IoT Standard Project

By
Musse Gebreezgabher(6015033)
and
Eduard Fast(7169977)

Introduction

The goal of this project is to create an application capable of showing different health-related data of individual teams. These should be represented on a dashboard and processed to give further information about the condition of the person. We decided to simulate the values of three different types of activities which are separated into teams. The first one is “Team 1” who is walking, then “Team 2” who is running and sometimes walking and at last “Team 3”, who is sprinting and sometimes running. Based on their activities we extract their vital parameters, including the heart rate and the body temperature, then their current position and travel distance, and also their speed and acceleration. All the data will not be collected by real sensors, but simulated by an mqtt broker realistically.
The received data will be processed in real-time and in batch processing to give as much information as possible to the users. In the following parts, the code and the features of the simulator will be explained.

Project Overview

The project simulates sensor data for different teams and streams this data to an MQTT broker. Node-RED and Node-RED Dashboard are used to process, analyze, and visualize the data. The project includes various types of sensors like position, acceleration, speed, and vital parameters, and simulates their data for three different teams. This data is then visualized and analyzed using Node-RED. The project incorporates both real-time and batch data processing to handle the sensor data effectively.

Technologies Used

Node-RED: An open-source flow-based development tool for visual programming.
Node-RED Dashboard: A set of nodes for creating dashboards in Node-RED.
MQTT: A lightweight messaging protocol for streaming the sensor data.
Node.js: A JavaScript runtime for simulating the sensor data.

Code and Explanation

The provided Node.js script simulates sensor data for three teams and streams it to an MQTT broker.

```
const mqtt = require("mqtt");
const client = mqtt.connect("mqtt://localhost"); // Connect to MQTT Mosquitto
running on localhost
const path = require("./coordinates"); // Adjust the path to your actual file
location

let currentIndexTeam1 = 0;
let currentIndexTeam2 = 0;
let currentIndexTeam3 = 0;
let team = 1;
// Simulated sensor data

const sensorsTeams = [
  {
    team: 1,
    sensors: [
```

```
      {
        sensor_team: "team_1",
        sensor_type: "position",
        sensor_id: "gps_001",
        data: () => {
          // Get current position from path array and increment index
          const currentPosition = path[currentIndexTeam1];
          currentIndexTeam1 = (currentIndexTeam1 + 1) % path.length; // Wrap around
using modulo
          return {
            latitude: currentPosition.latitude,
            longitude: currentPosition.longitude,
          };
        },
      },
      {
        sensor_team: "team_1",
        sensor_type: "acceleration",
        sensor_id: "accel_002",
        data: () => ({
          x: (Math.random() - 0.5) * 2,
          y: (Math.random() - 0.5) * 2,
          z: (Math.random() - 0.5) * 2,
        }),
      },
      {
        sensor_team: "team_1",
        sensor_type: "speed",
        sensor_id: "speed_003",
        data: () => ({
          speed: parseFloat(4 + (Math.random() - 0.5) * 0.5).toFixed(1),
        }),
      },
      {
        sensor_team: "team_1",
        sensor_type: "vital_parameters",
        sensor_id: "heart_rate_004",
        data: () => ({
          heart_rate: 72 + Math.floor((Math.random() - 0.5) * 10),
          body_temperature: parseFloat(
            (36.6 + (Math.random() - 0.5) * 0.5).toFixed(1)
          ),
        }),
      },
    ],
  },
  {
```

```
  team: 2,
  sensors: [
    {
      sensor_team: "team_2",
      sensor_type: "position",
      sensor_id: "gps_001",
      data: () => {
        // Get current position from path array and increment index
        const currentPosition = path[currentIndexTeam2];
        currentIndexTeam2 = currentIndexTeam2 + 2;
        if (currentIndexTeam2 >= path.length - 1) {
          currentIndexTeam2 = 0; // Wrap around to the beginning if at the end
        }

        return {
          latitude: currentPosition.latitude,
          longitude: currentPosition.longitude,
        };
      },
    },
    {
      sensor_team: "team_2",
      sensor_type: "acceleration",
      sensor_id: "accel_002",
      data: () => ({
        x: (Math.random() - 1) * 4,
        y: (Math.random() - 1) * 4,
        z: (Math.random() - 1) * 4,
      }),
    },
    {
      sensor_team: "team_2",
      sensor_type: "speed",
      sensor_id: "speed_003",
      data: () => ({
        speed: parseFloat(8 + (Math.random() - 0.5) * 1).toFixed(1),
      }),
    },
    {
      sensor_team: "team_2",
      sensor_type: "vital_parameters",
      sensor_id: "heart_rate_004",
      data: () => ({
        heart_rate: 100 + Math.floor((Math.random() - 0.5) * 15),
        body_temperature: parseFloat(
          (37.0 + (Math.random() - 0.5) * 0.5).toFixed(1)
        ),
```

```
      }),
    },
  ],
},
{
  team: 3,
  sensors: [
    {
      sensor_team: "team_3",
      sensor_type: "position",
      sensor_id: "gps_001",
      data: () => {
        // Get current position from path array and increment index
        const currentPosition = path[currentIndexTeam3];
        currentIndexTeam3 = currentIndexTeam3 + 4;
        if (currentIndexTeam3 >= path.length - 1) {
          currentIndexTeam3 = 0; // Wrap around to the beginning if at the end
        }
        return {
          latitude: currentPosition.latitude,
          longitude: currentPosition.longitude,
        };
      },
    },
    {
      sensor_team: "team_3",
      sensor_type: "acceleration",
      sensor_id: "accel_002",
      data: () => ({
        x: (Math.random() - 1) * 4,
        y: (Math.random() - 1) * 4,
        z: (Math.random() - 1) * 4,
      }),
    },
    {
      sensor_team: "team_3",
      sensor_type: "speed",
      sensor_id: "speed_003",
      data: () => ({
        speed: parseFloat(16 + (Math.random() - 0.5) * 1).toFixed(1),
      }),
    },
    {
      sensor_team: "team_3",
      sensor_type: "vital_parameters",
      sensor_id: "heart_rate_004",
      data: () => ({
```

```
          heart_rate: 120 + Math.floor((Math.random() - 0.7) * 15),
          body_temperature: parseFloat(
            (37.5 + (Math.random() - 0.5) * 0.5).toFixed(1)
          ),
        }),
      },
    ],
  },
];

client.on("connect", () => {
  console.log("Connected to MQTT broker");

  setInterval(() => {
    sensorsTeams.forEach((teamSensors) => {
      const sensorData = teamSensors.sensors.map((sensor) => ({
        ...sensor,
        data: sensor.data(),
        timestamp: new Date().toISOString(),
        timestampRegular: Math.floor(Date.now() / 1000),
        team: teamSensors.team,
      }));
      sensorData.forEach((data) => {
        client.publish("team/sensors", JSON.stringify(data));
      });
    });
  }, 5000); // Send data every 5 seconds
});

client.on("error", (err) => {
  console.error("Connection error: ", err);
  client.end();
});

client.on("close", () => {
  console.log("Disconnected from MQTT broker");
});
```

Explanation of the Code

Dependencies:

The script uses the MQTT library to connect to an MQTT broker. It also requires a coordinates file to simulate GPS data.

MQTT Connection:

Connects to the MQTT broker running on localhost.

Simulated Sensor Data:

Defines sensor data for three teams. Each team has sensors for position, acceleration, speed, and vital parameters.

Data Simulation:

Each sensor has a data function that generates simulated data. Position data is generated using a predefined path of coordinates. Acceleration, speed, and vital parameters are generated using random values within a specified range.

Data Publishing:

The script publishes the simulated sensor data to the MQTT topic "team/sensors" every 5 seconds.
The data includes timestamp, sensor type, sensor ID, and team information.

Error Handling:

The script includes basic error handling for MQTT connection errors.

