**IoT Weather Warning System - Technical Documentation**

This document details the specific modifications and additions made to the IoT Weather Warning System's components, explaining the reason behind each change.

**Fire Data Node (fire\_node.js – Code Snippet 1) - New Implementation**

The fire\_node.js was introduced as a new component to fetch real-time fire danger ratings from the Country Fire Authority (CFA) Victoria and send them to the central Weather Service.

**Key Implementation Details and Reasoning:**

* **Fetching RSS Feed:** We used the rss-parser library to retrieve the XML RSS feed from https://www.cfa.vic.gov.au/cfa/rssfeed/tfbfdrforecast\_rss.xml.
  + **Reasoning:** rss-parser simplifies the process of parsing RSS/Atom feeds into manageable JavaScript objects, allowing us to easily access feed items.
* **Handling Status code 406 Error:** Initially, requests to the CFA RSS feed resulted in a Status code 406 (Not Acceptable) error. This was resolved by configuring rss-parser to send standard HTTP headers.
  + **Explanation:** Many web servers, including those hosting RSS feeds, inspect HTTP request headers (like User-Agent and Accept) to determine if the client is a legitimate browser or an automated script. By sending common browser-like headers, we made our requests appear valid to the CFA server.
* **Correcting latestItem.description to latestItem.content:** After resolving the 406 error, a new issue arose where cheerio.load() expected a string, but latestItem.description was undefined. Debugging revealed that rss-parser mapped the RSS item's <description> XML tag to the content property of the parsed JavaScript object, not description.
  + **Explanation:** rss-parser has conventions for how it maps XML elements to JavaScript object properties. In this case, the main HTML content within the <description> tag was made available via the content property.
* **Parsing HTML with cheerio:** The extracted content (which is an HTML string) contains the fire danger ratings. cheerio was used to navigate this HTML structure.
  + **Reasoning:** cheerio provides a jQuery-like syntax for parsing and manipulating HTML, making it easy to find specific elements (like <p> tags) and extract their text or HTML content.
  + **Process:** We specifically looked for the <p> tag containing "Fire Danger Ratings", then extracted data from the *next* <p> tag, splitting it by <br> tags to get individual area ratings.
* **Data Transmission to Weather Service:** The extracted fire warning levels are formatted as a JSON object (e.g., {"Central": "NO RATING", "Mallee": "LOW"}) and sent via TCP socket to the Weather Service using the command fire,{"JSON\_DATA"}.
  + **Reasoning:** JSON is a lightweight and human-readable format for transmitting structured data, making it suitable for this purpose.
* **Periodic Updates:** The fetchFireWarning function is called every 5 minutes using setInterval.
  + **Reasoning:** Fetching too frequently (e.g., every few seconds) can lead to IP blocking or rate-limiting by the external RSS feed provider. A 5-minute interval is generally more polite and sustainable for public data sources.

**Weather Service Node (weather\_service.js – Code Snippet 2) - Modifications**

The central weather\_service.js was modified to correctly receive and process the new fire data from fire\_node.js and to handle request commands for localized warnings.

**Key Modifications and Reasoning:**

* **Correct JSON Payload Parsing:** The primary issue here was that the data.toString().split(',') method incorrectly split the incoming JSON string from the fire\_node.js.
  + **Explanation:** When fire\_node.js sends fire,{"Central":"NO RATING", ...}, the JSON string itself contains commas. A simple split(',') would break this JSON into multiple pieces (e.g., command[1] would be {"Central":"NO RATING", command[2] would be "East Gippsland":"NO RATING"}). This led to JSON.parse() receiving an incomplete and invalid string.
  + **Solution:** We implemented a more robust parsing method using indexOf(',') and substring() to ensure only the *first* comma separates the command name from the entire JSON payload.
* **Handling fire command:** A new case "fire": was added to the switch statement to process the incoming fire warning data.
  + **Explanation:** This case is responsible for parsing the rawCommandValue (which is now the full JSON string) into the fireWarningLevels object using JSON.parse(). This object then stores the fire danger ratings for each region.
* **Handling request command with Localization:** The request command was updated to accept an areaToRequest parameter.
  + **Explanation:** Previously, request might have been generic. Now, rawCommandValue after the split for a request command directly contains the area name (e.g., "Central"). The service uses this area name to look up the specific fire warning from the fireWarningLevels object and includes it in the response.

