Challenge

Chiara Thien Thao Nguyen Ba 10727985

April 26, 2025

1 Node-Red

The following picture illustrates the obtained Node-Red flow.

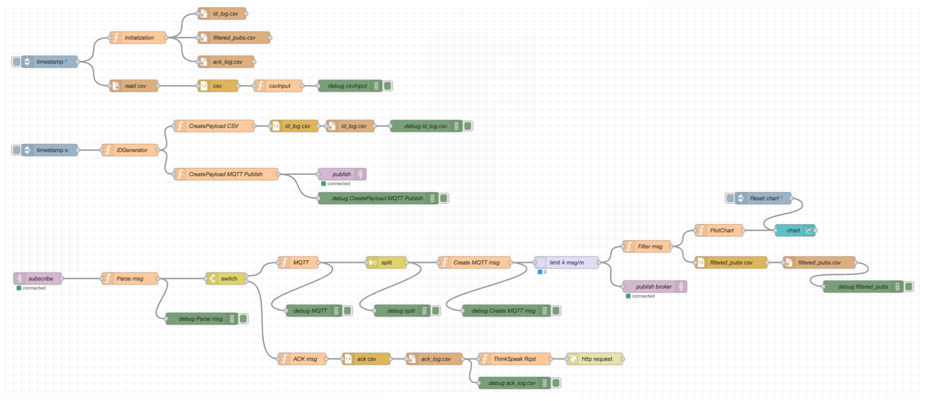

Figure 1: The obtained Node-Red flow

1.1 Nodes explanation

1.1.1 Initialization

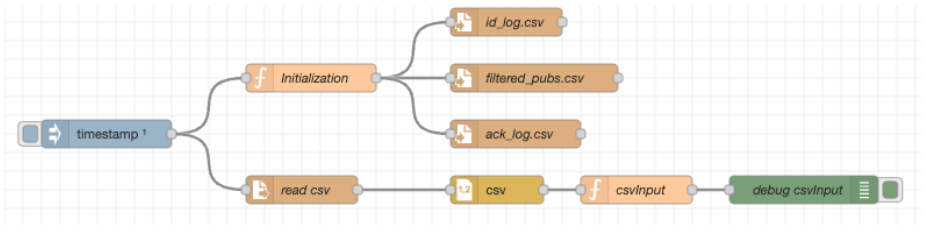

Figure 2: Initialization of the output files and reading of the input file

In the first part, there is the initialization of the csv output files and the reading of the csv input file. In particular, the inject node is used to start the flow, which is split in two parallel branches. The first branch is composed by the *Initialization* function, which simply has On Start the following code:

```
    global.set("Status", 0);
```

for setting the global Status to 0, meaning the flow can be processed, and On Message has the following code:

```
    msg.payload = "";
    return msg;
```

Then, there are three write file nodes *id_log.csv, filtered_pubs.csv, ack_log.csv*, which initialize the output files with the correct output path and by overwriting the files from previous executions.

The second branch is made by a read file node, which reads the input file *challenge3.csv* from the specified path, a csv node that parses the csv input file and the *csvInput* function with On Message the following code:

```
    flow.set("input", msg.payload);
    return msg;
```

which simply sets the variable `input` with the array content of the input file.

1.1.2 Generation of random IDs and Publish MQTT messages

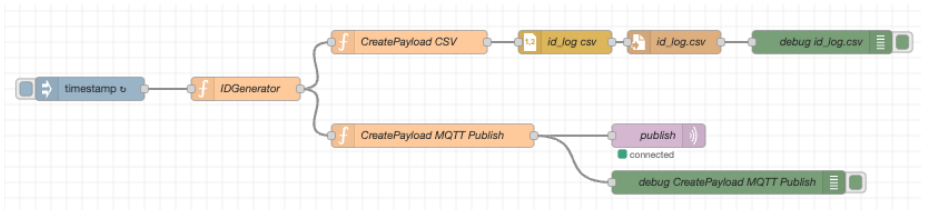

Figure 3: Generation of random IDs and publish MQTT messages

In the second part, there are the generation of the random ID numbers for the MQTT Publish messages and the publish of these messages on topic *challenge3/id_generator*. The inject node is set to create a new message every 5 seconds, it is followed by the *IDGenerator* function, which generates the message payload with the random ID number and the current timestamp. In the following, there is the On Message code of this function.

```
if (global.get("Status")===1) return null;

var randomNumber = Math.round(Math.random() * 30000);

msg.payload = {
    id: randomNumber,
    timestamp: msg.payload,
};

return msg;
```

