**Text

Description automatically generated**

**Development Approach**

Mohamed El Mehdi Sabor

Homework 3

Adv. & Distributed Programming Paradigms CSC 3374 – 01

**Technologies Enablers :**

Java: Java is used as the programming language for developing the server-side Spring Boot application.

Spring Boot: Spring Boot is an open-source Java-based framework used to develop microservices and web applications quickly. It simplifies application development by providing default configurations and reducing boilerplate code. In this project, Spring Boot is used to create a RESTful API for interacting with the shared folder.

JavaScript: JavaScript is the programming language used to create the command-line interface (CLI) for the client-side. It is a versatile language that can run both in the browser and on the server (using Node.js). JavaScript enables asynchronous communication with the server, which is a requirement for this project.

Node.js: Node.js is a JavaScript runtime that allows the language to run on the client-side, enabling the creation of the CLI for this project. Node.js also provides built-in modules like readline for handling user input and output in the console.

**Development approach:**

Initially, I made modifications to two server-side functions - the download and upload functions. Although they were functional in the previous assignment, I updated them for greater scalability. Specifically, I made sure that they now perform downloads and uploads with a high degree of accuracy by using bytes.

As for the JavaScript client-side code uses Promise-based asynchronous operations to ensure that the API calls don't block the execution of the program. Promises are a modern way to handle asynchronous operations in JavaScript, allowing for more readable and maintainable code that follows a modular, asynchronous, and event-driven structure. Here is a breakdown of the development approach with examples from the code:

**The functions:**

The code is organized into modular functions, each responsible for a specific task, such as browsing a folder, renaming a file, downloading a file, uploading a file, or deleting a file. These functions simplify the code and make it easier to understand, maintain, and extend.

It also uses Promise-based asynchronous operations to handle communication with the server. Promises allow the code to run non-blocking operations, ensuring that the API calls don't lock up the execution of the program. For example, in the browseFolder function, the fetch function returns a Promise that resolves with the response, which is then converted to JSON:

   return fetch(url)

        .then(response => response.json())

        .catch(error => console.error("Error while browsing folder:", error));

Here the stanrd method is GET, we can use POST or PUT with:

    return fetch(url, { method: "POST" })

        .then(response => response.text())

        .catch(error => console.error("Error while uploading file:", error));

Some are converted to text, files(blob) no always json depending on the function.

**Command Line Interface**

The client-side application uses a command-line interface to interact with the user. The CLI is built using Node.js and JavaScript, with the readline module handling user input and output.

It also prints a menu similar to the one that we build with python in the previous assignment.

The only difference is that in the main function, the modular functions are called using the async/await syntax. This syntax allows for writing asynchronous code in a more synchronous style, making it easier to read and understand.

async: The async keyword is used to declare a function as asynchronous. An async function always returns a Promise, even if the return value is not a Promise.

await: The await keyword is used inside an async function to wait for a Promise to resolve before continuing execution. It effectively "pauses" the function until the Promise is resolved, without blocking the rest of the program.

In the main function, the modular functions are called using await, which ensures that the user interaction is sequential, and the next prompt isn't displayed until the current action is completed.

rl.question("Enter subfolder (leave empty for root): ", async (subFolder) => {

          const files = await browseFolder(subFolder);

          console.log("Files and folders:", files);

          main();

        });

You can observe that ‘then’ and ‘catch’ are used with Promises to handle asynchronous code in the modular functions, while async/await is used in the main function to handle the sequential user interaction. Both methods are used to manage asynchronous operations, but they are used in different parts of the code and have different syntax styles.

But the main purpose of using async await in the main is due to this line:

  rl.question("Enter your choice: ", async (choice) => {

Here we ask the user to prompt his choice and we have to wait for a callback from the function, and the use nested then() and catch() blocks, which can lead to less readable code.

By using async and await, the code becomes cleaner and more readable.