Sensor Data

The sensor data published to the MQTT broker includes the following fields:
timestamp: The current date and time.
team: The team number (1, 2, or 3).
timestampRegular: for calculating the acceleration
sensor_team: The team identifier (e.g., "team_1").
sensor_type: The type of sensor (position, acceleration, speed, or vital parameters).
sensor_id: The sensor identifier.
data: The sensor data, which includes:
For position sensors: latitude and longitude.
For acceleration sensors: x, y, and z-axis values.
For speed sensors: speed value.
For vital parameters sensors: heart rate and body temperature.

Hot Path Data Processing

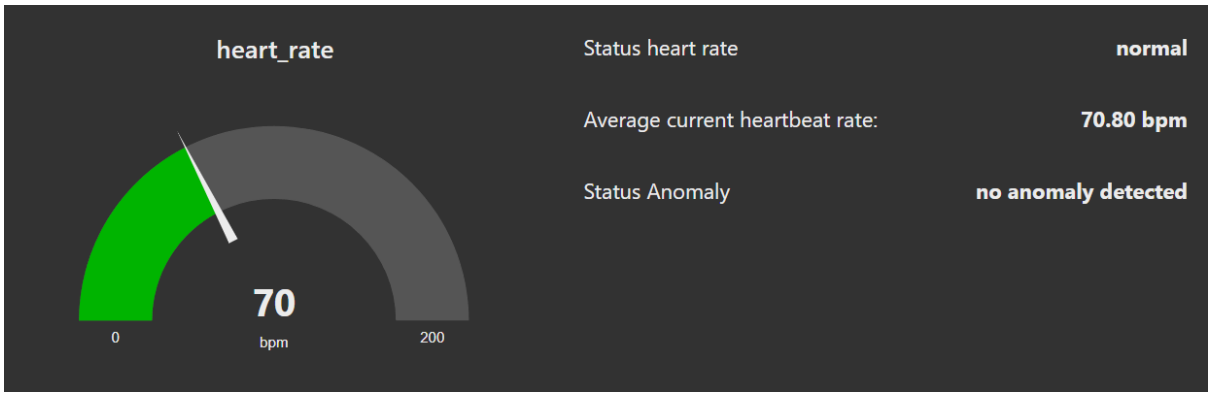
Hot Path Data Processing processes data in real-time. This is important for systems that need to get their data processed immediately to give necessary information if needed.
In the context of our Health Data Simulator, we wanted to show all the sensor data on the Node-Red Dashboard displayed right away. This means the vital parameters, the speed, acceleration, and position sensor should show the data right as they came out of the MQTT Broker. An important factor here is, that the data shouldn’t be only streamed as it is, but also processed and analyzed, to make conclusions about the current status of the person using the health device.
Since there are three different teams to choose from, the dashboard contains a dropdown list. It contains the values “Team 1”, “Team 2”, “Team 3” and “All teams”, to give the possibility to choose between individual groups and all the groups combined.

1. Vital Data

A. Heart Rate Flow

First of all the data from the MQTT Broker had to be separated. Therefore the function “Hot Path Heartrate” extracted the needed values for the heart rate portrayal. The function processes the data

according to the features which were planned. These include the representation of the current status of the heart rate, whether it is “normal” or “high”, the current trend of the values which were shown by gathering the last ten received transmissions and calculating their average value, and at last the current value should be displayed in a gauge.
The heart rate dashboard can be seen in the following picture:



To extract the data out of the mqtt broker, the right sensor “vital_parameters” needs to be addressed. It contains information about the heart rate and the body temperature. Furthermore, the chosen team needs to be filtered to get the right values from the desired group.

```
if (msg.payload.sensor_type === "vital_parameters") {
if(msg.payload.sensor_team === teamId && teamId !== "all"){
    let { heart_rate, body_temperature } = msg.payload.data;
```

Afterward, the status of the person needs to be determined by evaluating the heart rate and the body temperature value. If the heart rate goes beyond 100 or the body temperature goes above 37.5, the status of the person's vital parameters will change to high, otherwise, it will remain normal.

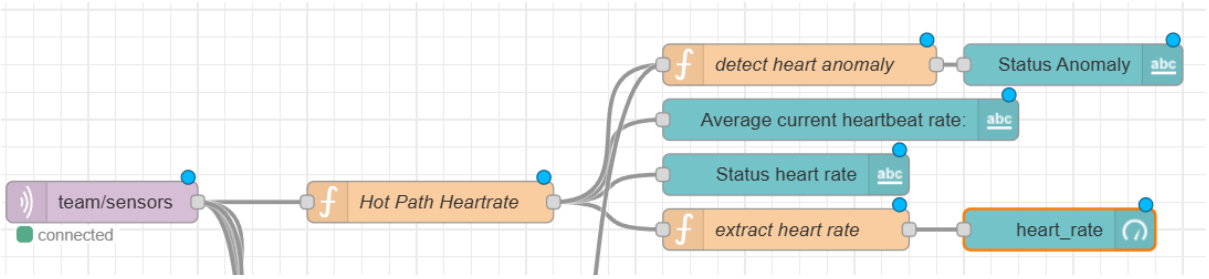
```
if (heart_rate > 100 || body_temperature > 37.5) {
        node.warn("Heart Rate: critical, value: "+ heart_rate);
        msg.payload = {
            status: "high",
            heart_rate: heart_rate,
            average : average
        };   return msg;
    }else{
        msg.payload = {
            status: "normal",
            heart_rate: heart_rate,
            average : average
        };  return msg;
    }
```

The average value will be calculated by adding the received value into an array called “values”. It will get saved in the context of the program so the values remain there and can be used for each sent message from the MQTT broker. Each heart rate value will get added together in a variable “sum”. Since we only

want the last 10 values, the array will push the first stored value out, when it gets larger than 10. If that happens, the value of the removed heart rate will also get subtracted from the “sum”. Afterwards, the average will be calculated by dividing the sum by the length of the “value” array. At the end the “sum” and the “values” have to be saved, so they can be used for the next incoming value.

```
function calculateAverage(heart_rate){
    var values = context.get('values') || [];
     var sum = context.get('sum') || 0;

var value = parseFloat(heart_rate);

if (!isNaN(value)) {
    values.push(value)
    sum += value;

    if (values.length > 10) {
        var removedValue = values.shift();
        sum -= removedValue;
    }

    var average = (sum / values.length).toFixed(2);

    context.set('values', values);
    context.set('sum', sum);
    return average;
    }
}
}
```

To display the value the msg.payload has to be delivered to the desired output nodes. The “status” message is sent to the “Average current heartbeat rate” node. The “average” message to the “Status heart rate” node, and the actual heart rate to the “heart_rate” gauge after getting extracted from the msg.payload by the “extract heart rate” function.



Another feature of the heart_rate is the possibility to detect heart anomalies. Therefore the “detect heart anomaly” function compares the calculated average value from the “Hot Path Heartrate” msg.payload and also uses the “activity” value from the speed sensor which will be discussed later. To simulate it, the

dashboard contains a “simulate anomaly” switch, which raises the current heart rate by 50 to trigger the anomaly message.

```
if(average >= 110 && activitySpeed === "walking"){
    statusAnomaly = "Heart anomaly detected";
}else if(average >= 130 && activitySpeed === "running"){
    statusAnomaly = "Heart anomaly detected";
}else if(average >= 160 && activitySpeed === "sprinting"){
    statusAnomaly = "Heart anomaly detected";
}else{
    statusAnomaly = "no anomaly detected";
}
```

In case any of the if clauses are fulfilled, the message will be displayed on the dashboard via a text message.