**Warning Request Node (warning\_request.js – Code Snippet 3) - Modifications**

The warning\_request.js client was modified to enable requesting warnings for specific localized areas.

**Key Modifications and Reasoning:**

* **Dynamic Area Request:** A new const areaToRequest variable was introduced.
  + **Explanation:** This allows the client to specify which Victorian region it wants to query (e.g., "Central"). This makes the request node more flexible and capable of simulating requests from different geographical locations.
* **Purpose:** This modification ensures that the system can demonstrate the delivery of **localized warnings**, which was a key requirement.

**Code of fire\_node.js, modified weather\_service.js, modified warning\_request.js**

// fire\_node.js

// This code implements a TCP socket client that connects to a weather service

// and fetches fire warning levels from an RSS feed, sending updates to the server.

const net = require("net");

const Parser = require("rss-parser");

const cheerio = require("cheerio");

const host = "127.0.0.1";

const port = 6000;

const CFA\_RSS\_FEED\_URL = "https://www.cfa.vic.gov.au/cfa/rssfeed/tfbfdrforecast\_rss.xml";

// Create a TCP client

const client = new net.Socket();

// Initialize variables to store fire warning levels

let fireWarningLevels = {};

// Function to fetch fire warning levels from the RSS feed

async function fetchFireWarning() {

    try {

        // Use rss-parser to fetch and parse the RSS feed

        // Ensure the parser is configured correctly

        // to handle the specific structure of the CFA RSS feed

        // Note: The CFA RSS feed may have specific headers or content types

        // that need to be handled, so we use a custom parser configuration

        let parser = new Parser({

            headers: {

                'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/91.0.4472.124 Safari/537.36',

                'Accept': 'application/xml, text/xml, \*/\*'

            }

        });

        let feed = await parser.parseURL(CFA\_RSS\_FEED\_URL);

        // Check if the feed has items

        // If items are present, process the first item (usually today)

        // If no items are found, set a default warning level

        if (feed.items.length > 0) {

            const latestItem = feed.items[0];

            // Use cheerio to parse the content of the latest item

            // and extract the fire warning levels

            // This assumes the content is HTML and contains the relevant information

            const $ = cheerio.load(latestItem.content);

            // Initialize fireWarningLevels to an empty object

            fireWarningLevels = {};

            // Find the paragraph that contains the "Fire Danger Ratings"

            const fireDangerRatingsHeaderP = $('p').filter(function() {

                return $(this).text().trim() === 'Fire Danger Ratings';

            });

            // If the header is found (or there is a sentence called "Fire Danger Ratings"), proceed to extract the ratings

            if (fireDangerRatingsHeaderP.length > 0) {

                // Find the next paragraph that contains the actual ratings

                // This assumes the ratings are in the next paragraph after "Fire Danger Ratings"

                const actualRatingsP = fireDangerRatingsHeaderP.next('p');

                if (actualRatingsP.length > 0) {

                    // Split the ratings by line breaks and process each line

                    // to extract area names and their corresponding ratings

                    // e.g., "Central: NO RATING<br>East Gippsland: NO RATING<br>Mallee: NO RATING<br>North Central: NO RATING<br>..."

                    const rawRatings = actualRatingsP.html();

                    // Split the raw ratings by line breaks and store them in an array

                    // This assumes the ratings are separated by <br> tags

                    // Outcome: ratingLines = ["Central: NO RATING", "East Gippsland: NO RATING", ...]

