The IOT Based Jacket

A Senior Project Report

Presented in Partial Fulfillment of the Requirements for

the Degree Bachelor of Engineering in the

Faculty of Engineering of Notre Dame University

By:

Atieh Atieh 20197021
Elie El Arou 20197061

Notre Dame University
Fall 2023

Advised by:
Dr. Chady El-Moucary

Department of Electrical, Computer, and Communication Engineering

Committee Member

Dr. Abdallah Kassem

Dr. Walid Zakhem

Advised by:

Dr. Chady El-Moucary

Department of Electrical, Computer, and

Communication Engineering

Abstract

This Project details the design and implementation of a novel IoT-enabled jacket for continuous health monitoring emphasizing the integration of Arduino microcontrollers and specialized sensors. The hardware architecture captures vital health metrics, such as heart rate, oxygen levels, and body temperature, with communication facilitated through HTTPS and socket protocols. The jacket's hardware system is augmented with an Angular-based user interface, providing real-time visualization of health parameters and facilitating user interaction. Additionally, Node.js serves as the backend, orchestrating seamless connectivity between the wearable device, the frontend and the database.

This work contributes to the field of wearable healthcare technology by providing a comprehensive hardware-software solution, emphasizing the integration of Arduino, specialized sensors, and effective communication protocols. The presented IoT jacket serves as a prototype showcasing advancements in continuous health monitoring systems and their potential impact on user well-being.

Acknowledgements

First and foremost, we would like to express our deepest gratitude to our project supervisor, Dr. Chady El Moucary, whose expertise, understanding, and patience, added considerably to our project fulfillment. We truly appreciate his guidance, vast knowledge and skills in numerous areas and his assistance in every step of this project.

We would also like to thank Dr. Abdallah Kassem and Mr. Walid Zakhem whose continuous support and guidance throughout the project completion were pivotal for the project design and implementation.

Additionally, we owe a special thanks to all our instructors in the department of electrical, computer and communication engineering for their support all through our Bachelor of Engineering journey.

Finally, we wish to express our love and gratitude to our families whose support and encouragement allowed us to excel during our university journey and mainly for the accomplishment of this project.

Table of Contents

LIST OF FIGURES

LIST OF TABLES

1 Introduction

The elderly population around the world is facing an increasing number of health challenges, with falls and cardiovascular issues being significant contributors to morbidity and mortality among the elderly. In the United States, the impact of falls on the elderly is particularly alarming, as evidenced by the fact that approximately 300,000 elderly individuals are hospitalized annually due to broken hips, a prevalent consequence of falls [1]. Most notably, the financial burden associated with medical costs for broken hips in the U.S. amounts to a staggering six billion dollars per year [2].

The implications of falls extend beyond the immediate physical injuries, as studies have revealed a disturbingly high mortality rate among elderly patients treated for hip fractures. Shockingly, up to 33% of elderly individuals do not survive beyond one year after experiencing a hip fracture, underscoring the severity of the issue [3]. The first 12 months post-hip fracture are especially critical, with one in three adults aged 50 and over dying to the consequences of this injury. Moreover, older adults face a five-to-eight times higher risk of mortality within the first three months of a hip fracture compared to their counterparts without such injuries, and this elevated risk persists for nearly a decade [4].

In addition to the challenges posed by falls, cardiovascular issues pose a significant threat to the well-being of the elderly population. Individuals aged 65 and older are at a heightened risk of experiencing heart attacks, strokes, coronary heart disease, and heart failure [2]. The magnitude of this issue is further emphasized by the fact that, in 2002, heart disease emerged as the leading cause of death among the elderly in the United States, accounting for a staggering 31.8% of mortality [1].

Recognizing the pressing need to address these health concerns among the elderly, our project focuses on the development of an innovative jacket equipped with advanced monitoring capabilities. By integrating features such as heart rate monitoring, fall detection with airbag deployment, blood oxygen level measurement, and temperature monitoring, our project aims to provide a comprehensive solution to enhance the health and well-being of the elderly population. This paper delves into the details of the project, outlining its objectives, design, and potential impact on mitigating the health risks faced by the elderly.

2 Standards, Constraints, and Requirements Specification

In the development of the IOT Based Jacket, a series of design constraints have been carefully considered to ensure the feasibility, ethical soundness, and environmental responsibility of the product. These constraints encompass several key aspects, including economic viability, environmental impact, ethical considerations, user safety, social acceptability, and sustainability.

From an economic perspective, our primary aim is to create an affordable solution suitable for elderly individuals and their families. To achieve this, we have utilized readily available components such as Arduino and sensors, which effectively reduce the cost of production, making the product financially accessible to the target demographic.

Environmental responsibility is a fundamental concern, and we have addressed this by using power banks as the primary energy source for the jacket, significantly reducing the need for disposable batteries. Furthermore, the design of the jacket places a strong emphasis on energy efficiency and low power consumption, ensuring that it minimizes its environmental footprint.

Ethical considerations are at the core of this project. We are committed to delivering a solution that not only enhances the quality of life for elderly individuals but also aligns with ethical principles. This involves ensuring their well-being, safety, and comfort throughout the use of the All-in-One Jacket.

Safety for the user is paramount in our design. The jacket is equipped with features that mitigate the risk of injury from falls, such as an airbag deployment mechanism that activates in case of a fall or sudden impact.

Social acceptability plays a crucial role in the adoption of the product. The design of the jacket is not only functional but also stylish and comfortable, suitable for everyday wear. We've taken into account the specific needs of elderly individuals, such as the provision of warmth and easy mobility, to ensure it is socially acceptable.

Sustainability is another significant constraint in our project. To facilitate ease of manufacturing and replication, we have designed the system using readily available components, thereby contributing to the sustainability of the product's production and distribution.

In addition to the design constraints, the project requires a comprehensive set of specifications to guide its development. The central objective of this system is to ensure the well-being and safety of elderly individuals by continuously monitoring their vital signs and environmental conditions. Simultaneously, it aims to provide comfort and peace of mind to the elderly wearer and their caregivers through an array of advanced functionalities.

The IOT Based Jacket is a multidisciplinary effort that combines knowledge and expertise from various fields to create a useful and practical solution. The project involves elements of electrical engineering, software development and mechanical engineering.

In terms of electrical engineering, the project incorporates the use of sensors such as the fall sensor, temperature sensor, and heart rate and oxygen level sensors, as well as resistors to provide warmth. The project also involves the use of an Arduino microcontroller to process sensor data and control the various functions of the jacket.

In software development, the project requires the development of a program to interpret and display sensor data in a meaningful way. This includes programming the Arduino to trigger an emergency call when necessary.

Mechanical engineering comes into play in the design of the jacket itself, which must be comfortable, functional, and aesthetically pleasing. The design must also consider factors such as durability, mobility, and ease of use for the elderly user.

To achieve our goal of developing an effective healthcare solution for the elderly population, it is important to outline a set of rigorous requirements that will guide the design and implementation of the proposed system. The primary objective of this system is to ensure the well-being and safety of elderly individuals by monitoring their vital signs and environmental conditions. Additionally, the system aims to provide comfort and peace of mind to the elderly wearer and their caregivers through a range of advanced functionalities.

1. Room Temperature Tracking: The system shall also monitor and record ambient room temperature, allowing to maintain a suitable environment for the elderly.
2. Heating function: To ensure the warmth of the elderly.
3. Health Parameters Tracking: The system will continuously monitor and report essential health parameters, including:
    a. Heart Rate.
    b. Blood Oxygen Level.
    c. ECG (Electrocardiogram).

   d. Body temperature.

4. Fall Mitigation: To reduce the risk of injury from falls, the system shall be equipped with an airbag deployment mechanism that activates in case of a fall or sudden impact.
5. User-Friendly App: providing a dedicated website and mobile application for caregivers to be able to access all the important data, and act when needed.

These specifications constitute the fundamental requirements for the product. By adhering to these requirements, the system will enable caregivers to provide optimal care for elderly individuals, ensuring their safety, comfort, and peace of mind.

3 Functional Decomposition

At the foundational level (Level 0) of the project system, data is collected from an array of sensors. This data is subsequently channeled to the system's core processing unit for comprehensive analysis and processing. Once this processing is complete, the system generates user-friendly outputs in the form of both a website and a mobile application, ensuring real-time updates and results are readily available.

At a more profound level, the system is composed of an array of sensors integrated with two Arduino/ESP, which will be the main responsible for processing and output generation, in its different forms. Staring with the sensors used:

1. AD8232: Responsible for monitoring ECG and heart rate.
2. MAX30100: Dedicated for the blood oxygen level.
3. MPU6050: Enabling fall detection and ambient room temperature tracking.
4. DS18B20: Accurate measurement of the patient's body temperature.

The Arduino/ESP processors play a central role in processing the collected data. The resulting output is available in multiple forms, enhancing patient care:

1. Website and mobile application.
2. Deploying airbag.
3. Turning on/off the heated sheets.

This integrated system empowers caregivers with a comprehensive monitoring mechanism, providing data-driven insights and enhancing patient safety and comfort.