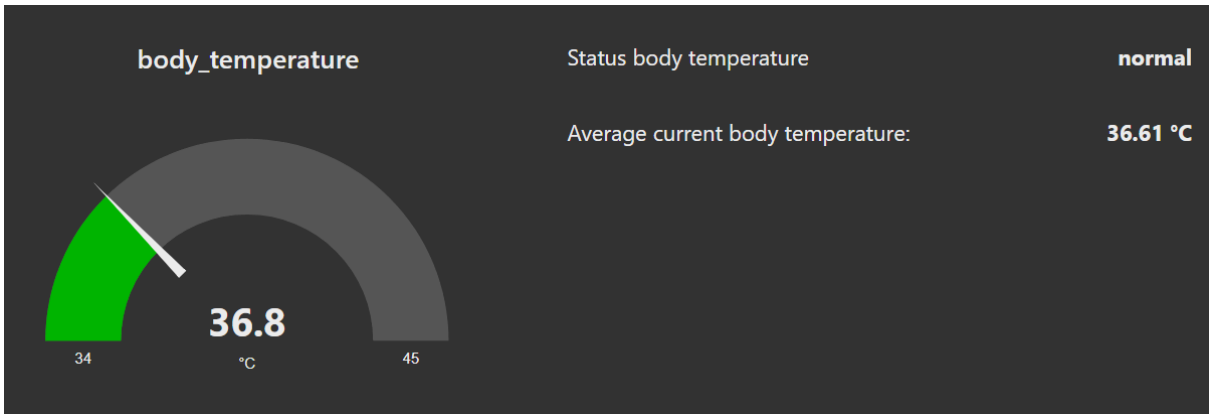
Furthermore, the displayed of all the values combined is also covered in the “Hot Path Heartrate” function. Therefore all the incoming values from each team are collected, and the average for each numerical output is generated. The gauge will display the average current value of all teams and the “average” will be the average ten values of each team combined. To achieve this each value has to be saved as a variable as seen in the code below by setting them in the context. Since the values arrive sequentially the transmission of team 3 would otherwise not know what the values of team 2 and team 1 were. After all values are collected the data processing will proceed and the msg.payload will be created.

```
else if(teamId === "all"){
   let heart_rate1 = context.get('HRTeam1') || 0;
   let heart_rate2 = context.get('HRTeam2') || 0;
   let heart_rate3 = context.get('HRTeam3') || 0;

    if(msg.payload.sensor_team === "team_1"){
        heart_rate1 = msg.payload.data.heart_rate;
        context.set('HRTeam1', heart_rate1);
    }
    if(msg.payload.sensor_team === "team_2"){
        heart_rate2 = msg.payload.data.heart_rate;
        context.set('HRTeam2', heart_rate2);
    }
    if(msg.payload.sensor_team === "team_3"){
        heart_rate3 = msg.payload.data.heart_rate;
        context.set('HRTeam3', heart_rate3);
    }

    const heart_rate = ((heart_rate1 +  heart_rate2 + heart_rate3)/3).toFixed(2);
    var average = calculateAverage(heart_rate);
    msg.payload = {
        status : "combined values",
        heart_rate : heart_rate,
```

```
        average : average,
        };
    return msg;
}
```

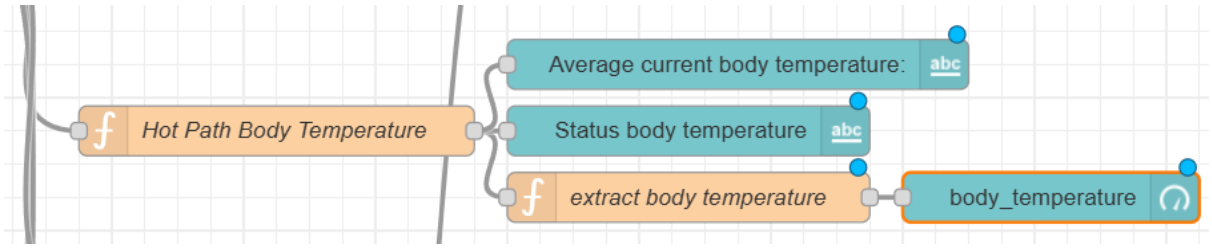
B. Body temperature flow

The body temperature dashboard and flow are very similar to the heart rate one. It gets its data from the same sensor, so even the code is very similar. It uses the body temperature variable instead of the heart rate variable and processes it in the same way. The msg.payload from the “Hot Path Body Temperature” function delivers the same output with the current status of the person, the average body temperature of the last ten received values, and the current body temperature for representation on a gauge



The flow is connected in a similar way and the “extract body temperature” function extracts the body temperature from the msg.payload for the gauge.

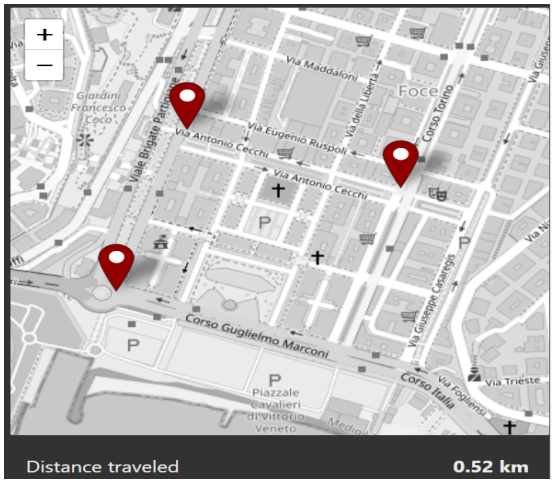
The combined values also work with the same code structure as above and the average values of the three teams will be displayed for all numerical values.



2. Position Flow

The next feature shows the position of each team on a node-red worldmap. When the representation of all teams was chosen, the goal was to display all teams at once and they should move according to the delivered latitude and longitude values. Below the map, a distance counter should be implemented which counts the overall distance of all teams.

If an individual team is chosen, it should be the only team displayed on the worldmap and the traveled distance should be the one from the specific team. Each team also should calculate their own distance traveled in real-time and display the right value when needed.
The result is shown in the figure below. The teams are represented as red markers and they are moving across the worldmap at the designated pace which is contributed by the mqtt broker.



A. Calculate Distance

To achieve the functionality we created two separate functions to handle the data from the mqtt broker. The first one “Calculate Distance” calculates the distance for each team individually and stores it. If it is chosen by the Dropdown list, it will send the data as msg.payload to the text node and display it on the dashboard. In the code below the example of the distance of team 1 is shown. It is determined by the incoming latitude and longitude values. When they get broadcasted by the MQTT broker, they will be stored immediately in a “latArray” and “lonArray”. Each team has its own array to be able to distinguish the calculations. When each array reaches the size of two. The function “haversineDistance(lat1, lon1, lat2, lon2)” takes the values and calculates the traveled distance which will get summed in the “totalDistance” variable. Afterward, when new latitude and longitude values arrive, the oldest lat/lon pair will be deleted from the array, to start the calculation again with the new values.

```
var teamId = global.get("selectedTeam");
var totalDistanceTeam1 = context.get('totalDistanceTeam1') || 0;
let latArrayTeam1 = context.get('previous_latArrayTeam1') || [];
let lonArrayTeam1 = context.get('previous_lonArrayTeam1') || [];
var totalDistanceTeam2 = context.get('totalDistanceTeam2') || 0;
let latArrayTeam2 = context.get('previous_latArrayTeam2') || [];
let lonArrayTeam2 = context.get('previous_lonArrayTeam2') || [];
var totalDistanceTeam3 = context.get('totalDistanceTeam3') || 0;
let latArrayTeam3 = context.get('previous_latArrayTeam3') || [];
let lonArrayTeam3 = context.get('previous_lonArrayTeam3') || [];

if(msg.payload.sensor_type === "position"){
```

```
    if(msg.payload.sensor_team === "team_1"){
        const { latitude, longitude, altitude } = msg.payload.data;
        latArrayTeam1.push(latitude);
          if(latArrayTeam1.length > 2){
             latArrayTeam1.shift();
           }
          lonArrayTeam1.push(longitude);
          if(lonArrayTeam1.length > 2){
             lonArrayTeam1.shift();
}
          if(latArrayTeam1.length == 2){
            var distanceCalc = haversineDistance(latArrayTeam1[0],
lonArrayTeam1[0], latArrayTeam1[1], lonArrayTeam1[1]);
    totalDistanceTeam1 += distanceCalc;
}
}
context.set('totalDistance', totalDistanceTeam1);
context.set('totalDistanceTeam1', totalDistanceTeam1);

context.set('previous_latArrayTeam1', latArrayTeam1);
context.set('previous_lonArrayTeam1', lonArrayTeam1);
}
```

When all the “totalDistance” values are generated, it is possible to choose the desired value based on the code below. Each value gets divided by 1000 to get the kilometer value. For the total value when all the teams are displayed, all the individual values are added together.

```
if (teamId === "team_1"){
         msg.payload = (totalDistanceTeam1 / 1000).toFixed(2);
    }else if(teamId === "team_2"){
        msg.payload = (totalDistanceTeam2 / 1000).toFixed(2);
    }else if(teamId === "team_3"){
        msg.payload = (totalDistanceTeam3 / 1000).toFixed(2);
    }else if(teamId === "all"){
        msg.payload = ((totalDistanceTeam1 + totalDistanceTeam2 +
totalDistanceTeam3)/1000).toFixed(2);
    }
    return msg;
```

B. Hot Path Positon function

The next function is called “Hot Path Position”. It shows the current position of each team on the node-red worldmap.