                    const ratingLines = rawRatings.split('<br>');

                    // Process each element of ratingLines to extract area names and ratings

                    // and store them in the fireWarningLevels object

                    ratingLines.forEach(element => {

                        // Trim whitespace and match the pattern "Area: Rating"

                        // e.g., "Central: NO RATING"

                        // Outcome: fireWarningLevels = { "Central": "NO RATING", "East Gippsland": "NO RATING", ... }

                        const trimmedElement = element.trim();

                        const match = trimmedElement.match(/^(.+?):\s\*(.+)$/);

                        // If the match is successful, extract area name and rating

                        // and store them in the fireWarningLevels object

                        // eg., match = ["Central: NO RATING", "Central", "NO RATING"] where:

                        // match[0] = "Central: NO RATING" (full match) comes from ^

                        // match[1] = "Central" (area name) comes from (.+?)

                        // match[2] = "NO RATING" (rating) comes from \s\*(.+)$

                        if (match && match.length === 3) {

                            let areaName = match[1].trim();

                            let rating = match[2].trim();

                            // fireWarningLevels["Central"] = "NO RATING"

                            fireWarningLevels[areaName] = rating;

                        }

                    });

                } else {

                    console.warn("Warning: Could not find the actual fire ratings paragraph.");

                    fireWarningLevels = { "ALL": "NO RATING - PARAGRAPH\_NOT\_FOUND" };

                }

            } else {

                console.warn("Warning: Could not find 'Fire Danger Ratings' header.");

                fireWarningLevels = { "ALL": "NO RATING - HEADER\_NOT\_FOUND" };

            }

            console.log("Fire warning levels fetched successfully:", fireWarningLevels);

        } else {

            console.log("No items found in the RSS feed.");

            fireWarningLevels = { "ALL": "NO RATING - NO\_ITEMS" };

        }

    } catch (error) {

        console.error("Error fetching fire warning levels:", error.message);

        fireWarningLevels = { "ALL": "ERROR\_FETCHING" };

    }

    // Send the fire warning levels to the server

    // Check if the client is connected before sending data

    // If the client is not connected, attempt to reconnect and send data

    if (client.readyState === 'open') {

        client.write(`fire,${JSON.stringify(fireWarningLevels)}`);

        console.log(`Sent: fire,${JSON.stringify(fireWarningLevels)}`);

    } else {

        console.log("Client is not connected, attempting to reconnect and send data.");

        // Attempt to reconnect if the client is not connected

        // This is a simple retry mechanism to ensure the client can send data

        // after reconnecting to the server

        if (client.connecting === false && client.destroyed === false) {

            client.connect(port, host, () => {

                console.log("Reconnected to Weather Service");

                client.write(`fire,${JSON.stringify(fireWarningLevels)}`);

                console.log(`Sent after reconnect: fire,${JSON.stringify(fireWarningLevels)}`);

            });

        } else {

            console.log("Client is already connecting or destroyed. Will try again next interval.");

        }

    }

}

// Connect to the weather service

client.connect(port, host, () => {

    console.log("Connected to Weather Service");

    fetchFireWarning();

    setInterval(fetchFireWarning, 300000);

});

// Handle incoming data from the server

client.on("data", (data) => {

    console.log(`Received from Weather Service: ${data.toString()}`);

});

// Handle socket error event

client.on("error", (error) => {

    console.error(`Client Error: ${error.message}`);

});

// Handle socket end event

client.on("close", () => {

    console.log("Connection closed");

});

Code Snippet 1 – fire\_node.js

// weather\_service.js

// This code implements a TCP socket server that listens for weather data updates

// and responds to requests for weather conditions.

const net = require("net");

const port = 6000;

// Initialize variables to store weather data

let temp;

let wind;

let rain;

let fireWarningLevels = {};

// Create a TCP server

const server = net.createServer((socket) => {

    console.log("Client connected");