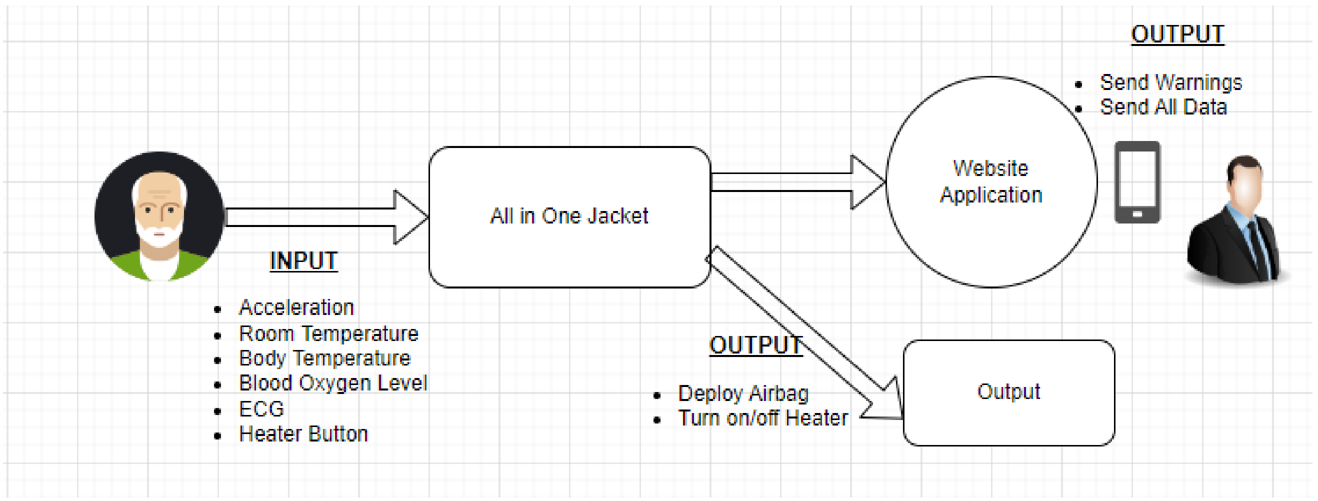


Figure 3-1: functional decomposition level 0

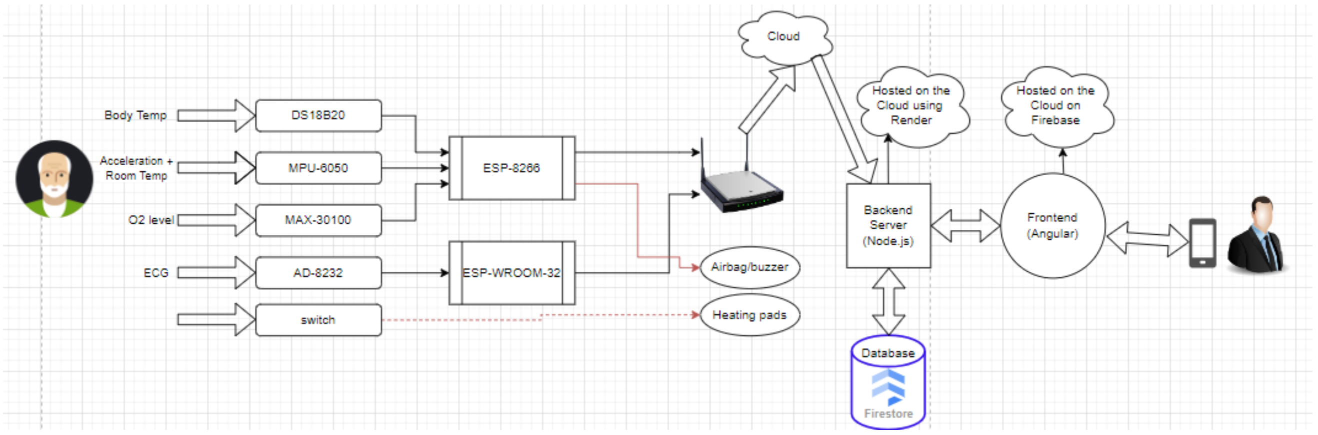


Figure 3-2: functional decomposition level 1

4 Project Management (Gantt chart, WBS, Cost Analysis)

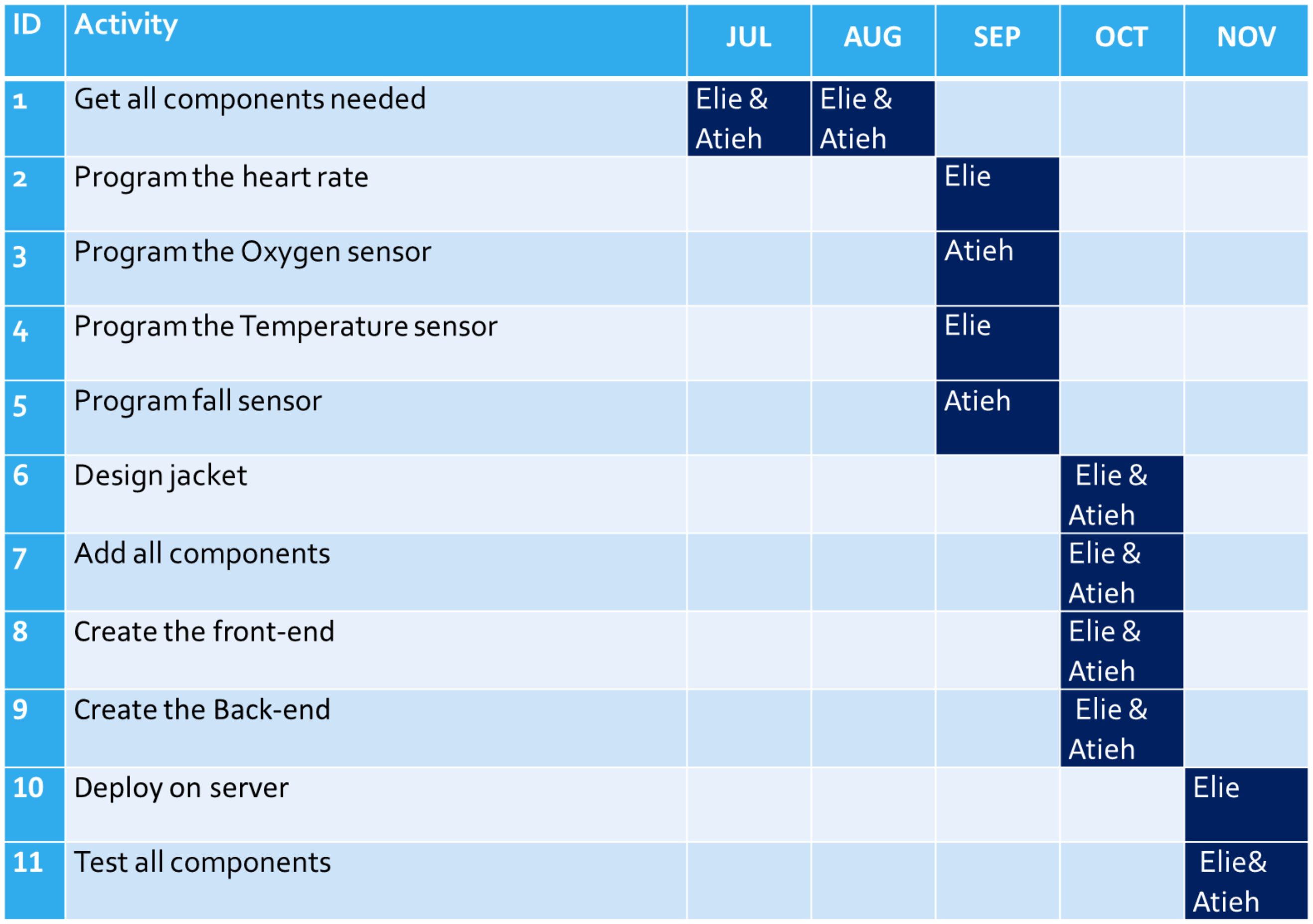
In the first phase of our project management plan, spanning July and August, our primary objective is to "Get and Verify." This involves examining the project's requirements and specifications to ensure clarity and alignment with our goals. Simultaneously, we will begin gathering the necessary materials and resources required for the project's successful execution.

As we transition into September, the project enters a critical phase titled "Start with All Sensors." During this month, we will commence the integration of various sensors into our project. Elie El Arou will take charge of the heart rate and temperature sensors, ensuring their seamless incorporation, while Atieh Atieh will be responsible for overseeing the integration of the oxygen and fall sensors, contributing to the project's overall sensor functionality.

Moving into October, the project diverges into two parallel tracks. First, we embark on the "Design of the Jacket." This phase is important to the project, as it lays the foundation for the physical component of our solution. Simultaneously, we will initiate the development of the web application, essential for data processing and user interaction. These concurrent efforts will be essential in ensuring a well-rounded and functional project.

In the concluding month of our project, November, our focus will be on "Finalizing the Design and App and Writing Reports." The design of the jacket will be completed, with any necessary refinements made to ensure it aligns with our project goals. Simultaneously, the web application development will be finalized, ensuring its readiness for deployment. Furthermore, this month is dedicated to the crucial task of documenting the entire project, with the team focused on writing comprehensive reports to summarize the project's journey, results, and outcomes. This organized project management plan encompasses a structured timeline and clear task delegation, allowing for effective coordination and successful project delivery.

In the following we can see the Gantt chart and the detailed project cost.



ID	Activity	JUL	AUG	SEP	OCT	NOV
1	Get all components needed	Elie & Atieh	Elie & Atieh			
2	Program the heart rate			Elie		
3	Program the Oxygen sensor			Atieh		
4	Program the Temperature sensor			Elie		
5	Program fall sensor			Atieh		
6	Design jacket				Elie & Atieh	
7	Add all components				Elie & Atieh	
8	Create the front-end				Elie & Atieh	
9	Create the Back-end				Elie & Atieh	
10	Deploy on server					Elie
11	Test all components					Elie& Atieh

Figure 4-1: Gantt chart

Table 4-1: Cost estimation

Components:	Cost in $
ESP-8266	3.5
ESP-WROOM-32	7.77
AD-8232 + electrodes	11 + 6
MAX-30100	7
MPU-6050	4
DS18B20	2
Heating pads	6 * 2
Jacket	15
Power bank	20
Additional material (breadboard, wires etc..)	20
Total cost	108.27

5 System Failure Modes and Effects Analysis (FMEA)

The project is a smart jacket designed for elderly people, containing a heater, a heart rate device, a pressure device, and an airbag. The jacket is intended to keep the wearer warm and safe, while also monitoring their health and providing an emergency notification system in case of any hazard.

Possible failure modes that could occur within the system:

- Heater malfunctions, causing burns or fire.
- Heart rate device malfunctions, providing inaccurate or false readings.
- Airbag malfunctions, deploying at the wrong time or failing to deploy when needed.
- Software glitch, causing the emergency notification system to malfunction or fail.

The severity of each failure mode: (all scales from 1-10)

- Heater malfunction: High severity: 9 (could cause burns or fire)
- Heart rate device malfunction: Moderate severity: 6 (could cause inaccurate readings)
- Oxygen device malfunction: Low severity: 3 (could cause discomfort, but not serious injury)
- Airbag malfunction: High severity 10 (could cause injury or death)
- Software glitch: Moderate severity 7.5 (could delay or prevent emergency notification)

Probability of each failure mode occurring:

- Heater malfunction: Moderate occurrence: 5 (due to quality control measures)
- Heart rate device malfunction: Low occurrence: 1 (due to quality control

measures)

- Oxygen device malfunction: Low occurrence 1 (due to quality control measures)
- Airbag malfunction: moderate occurrence 6.5 (due to quality control measures)
- Software glitch: Moderate occurrence 5.5 (due to the complexity of software development)

Detection ratings:

- Heater malfunction: High detection: 8 (due to temperature sensors and safety switches)
- Heart rate device malfunction: High detection: 8 (due to calibration and testing)
- Oxygen device malfunction: High detection 8 (due to calibration and testing)
- Airbag malfunction: low detection: 8
- Software glitch: Moderate detection: 6 (due to the complexity of software testing)

Risk Priority Number Calculation: RPN = Severity x Occurrence x Detection

- Heater malfunction: RPN = 9 (severity) x 5 (occurrence) x 8 (detection) = 360
- Heart rate device malfunction: RPN = 6 (severity) x 1 (occurrence) x 8 (detection) = 48
- Oxygen device malfunction: RPN = 3 (severity) x 1 (occurrence) x 8 (detection) = 24
- Airbag malfunction: RPN = 10 (severity) x 6.5 (occurrence) x 8 (detection) = 520
- Software glitch: RPN = 7.5 (severity) x 5.5 (occurrence) x 6 (detection) = 247.5

Based on the RPNs calculated, the Airbag malfunction has the highest RPN score and is therefore the highest priority for mitigation actions. So, for that reason, we decided to do progressive testing to ensure it detects the elderly fall, and for the long term we have plans to implement a machine learning approach to detect falls for everyone not only old people.

For the second highest failure mode, which is the heater malfunction, so we decided to improve quality control measures and add more safety features such as automatic shut-off switches.

As for the Software glitch we decided to increase the frequency and rigor of software testing and add redundant systems for emergency notification. Plus, to do a beta version and give it to our friends to test it and simulate it.

6 Hardware Component

In this chapter, all the hardware components used will be listed. As seen in the previous chapter (Functional Decomposition), we will be using the following components:

1. ESP-8266: Responsible for most of the processing in the project.
2. ESP-WROOM-32: responsible for the ECG readings.
3. AD-8232: responsible for monitoring ECG and heart rate.
4. MAX-30100: dedicated for the blood oxygen level.
5. MPU-6050: enabling fall detection and ambient room temperature tracking.
6. DS18B20: accurate measurement of the patient's body temperature.
7. Heating pads: responsible for the heating function.

6.1 ESP-8266

The ESP8266 is a popular component in the world of IoT (Internet of Things). It's a low-cost Wi-Fi module that enables microcontrollers and sensors to connect to a Wi-Fi network, facilitating communication with other devices and the internet. The ESP8266 has a built-in microcontroller that can be programed using Arduino IDE.

This is a list of the ESP8266 specifications [5]:

- Price: 3.5$
- Microcontroller: Tensilica 32-bit RISC CPU Xtensa LX106
- Operating Voltage: 3.3V
- Digital I/O Pins (DIO): 16
- Analog Input Pins (ADC): 1
- UARTs: 1
- SPIs: 1
- I2Cs: 1
- Flash Memory: 4 MB
- SRAM: 64 KB
- Clock Speed: 80 MHz
- PCB Antenna

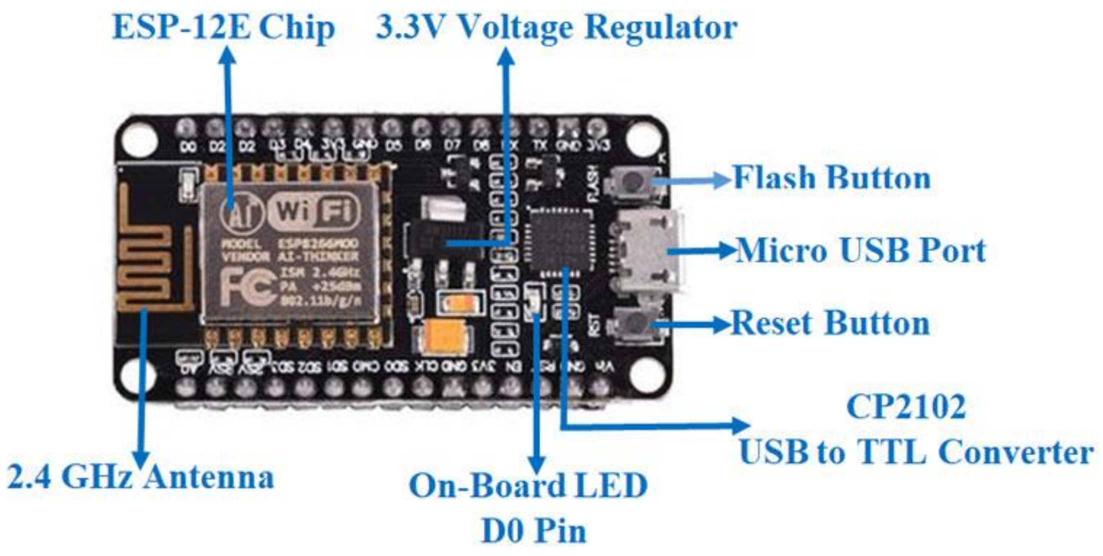


Figure 6-1: Detailed circuit of ESP-8266

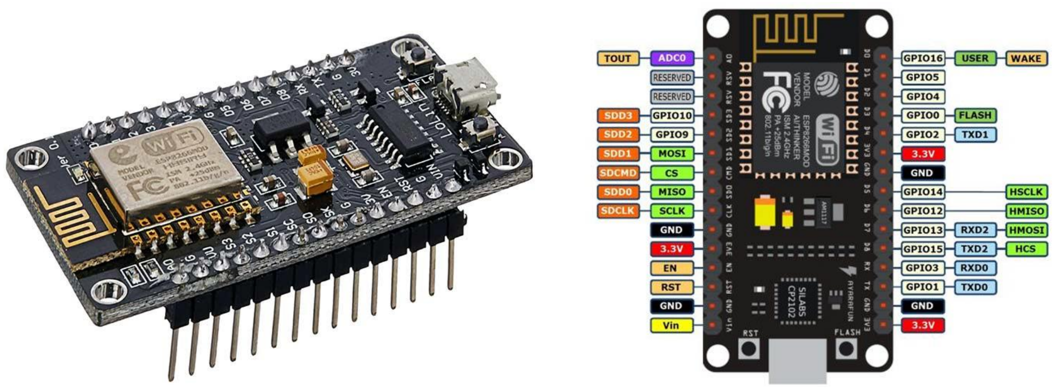


Figure 6-2:ESP-8266 and the connection pins

6.2 ESP-WROOM-32:

The second microcontroller we used is the ESP-WROOM-32. It is another popular component in the world of IoT. It has a similar low-cost Wi-Fi model.