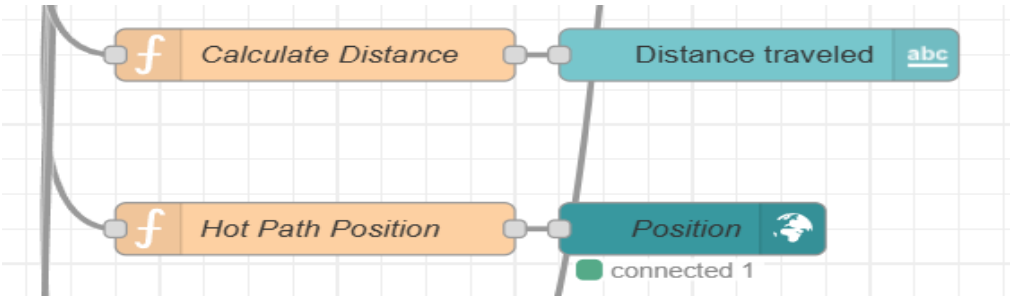
In this part, we encountered the most problems. The main difficulty in this feature was the right deletion of the markers that should not be displayed. After several trials, we found a solution which is shown in the

code below. We came across the possibility of having the previously chosen team stored to determine which values are not needed in the worldmap anymore. So after each change, the previous value would get stored in the variable “previousTeamId”. Now if the “previousTeamId” is “all” and the current “teamId” is not all, the current displayed value will get deleted. If the current teamId is “all”, all three position markers will be displayed with the next data flow. If the current “teamId” is a specific team, then only the chosen team will be shown.
In the end, also the switching between the teams had to be covered. Therefore an if clause checks if the “previousValue” is not “all” and the current “teamId” and that “teamId” is not “all”. If so the previous marker will get deleted.
Therefore the array “deletePayload” exists, which stores all the team markers that are displayed in the msg.payload and deletes them in the function “addDeletePayload if needed.

```
// Retrieve the teamId from the global context
var teamId = global.get("selectedTeam");
var previousTeamId = context.get("previousTeamId");

var deletePayloads = [];
var updatePayload = null;

// Function to add a deletion message
function addDeletePayload(name) {
    deletePayloads.push({
        name: name,
        deleted: true
    });
}

// Handle incoming payload
if (msg.payload.sensor_type === "position") {
    // Handle the case where 'all' was previously selected
    if (previousTeamId === "all" && teamId !== "all") {
        // If switching from "all" to a specific team, delete all previous team
markers
        ["team_1", "team_2", "team_3"].forEach(team => {
            if (team !== teamId) {
                addDeletePayload(team);
            }
        });
    }
    // Handle the case where 'all' is currently selected
    if (teamId === "all") {
        // Add/update markers for all teams
        updatePayload = {
            name: msg.payload.sensor_team,
            lat: msg.payload.data.latitude,
            lon: msg.payload.data.longitude
```

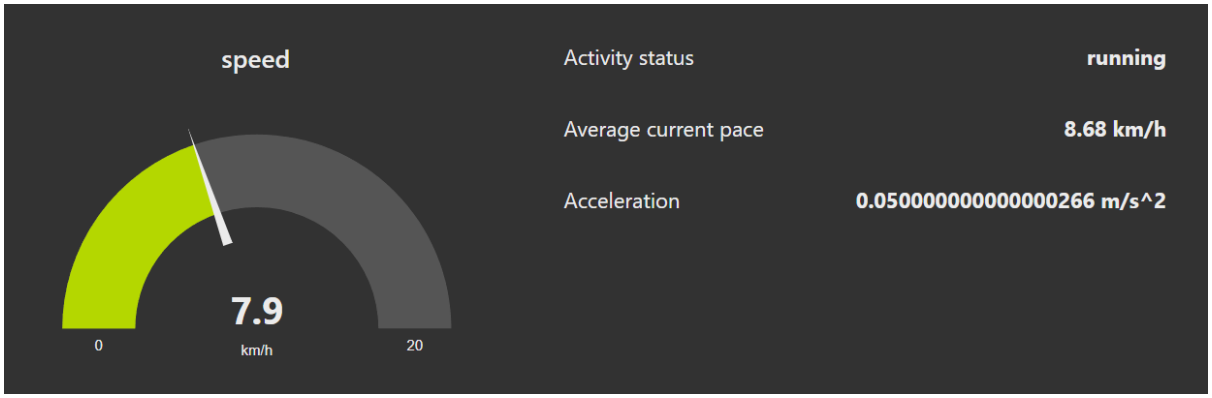
```
        };
    } else if (teamId !== "all" && msg.payload.sensor_team === teamId) {
        // Add/update marker for the selected team
        updatePayload = {
            name: teamId,
            lat: msg.payload.data.latitude,
            lon: msg.payload.data.longitude,
            altitude: msg.payload.data.altitude
        };
    }
}
// Handle the case where switching from a specific team to another
if (previousTeamId !== "all" && previousTeamId !== teamId && teamId !== "all") {
    addDeletePayload(previousTeamId);
}
// Update the context with the current teamId
context.set("previousTeamId", teamId);

msg.payload = deletePayloads;
if (updatePayload) {
    msg.payload.push(updatePayload);
}
return msg;
```

By connecting the flows to the worldmap and the text field, the data can be seen.



3. Speed and acceleration Flow

The speed sensor shows the current speed of the running person in km/h. The function “Hot Path Data” determines the activity status, the average current pace, and the acceleration of the team.
The three activities were “walking”, “running” and “sprinting. These activities are further used to detect heart anomalies as mentioned earlier.