Then, there are two parallel branches. The first one is used to create the payload of the *id_log.csv* file, with the *CreatePayload CSV* function, which has the following code for On Start:

```
if (context.get("rowCount") === undefined) {
    context.set("rowCount", 0)
}
```

and the following code for On Message:

```
var rowCount = context.get("rowCount");
rowCount++;
context.set("rowCount", rowCount);

msg.payload = {
    "No.": rowCount,
    "ID": msg.payload.id,
    "TIMESTAMP": msg.payload.timestamp,
};

return msg;
```

It creates the required payload for saving the generated IDs in the output file. Thus, it is followed by the csv node for parsing the values and writing them in the correct output path in the *id_log.csv* file with the write file node.

The second branch is composed by the *CreatePayload MQTT Publish* function, which has this On Message code:

```
    msg.payload = JSON.stringify(msg.payload);
    return msg;
```

It converts the message payload received from the *IDGenerator* function into a JSON-formatted string for the following publish broker. The publish broker has Server: localhost:1884 and Topic: *challenge3/id_generator*.

1.1.3 Subscription messages

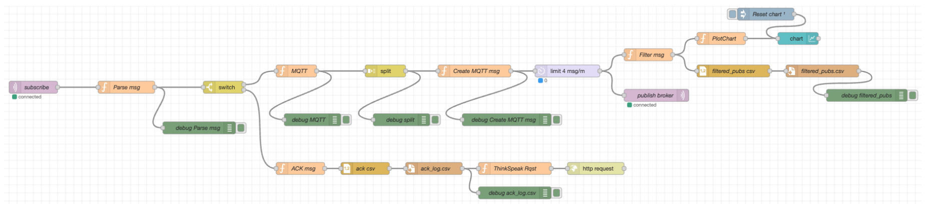

Figure 4: Handling of MQTT subscribe messages

The third part aims to handle the subscription messages. It starts with the mqqt in node, with Server: localhost:1884 and Topic: *challenge3/id_generator*. The mqqt in node is followed by the *Parse msg* function, which has On Start code the initialization of the packet counter:

```
if (context.get("packetCounter") === undefined) {
    context.set("packetCounter", 0)
}
```

and On Message the following code:

```
// Update the packet counter from context
let packetCounter = context.get("packetCounter");
packetCounter += 1;
context.set("packetCounter", packetCounter);

// Bypass further processing after 80 packets
if (packetCounter >= 80) {
    node.warn("Processed 80 packtes");
    global.set("Status", 1);
    msg.reset = true;
    return msg ;
}

// Retrieve the input CSV array
var input = flow.get("input");

// Store the Publish payload
flow.set("publishMessage", msg.payload);

// Compute the row index using modulo operation
var remainder = msg.payload.id % 7711;

// Retrieve the input row from CSV array
var inputRow = input[remainder - 1];

// Store the selected row of CSV input
flow.set("inputRow", inputRow);
```

```

// Initialize message type
msg.type = "";

// Determine the message type based on content
const info = inputRow["Info"];

if (info.includes("Publish Message")) {
    msg.type = "PUBLISH";
    return msg;
}

if (info.includes("Ack")) {
    msg.type = "ACK";
    return msg;
}

// If no known type matched but still within packet limit
node.warn("Skipped");

return msg;
```

which performs the following actions:

- For each subscribe message, it increments the `packetCounter` variable, and then checks if this variable is 80, if not it continues the subsequent actions, otherwise it stops processing new subscribe messages by setting the global `Status` to 1, meaning that the subscription flow is stopped.
- It computes the remainder with the modulo operation and uses the remainder for retrieving the corresponding row in the CSV input array from the `input` variable. It sets `inputRow` variable with the selected row.
- It determines the message type of `inputRow` by looking at the Info column and sets the return message type PUBLISH, ACK or ””, if the `inputRow` is a Publish message, an Ack message or none of the previous, respectively.