This is a list of the ESP-WROOM-32 specifications [6]:

- 7.77$
- Microprocessor: Tensilica Xtensa LX6
- Maximum Operating Frequency: 240MHz
- Operating Voltage: 3.3V
- Analog Input Pins: 12-bit, 18 Channel
- DAC Pins: 8-bit, 2 Channel
- Digital I/O Pins: 39 (of which 34 is normal GPIO pin)
- DC Current on I/O Pins: 40 mA
- DC Current on 3.3V Pin: 50 mA
- SRAM: 520 KB
- Communication: SPI(4), I2C(2), I2S(2), CAN, UART(3)
- Wi-Fi: 802.11 b/g/n
- Bluetooth: V4.2 – Supports BLE and Classic Bluetooth

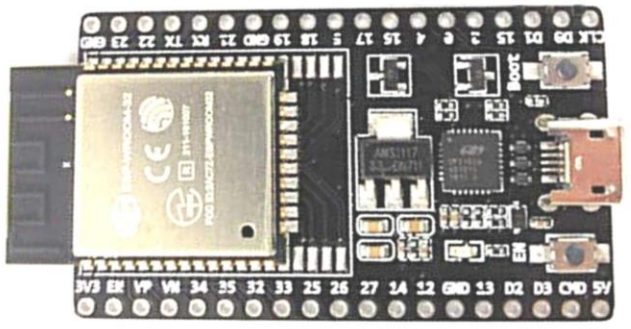

GND 23 22 TX RX 21 GND 19 18 5 17 16 4 0 2 15 D1 D0 CLK
Boot
3V3 EN VP VN 34 35 32 33 25 26 27 14 12 GND 13 D2 D3 CMD 5V

Figure 6-3:ESP-Wroom-32

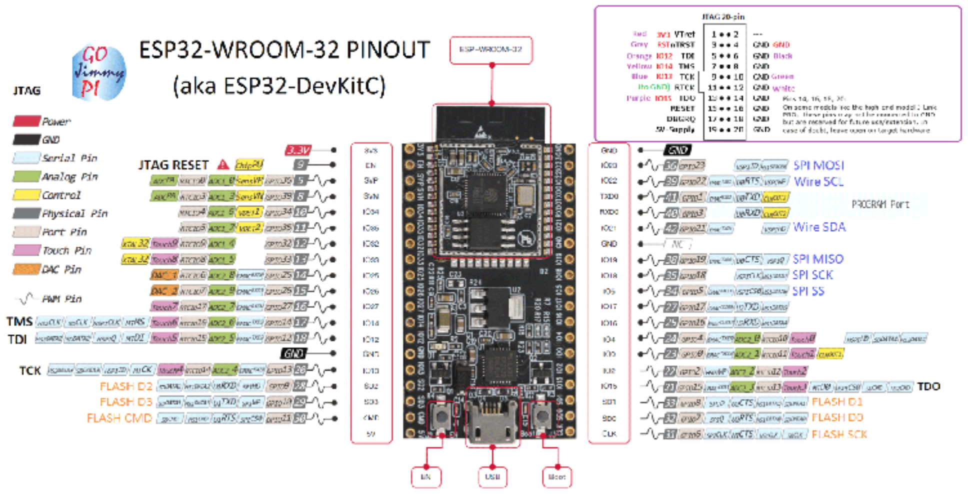


Figure 6-4: ESP-Wroom-32 connection pins

6.2.1 ECG sensor AD-8232:

Monitoring the patient's heart activity was one of our primary objectives. To achieve this, we needed to examine the patient's Electrocardiography (ECG). This was accomplished using a lead system, which consists of a set of electrodes attached to the patient's body for the purpose of recording the heart's electrical activity. The standard lead system used for this purpose is the 12-lead ECG, which can capture most of the heart's electrical activity and identify abnormalities in various regions of the heart, including the anterior, lateral, and inferior walls. This offers a more comprehensive assessment of the patient's cardiac condition [7].

In our project, we were constrained to using a single-lead ECG, as it was the only available option. It's important to note that single-lead ECG has a lower sensitivity in detecting ST-segment elevation, approximately 44%, as compared to the 79% sensitivity achieved with the 12-lead ECG [7]. The two main electrodes of the sensor will be connected to the chest of the patient or to the arms of him, and a ground electrode to the right leg according to the following photo [10]:

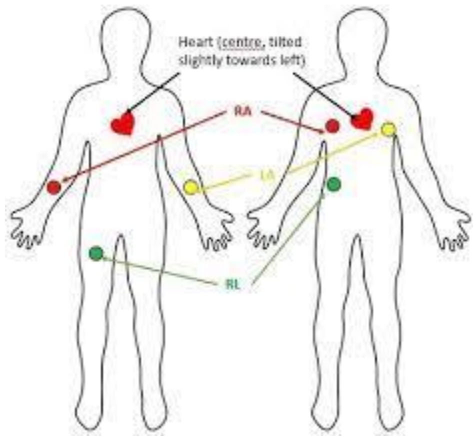


Figure 6-5: ECG connection to body

This connection is made to create Einthoven's triangle, positioning the heart in the center of the triangle, which enables us to measure the electrical voltage between the electrodes and obtain the ECG signal [11].

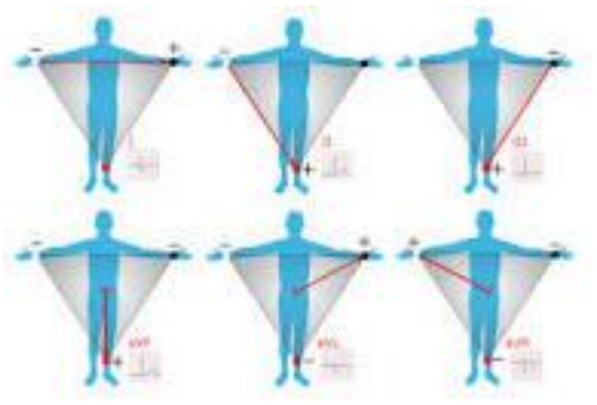


Figure 6-6: Einthoven's triangle

In our project, we utilized the AD8232 sensor, which serves as a signal conditioning block for ECG measurements in analog format. The sensor has 9 pins, including SDN, LO+, LO-, OUTPUT, 3.3V, GND, right arm (RA), left arm (LA), and right leg (RL). The first four pins are used for connecting to the Arduino to retrieve data, while the last three pins (RA, LA, and RL) are used to interface with the electrodes placed on the human body [8-9].

The sensor has the following specification [8]:

- Price: 11$
- Fully integrated single-lead ECG front end
- Supply current: 170 µA
- Common-mode rejection ratio: 80 dB (dc to 60 Hz)
- Two or three electrode configurations
- High signal gain (G = 100) with dc blocking capabilities
- 2-pole adjustable high-pass filter
- Accepts up to ±300 mV of half cell potential
- Fast restore feature improves filter settling
- Uncommitted op amp
- 3-pole adjustable low-pass filter with adjustable gain
- Leads off detection: ac or dc options
- Integrated right leg drive (RLD) amplifier
- Single-supply operation: 2.0 V to 3.5 V
- Integrated reference buffer generates virtual ground
- Rail-to-rail output
- Internal RFI filter
- 8 kV HBM ESD rating
- Shutdown pin
- 20-lead 4 mm × 4 mm LFCSP package

For optimal performance with this sensor, it is recommended to position the electrodes on the chest, as shown in the provided picture, as this helps reduce noise in the ECG signal [10].

To ensure proper connectivity, we followed the circuit configuration as depicted in the provided picture:

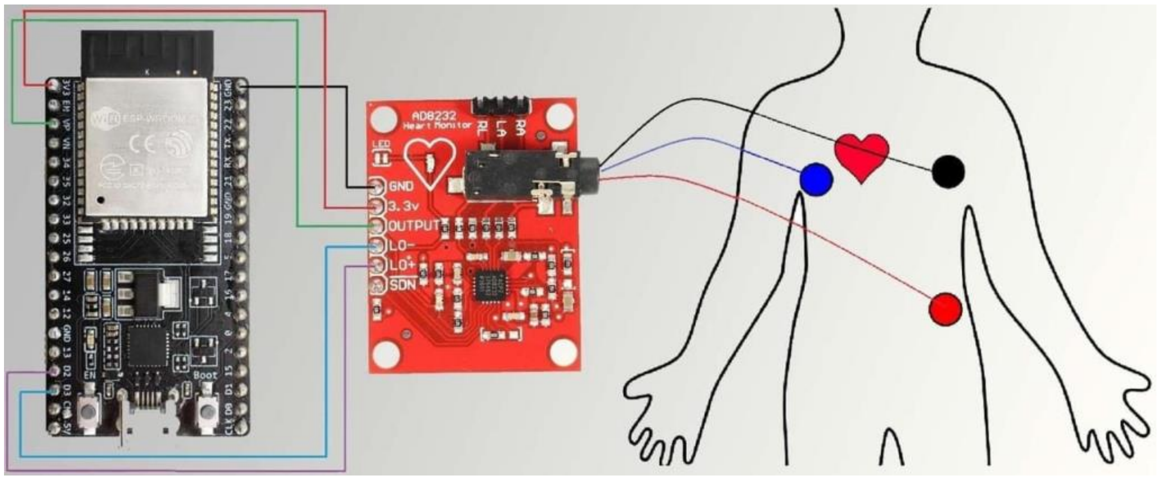


Figure 6-7: AD-8262 circuit.

We developed a program to extract data, generate the ECG waveform, and compute the heart rate based on the R-to-R interval:

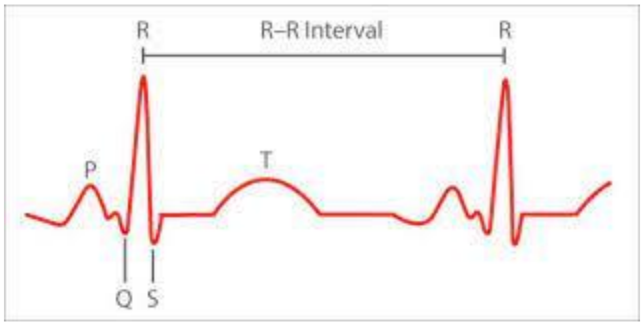


Figure 6-8: R-to-R interval representation

This process is achieved through the application of the following formula:

Heart rate (bpm) = 60000 / R-to-R time (in milliseconds).

The R-to-R time represents the duration between two successive heartbeats. Utilizing this formula allows us to obtain the patient's instantaneous heart rate.

After conducting several tests, we have pinpointed the factors contributing to margin errors in our observations. These include the placement of the electrodes, particularly when positioned on hairy skin, as it can introduce interference. Furthermore, the connection of a charger to the power supply can disrupt the functionality of the AD8232 sensor, causing it to fail to obtain data accurately. Lastly, the movement of muscles generates electrical charges that may impact the quality of the data collected. By addressing these issues, we aim to enhance the precision and reliability of our measurements and minimize margin errors.

6.3 Oxygen blood level sensor MAX-30100:

One critical parameter that requires monitoring is the blood oxygen level. The blood oxygen level is indispensable for assessing the overall health of an individual. Red blood cells play a pivotal role in transporting oxygen from the respiratory system to various cells throughout the body, including the brain and muscles [13]. Monitoring blood oxygen levels is particularly crucial for the well-being of the elderly, as a low oxygen level often signifies Hypoxemia, indicating a critical issue in the respiratory system.

Moreover, monitoring oxygen levels becomes important in various cases, especially for individuals with underlying heart or lung conditions such as congestive heart failure, COPD, or asthma. Additionally, those afflicted with contagious illnesses like influenza, pneumonia, and COVID-19 face an elevated risk of hypoxemia [14].

For this purpose, we will utilize the MAX-30100 to monitor blood oxygen levels. The MAX-30100 serves as an integrated pulse oximeter and heart rate monitor sensor. The sensor features seven pins: VCC and GND for power supply, SCL and SDA for data transmission through the I2C serial communication protocol, and additional pins INT, IRD, and RD, which will not be utilized in this application (www.analog.com). The measurement process involves the use of two light-emitting diodes and one photodiode [15].

The sensor has the following specification [16]:

- Price: 7$
- Power supply: 3.3V to 5.5V
- Current draw: ~600µA (during measurements)
- Current draw: ~0.7µA (during standby mode)
- Red LED Wavelength: 660nm
- IR LED Wavelength: 880nm
- Temperature Range: -40˚C to +85˚C
- Interface: I^{2}C

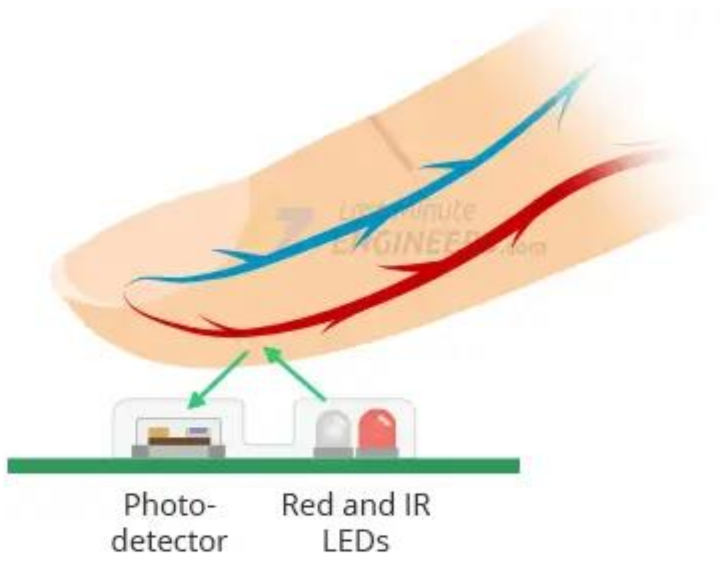


Figure 6-9: MAX-30100 working

The sensor employs a method called Photoplethysmogram to read data. In this process, the photodiode gauges the amount of light reflected from blood vessels after being emitted by the two LEDs [16]. The circuit diagram of the sensor connected to the Arduino is illustrated in the following picture.

