# Project Report
folderwatcher

## Table of Contents



# Solution Description

## Overall Description of the project

The aim of the project is to apply reactive programming techniques with RxJS to produce a responsive web application that can track and display change in a remote shared folder.

The project is a web application that allows user to view and track change in a remote shared folder. The application uses a Pseudo-RESTful service developed using Java Spring on the backend to retrieve data about the remote folder. On the frontend, the application is built using HTML, CSS, and JavaScript, with the use of RxJS for implementing reactive programming.

The primary functionality of the application is to display the contents of the remote shared folder on the client-side and update the presentation if any changes are made to the remote folder. The user can set the time interval for update by inputting a value n in a text field. The content is updated each n seconds, where n is an input parameter. The application uses the RxJS interval operator to trigger update at the set interval.

The client will have a simple user interface with an input field for the n parameter and a start/stop button to control the updates. When the user clicks the start button, the application starts updating the presentation at the set interval. The button text changes to "Stop," and when the user clicks the end button, the application stops updating the display. The content display will be a grid of icons representing files and folders. The icon will be created and modified using JavaScript, and the presentation will be updated in real-time using the RxJS operators.

To implement this functionality, the application uses the RxJS Subject and takeUntil operator to manage the start and end events. The Subject is used to emit the start and end events, and the takeUntil operator is used to stop update the presentation when the end result is emitted. In addition to displaying the remote folder contents, the application also tracks change in the remote folder by using the RxJS fetch method to periodically retrieve the contents of the remote folder and compare them to the previous contents. If any changes are detected, the application update the presentation accordingly.

The main part of the client can be split into the index.js and the browse-folder.js.

Index.js is responsible for starting and stopping the interval that updates the content display on the browser. It sets up a listener on the click of a button, check if the input field is valid, and then create an interval using the rxjs.interval function. The interval emits value at the specified time interval and is subscribed to using the takeUntil operator, which ensures that the subscription is closed when the stopClick$ observable emits a value. The tap operator is used to update the countdown timer and call the browseFolder function when the countdown reaches zero. The browseFolder function makes a request to a remote server using the fetch function and updates the display with the return data.

browse-folder.js contained the browseFolder function, which is called when the countdown reaches zero to update the local content display (inside the browser) with the content of the remote shared folder. It makes an HTTP request to the remote server using the fetch function and then parses the return data to create HTML element representing the files and folders in the remote shared folder. The elements are then appended to the document body.

The two codes interact by the browseFolder function being called when the countdown timer reaches zero, causing the browser to request and display the content of the remote shared folder. This satisfies the necessity of displaying the content of the remote shared folder each n seconds. The rxjs.interval function is used to check the tempo of the update and to provide for the easy start and end of the updates, satisfying the requirements of the app being developed using RxJS and the user being able to begin and end the update by clicking the same button.

RxJS Operators

The code implements Reactive programming using RxJS by making use of RxJS's operators and observables.

It starts by defining the necessary observables and operators using the **rxjs** and **rxjs.operators** packages. The **interval** operator is used to create a time-based observable that emits a value every n second, where n is the value inputted by the user.

The **tap** operator is then used to perform side effects, such as updating the remaining time and the timer display, every time the **countdown$** observable emits a value. The **takeUntil** operator is used to complete the observable sequence when the **stopClick$** observable emits a value, indicating that the user has clicked the stop button.

When the user clicks the start button, the code sets up the **countdown$** and **updateInterval$** observables to start emitting values, and when the user clicks the stop button, the observables are unsubscribed, and the sequence is completed.

In this way, the code uses the power of observables and operators to create a reactive program that responds to user input and updates the display accordingly, while also allowing for easy handling of asynchronous events such as HTTP requests.

In this code, we are utilizing several RxJS operators and concepts to implement reactive programming:

interval: This is an RxJS function that creates an Observable that emits sequential numbers at a specified interval (in milliseconds). We use it to create an Observable that emits every **n** seconds (where **n** is the user input).