The switch node is used to create and handle two different branches based on the message type, one for the MQTT Publish messages and the other one for the ACK messages.

1.1.4 MQTT Publish Messages

The MQTT branch is composed by the *MQTT* function with the following On Message code:

```
// Retrieve stored publish message and input row from flow
let publishMessage = flow.get("publishMessage");
let inputRow = flow.get("inputRow");

```

```
// Extract all topics from the Info field
let infoField = inputRow["Info"];
let topics = [];
let regex = /\[([^\]]+)\]/g;
let match;

while ((match = regex.exec(infoField)) !== null) {
    topics.push(match[1]);
}

msg.topic = topics;

// Get the raw payload string
let raw = inputRow["Payload"];

if (raw !== undefined) {
    // Parse individual JSON objects from raw string
    var payloadChunks = [];
    let current = "";
    let depth = 0;
    let inString = false;
    let collecting = false;

    for (let i = 0; i < raw.length; i++) {
        let char = raw[i];

        // Toggle inString when encountering unescaped quotes
        if (char === '"' && raw[i - 1] !== '\\') {
            inString = !inString;
        }

        if (!inString) {
            if (char === '{') {
                if (depth === 0) collecting = true; // Start collecting a new object
                depth++;
            } else if (char === '}') {
                depth--;
            }
        }

        if (collecting) {
            current += char;
        }

        // End of a complete object
        if (collecting && depth === 0) {
            payloadChunks.push(current.trim());
            current = "";
            collecting = false;
        }
    }


    // Parse each chunk into JSON or null if malformed
    let payloadArray = payloadChunks.map(chunk => {
        try {
            return JSON.parse(chunk);
        } catch (e) {
            return null;
        }
    });


    // Build result array
    let maxLength = Math.max(topics.length, payloadArray.length);
    var result = [];

    for (let i = 0; i < maxLength; i++) {
        result.push({
            timestamp: Date.now(),
            id: publishMessage.id,
            topic: topics[i] || null,
            payload: payloadArray[i] || "{}"
        });
    }

    // Set msg.payload as array
    msg.payload = result;
} else {
    for (let i = 0; i < topics.length; i++) {
        msg = {
            timestamp: Date.now(),
            id: publishMessage.id,
            topic: topics[i] || null,
            payload: "{}"
        }
    }
}

return msg;
```

It extracts the topics from the Info column of the input array and parses one or more concatenated JSON objects from the payload string. Each extracted topic is then paired with its corresponding payload (or an empty object with "{}" if the payload is incomplete or missed), along with the current timestamp and message ID from the publish message. The output is an array of these structured message objects assigned to the message payload.

It is followed by the split node, which divides the message payload into more messages if there are more publish messages in one single message. The *Create MQTT msg* function has the following On Message code:

```
// Message payload empty
if (msg.payload === "{}") {
    msg.topic = msg.topic;
    return msg;
}
msg.topic = msg.payload.topic;

// Create message payload
msg.payload = {
    timestamp: new Date().toISOString(),
    id: msg.payload.id,
    topic: msg.payload.topic,
    payload: msg.payload.payload
}

return msg;
```

which creates the message payload if the current message has not an empty payload, by setting the current timestamp, the id, the topic and the payload, as required. It is followed by the delay node, which sets a rate limit of 4 messages per minute. The mqtt out node is used to publish the messages with their corresponding topic and the Server is localhost:1884.