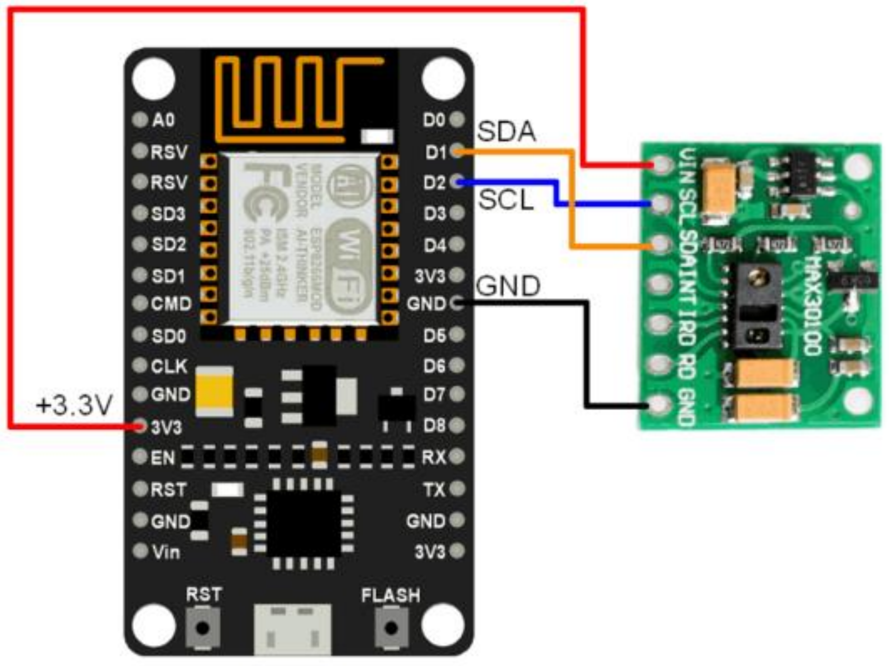


Figure 6-10: MAX-30100 circuit

Put example of the difference between an accurate reading and a reading with errors discussed above.

6.4 Fall detection using accelerometer MPU-6050:

In addressing the critical issue of fall detection among the elderly, our project introduces an innovative approach by incorporating the MPU6050 sensor. The paramount objective is to deploy an airbag preemptively, ensuring the patient is shielded from impact with the ground. This preventative measure is particularly focused on safeguarding vulnerable areas, such as the hips. The MPU-6050 sensor, equipped with 8 pins, plays a pivotal role in this system. Vcc and GNG pins handle power input, while SCL and SDA facilitate data reading through the I2C protocol. The remaining pins, namely XDA, XCL, AD0, and INT, are not utilized in our project.

Figure 6-11:MPU-6050

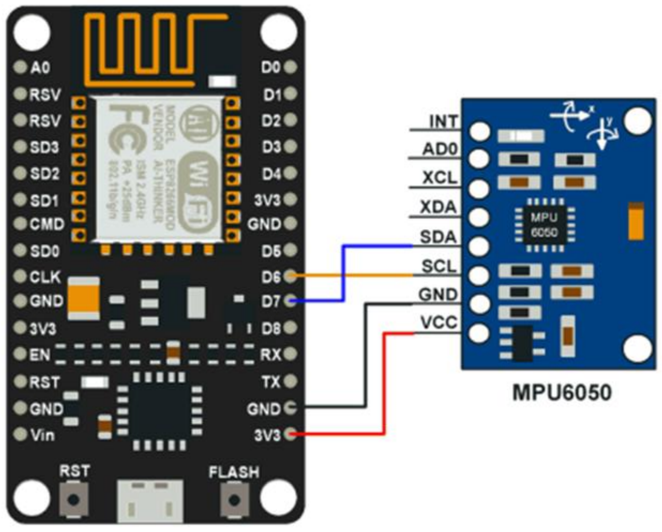


Figure 6-12: MPU-6050 circuit

For fall detection, our emphasis lies on the accelerometer function of the MPU6050. Positioned at the front of the patient's neck, the sensor calculates the total acceleration using the formula ACC = sqrt(ACC_X^2 + ACC_Y^2 + ACC_Z^2) – 9.8. Here, ACC_X, ACC_Y, and ACC_Z represent the accelerations along the respective axes. The subtraction of 9.8, the value of gravitational force, is crucial to isolate the specific acceleration of the body. By monitoring these acceleration levels, our system can accurately and swiftly detect a fall, triggering the deployment of the protective airbag to mitigate potential injuries, especially to critical areas like the hips. This technological intervention represents a significant advancement in addressing the safety concerns of the elderly, enhancing their overall well-being.

In addition, the MPU6050 is equipped with a temperature sensor. This sensor will be used to monitor the ambient temperature to ensure the elderly is feeling comfortable.

The sensor has the following specifications [17]:

- Price: 4$
- Voltage supply: 3.3 - 5 V DC
- Logic level: 3.3 V
- Degree of freedom: 6 x
- Interface: I²C
- Bult-in chip: 16-bit AD converter
- Pins: 8 x
- Accelerator measurement range: ± 2, ± 4, ± 8, ± 16 g
- Accelerator sensitivity: 16384 LSB/g ±2g, 8192 LSB/g ± 4g, 4096 LSB/g ± 4g, 2048LSB/g ±16g
- Gyroscope measurement range: ± 250. ± 500,± 1000, ± 2000 °/s
- Gyroscope sensitivity: 131 LSB/dps ± 250 dps, 65,5 LBS/dps ± 500 dps, 32,8 LBS/dps ± 1000 dps, 16,4 LBS/dps ± 2000 dps
- Dimension: 25 x 20 x 7 mm

6.5 Temperature using DS18B20:

Monitoring body temperature is a crucial aspect of elderly health management. To achieve this, we have opted to utilize the DS18B20 sensor. Housed in a metallic casing, the sensor's design is highly user-friendly. It is affixed to the jacket and positioned under the arm for optimal temperature monitoring. The sensor is equipped with three pins: VCC and GND for power supply, and DATA, which connects to the Arduino for temperature readings from the body.

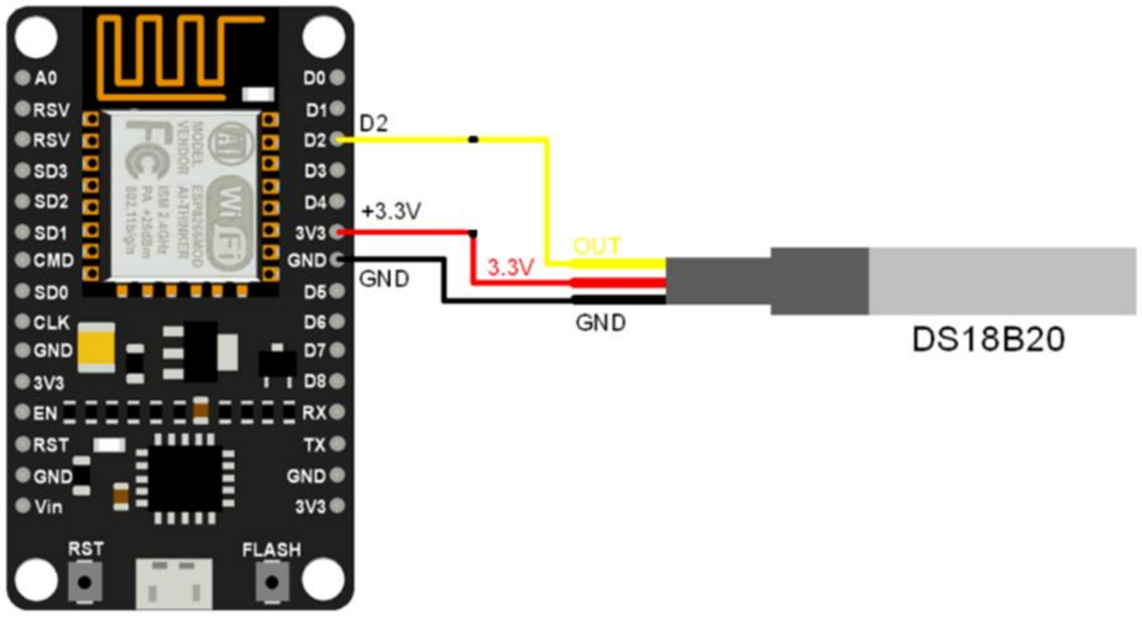


Figure 6-13: DS18B20 circuit

The sensor's specifications include [18]:

- Price: 2$
- 1-Wire interface requires only one port pin for communication
- Requires no external components
- Power supply range is 3.0V to 5.5V
- Zero standby power required
- Measures temperatures from -55°C to +125°C(-67°F to +257°F)
- ±0.5°C accuracy from -10°C to +85°C
- Thermometer resolution is programmable from 9 to 12 bits

6.6 Heating Pad:

Figure 6-14: Heating Pad

For the heating pads, we choose the 412 Heating Pad 12V 50x50. This component functions as a simple resistor, with higher voltage there is a higher high. This enable us to reach comfortable temperatures for the body. Will explain the full connection of the heated pads in later parts.

The component specifications include [19]:

- Price: 6$
- Material: Silicon Rubber
- Dimensions: 5x5cm
- Voltage input: 12VDC
- Power consumption: 10W

7 Full Circuit Implementation

This is the full circuit connected:

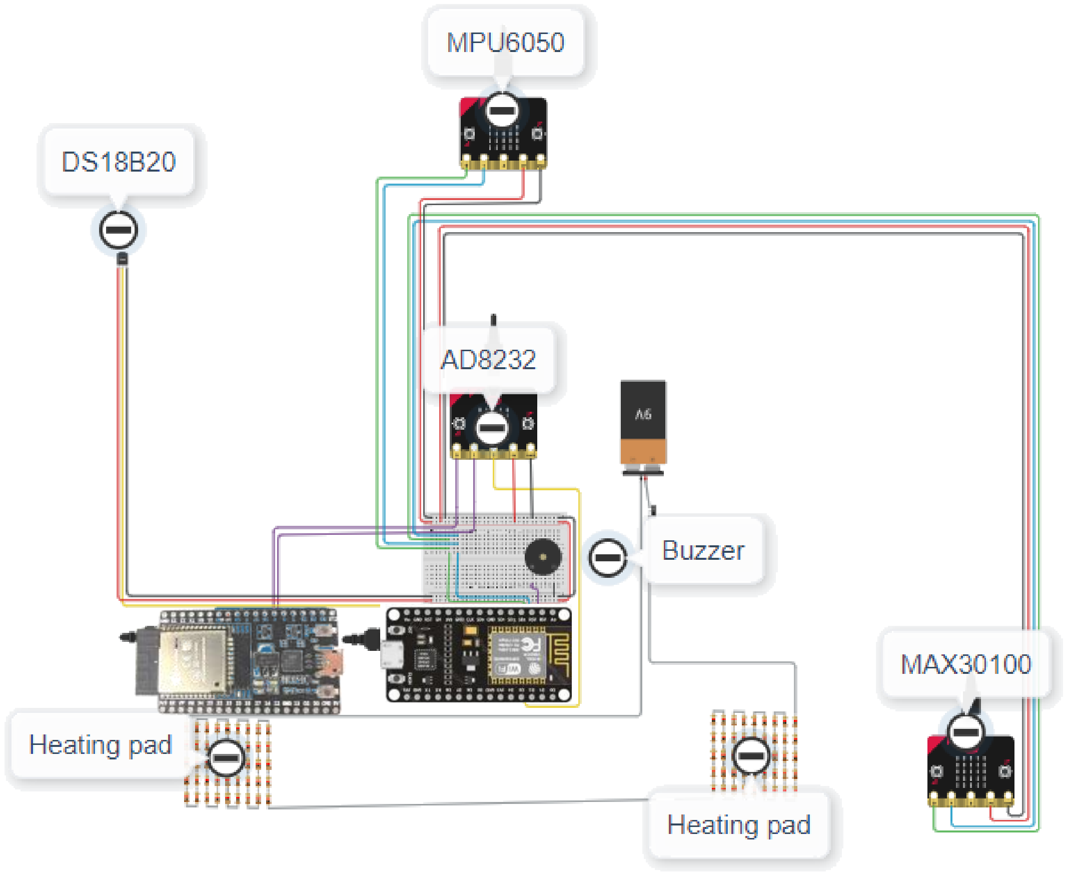


Figure 7-1: Full circuit

We start with the full circuit implementation. As we can see, the sensors are connected to the ESPs. The MAX-30100 and MPU-6050 are both connected to the ESP-8266 using SCL and SDA pins which is the I2C protocol. Each sensor has a different I2C protocol address, which makes the connection feasible. The AD-8232 is connected using two digital pins for L+ and L- and one analog pin for data to the ESP-WROOM-32. The AD18B20 is connected to one digital pin. Additionally, the buzzer, which represents the airbag, is connected to a digital pin used as output. The heating pads, represented as a chunk of resistors, are connected to a power supply and a switch to turn them on and off.