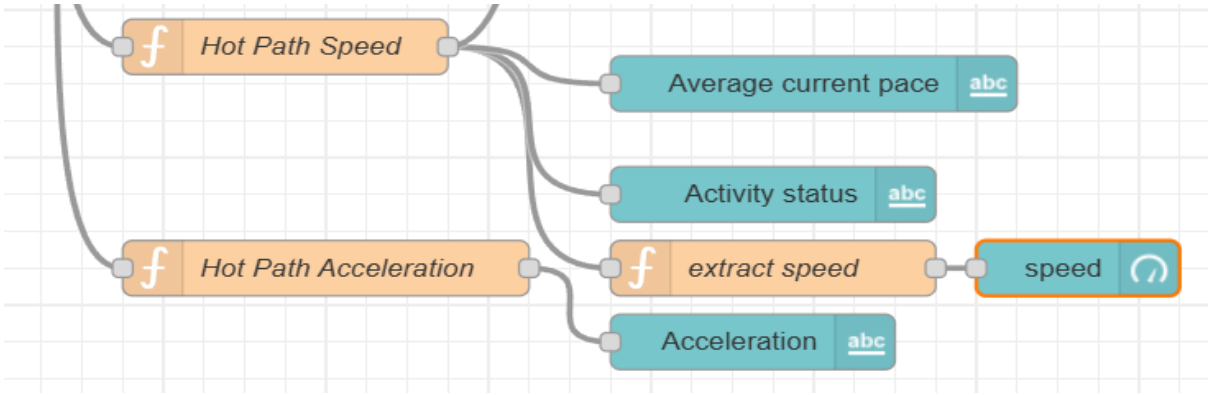


The average value is calculated in the same way as the heart rate and the body temperature.
Also, the values for the combined teams are calculated in the same way.
The acceleration is generated in a separate function “Hot Path Acceleration”. There it is calculated by using the speed data, and the timestamp data from the mqtt broker as seen in the code below with an example for team 1.
The formula for the acceleration is the delta of the previous velocity the current velocity and the delta of the previous timestamp and the current time stamp. To separate the data each team needs to have their own variables which need to be saved in the context individually. If all the values are gathered correctly and the formula is used right the right acceleration will be displayed.
If all the teams are selected the acceleration of each team will be shown summed up.

```
var current_timestamp = msg.payload.timestampRegular;
var previous_velocityTeam1 = context.get('previous_velocityTeam1') || null;
var previous_velocityTeam2 = context.get('previous_velocityTeam2') || null;
var previous_velocityTeam3 = context.get('previous_velocityTeam3') || null;
var previous_timestampTeam1 = context.get('previous_timestampTeam1') || null;
var previous_timestampTeam2 = context.get('previous_timestampTeam2') || null;
var previous_timestampTeam3 = context.get('previous_timestampTeam3') || null;
var accelerationTeam1 = context.get('accelerationTeam3') || null;;
var accelerationTeam2 = context.get('accelerationTeam3') || null;;
var accelerationTeam3 = context.get('accelerationTeam3') || null;;

var teamId = global.get("selectedTeam");
let current_velocityTeam1 = null;
let current_velocityTeam2 = null;
let current_velocityTeam3 = null;

if (msg.payload.sensor_type === "speed") {

if(msg.payload.sensor_team === "team_1"){
    current_velocityTeam1 = msg.payload.data.speed;

if (previous_velocityTeam1 !== null && previous_timestampTeam1 !== null) {
```

```
    var velocity = current_velocityTeam1 - previous_velocityTeam1;
    var time = current_timestamp - previous_timestampTeam1;

    // Check if delta_t is not zero to avoid division by zero
    if (time !== 0) {
        // Calculate acceleration
        accelerationTeam1 = velocity / time;
        context.set('accelerationTeam1', accelerationTeam1);
        context.set('previous_velocityTeam1', current_velocityTeam1);
        context.set('previous_timestampTeam1', current_timestamp);
    } else {
        return null; // Ignore the message if delta_t is zero
    }
} else {
    // Initialize context variables if they are not set
    context.set('previous_velocityTeam1', current_velocityTeam1);
    context.set('previous_timestampTeam1', current_timestamp);
    return null; // No acceleration can be calculated on the first data point
}
```

The flow for the speed works the same way as the flow for the heartbeat and the temperature. However, the acceleration flow only displays its value on a single text field.



Cold Path (Batch Data Processing) for IoT Project Report

The cold path, or batch data processing, handles data that isn't time-sensitive but requires analysis to uncover valuable insights over extended periods. This method is useful for creating reports, conducting historical data analysis, and identifying trends from data gathered by IoT sensors.

In this project, we implemented a batch processing system using Node-RED to handle data from various sensors. Below is a detailed description of the process.

1. Listening to Streamed Data

The data from IoT sensors is received in real time through MQTT. We set up an mqtt-in node to subscribe to the topic "team/sensors" and listen for incoming messages. Each message contains data from different types of sensors such as position, acceleration, speed, and vital parameters.

2. Preparing and Writing Data to CSV Files

Once the data is received, we prepare it to be written to CSV files for each sensor type. This ensures that the data is stored in a structured format for batch processing. The following code handles this preparation:

```
var sensorData = msg.payload;
// Initialize context.headersWritten if it doesn't exist
context.headersWritten = context.headersWritten || {};
switch(sensorData.sensor_type) {
   case 'position':
       msg.filename = '/home/musse/Documents/Class/IOT/Final
Project/IoT-Sensor-Data-Processing-NodeRED/batch/gps_data.csv';
       if (!context.headersWritten.gps) {
           msg.payload =
`timestamp,timestampRegular,sensor_id,sensor_team,latitude,longitude\n${sensorData.
timestamp},${sensorData.timestampRegular},${sensorData.sensor_id},${sensorData.sens
or_team},${sensorData.data.latitude},${sensorData.data.longitude}`;
           context.headersWritten.gps = true;
       } else {
           msg.payload =
`${sensorData.timestamp},${sensorData.timestampRegular},${sensorData.sensor_id},${s
ensorData.sensor_team},${sensorData.data.latitude},${sensorData.data.longitude}`;
       }
       break;
   case 'acceleration':
       msg.filename = '/home/musse/Documents/Class/IOT/Final
Project/IoT-Sensor-Data-Processing-NodeRED/batch/accel_data.csv';
       if (!context.headersWritten.acceleration) {
           msg.payload =
`timestamp,timestampRegular,sensor_id,sensor_team,x,y,z\n${sensorData.timestamp},${
sensorData.timestampRegular},${sensorData.sensor_id},${sensorData.sensor_team},${se
nsorData.data.x},${sensorData.data.y},${sensorData.data.z}`;
           context.headersWritten.acceleration = true;
       } else {
           msg.payload =
`${sensorData.timestamp},${sensorData.timestampRegular},${sensorData.sensor_id},${s
ensorData.sensor_team},${sensorData.data.x},${sensorData.data.y},${sensorData.data.
```

```
z}`;
        }
        break;
    case 'speed':
        msg.filename = '/home/musse/Documents/Class/IOT/Final
Project/IoT-Sensor-Data-Processing-NodeRED/batch/speed_data.csv';
        if (!context.headersWritten.speed) {
            msg.payload =
`timestamp,timestampRegular,sensor_id,sensor_team,speed\n${sensorData.timestamp},${
sensorData.timestampRegular},${sensorData.sensor_id},${sensorData.sensor_team},${se
nsorData.data.speed}`;
            context.headersWritten.speed = true;
        } else {
            msg.payload =
`${sensorData.timestamp},${sensorData.timestampRegular},${sensorData.sensor_id},${s
ensorData.sensor_team},${sensorData.data.speed}`;
        }
        break;
    case 'vital_parameters':
        msg.filename = '/home/musse/Documents/Class/IOT/Final
Project/IoT-Sensor-Data-Processing-NodeRED/batch/vital_data.csv';
        if (!context.headersWritten.vital_parameters) {
            msg.payload =
`timestamp,timestampRegular,sensor_id,sensor_team,heart_rate,body_temperature\n${se
nsorData.timestamp},${sensorData.timestampRegular},${sensorData.sensor_id},${sensor
Data.sensor_team},${sensorData.data.heart_rate},${sensorData.data.body_temperature}
`;
            context.headersWritten.vital_parameters = true;
        } else {
            msg.payload =
`${sensorData.timestamp},${sensorData.timestampRegular},${sensorData.sensor_id},${s
ensorData.sensor_team},${sensorData.data.heart_rate},${sensorData.data.body_tempera
ture}`;
        }
        break;
    default:
        return null; // Ignore unknown sensor types
}
return msg;
```

A file write node is then used to write the prepared data to the respective CSV files.

3. Data Processing Trigger

To ensure that batch processing occurs periodically, an inject node is configured to trigger the process once every hour. The inject node is set up with the following options:

```
Inject once after 0.1 seconds, then repeat interval of 60 minutes
```

4. Team Selection

A dropdown node is provided for selecting the team whose data should be processed and displayed. The options include "all", "team1", "team2", and "team3". The selected team is stored in the global context:

```
// Function node to handle team selection from the dropdown
var teamId = msg.payload;
global.set("selectedTeam", teamId);
return msg;
```

Another function node retrieves the selected team from the global context for use in processing:

```
var teamId = global.get("selectedTeam");
msg.payload = teamId;
return msg;
```

5. Reading and Parsing CSV Data

For each type of sensor data, we use a file read node to read the respective CSV file. Below are the details for each sensor type.