    // Set the encoding for the socket

    socket.on("data", (data) => {

        const strData = data.toString().trim();

        console.log(`Received: ${strData}`);

        const firstCommaIndex = strData.indexOf(",");

        if (firstCommaIndex === -1) {

            console.error("Invalid command format: No comma found.");

            socket.write("error");

            return;

        }

        const name = strData.substring(0, firstCommaIndex);

        const rawCommandValue = strData.substring(firstCommaIndex + 1);

        // Process the command based on the name

        switch (name) {

            case "temp":

                temp = parseFloat(rawCommandValue);

                console.log(name + “ : “ + temp);

                result = "ok";

                break;

            case "rain":

                rain = parseFloat(rawCommandValue);

                console.log(name + “ : “ + rain);

                result = "ok";

                break;

            case "wind":

                wind = parseFloat(rawCommandValue);

                console.log(name + “ : “ + wind);

                result = "ok";

                break;

            case "fire":

                try {

                    fireWarningLevels = JSON.parse(rawCommandValue);

                    console.log("Received Fire Warning Levels:", fireWarningLevels);

                    result = "ok";

                } catch (error) {

                    console.error("Error parsing fire warning levels:", error);

                    result = "error";

                }

                break;

            case "request":

                const requestedArea = rawCommandValue;

                console.log(`Request for weather conditions in area: ${requestedArea}`);

                const currentFireWarning = fireWarningLevels[requestedArea] || "NO DATA";

                // Respond with the current weather conditions

                if(temp > 20 && rain < 50 && wind > 30){

                    result = `Weather Warning. Fire Warning Level: ${currentFireWarning}`;

                }

                else {

                    result = `Everything is fine. Fire Warning Level: ${currentFireWarning}`;

                }

                break;

        }

        // Send the result back to the client

        socket.write(result.toString());

    });

    // Handle socket end event

    socket.on("end", () => {

        console.log("Client disconnected");

    });

    // Handle socket error event

    socket.on("error", (error) => {

        console.log(`Socket Error: ${error.message}`);

    });

});

// Handle server errors

server.on("error", (error) => {

    console.log(`Server Error: ${error.message}`);

});

// Start the server and listen on the specified port

server.listen(port, () => {

    console.log(`TCP socket server is running on port: ${port}`);

});

Code Snippet 2 – Modified weather\_service.js to complement with fire\_node.js

// warning\_request.js

// This code implements a TCP socket client that sends a request for weather conditions

// and listens for responses from the server.

const net = require("net");

const host = "127.0.0.1";

const port = 6000;

// Specify the area for which to request weather conditions

// List of areas can be defined based on the fire warning levels:

// Central, East Gippsland, Mallee, North Central, North East, Northern Country, South West, West and South Gippsland, Wimmera

const areaToRequest = "Central"; // Specify the area for which to request weather conditions

// Create a TCP client that connects to the server

const client = net.createConnection(port, host, () => {

    console.log("Connected to Weather Service");

    // Set the encoding for the client

    setInterval(() => {

        client.write(`request,${areaToRequest}`);

        console.log(`Sent request for weather conditions in area: request,${areaToRequest}`);

    }, 2000); // Interval in milliseconds (2000ms = 2 seconds)

});

// Handle incoming data from the server

client.on("data", (data) => {

    console.log(`Received: ${data.toString()}`);

//    process.exit(0);

});

// Handle socket error event

client.on("error", (error) => {

    console.log(`Error: ${error.message}`);

});

// Handle socket end event

client.on("close", () => {

    console.log("Connection closed");

});