8 Full Application Implementation

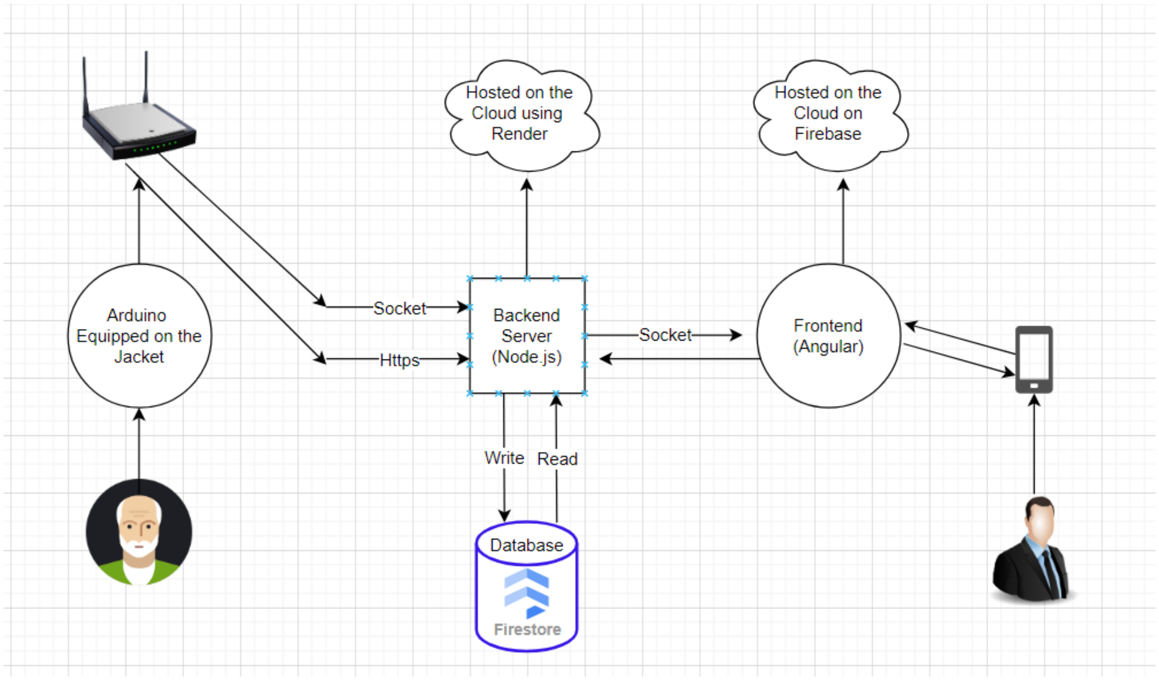

Hosted on the Cloud using Render
Hosted on the Cloud on Firebase
Socket
Https
Backend Server (Node.js)
Socket
Frontend (Angular)
Arduino Equipped on the Jacket
Write
Read
Database
Firestore

Figure 8-1: Full Application flow chart

Firstly, let's overview the system architecture. The Arduino embedded in the jacket serves as the primary data monitor. It establishes a continuous connection to the internet via Wi-Fi, facilitated by the router. This connectivity allows interaction with the cloud-based backend. Two communication protocols are employed by the Arduino: Socket communication and HTTPS REST API. The rationale behind this dual approach is determined by the nature of the data being transmitted. For example, for data like temperature, which is less time-sensitive and doesn't demand real-time updates, a periodic POST request made every 30 seconds suffices. This method ensures efficient use of resources. However, for critical data such as heart rate and oxygen levels, the need for real-time updates is necessary. To fulfill this requirement, a live communication channel using sockets is employed. Socket communication allows for instantaneous transmission of sensitive data, ensuring timely and accurate updates on the user's end.

Secondly and Upon reaching the Node.js server in the backend, data undergo preprocessing for storage in the database because using our Node.js server make it easier to manipulate data and interface with the database, in addition the socket.io [1] from the node part is way better and more efficient than the one in Arduino because we can send over it many and distinct channels and we can define separate events for different types of data such as "temperature", "falls", “Oxygen” … Each of these events can carry specific data payloads related to the respective category. This provides a more organized and structured way of handling different types of information, and therefore provides them to the frontend that is responsible for displaying data using the latest UI/UX designs.

Lastly, users interact with the system through a Progressive Web App (PWA), enabling a responsive and seamless user experience on both mobile devices and web browsers. This PWA system acts as the interface for users to visualize and interact with the data generated by the Arduino equipped on their old parents.

9 Arduino Implementation

9.1 Arduino in depth flow chart:

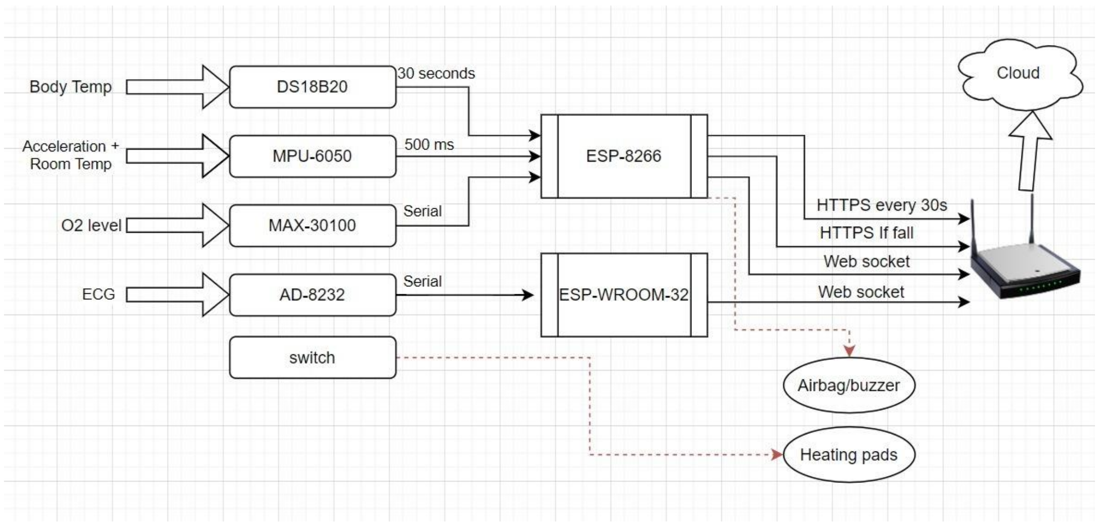
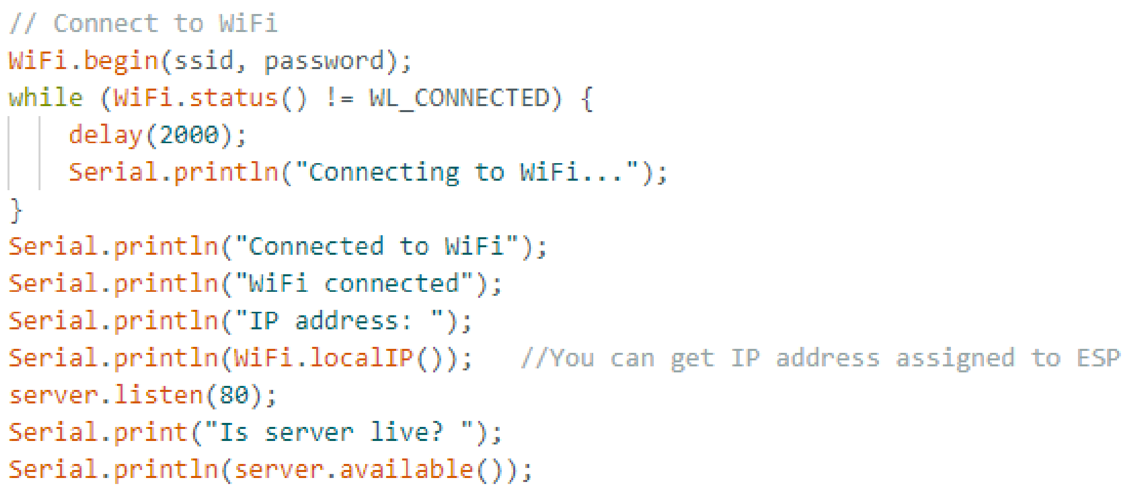


Figure 9-1: Arduino flow chart

In the above figure we can see all the data connection. In the first part we can see the sensors connection to the ESPs and the type of sending data. In the second part we can see the way the ESPs send the data to the cloud remote server. Everything will be explained in detail.

9.2 Arduino Setup:

9.2.1 Wi-Fi connection:

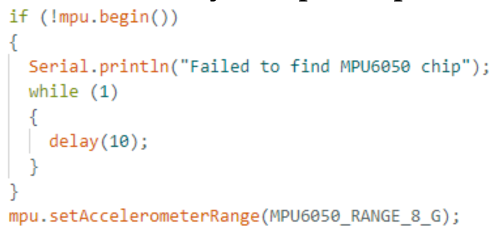


```
// Connect to WiFi
WiFi.begin(ssid, password);
while (WiFi.status() != WL_CONNECTED) {
    delay(2000);
    Serial.println("Connecting to WiFi...");
}
Serial.println("Connected to WiFi");
Serial.println("WiFi connected");
Serial.println("IP address: ");
Serial.println(WiFi.localIP());   //You can get IP address assigned to ESP
server.listen(80);
Serial.print("Is server live? ");
Serial.println(server.available());
```

Figure 9-2: Arduino Wi-Fi setup

First, we start by setting up the connection to WIFI. In the above code, we can see how the Wi-Fi connection is established using the wifi.begin(ssid, password); command.

9.2.2 MPU-Gyroscope setup:

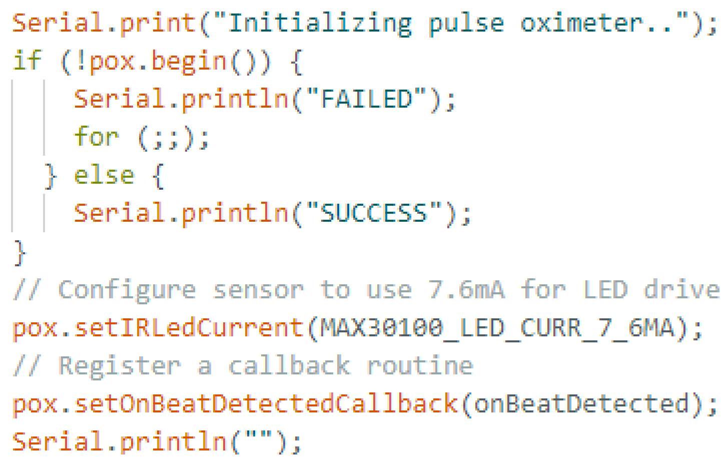


```
if (!mpu.begin())
{
  Serial.println("Failed to find MPU6050 chip");
  while (1)
  {
    delay(10);
  }
}
mpu.setAccelerometerRange(MPU6050_RANGE_8_G);
```

Figure 9-3: Arduino MPU setup.

As we can see in the figure above, we have set up the Accelerometer Range using mpu.setAccelerometerRange(MPU6050_RANGE_8_G), it means that the accelerometer sensor can measure accelerations in three dimensions (X, Y, and Z axes) within the range of -8 G to +8 G. The "G" stands for gravity, and in this context, it represents the standard acceleration due to gravity on Earth, approximately 9.8 meters per second squared (m/s^2). This range is suitable for detecting a variety of movements, including those associated with falls.

9.2.3 MAX30100 setup:



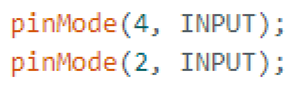
```
Serial.print("Initializing pulse oximeter..");
if (!pox.begin()) {
    Serial.println("FAILED");
    for (;;);
  } else {
    Serial.println("SUCCESS");
}
// Configure sensor to use 7.6mA for LED drive
pox.setIRLedCurrent(MAX30100_LED_CURR_7_6MA);
// Register a callback routine
pox.setOnBeatDetectedCallback(onBeatDetected);
Serial.println("");
```

Figure 9-4: Arduino MAX setup

The code snippet, pox.setIRLedCurrent(MAX30100_LED_CURR_7_6MA), plays a major role in configuring the current for the infrared (IR) LEDs, a critical parameter in both pulse oximetry and photoplethysmography (PPG). As discussed earlier in this paper, selecting the appropriate LED current involves a delicate balance between achieving a robust signal for precise measurements and mitigating power consumption and potential tissue damage. Higher LED currents typically yield more robust signals, yet they may elevate power consumption and the risk of tissue heating. Therefore, after much testing, the specific value of 7.6mA was deemed optimal. This choice represents a judicious compromise between signal strength and power efficiency for the given sensor.

9.2.4 AD-8232:

For this sensor we only specify the pin mode of the digital pins as inputs.



```
pinMode(4, INPUT);
pinMode(2, INPUT);
```

Figure 9-5: Arduino AD-8232 setup

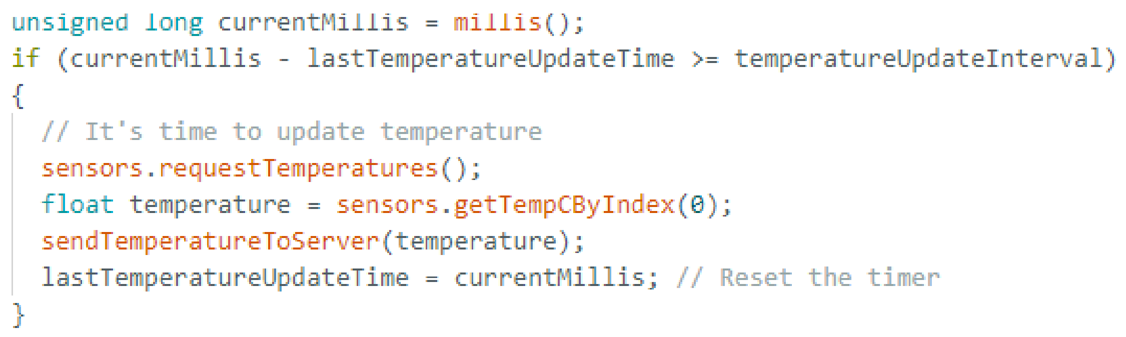
9.2.5 DSB18b20:

There was no need to do any direct setup in the setup function in the Arduino.

9.3 Implementation:

9.3.1 Temperature Sensor sending data Using POST https:

For this sensor, the millis function is utilized in such a way that data is sent at regular intervals of time, specifically set to 30 seconds. This approach is suitable because the data being transmitted is not constantly changing or highly sensitive.