Subject: This is a type of Observable that allows for multicasting, meaning that it can have multiple subscribers, and it can also emit values manually through its **next** method. We use it to create a Subject called **stopClick$**, which will emit a value whenever the user clicks the "Stop" button, thus stopping the update process.

tap: This is an RxJS operator that allows us to perform a side effect (i.e., execute a function) for each emitted value of an Observable without modifying the emitted values. We use it to decrement the remaining time and update the timer display every second and also to call **browseFolder** function whenever the countdown reaches zero.

takeUntil: This is an RxJS operator that allows us to complete (i.e., unsubscribe) from an Observable when another Observable emits a value. We use it to stop the countdown and update interval Observables when the user clicks the "Stop" button.

subscribe: This method is used to subscribe to an Observable and receive its emitted values. We use it to subscribe to both the countdown and update interval Observables.

By using these RxJS concepts and operators, we are able to implement reactive programming in our code. Instead of relying on imperative programming techniques, we are using Observables and operators to react to user input and emit values accordingly.

## Breakdown of index.js

Importing necessary modules: The code begins by importing the necessary modules from the RxJS library, including **interval** and **Subject**. It also imports the **tap** and **takeUntil** operators from the **rxjs.operators** module.

Selecting DOM elements: The code selects the relevant DOM elements from the HTML using **querySelector**.

Creating an observable for the update interval: The code creates an observable **updateInterval$** that emits a value every **inputField.value** seconds using the **interval** method.

Creating a subject for stopping the countdown: The code creates a **Subject** named **stopClick$** that will be used to stop the countdown.

Creating an event listener for the button: The code creates an event listener for the button, which toggles between 'Start' and 'Stop' text on click.

Starting the countdown: When the button is clicked and its text is 'Start', the code checks if the input field value is a valid number greater than 0. If it is, it updates the button text to 'Stop' and creates an observable **countdown$** that emits a value every second using the **interval** method. It also subscribes to the **countdown$** observable, which triggers the **tap** operator for every emitted value. The **tap** operator decrements the remaining time by 1, updates the timer display, and triggers the **browseFolder** function when the remaining time reaches 0. Finally, the **takeUntil** operator is used to stop the countdown when **stopClick$** emits a value.

Updating the update interval: The **updateInterval$** observable is also subscribed to and stopped by **takeUntil(stopClick$)**, but it doesn't do anything else.

Stopping the countdown: When the button is clicked and its text is 'Stop', the code updates the button text to 'Start', removes the 'stop' class and adds the 'start' class. It then emits a value from the **stopClick$** subject to stop the countdown using **next()**. Finally, it unsubscribes from the **countdown$** observable to clear the timer.

Detailed Breakdown:

Importing necessary modules
The code begins by importing two modules from the RxJS library: **interval** and **Subject**. The **interval** module emits a sequence of values at a specified time interval, and the **Subject** module is a special type of observable that allows values to be manually emitted to its subscribers. The **tap** and **takeUntil** operators from the **rxjs.operators** module are also imported. The **tap** operator

performs a side effect for each emission, and the **takeUntil** operator completes the observable sequence when another observable emits a value.

```
const { interval, Subject } = rxjs;
const { tap, takeUntil } = rxjs.operators;
```

Selecting DOM elements
The code selects three DOM elements from the HTML using **querySelector**. The **inputField** variable selects an input field with the class **time-input**, the **clickButton** variable selects a button with the ID **clickButton**, and the **timerDisplay** variable selects an element with the ID **timerDisplay**.

```
const inputField = document.querySelector('
.time-input');
const clickButton = document.querySelector('
#clickButton');
const timerDisplay = document.querySelector('
#timerDisplay');
```

Creating an observable for the update interval
The code creates an observable **updateInterval$** that emits a value every **inputField.value** seconds using the **interval** method. This observable is used to update the timer display with the remaining time.

```
const updateInterval$ = interval(1000 * parseInt
(inputField.value));
```

Creating a subject for stopping the countdown
The code creates a **Subject** named **stopClick$** that will be used to stop the countdown. This subject is used in the **takeUntil** operators for both the countdown and the update interval observables.

```
const stopClick$ = new Subject();
```

Creating an event listener for the button
The code creates an event listener for the button that toggles between 'Start' and 'Stop' text on click. If the button text is 'Start' and a valid input value is entered, the countdown timer starts. If the button text is 'Stop', the countdown timer stops.