A. Acceleration

i. Reading and Parsing Data

For acceleration sensor data, a file read node is used to read the accel_data.csv file. The data is then parsed in a function node:

```
var csvData = msg.payload;
var lines = csvData.split('\n');
var selectedTeam = global.get('selectedTeam');

// Initialize an array to hold the processed data
var processedData = [];

// Iterate over each line of the CSV
for (var i = 1; i < lines.length; i++) {
    var line = lines[i].trim();
    if (line === '') continue; // Skip empty lines
    var values = line.split(',');
    var data = {};

    // Assuming the CSV format:
timestamp,timestampRegular,sensor_id,sensor_team,x,y,z
```

```
    data.timestamp = values[0].trim();
    data.timestampRegular = parseInt(values[1].trim(), 10);
    data.sensor_id = values[2].trim();
    data.sensor_team = values[3].trim();
    data.x = parseFloat(values[4].trim());
    data.y = parseFloat(values[5].trim());
    data.z = parseFloat(values[6].trim());

    if (!isNaN(data.x) && !isNaN(data.y) && !isNaN(data.z)) {
        data.magnitude = Math.sqrt(data.x * data.x + data.y * data.y + data.z *
data.z);

        if (selectedTeam !== "all" && selectedTeam != null && selectedTeam !=
undefined) {
            if (data.sensor_team === selectedTeam) {
                processedData.push(data);
            }
        } else {
            processedData.push(data);
        }
    }
}
// Set the processed data as the message payload
msg.payload = processedData;
return msg;
```

ii. Visualization

To visualize the acceleration data, specifically the magnitude of acceleration, we use a function node to prepare the data for charting:

```
var processedData = msg.payload;
// Initialize an array to hold the data for the chart
var chartData = [];
// Iterate over the processed data to extract timestamp and magnitude
for (var i = 0; i < processedData.length; i++) {
   var data = processedData[i];

   // Only process entries with calculated magnitude
   if (data.magnitude !== undefined) {
       chartData.push({
           x: new Date(data.timestamp),
           y: data.magnitude
       });
   }
}
// Set the processed data for the chart as the message payload
```

```
msg.payload = [{
   series: ["Magnitude"],
   data: [chartData],
   labels: [""]
}];
return msg;
```

The prepared data is then sent to a `line chart` node for visualization.

iii. Statistical Analysis

We also perform statistical analysis on the acceleration data to derive insights such as mean, median, variance, and standard deviation. This is done in a function node:

```
// Function node for statistical analysis
var data = msg.payload;

// Helper function to calculate median
function median(values) {
   if (values.length === 0) return 0;
   values.sort(function(a, b) {
       return a - b;
   });
   var half = Math.floor(values.length / 2);
   if (values.length % 2) {
       return values[half];
   } else {
       return (values[half - 1] + values[half]) / 2.0;
   }
}

// Initialize arrays to hold values for each axis and magnitude
var xValues = [];
var yValues = [];
var zValues = [];
var magnitudeValues = [];

// Populate arrays with data
for (var i = 0; i < data.length; i++) {
   xValues.push(data[i].x);
   yValues.push(data[i].y);
   zValues.push(data[i].z);
   magnitudeValues.push(data[i].magnitude);
}

// Function to calculate mean
function mean(values) {
```

```
    var sum = values.reduce((a, b) => a + b, 0);
    return sum / values.length;
}

// Function to calculate variance
function variance(values, meanValue) {
    return values.reduce((a, b) => a + Math.pow(b - meanValue, 2), 0) /
values.length;
}

// Calculate statistics for each axis and magnitude
var meanX = mean(xValues);
var meanY = mean(yValues);
var meanZ = mean(zValues);
var meanMagnitude = mean(magnitudeValues);

var varianceX = variance(xValues, meanX);
var varianceY = variance(yValues, meanY);
var varianceZ = variance(zValues, meanZ);
var varianceMagnitude = variance(magnitudeValues, meanMagnitude);

var stdDevX = Math.sqrt(varianceX);
var stdDevY = Math.sqrt(varianceY);
var stdDevZ = Math.sqrt(varianceZ);
var stdDevMagnitude = Math.sqrt(varianceMagnitude);

var medianX = median(xValues);
var medianY = median(yValues);
var medianZ = median(zValues);
var medianMagnitude = median(magnitudeValues);

// Add statistics to the message payload
msg.payload = {
    mean: {
        x: meanX.toFixed(2),
        y: meanY.toFixed(2),
        z: meanZ.toFixed(2),
        magnitude: meanMagnitude.toFixed(2)
    },
    median: {
        x: medianX.toFixed(2),
        y: medianY.toFixed(2),
        z: medianZ.toFixed(2),
        magnitude: medianMagnitude.toFixed(2)
    },
    variance: {
        x: varianceX.toFixed(2),
```

```
        y: varianceY.toFixed(2),
        z: varianceZ.toFixed(2),
        magnitude: varianceMagnitude.toFixed(2)
    },
    stdDev: {
        x: stdDevX.toFixed(2),
        y: stdDevY.toFixed(2),
        z: stdDevZ.toFixed(2),
        magnitude: stdDevMagnitude.toFixed(2)
    }
};

return msg;
```

The results are then formatted into an HTML table using a template node to display the statistics.

B. Position Sensor Data

i. Reading and Parsing Data

To read the position sensor data, we start by using a `file read` node to read the `gps_data.csv` file. The data is then parsed using the following function node:

```
var csvData = msg.payload;
var lines = csvData.split('\n');
var selectedTeam = global.get('selectedTeam');
// Initialize an array to hold the processed data
var processedData = [];
// Iterate over each line of the CSV
for (var i = 1; i < lines.length; i++) {
    var line = lines[i].trim();
    if (line === '') continue; // Skip empty lines
    var values = line.split(',');
    var data = {};
    // Assuming the CSV format:
timestamp,timestampRegular,sensor_id,sensor_team,latitude,longitude
    data.timestamp = values[0].trim();
    data.timestampRegular = parseInt(values[1].trim(), 10); // Assuming
timestampRegular is an integer
    data.sensor_id = values[2].trim();
    data.sensor_team = values[3].trim();
    data.latitude = parseFloat(values[4].trim());
    data.longitude = parseFloat(values[5].trim());
```

```
    if (!isNaN(data.latitude) && !isNaN(data.longitude)) {
        if (selectedTeam !== "all" && selectedTeam != null && selectedTeam !=
undefined) {
            if (data.sensor_team === selectedTeam) {
                processedData.push(data);
            }
        } else {
            processedData.push(data);
        }
    }
}
// Set the processed data as the message payload
msg.payload = processedData;
return msg;
```

This function processes the CSV data, filters it based on the selected team, and prepares it for further use.

ii. Visualization

To prepare the data for GPS map visualization, the following function node is used:

```
// Iterate over each GPS data point
for (var i = 0; i < msg.payload.length; i++) {
    var data = {
        name:msg.payload[i].sensor_team +  " point " + (i + 1),     // Unique name
for each point
        lat: msg.payload[i].latitude, // Latitude of the point
        lon: msg.payload[i].longitude // Longitude of the point
        // Optionally, you can add more properties like icon and layer
    };

    // Emit each data point individually to the Worldmap node
    node.send({
        payload: data
    });
}
return null; // Return null to suppress the original message
```

This function formats the parsed GPS data points for display on a GPS map using the worldmap node.