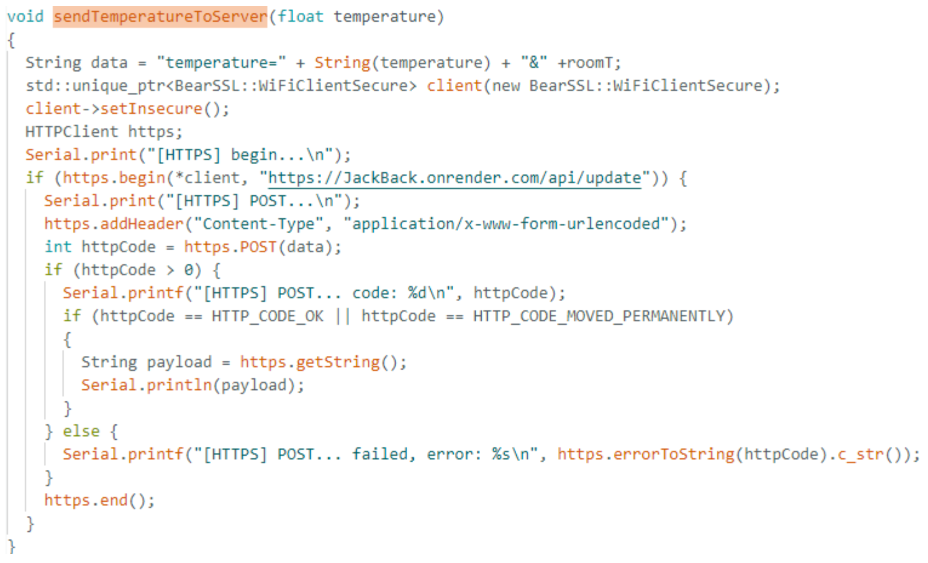


```
unsigned long currentMillis = millis();
if (currentMillis - lastTemperatureUpdateTime >= temperatureUpdateInterval)
{
  // It's time to update temperature
  sensors.requestTemperatures();
  float temperature = sensors.getTempCByIndex(0);
  sendTemperatureToServer(temperature);
  lastTemperatureUpdateTime = currentMillis; // Reset the timer
}
```

Figure 9-6: Arduino temperature reading code.

The sendTemperatureToServer() function manages the procedure of transmitting data to the Node.js Backend through HTTPS POST, as previously discussed. This approach is feasible because we can afford to send this type of data as a one-time current state.

Subsequently, we will explore the details of how we established communication with our cloud-hosted server using HTTPS from the Arduino to send real-time data. This real-time data transmission is fundamental to the Internet of Things (IoT) and serves as the core functionality of our system.



```
void sendTemperatureToServer(float temperature)
{
  String data = "temperature=" + String(temperature) + "&" +roomT;
  std::unique_ptr<BearSSL::WiFiClientSecure> client(new BearSSL::WiFiClientSecure);
  client->setInsecure();
  HTTPClient https;
  Serial.print("[HTTPS] begin...\n");
  if (https.begin(*client, "https://JackBack.onrender.com/api/update")) {
    Serial.print("[HTTPS] POST...\n");
    https.addHeader("Content-Type", "application/x-www-form-urlencoded");
    int httpCode = https.POST(data);
    if (httpCode > 0) {
      Serial.printf("[HTTPS] POST... code: %d\n", httpCode);
      if (httpCode == HTTP_CODE_OK || httpCode == HTTP_CODE_MOVED_PERMANENTLY)
      {
        String payload = https.getString();
        Serial.println(payload);
      }
    } else {
      Serial.printf("[HTTPS] POST... failed, error: %s\n", https.errorToString(httpCode).c_str());
    }
    https.end();
  }
}
```

Figure 9-7: Temperature sending data.

So, we start by constructing the data string that includes both the room and body temperature. We establish a secure WIFI client using BearSSL to connect to a designated IP address, which is our backend server at https://JackBack.onrender.com.

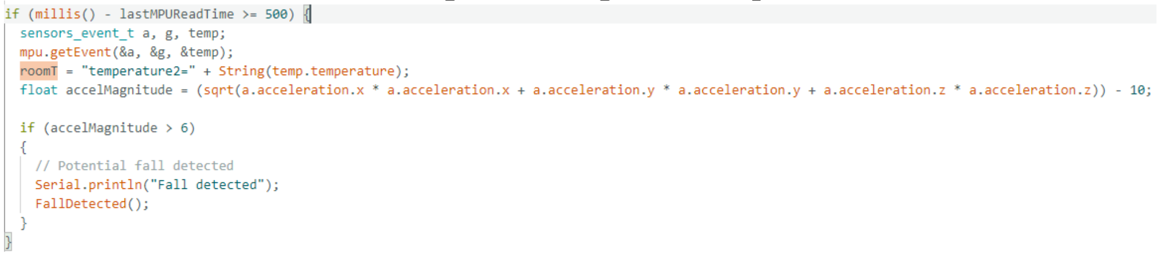
Following this, we initiate an HTTPS connection:

```
if (https.begin(*client, "https://JackBack.onrender.com/api/update"))
```

Note that the "/api/update" is used to route the connection to the specific API responsible for receiving the temperature values, parsing them, and subsequently transmitting them to the frontend.

Subsequently, we specify the content type to be "application/x-www-form-urlencoded," indicating the data form. Finally, we send the data via an HTTPS POST request: (`int httpCode = https.POST(data)`). In a separate section, we will elaborate on how the backend processes the received data and sends it to the database.

9.3.2 Fall detection and sending data Using POST https:

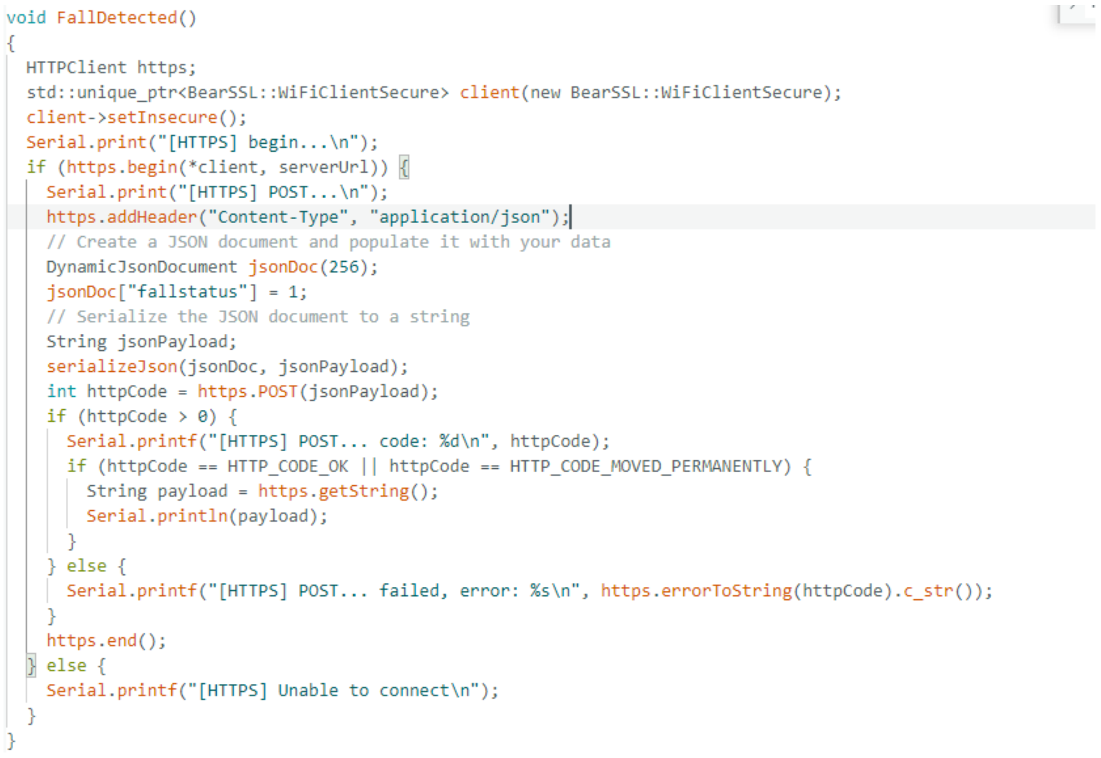


```
if (millis() - lastMPUReadTime >= 500) {
  sensors_event_t a, g, temp;
  mpu.getEvent(&a, &g, &temp);
  roomT = "temperature2=" + String(temp.temperature);
  float accelMagnitude = (sqrt(a.acceleration.x * a.acceleration.x + a.acceleration.y * a.acceleration.y + a.acceleration.z * a.acceleration.z)) - 10;

  if (accelMagnitude > 6)
  {
    // Potential fall detected
    Serial.println("Fall detected");
    FallDetected();
  }
}
```

Figure 9-8: Arduino fall detection code.

We initiated the process by getting the acceleration magnitude on every axis, specifically every 500 milliseconds utilizing the millis function. Then, we computed the total acceleration excluding the gravitational factor. We examined the magnitude, and if it surpasses the predefined threshold, a fall detection function is invoked.

The fall detection function triggers an HTTPS request to the server for sending a notification, following a similar methodology as explained previously in the temperature sensor. This ensures that the fall event is communicated to the server, and the notification process is activated. The fall notification will be explained in a later part.



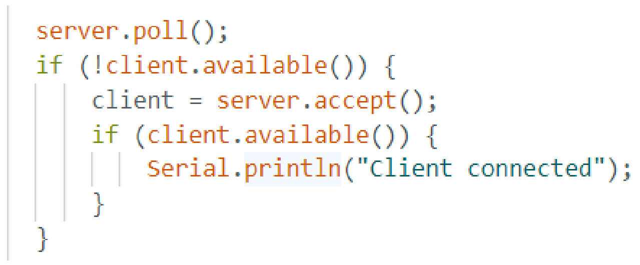
```
void FallDetected()
{
  HTTPClient https;
  std::unique_ptr<BearSSL::WiFiClientSecure> client(new BearSSL::WiFiClientSecure);
  client->setInsecure();
  Serial.print("[HTTPS] begin...\n");
  if (https.begin(*client, serverUrl)) {
    Serial.print("[HTTPS] POST...\n");
    https.addHeader("Content-Type", "application/json");
    // Create a JSON document and populate it with your data
    DynamicJsonDocument jsonDoc(256);
    jsonDoc["fallstatus"] = 1;
    // Serialize the JSON document to a string
    String jsonPayload;
    serializeJson(jsonDoc, jsonPayload);
    int httpCode = https.POST(jsonPayload);
    if (httpCode > 0) {
      Serial.printf("[HTTPS] POST... code: %d\n", httpCode);
      if (httpCode == HTTP_CODE_OK || httpCode == HTTP_CODE_MOVED_PERMANENTLY) {
        String payload = https.getString();
        Serial.println(payload);
      }
    } else {
      Serial.printf("[HTTPS] POST... failed, error: %s\n", https.errorToString(httpCode).c_str());
    }
    https.end();
  } else {
    Serial.printf("[HTTPS] Unable to connect\n");
  }
}
```

Figure 9-9: Arduino fall detection data sending.

In addition, the MPU is used to detect the room temperature, this value is saved and sent with the body temperature.

9.3.3 Sending Oxygen level continuously using WebSocket:

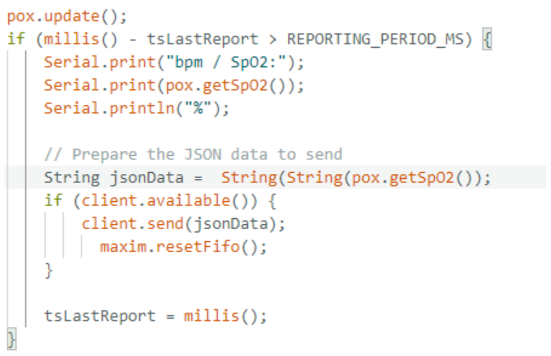
Firstly, we define the WebSocket server and client as follows: WebsocketsServer server; WebsocketsClient client; this is done after including the necessary WebSocket library. We set the server's IP and port that the client will use to connect to the socket. The server.poll() function is then called, as depicted in the figure below.



```
server.poll();
if (!client.available()) {
    client = server.accept();
    if (client.available()) {
        Serial.println("Client connected");
    }
}
```

Figure 9-10: Arduino WebSocket setup.

This function is an integral part of a WebSocket server, serving the purpose of checking for incoming WebSocket connections or messages. It enables the server to handle WebSocket events. Now, regarding the if condition, it checks whether a client (in this case, Node.js) is connected. Upon establishing a connection, it initiates the continuous transmission of data to the client. This instantaneous data transfer to the server exemplifies the potency of socket communication. The code snippet below demonstrates how data is sent:

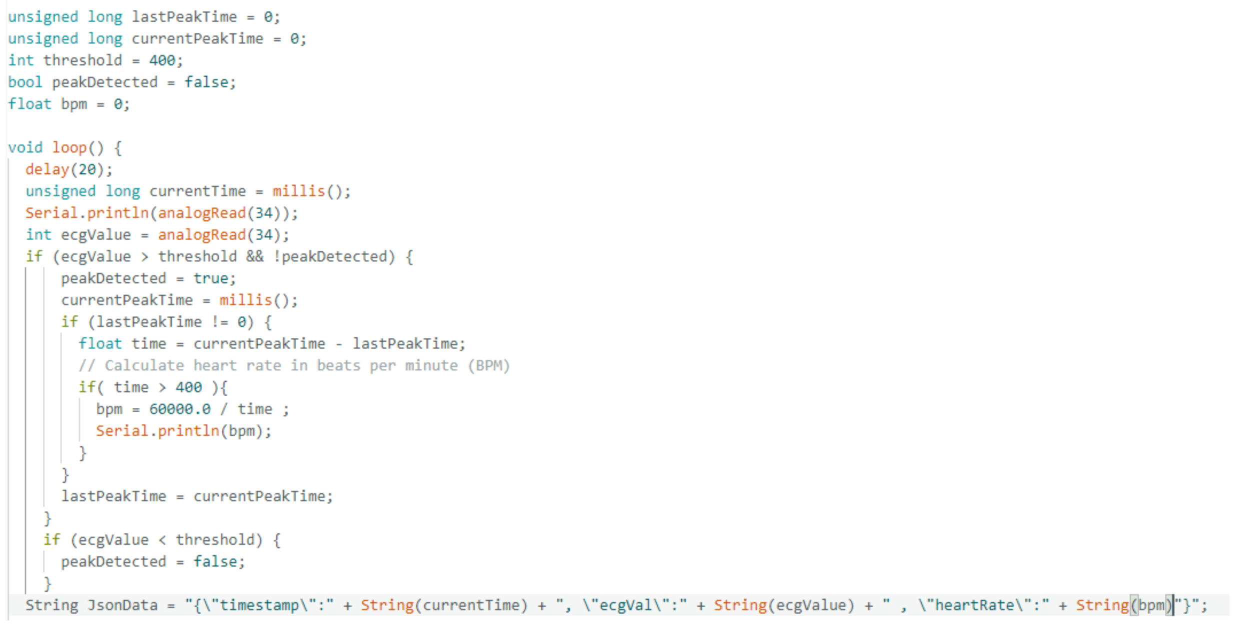


```
pox.update();
if (millis() - tsLastReport > REPORTING_PERIOD_MS) {
    Serial.print("bpm / SpO2:");
    Serial.print(pox.getSpO2());
    Serial.println("%");

    // Prepare the JSON data to send
    String jsonData =  String(String(pox.getSpO2());
    if (client.available()) {
        client.send(jsonData);
         maxim.resetFifo();
    }

    tsLastReport = millis();
}
```

Figure 9-11: Arduino reading and sending Oxygen level.