Starting the countdown
When the button is clicked and its text is 'Start', the code checks if the input field value is a valid number greater than 0. If it is, it updates the button text to 'Stop' and creates an observable **countdown$** that emits a value every second using the **interval** method. It also subscribes to the **countdown$** observable, which triggers the **tap** operator for every emitted value. The **tap** operator decrements the remaining time by 1, updates the timer display, and triggers the **browseFolder** function when the remaining time reaches 0. Finally, the **takeUntil** operator is used to stop the countdown when **stopClick$** emits a value.

```
if (clickButton.textContent === 'Start') {
    const inputInterval = parseInt(inputField
.value);
    if (isNaN(inputInterval) || inputInterval
<= 0) {
      alert('Please enter a valid interval');
      return;
    }

    clickButton.textContent = 'Stop';
    clickButton.classList.remove('start');
    clickButton.classList.add('stop');

    let remainingTime = inputInterval;
    timerDisplay.textContent = remainingTime;
    countdown$ = interval(1000).pipe(
      tap(() => {
        remainingTime--;
        timerDisplay.textContent =
remainingTime;
        if (remainingTime === 0) {
          browseFolder();
          remainingTime = inputInterval;
          timerDisplay.textContent =
remainingTime;
        }
      }),
      takeUntil(stopClick$)
    ).subscribe();

    updateInterval$.pipe(
      takeUntil(stopClick$)
    ).subscribe();
```

Stopping the countdown
When the button is clicked and its text is 'Stop', the code updates the button text to 'Start' and unsubscribes from the **countdown$** observable using its **unsubscribe()** method. It also emits a value to **stopClick$** using its **next()** method, which stops the **takeUntil** operators for both the countdown and the update interval observables.


Breakdown of browse-folder.js

Removing existing icons
The first thing the function does is select all elements with the classes **.folder-icon** and **.file-icon** and remove them from the DOM using the **forEach** method.

```
const icons = document.querySelectorAll('
.folder-icon, .file-icon');
icons.forEach(icon ⇒ icon.remove());
```

Defining the folder path and URL
The function sets the **folderPath** variable to an empty string and creates a URL variable **url** with the **folderPath** appended as a query parameter.

```
const folderPath = '';
const url =
`http://localhost:8080/remotestorage/browseFolde
r?folderPath=
${folderPath}`;
```

Fetching data from the server
The function makes a GET request to the URL using the **fetch** method. It checks if the response is OK and returns the response text as a promise. If the response is not OK, it throws an error.

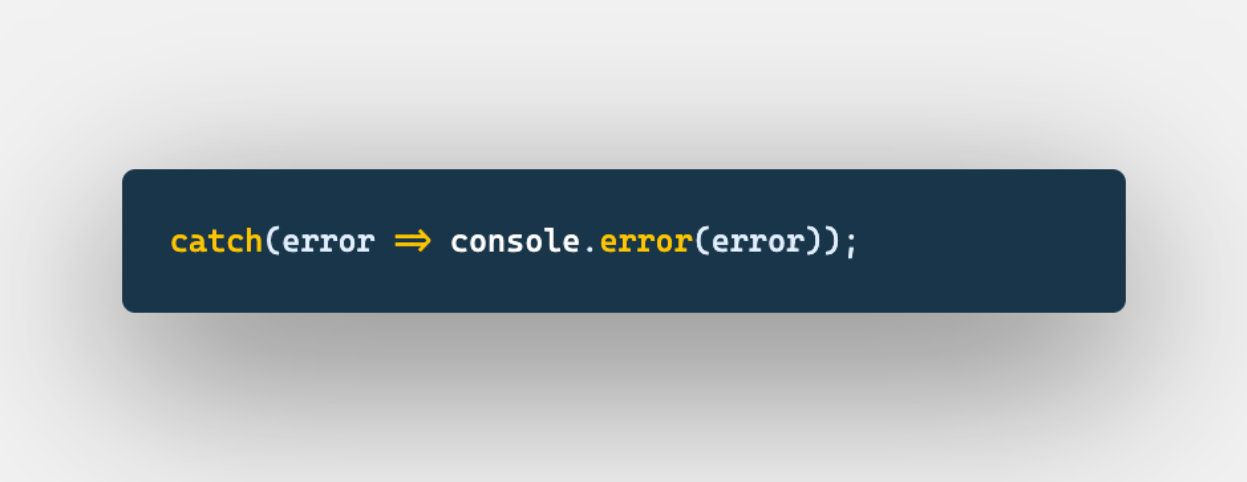
```
fetch(url)
    .then(response => {
      if (!response.ok) {
        throw new Error(`HTTP error! status: ${
response.status}`);
      }
      return response.text();
    })
```

