localhost/, the Lambda function is executed, but when you go to: localhost/[html file name], express goes to the public folder and serves you the static html file + image
10. Execute and observe using the developer toolbar (Network tab)

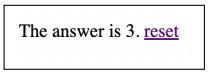
6 Complete example

We now want to build a simple calculator, but a calculator where the server is the one doing the calculations, not the client.

The UI will look like this:



And the server's response will look like this:



To that end:

- Create another project.
- The structure of the project should be like this:

- index.html:

```
<!DOCTYPE html>
<html>
    <head>
        <link href="index.css" rel="stylesheet">
    </head>
    <body>
        <form action="/calc" method="post">
            <input name="x" value="0">
            <select name="operation" value="0">
                <option value="0">plus</option>
                <option value="1">minus</option>
                <option value="2">times</option>
                <option value="3">divide</option>
            </select>
            <input name="y" value="0">
            <input type="submit" value="Calculate">
        </form>
        <div id="formula"></div>
        <script src="index.js"></script>
```

```
    </body>
</html>
```

- index.css:

```
*{
    margin: 10px;
    font-size: 18pt;
}
```

- index.js:

```
function updateFormula(){
    var x = document.getElementsByName('x')[0].value;
    var y = document.getElementsByName('y')[0].value;
    var operation = document.getElementsByName('operation')[0].value;
    var operationText = document.getElementsByTagName('option')[operation].innerText;
    document.getElementById('formula').innerText = `${x} ${operationText} ${y} = ?`
}

for (var i = 0; i < document.forms[0].elements.length; i++) {
    const element = document.forms[0].elements[i]
    element.onchange = updateFormula
}
```

- app.js:

```
const express = require('express')
const app = express()
app.use(express.static('public'))

const bodyParser = require('body-parser')
app.use(bodyParser.urlencoded({extended : true}))

function calc(x,y,operation){
    switch (operation) {
        case '0':
            return x+y
        case '1':
            return x-y
        case '2':
            return x*y
        case '3':
            return x/y
        default:
            return 'incalculable'
    }
}

app.post('/calc', function(req,res){
    var x = parseInt(req.body.x)
    var y = parseInt(req.body.y)
    var operation = req.body.operation

    var result = calc(x,y,operation)

    res.end(`<html>
                <body>
                    The answer is ${result}.
                    <a href="/index.html">reset</a>
                </body>
            </html>`)
})

app.listen(880)
```

Make sure you understand the entire code.
Execute in the browser and demonstrate.
Add comments to the entire code that explain the code

7 Get vs. Post

1. Create a login form. Namely, an html form with two input fields (username and password) and a submit button. No need for CSS or client side logic. Make sure to have a relevant 'name' attribute for these input fields. **Hint:** lecture slides.

2. Ensure your form tag has a method attribute with a value 'get' and an action attribute with the address of your local machine. For example:

   ```
   <form method="get" action="http://localhost:8080">
   ```

3. Create a NodeJS server that will handle login attempts.

4. In addition, the Web server needs to extract the username and password values sent to it, and check if they are correct (compare to a hardcoded username and password of your choosing). The server sends back a response indicating if the username/password was correct or not. **I didn't show in class how to do it. Please use Google, and see how you can extract from the 'request' object the username and password sent from the browser.**

Execute the code and demonstrate that it works. Look at the URL in the browser. Can you see the password you entered? (spoiler: yes, you can). Not very good... anyone looking at your computer could also see it!

1. Change the html form to send the data using post instead of get:

   ```
   <form method="post" action="http://localhost:8080">
   ```

2. **Adjust** the server's code to extract the username and password values sent to it, and check if they are correct (compare to a hardcoded username and password of your choosing). The server sends back a response indicating if the username/password was correct or not.

Note that you had to adjust the code in the server, because you extracted the values from a different location of the request. To understand this, go back to the browser, and look at the URL again. Do you still see the username/password? (spoiler: no, you don't). But, how come the server still gets it? If you'll go to the 'Network' tab in the developer toolbar and click on the request, you will see it, in the BODY of the request ('payload').

8 Model View Controller Architecture

The problem with the last server we just wrote, is that everything was in a single file. Therefore, whenever we needed to do a change, like supporting POST instead of GET, we would have to check the entire code again - because our changes might have 'broken' the code. Not very good...

We want to create a multi-layered application, that will separate our code to decrease dependency and to allow maintainability.