Here, every 1 second (reporting_period_ms = 1000 ms), we retrieve the Spo2 level using the previously set up pox instance through functions like pox.getSpO2(). Subsequently, we transmit them using the client.send() function. This function sends the data as bytes (buffer), which we then need to decode (Unbuffer) on our backend.

9.3.4 Sending ECG and heart rate using WebSocket:



```
unsigned long lastPeakTime = 0;
unsigned long currentPeakTime = 0;
int threshold = 400;
bool peakDetected = false;
float bpm = 0;

void loop() {
  delay(20);
  unsigned long currentTime = millis();
  Serial.println(analogRead(34));
  int ecgValue = analogRead(34);
  if (ecgValue > threshold && !peakDetected) {
      peakDetected = true;
      currentPeakTime = millis();
      if (lastPeakTime != 0) {
        float time = currentPeakTime - lastPeakTime;
        // Calculate heart rate in beats per minute (BPM)
        if( time > 400 ){
          bpm = 60000.0 / time ;
          Serial.println(bpm);
        }
      }
      lastPeakTime = currentPeakTime;
    }
    if (ecgValue < threshold) {
      peakDetected = false;
    }
  String JsonData = "{\"timestamp\":" + String(currentTime) + ", \"ecgVal\":" + String(ecgValue) + " , \"heartRate\":" + String(bpm)"}";
```

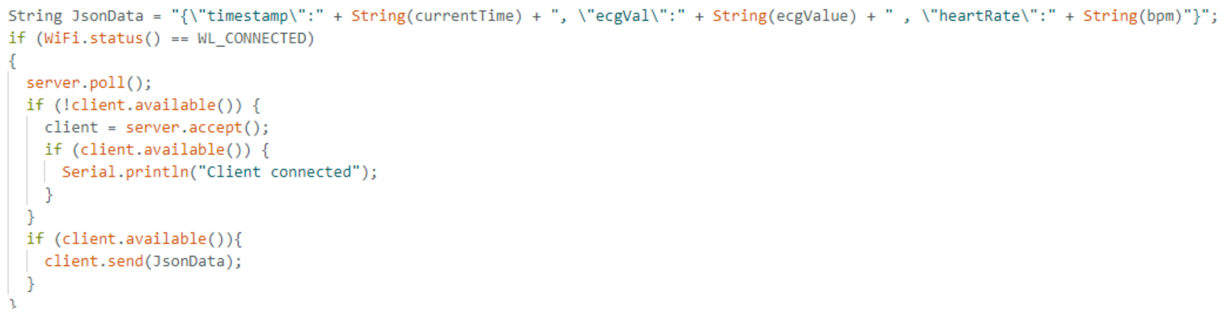
Figure 9-12: Arduino ECG reading and heart rate calculation.

For the AD-8232, we gather the ECG values and utilize them to calculate the heart rate. As explained in a previous part, we us the following formula to find the heart rate.

Heart rate (bpm) = 60000 / R-to-R time (in milliseconds).

The R-to-R time is determined through conditional statements in the preceding code. These conditions involve checking if the ecgValue exceeds a certain threshold and confirming that no peak was detected in the previous ecgValue reading. This precaution ensures that the detected peaks correspond to distinct heartbeats. Subsequently, the R-to-R time is calculated as follows: R-to-R = currentPeakTime - lastPeakTime;

After that, the ecgValue and time of it in addition to the heart rate are sent using WebSocket to the backend, in the same maner the oxygen level in the MAX-30100 data is sent.



```
String JsonData = "{\"timestamp\":" + String(currentTime) + ", \"ecgVal\":" + String(ecgValue) + " , \"heartRate\":" + String(bpm)"}";
if (WiFi.status() == WL_CONNECTED)
{
  server.poll();
  if (!client.available()) {
    client = server.accept();
    if (client.available()) {
      Serial.println("Client connected");
    }
  }
  if (client.available()){
    client.send(JsonData);
  }
}
```

Figure 9-13: Arduino ECG and heart rate data sending.

By employing these two different communication protocols, we successfully send all of our sensor's data to our cloud-hosted server. This accomplishment signifies the creation of an IoT product, as the sensors are now capable of seamlessly transmitting their data to the internet.

10 Arduino Seniors Testing

10.1 ECG sensor AD-8232:

To test the AD-8232, we attached disposable ECG Electrodes to the body and interfaced the sensor with an Arduino. Initially, we encountered corrupted data, and upon investigation, we identified that the charger connected to the laptop was causing noise interference. The AD8232 sensor is specifically designed for measuring the heart's electrical activity, and the corresponding ECG signal is inherently weak, typically in the microvolts (µV) range. Contrastingly, noise interference can be significantly stronger, ranging from millivolts (mV) to even volts (V). Consequently, the robustness of the noise can easily overpower the delicate ECG signal, making its detection challenging.

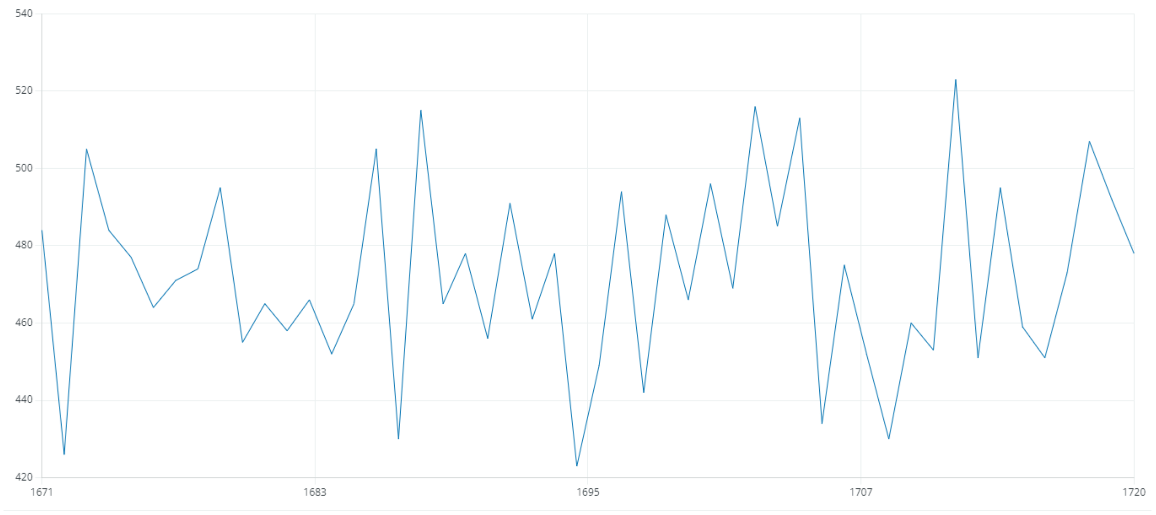


Figure 10-1: ECG with charger connected.

The solution was to use the battery of the pc without connecting the charger to reduce as much as we can the interference. This led us to the following output:

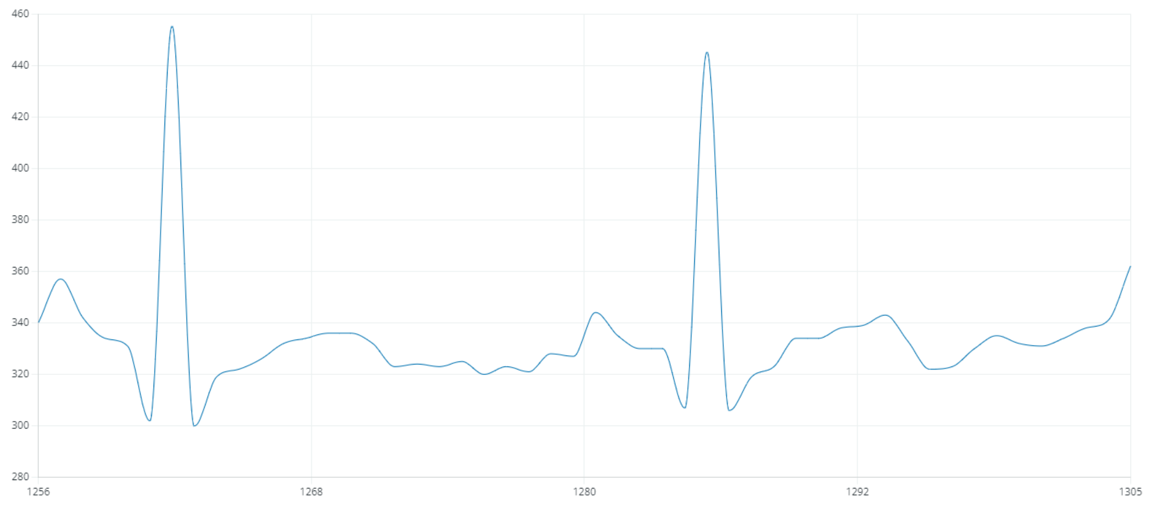


Figure 10-2: ECG with PC on battery only

Upon achieving acceptable results, we proceeded to assess the impact of muscle movement on the ECG. Our findings revealed that the ECG signal can be influenced by additional sources of electrical activity within the body, particularly from skeletal muscles. When these muscles contract or move, they generate electrical signals that have the potential to interfere with the ECG signal, giving rise to what is commonly known as motion artifacts or muscle artifacts. These artifacts have the capability to distort both the shape and amplitude of the ECG waves, posing challenges in their detection and subsequent analysis.

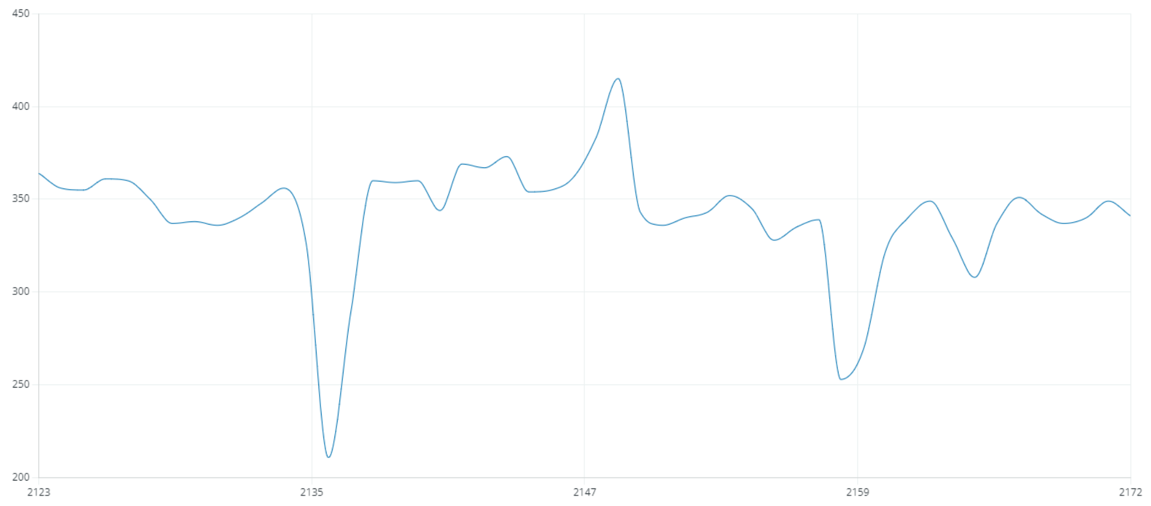


Figure 10-3: ECG with core contracted.