The next nodes are used to filter the Fahrenheit temperatures and plot them on the chart. The *Filter msg* function has the following On Start code:

```
if (context.get("rowCountFiltered") === undefined) {
    context.set("rowCountFiltered", 0)
}
```

for counting the filtered rows, and the following On Message code:

```
//  Message payload empty
if (msg.payload === "{}") {
    return null;
}

let inner = msg.payload.payload;

// Filter: Only temperature messages in Fahrenheit
if (inner === null || inner.type !== "temperature" || inner.
    unit !== "F") {
    return null;
}

// Extract mean value from the range array
```

```
if (!Array.isArray(inner.range) || inner.range.length !== 2)
     {
    node.warn("Invalid range format");
    return null;
}

let meanValue = (inner.range[0] + inner.range[1]) / 2;

var rowCountFiltered = context.get("rowCountFiltered");
rowCountFiltered++;
context.set("rowCountFiltered", rowCountFiltered);

// Construct payload for CSV output
msg.payload = {
    "No.": rowCountFiltered,
    LONG: inner.long,
    LAT: inner.lat,
    MEAN_VALUE: meanValue,
    TYPE: inner.type,
    UNIT: inner.unit,
    DESCRIPTION: inner.description
};

return msg;
```

which filters the messages with type ”Temperature” and unit ”Fahrenheit” and calculates the mean value with the range values. It increments each time the counter of the rows and constructs the payload for the csv output file, by setting the current row count, the longitude, the latitude, the mean value, the type, the unit and the description.

The values are saved in the *filtered_pubs.csv* file through the csv node and the write file node.

The mean values of the filtered temperatures are plotted using *PlotChart* function, which has this On Message code:

```
if (msg !== null) {
    msg.topic = "";
    msg.payload = msg.payload["MEAN_VALUE"];
}

return msg;
```

used for creating a single line in the chart and it is followed by the chart node for showing the values on the graph. The chart node has an inject node called *Reset chart* for cleaning the chart each time at the beginning of the execution of the flow.

1.1.5 ACK Messages

The ACK branch is composed by the *ACK msg* function, which has the following On Start code:

```javascript
if (context.get("rowCountACK") === undefined) {
    context.set("rowCountACK", 0)
}
```

and the following On Message code:

```javascript
// Retrieve previously stored data from flow context
let publishMessage = flow.get("publishMessage");
let inputRow = flow.get("inputRow");

var rowCountACK = context.get("rowCountACK");
rowCountACK++;
context.set("rowCountACK", rowCountACK);

// Construct payload for CSV output
msg.payload = {
    "No.": rowCountACK,
    TIMESTAMP: Date.now(),
    SUB_ID: publishMessage.id,
    MSG_TYPE: inputRow["Info"]
};

return msg;
```

which increments each time the row counter variable and creates the payload for the ACK messages by setting the corresponding row counter, current timestamp, subscribe id and message type of the input array. It is followed by the csv node for creating the csv file *ack_log.csv* to write the values with the subsequent write file node.

Then, there is the *ThinkSpeak Rqst* function which has the following On Start code:

```javascript
// Initialize ACK packet counter
if (context.get("ackCounter") === undefined) {
    context.set("ackCounter", 0)
}
```

for setting up the counter of ACK messages received, and the following On Message code:

```javascript
var apiKey = "LAWGNK69EQ5JVH1C";

// Increase ACK counter
var ackCounter = context.get("ackCounter");
ackCounter++;
context.set("ackCounter", ackCounter);
```

```

msg.url = "https://api.thingspeak.com/update?api_key="
msg.url += apiKey;
msg.url += "&field1="
msg.url += ackCounter;

return msg;
```

for increasing each time the ACK counter, creating the HTTP request to the specified field channel on ThinkSpeak and plotting the updated value of the ACK counter. Thus, this is followed by the http request node with GET method.

1.2 Temperature chart

The following picture illustrates the temperature chart obtained by deploying the flow.

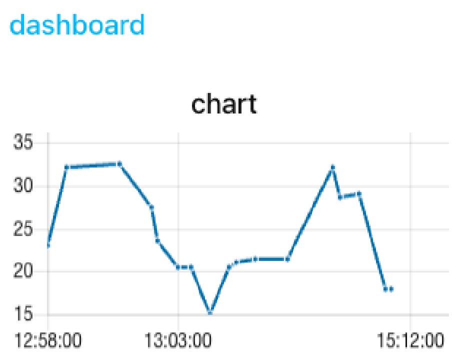

Figure 5: The obtained temperature chart from Node-Red