iii. Distance Calculation

For distance calculation based on the position data, the following function node is used:

```
var gpsData = msg.payload;
var selectedTeam = global.get('selectedTeam');
var teamDistances = {};
var totalDistance = 0;
```

```
// Helper function to calculate distance using Haversine formula
function haversineDistance(lat1, lon1, lat2, lon2) {
    var R = 6371000; // Radius of the Earth in meters
    var phi1 = toRadians(lat1);
    var phi2 = toRadians(lat2);
    var deltaPhi = toRadians(lat2 - lat1);
    var deltaLambda = toRadians(lon2 - lon1);

    var a = Math.sin(deltaPhi / 2) * Math.sin(deltaPhi / 2) +
        Math.cos(phi1) * Math.cos(phi2) *
        Math.sin(deltaLambda / 2) * Math.sin(deltaLambda / 2);
    var c = 2 * Math.atan2(Math.sqrt(a), Math.sqrt(1 - a));

    return R * c; // Distance in meters
}

function toRadians(degrees) {
    return degrees * Math.PI / 180;
}
// Process distances based on the selected team
if (selectedTeam !== "all") {
    // Calculate distance for the selected team only
    for (var i = 1; i < gpsData.length; i++) {
        var prevPoint = gpsData[i - 1];
        var currPoint = gpsData[i];
        var distance = haversineDistance(prevPoint.latitude, prevPoint.longitude,
currPoint.latitude, currPoint.longitude);
        totalDistance += distance;
    }
    totalDistance = totalDistance / 1000; // Convert to kilometers
    msg.payload = totalDistance.toFixed(2) + " km"; // Format the payload as a
string with two decimal places
} else {
    // Group data by sensor_team and calculate distances
    gpsData.forEach(function(data) {
        if (!teamDistances[data.sensor_team]) {
            teamDistances[data.sensor_team] = 0;
        }
    });
    for (var team in teamDistances) {
        var teamData = gpsData.filter(point => point.sensor_team === team);
        var teamDistance = 0;
        for (var i = 1; i < teamData.length; i++) {
            var prevPoint = teamData[i - 1];
            var currPoint = teamData[i];
            var distance = haversineDistance(prevPoint.latitude,
```

```
prevPoint.longitude, currPoint.latitude, currPoint.longitude);
            teamDistance += distance;
        }
        teamDistances[team] = teamDistance / 1000; // Convert to kilometers
        totalDistance += teamDistances[team]; // Sum up the distance for all teams
    }
    msg.payload = totalDistance.toFixed(2) + " km"; // Format as a string with two
decimal places
}
return msg;
```

This function calculates the total distance traveled by each team or by a specific team using the Haversine formula and then formats the result for display.

iv. Displaying Distance on Text Node

The calculated distance is displayed using a text node, which shows the total distance traveled in kilometers. This provides a clear and concise way to present the distance information to the user.

C. Vital Data Sensor

i. Reading and Parsing Data

For the vital data sensor, we start by reading the vital_data.csv file using a file read node. The data is then parsed using the following function node:

```
var csvData = msg.payload;
var lines = csvData.split("\n");
var selectedTeam = global.get("selectedTeam");

// Initialize an array to hold the processed data
var processedData = [];

// Iterate over each line of the CSV, starting from the second line to skip the
header
for (var i = 1; i < lines.length; i++) {
  var line = lines[i].trim();
  if (line === "") continue; // Skip empty lines

  var values = line.split(",");
  var data = {};

  // Assuming the CSV format:
timestamp,timestampRegular,sensor_id,sensor_team,heart_rate,body_temperature
  data.timestamp = values[0].trim();
  data.timestampRegular = parseInt(values[1].trim(), 10); // Assuming
```

```
timestampRegular is an integer
  data.sensor_id = values[2].trim();
  data.sensor_team = values[3].trim();
  data.heart_rate = parseInt(values[4].trim(), 10);
  data.body_temperature = parseFloat(values[5].trim());

  // Check for valid heart_rate and body_temperature values
  if (!isNaN(data.heart_rate) && !isNaN(data.body_temperature)) {
    if ( selectedTeam !== "all" &&
      selectedTeam != null &&
      selectedTeam != undefined ) {
      if (data.sensor_team === selectedTeam) {
        processedData.push(data);
      }
    } else {
      processedData.push(data);
    }
  }
}
// Set the processed data as the message payload
msg.payload = processedData;
return msg;
```

This function processes the CSV data, filters it based on the selected team, and prepares it for further analysis and visualization.

ii. Visualization of Heart Rate Data

To visualize the heart rate data, we use the following function node to prepare the data for a line chart:

```
var processedData = msg.payload;
// Initialize arrays to hold the data for the heart rate chart
var heartRateData = [];
// Iterate over the processed data to extract timestamp and heart rate
for (var i = 0; i < processedData.length; i++) {
  var data = processedData[i];
  // Prepare data for heart rate chart
  heartRateData.push({
    x: new Date(data.timestamp),
    y: data.heart_rate,
  });
}
// Set the processed heart rate data as the message payload
msg.payload = [
  {
    series: ["Heart Rate"],
    data: [heartRateData],
```

```
    labels: [""],
  },
];
return msg;
```

This function formats the heart rate data into a suitable format for line chart visualization, plotting heart rate against timestamp.

iii. Visualization of Body Temperature Data

To visualize the body temperature data, the following function node is used to prepare it for a line chart:

```
var processedData = msg.payload;
// Initialize array to hold the data for the body temperature chart
var bodyTemperatureData = [];
// Iterate over the processed data to extract timestamp and body temperature
for (var i = 0; i < processedData.length; i++) {
  var data = processedData[i];
  // Prepare data for body temperature chart
  bodyTemperatureData.push({
    x: new Date(data.timestamp),
    y: data.body_temperature,
  });
}
// Set the processed body temperature data as the message payload
msg.payload = [
  {
    series: ["Body Temperature"],
    data: [bodyTemperatureData],
    labels: [""],
  },
];
return msg;
```

This function formats the body temperature data into a suitable format for line chart visualization, plotting body temperature against the timestamp.

iv. Statistical Analysis

For statistical analysis of the vital data, the following function node calculates key metrics such as average, maximum, and minimum heart rate and body temperature:

```
var processedData = msg.payload;
// Initialize arrays to hold heart rate and body temperature data
var heartRates = [];
var bodyTemperatures = [];
// Iterate over processed data to extract heart rates and body temperatures
```

```
for (var i = 0; i < processedData.length; i++) {
    var data = processedData[i];

    // Check if heart_rate is a valid number before collecting it
    if (!isNaN(data.heart_rate)) {
        heartRates.push(data.heart_rate);
    }

    // Check if body_temperature is a valid number before collecting it
    if (!isNaN(data.body_temperature)) {
        bodyTemperatures.push(data.body_temperature);
    }
}

// Calculate average heart rate if the array is not empty
var averageHeartRate = heartRates.length > 0 ?
    (heartRates.reduce((acc, val) => acc + val, 0) / heartRates.length).toFixed(2)
:
    "N/A";

// Calculate maximum and minimum heart rate if the array is not empty
var maxHeartRate = heartRates.length > 0 ? Math.max(...heartRates) : "N/A";
var minHeartRate = heartRates.length > 0 ? Math.min(...heartRates) : "N/A";

// Calculate average body temperature if the array is not empty
var averageBodyTemperature = bodyTemperatures.length > 0 ?
    (bodyTemperatures.reduce((acc, val) => acc + val, 0) /
bodyTemperatures.length).toFixed(2) :
    "N/A";

// Calculate maximum and minimum body temperature if the array is not empty
var maxBodyTemperature = bodyTemperatures.length > 0 ?
Math.max(...bodyTemperatures).toFixed(2) : "N/A";
var minBodyTemperature = bodyTemperatures.length > 0 ?
Math.min(...bodyTemperatures).toFixed(2) : "N/A";

// Prepare the payload with calculated statistics
msg.payload = {
    averageHeartRate: averageHeartRate,
    maxHeartRate: maxHeartRate,
    minHeartRate: minHeartRate,
    averageBodyTemperature: averageBodyTemperature,
    maxBodyTemperature: maxBodyTemperature,
    minBodyTemperature: minBodyTemperature
};
return msg;
```

This function calculates statistical metrics such as the average, maximum, and minimum values for heart rate and

body temperature, providing insights into the vital data.

v. Displaying Statistics on Template Node

The calculated statistics are then displayed using a template node. This node formats and presents the statistical data in a readable format.