Processing the data

The function takes the response data and trims it, then splits it into an array of items using the newline character **\n** as the delimiter. It then iterates over the array using a **for...of** loop and creates a new **div** element with the class **icon** for each item. It also creates a new **a** element with either the class **folder-icon** or **file-icon**, depending on whether the item starts with the string **'folder'** or not. The **href** attribute of the **a** element is set to the **folderPath** concatenated with the item, and the **innerHTML** of the **a** element is set to the item. Additionally, the **a** element has an **onclick** event listener that selects the clicked element and removes the **selected** class from any previously selected elements.

```
fetch(url)
  .then(response => {
    if (!response.ok) {
      throw new Error(`HTTP error! status: ${
response.status}`);
    }
    return response.text();
  })
  .then(data => {
    const items = data.trim().split('\n');
    for (const item of items) {
      const newDiv = document.createElement('
div');
      newDiv.classList.add('icon');
      const newAnchor = document.createElement(
'a');
      if (item.startsWith('.')) {
        // hidden file
        continue;
      } else if (item.startsWith('folder')) {
        // folder
        newAnchor.classList.add('folder-icon');
        newAnchor.href = folderPath + item + '/
';
        newAnchor.onclick = function() { const
elements = document.querySelectorAll('
.file-icon, .folder-icon');elements.forEach(
element => {element.classList.remove('selected'
);});newAnchor.classList.add('selected');
return false; };
        newAnchor.innerHTML = item;
      } else {
        // file
        newAnchor.classList.add('file-icon');
        newAnchor.href = folderPath + item;
        newAnchor.onclick = function() { const
elements = document.querySelectorAll('
.file-icon, .folder-icon');elements.forEach(
element => {element.classList.remove('selected'
);});newAnchor.classList.add('selected');
return false; };
        newAnchor.innerHTML = item;
      }
      newDiv.appendChild(newAnchor);
      document.body.appendChild(newDiv);
    }
  })
```

Catching errors
If there is an error during the fetch operation, the function logs the error to the console using **console.error**.




## Technology enablers

This project uses several technology enablers to accomplish the goal of displaying the content of a remote shared folder in a browser-based HTML/JavaScript client using Reactive programming with RxJS.

Java Spring:
The Java Spring Framework is used to develop the RESTful service that provides the API for accessing the remote shared folder.

RESTful API: enables the communication between the Java Spring service and the HTML/JavaScript client.

RxJS: RxJS is a library for Reactive programming in JavaScript. It provides a powerful set of operators for composing asynchronous and event-based programs using observable sequences. It is used in this project to implement the Reactive behavior of the client, enabling it to respond to changes in the remote shared folder and update the display accordingly. RxJS uses reactive extensions (Rx) and brings reactive programming to JavaScript. The library provides a lot of operators that help developers to create, filter, transform, and combine observable sequences.

In this project, RxJS is used to implement the client-side of the application. It provides an easy way to create an observable that emits values at a given interval, which is used to periodically fetch the remote shared folder's content. RxJS also provides operators like takeUntil, which helps to stop the observable sequence when a certain condition is met, and tap, which helps to perform side-effects when values are emitted.

HTML and CSS: used to create the user interface of the client.

JavaScript: used in this project to handle user interactions, communicate with the Java Spring service through the RESTful API, and implement the Reactive behavior using RxJS. It is used to handle user input, fetch the remote shared folder's content, and update the user interface.

Fetch API: The Fetch API provides a way to fetch resources asynchronously across the network. It provides a simple interface for fetching resources, and its response is a promise that resolves to the actual response.
In this project, the Fetch API is used to fetch the remote shared folder's content asynchronously. It is used to send an HTTP GET request to the RESTful service that provides the API for accessing the remote shared folder.

DOM Manipulation: The Document Object Model (DOM) is a programming interface for HTML and XML documents. It represents the page so that programs can change the document structure, style, and content. In this project, DOM manipulation is used to update the display of the client with the content of the remote shared folder.

Visual Studio Code: was used as the primary code editor for developing the client-side HTML/JavaScript application.

Gradle: Gradle is an open-source build automation tool that is designed to be flexible, scalable, and easy to use.