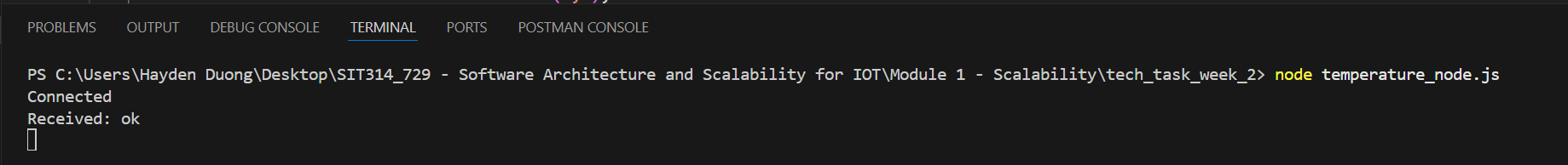
Code Snippet 3 – Modified warning\_request.js

Photo:

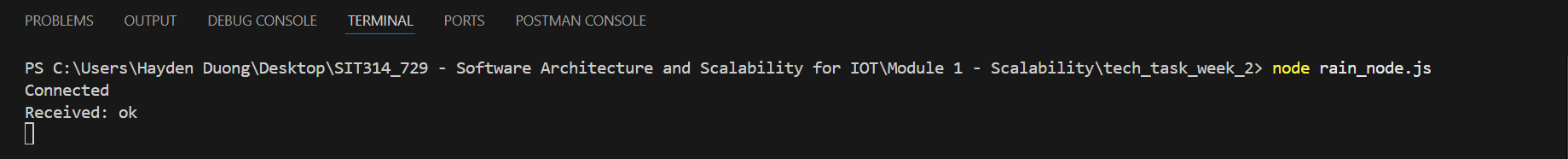
A screenshot of a computer

AI-generated content may be incorrect.

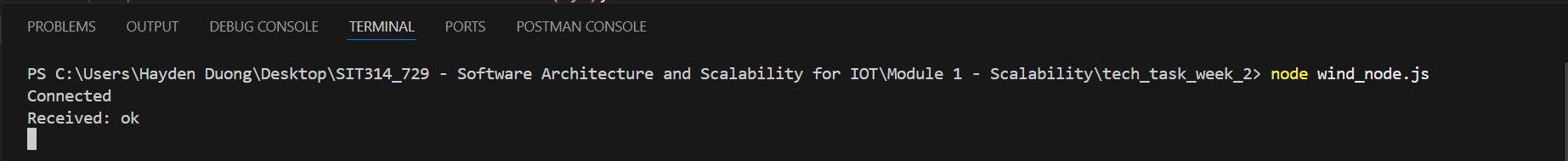
Picture 1 – weather\_service.js server received data from temperature\_node, rain\_node, wind\_node, fire\_node, and from warning\_request



Picture 2 – temperature\_node is successfully connected to weather\_service server.



Picture 3 – rain\_node is successfully connected to weather\_service server.

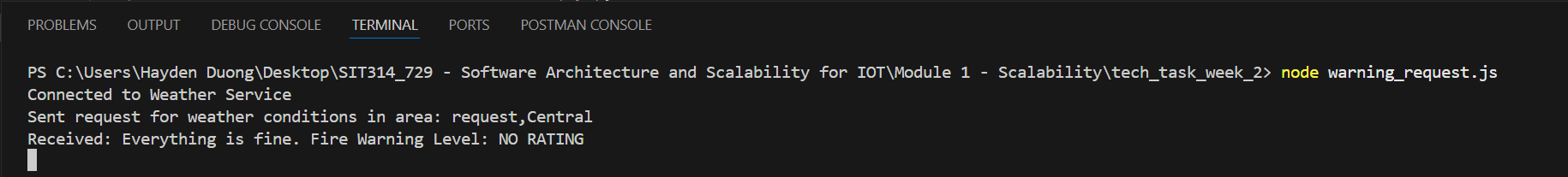


Picture 4 - wind\_node is successfully connected to weather\_service server.

A black screen with white text

AI-generated content may be incorrect.

Picture 5 – fire\_node is successfully connected to weather\_service server.



Picture 6 – warning\_request is successfully connected to weather\_service server and received warning result.