D. Speed Data Sensor

i. Reading and Parsing Data

For the speed data sensor, we start by reading the speed_data.csv file using a file read node. The data is then parsed using the following function node:

```
var csvData = msg.payload;
var lines = csvData.split('\n');
var selectedTeam = global.get('selectedTeam');

// Initialize an array to hold the processed data
var processedData = [];

// Iterate over each line of the CSV, starting from the second line to skip the
header
for (var i = 1; i < lines.length; i++) {
   var line = lines[i].trim();
   if (line === '') continue; // Skip empty lines

   var values = line.split(',');
   var data = {};

   // Assuming the CSV format:
timestamp,timestampRegular,sensor_id,sensor_team,speed
   data.timestamp = values[0].trim();
   data.timestampRegular = parseInt(values[1].trim(), 10); // Assuming
timestampRegular is an integer
   data.sensor_id = values[2].trim();
   data.sensor_team = values[3].trim();
   data.speed = parseFloat(values[4].trim());

   // Validate the parsed values
   if (!isNaN(data.timestampRegular) && !isNaN(data.speed)) {
       if (selectedTeam !== "all" && selectedTeam != null && selectedTeam !=
undefined) {
           if (data.sensor_team === selectedTeam) {
               processedData.push(data);
           }
```

```
        } else {
            processedData.push(data);
        }
    }
}

// Set the processed data as the message payload
msg.payload = processedData;
return msg;
```

This function processes the CSV data and filters it based on the selected team. It prepares the data for further analysis and visualization.

ii. Visualization of Speed Data

To visualize the speed data, we use the following function node to prepare it for a line chart:

```
var processedData = msg.payload;
// Initialize an array to hold the data for the speed chart
var speedData = [];
// Iterate over the processed data to extract timestamp and speed
for (var i = 0; i < processedData.length; i++) {
    var data = processedData[i];
    // Prepare data for speed chart
    speedData.push({
        x: new Date(data.timestamp),
        y: data.speed
    });
}
// Set the processed speed data as the message payload
msg.payload = [{
    series: ["Speed"],
    data: [speedData],
    labels: [""]
}];
return msg;
```

This function formats the speed data into a suitable format for line chart visualization, plotting speed against the timestamp.

iii. Basic Speed Statistics

To calculate basic statistics such as average speed, maximum speed, minimum speed, and standard deviation, we use the following function node:

```
var speedData = msg.payload;
// Initialize variables for calculations
var sumSpeed = 0;
var maxSpeed = -Infinity;
var minSpeed = Infinity;
var speedValues = [];

// Iterate over speed data to collect statistics
for (var i = 0; i < speedData.length; i++) {
    var speed = speedData[i].speed;
    // Sum for average calculation
    sumSpeed += speed;
    // Check for max speed
    if (speed > maxSpeed) {
        maxSpeed = speed;
    }
    // Check for min speed
    if (speed < minSpeed) {
        minSpeed = speed;
    }
    // Collect speeds for standard deviation calculation
    speedValues.push(speed);
}

// Calculate average speed
var averageSpeed = sumSpeed / speedData.length;

// Calculate standard deviation
var mean = averageSpeed;
var sumSquaredDiffs = speedValues.reduce((acc, val) => acc + Math.pow(val - mean,
2), 0);
var standardDeviation = Math.sqrt(sumSquaredDiffs / speedValues.length);

// Prepare the payload with calculated statistics
msg.payload = {
    averageSpeed: averageSpeed.toFixed(2),
    maxSpeed: maxSpeed.toFixed(2),
    minSpeed: minSpeed.toFixed(2),
    standardDeviation: standardDeviation.toFixed(2)
};
return msg;
```

This function calculates the average speed, maximum speed, minimum speed, and standard deviation, which are then displayed on a template node.

iv. Average Speed by Hour

To analyze the average speed by hour, we use the following function node to aggregate and compute the

average speed for each hour:

```
var speedData = msg.payload;
var hourlySpeeds = {};

// Aggregate speeds by hour
for (var i = 0; i < speedData.length; i++) {
    var timestamp = new Date(speedData[i].timestamp);
    var hour = timestamp.getUTCHours();
    if (!hourlySpeeds[hour]) {
        hourlySpeeds[hour] = { sumSpeed: 0, count: 0 };
    }
    hourlySpeeds[hour].sumSpeed += speedData[i].speed;
    hourlySpeeds[hour].count += 1;
}

// Calculate average speed for each hour
var averageHourlySpeeds = [];
for (var hourStr in hourlySpeeds) {
    var hour = parseInt(hourStr); // Ensure hour is treated as a number
    var avgSpeed = hourlySpeeds[hour].sumSpeed / hourlySpeeds[hour].count;
    averageHourlySpeeds.push({ hour: hour, averageSpeed: avgSpeed.toFixed(2) });
}
// Set the average hourly speeds as the message payload
msg.payload = averageHourlySpeeds;
return msg;
```

This function aggregates the speed data by hour and calculates the average speed for each hour, which is then displayed on a template node.

v. Activity Recognition (Threshold-Based)

To classify activities based on speed, we use the following function node to categorize activities into walking, running, or sprinting:

```
// Function node for activity recognition (threshold-based)
var data = msg.payload;

// Initialize counters for each activity
var walkingCount = 0;
var runningCount = 0;
var sprintingCount = 0;

// Realistic thresholds based on typical speed ranges (m/s)
var walkingThreshold = 8;  // Speed < 8 m/s for walking
var runningThreshold = 16;  // Speed between 8 and 16 m/s for running

for (var i = 0; i < data.length; i++) {
```

```
    var speed = parseFloat(data[i].speed);

    // Classify activities based on thresholds
    if (speed < walkingThreshold) {
        data[i].activity = "Walking";
        walkingCount++;
    } else if (speed >= walkingThreshold && speed < runningThreshold) {
        data[i].activity = "Running";
        runningCount++;
    } else {
        data[i].activity = "Sprinting";
        sprintingCount++;
    }
}

// Calculate percentages
var totalCount = walkingCount + runningCount + sprintingCount;
var walkingPercentage = (totalCount > 0) ? (walkingCount / totalCount) * 100 : 0;
var runningPercentage = (totalCount > 0) ? (runningCount / totalCount) * 100 : 0;
var sprintingPercentage = (totalCount > 0) ? (sprintingCount / totalCount) * 100 :
0;

// Add the counts and percentages to the message payload
msg.payload = {
    data: data,
    counts: {
        walking: walkingCount,
        running: runningCount,
        sprinting: sprintingCount
    },
    percentages: {
        walking: walkingPercentage.toFixed(2),
        running: runningPercentage.toFixed(2),
        sprinting: sprintingPercentage.toFixed(2)
    }
};
return msg;
```

This function classifies activities based on speed thresholds and calculates the percentage of time spent walking, running, and sprinting. The results are prepared for visualization.

vi. Preparing Data for Pie Chart

Here is the code for extracting the percentage of time spent walking for visualization in a pie chart:

```
// Function node to prepare data for the pie chart
var percentages = msg.payload.percentages;
msg.payload = percentages.walking;
```

```
msg.topic = "Walking";
return msg;
```

The process for extracting the percentages of time spent running and sprinting is similar, with the only difference being the specific activity (running or sprinting) referenced in the code. These functions prepare the percentages of walking, running, and sprinting for visualization in a pie chart, providing a clear view of the distribution of activities.

Conclusion

The finished project uses all the described sensors and is displaying it on the dashboard as intended. By separating the values into different groups, it was very interesting to see the data change and by combining them even more information could be obtained. Through the project, we were able to gain a deep insight into the world in the development of IOT systems. We also noticed that the planning and working together on the system is very complex and you have to act carefully in order to achieve the desired result.