1. Create another NodeJS project.
2. Create the following structure (you will also have package.json, node_modules etc.):

3. Notice the separation between models, views, and controllers. In addition, we will also use another layer called routes, to simplify our controllers.
4. Lets begin with our app.js file:

```
1 const express = require('express');
2
3 const app = express();
4
5 app.set('view engine', 'ejs');
6
7 app.use('/', require('./routes/articles'));
8
9 app.listen(8080)
```

5. You should already understand lines 1,3,9. Line 5 tells express that we would like to use 'ejs' as our view engine. This will allow us to define templates, and generate dynamic views based on these templates. Line 7 tells express that instead of defining the routes of our server here, we define them in a file called articles, under the routes folder. In other words, instead of defining here such code directly in app.js:

```
app.get('/', () => {
 // ...
}

app.post('/login', () => {
 // ...
}
```

We will define it in the routes folder. This helps make the app.js file short and concise.

6. Lets move to the file in the routes folder:

```
1 const express = require('express');
2 const { index } = require('../controllers/articles');
3 const router = express.Router();
4
5 router.get('/articles', index);
6
7 module.exports = router;
```

In line 1, we load express and then use its Router in line 3. This will allow us to define the URLs of our server. Line 2 loads the articles file from the controllers folder. This file will contain the controller functions for each url (endpoint). In line 5 define that if the browser asked for '/articles' using a GET request, the controller function that will handle this request is called 'index', and we loaded it in line 2. Finally, in line 7, we export the router object we defined here, so it can be use in the app.js file (line 7).

7. Lets move to the file in the controllers folder:

```
1 const Article = require('../models/articles');
2
3 const index = (req, res) => {
4     res.render("../views/articles", { articles: Article.getArticles() });
5 }
6
7 module.exports =  {
8     index
9 };
```

In line 1 we load the articles file from the models folder. I lines 3-5 we define a function called index. The index function handles the request/response to the browser (notice the req and res objects). In fact, line 4 performs several things:

- It obtains all the current articles from the model using Article.getArticles()
- The response is inserted into a JSON object with one key called articles.
- The response to the browser is **rendered**, the **ejs** view engine we mentioned earlier will take the articles file from the views folder and **render** it dynamically according to the JSON object we created.

8. Lines 7-9 export the index function, so we can use it in line 2 of the controller file.

9. Lets see the file in the views folder to understand this:

```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4     <meta charset="utf-8" />
5     <meta name="viewport" content="width=device-width, initial-scale=1.0" />
```

```
6     <title></title>
7 </head>
8 <body>
9     <div class="container-fluid">
10        <main role="main" class="pb-3">
11            <%for (var i = 0; i < articles.length; i++) {%>
12                <article>
13                    <span><%=articles[i].id%></span>
14                    <span><%=articles[i].title%></span>
15                    <span><%=articles[i].content%></span>
16                </article>
17            <%}%>
18        </main>
19    </div>
20 </body>
21 </html>
```

10. Most of the file is a plain html file. However, look at lines 11-17. In line 11 we define a for loop, and the **ejs view engine** understands it because we write it inside $< \% \ldots \% >$. This means that when line 4 in the controller file executes, the ejs file will generate lines 12-16 as many times as needed - depending on the JSON object we created and sent to the renser function.

11. Finally, lets see the file in the models folder:

```
1 const articles = [
2     { id : 1, title : "Hello", content : "world" },
3     { id : 2, title : "Hello2", content : "world2" },
4     { id : 3, title : "Hello3", content : "world3" }
5   ];
6
7 exports.getArticles = () => {
8     return articles;
9 }
```

    We created a simple JavaScript array and returned it. Next week, we will replace this logic with an access to a DB server. The cool thing is, that **none of the other files will need to change** to make it happen. We will simply change the model - and the rest of the code is oblivious to the change.

9 Serve React App from Node

When you run your React app (like, by pressing F5), the application is automatically launched in your default browser. But, in order for that to work, behind the scenes, your code is served using a **Web server**. Lets see how can we do it ourselves.

1. Create the default react app.

2. Replace the code of the default React app with the login form from before.

3. Instead of running the app using: npm start, create a build version using: npm run build

4. This creates a directory called ‘build’, which has the transpiled code, ready to be served on a web server. Observe the files in it, and see the difference between the content of the build folder, and the content of the project with the code.

5. Add to the node server that we just created, the relevant code to serve static files from a ‘public’ folder (**Hint**: you did it earlier)

6. Put the contents of the ‘build’ folder inside the ‘public’ folder and run the server. Demonstrate and observe that the app is now served from YOUR Web server, and not from the default web server.