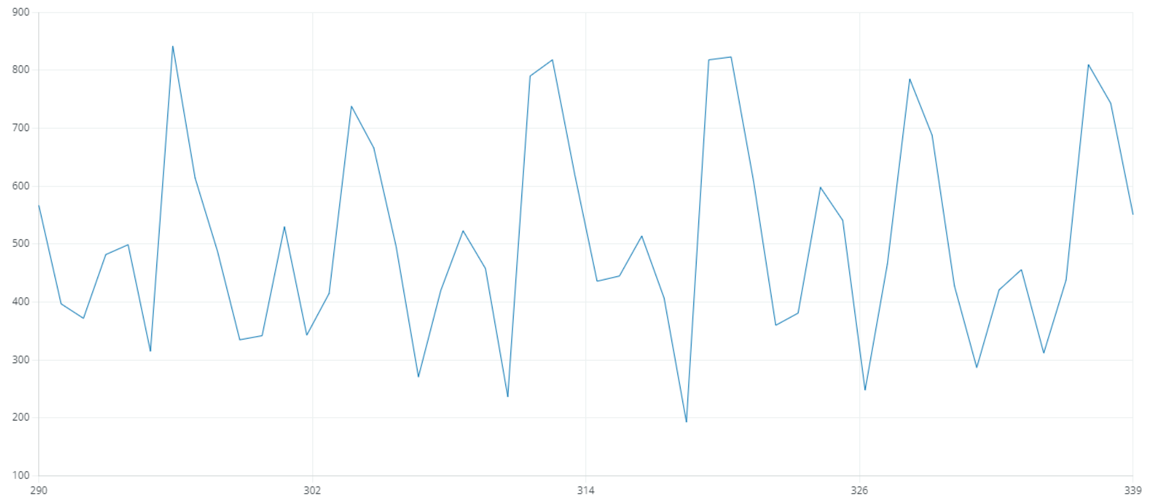
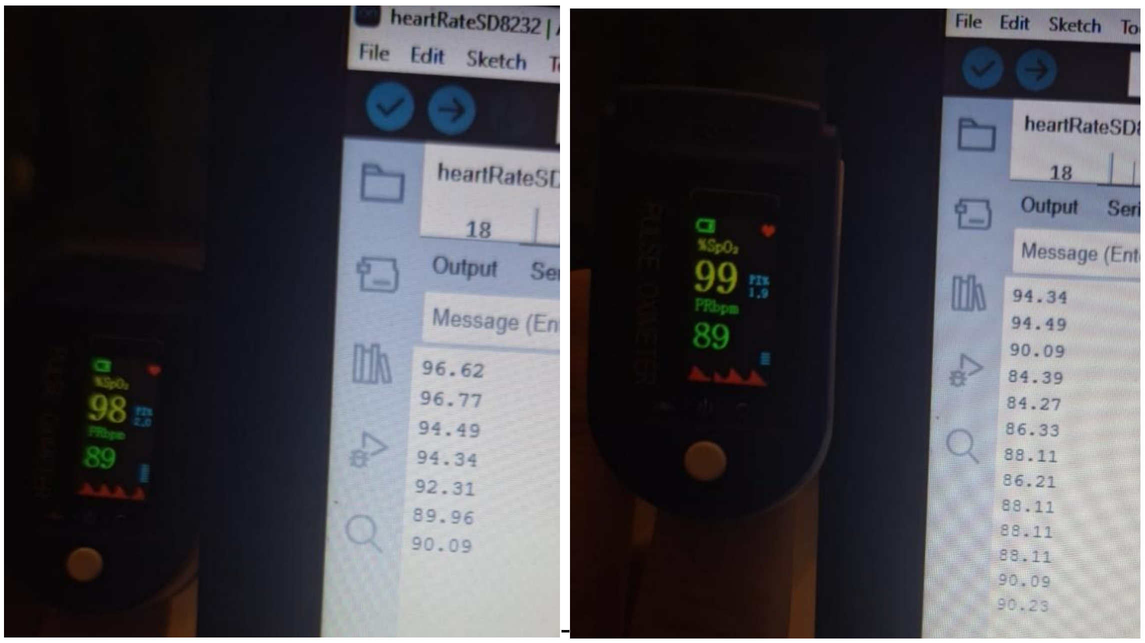


Figure 10-4: ECG will the person is walking.

After getting the most acceptable data of the ECG, we used the data to calculate the heart rate. This is a comparison of the output data and the values of the pulse oximeter.



```
96.62
96.77
94.49
94.34
92.31
89.96
90.09
```

```
94.34
94.49
90.09
84.39
84.27
86.33
88.11
86.21
88.11
88.11
88.11
90.09
90.23
```

Figure 10-5: Comparison between Pulse Oximeter and ECG heart rate.

According to the above picture, we can use the values to calculate the error percentage of the sensor:

$$Error = \frac{Sensor\ Reading - Expected\ Value}{Expected\ Value}$$

This formula yields a 1% and up to 8% error in the worst scenario, which is very acceptable according to the university doctor. It indicates that the sensor has a high level of accuracy.

10.2 Oxygen blood level sensor MAX-30100:

In this section, we will be utilizing the same pulse oximeter as used in the previous testing section. First, place the AD-30100 on one finger, and simultaneously place the pulse oximeter on another finger, as shown in the following picture:

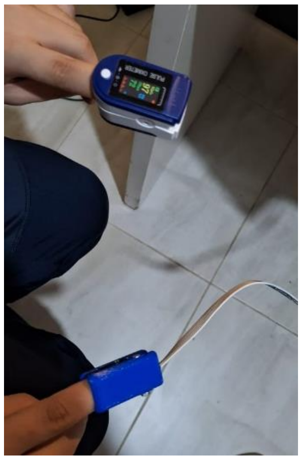
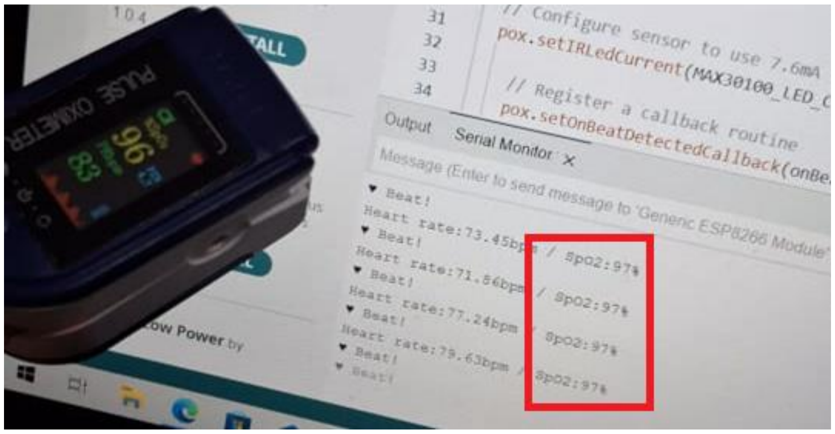


Figure 10-6: Testing MAX-30100

Next, refer to the image below. The yellow indicates the oxygen level in the pulse oximeter, while the laptop screen displays the level measured using the AD-30100.



```
// Configure sensor to use 7.6mA
pox.setIRLedCurrent(MAX30100_LED_C

// Register a callback routine
pox.setOnBeatDetectedCallback(onBe
```

```
♥ Beat!
Heart rate:73.45bpm / SpO2:97%
♥ Beat!
Heart rate:71.56bpm / SpO2:97%
♥ Beat!
Heart rate:77.24bpm / SpO2:97%
♥ Beat!
Heart rate:79.63bpm / SpO2:97%
♥ Beat!
♥ Beat!
```

Figure 10-7: Comparison between Pulse Oximeter and MAX-30100 oxygen level

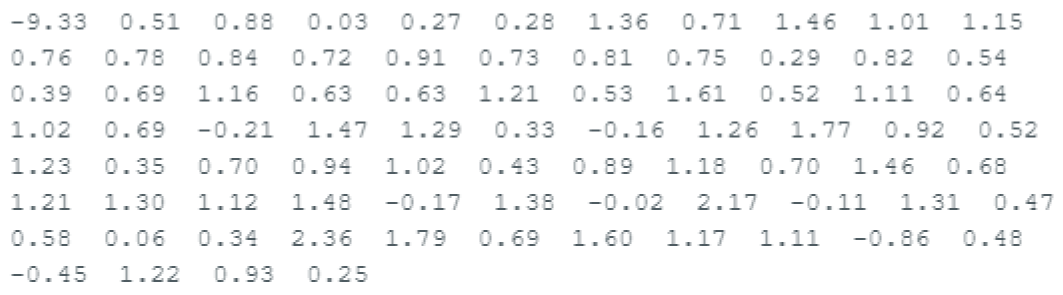
According to the above picture, we can use the values to calculate the error percentage of the sensor:

$$Error = \frac{Sensor\ Reading - Expected\ Value}{Expected\ Value}$$

This formula yields a 1% error, which is very acceptable. It indicates that the sensor has a high level of accuracy.

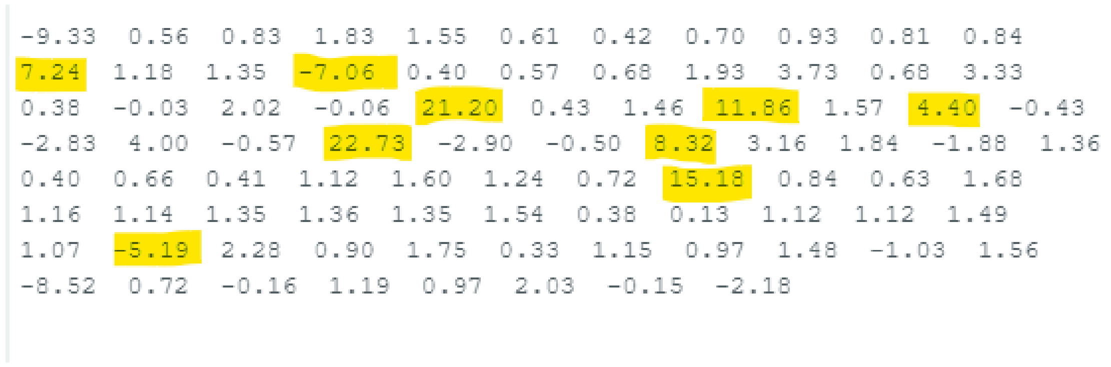
10.3 Fall detection using accelerometer MPU-6050:

For this sensor, we initiated the process by obtaining the acceleration magnitude corresponding to the body's movement under normal conditions, such as walking, sitting, and standing. The collected data reflects the acceleration of the body over the last 500 milliseconds for each scenario. Subsequently, we conducted experiments involving falling movements to determine the acceleration magnitude in those scenarios.



```
-9.33  0.51  0.88  0.03  0.27  0.28  1.36  0.71  1.46  1.01  1.15
0.76  0.78  0.84  0.72  0.91  0.73  0.81  0.75  0.29  0.82  0.54
0.39  0.69  1.16  0.63  0.63  1.21  0.53  1.61  0.52  1.11  0.64
1.02  0.69  -0.21  1.47  1.29  0.33  -0.16  1.26  1.77  0.92  0.52
1.23  0.35  0.70  0.94  1.02  0.43  0.89  1.18  0.70  1.46  0.68
1.21  1.30  1.12  1.48  -0.17  1.38  -0.02  2.17  -0.11  1.31  0.47
0.58  0.06  0.34  2.36  1.79  0.69  1.60  1.17  1.11  -0.86  0.48
-0.45  1.22  0.93  0.25
```

Figure 10-8: MPU acceleration magnitude in normal conditions.



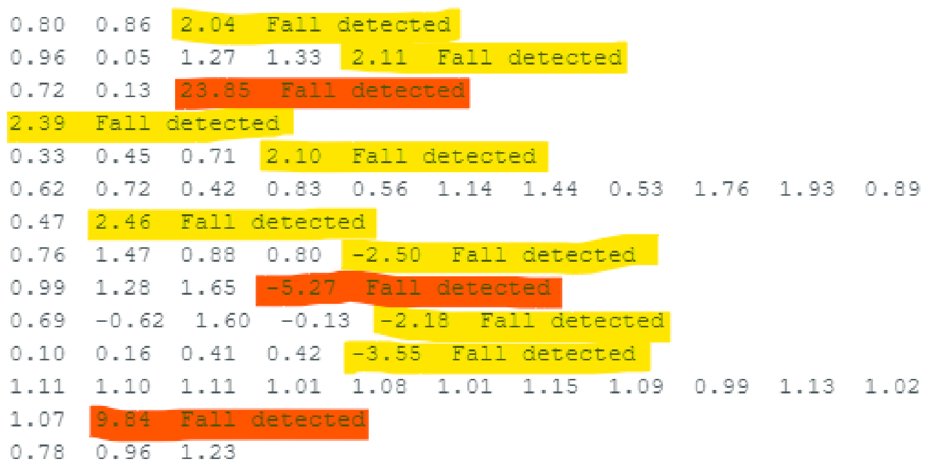
```
-9.33  0.56  0.83  1.83  1.55  0.61  0.42  0.70  0.93  0.81  0.84
7.24  1.18  1.35  -7.06  0.40  0.57  0.68  1.93  3.73  0.68  3.33
0.38  -0.03  2.02  -0.06  21.20  0.43  1.46  11.86  1.57  4.40  -0.43
-2.83  4.00  -0.57  22.73  -2.90  -0.50  8.32  3.16  1.84  -1.88  1.36
0.40  0.66  0.41  1.12  1.60  1.24  0.72  15.18  0.84  0.63  1.68
1.16  1.14  1.35  1.36  1.35  1.54  0.38  0.13  1.12  1.12  1.49
1.07  -5.19  2.28  0.90  1.75  0.33  1.15  0.97  1.48  -1.03  1.56
-8.52  0.72  -0.16  1.19  0.97  2.03  -0.15  -2.18
```

Figure 10-9:MPU acceleration magnitude with falling simulation.

.

We observed that the values in normal conditions consistently fall within the range of [-2, +2]. However, during a fall, they exceed +6 or fall below -6. Our initial testing for fall detection involved scenarios where the acceleration magnitude was outside the range [-2,

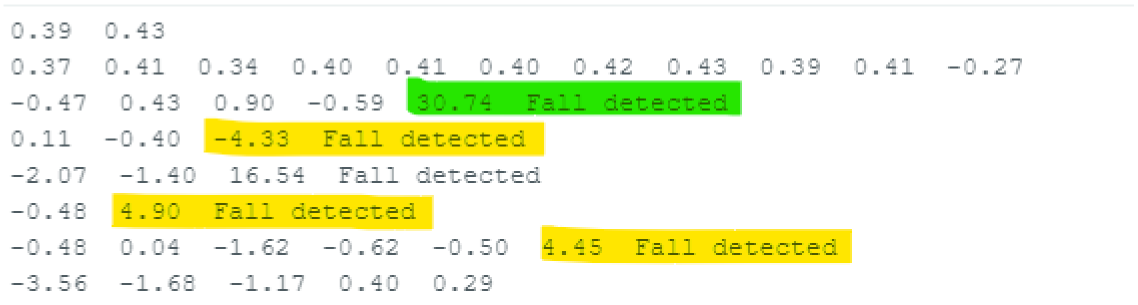
+2]. Unfortunately, this approach resulted in numerous false alarms. In our visual representation, false alarms are denoted by yellow, while true alarms are marked in red.



```
0.80   0.86   2.04   Fall detected
0.96   0.05   1.27   1.33   2.11   Fall detected
0.72   0.13   23.85   Fall detected
2.39   Fall detected
0.33   0.45   0.71   2.10   Fall detected
0.62   0.72   0.42   0.83   0.56   1.14   1.44   0.53   1.76   1.93   0.89
0.47   2.46   Fall detected
0.76   1.47   0.88   0.80   -2.50   Fall detected
0.99   1.28   1.65   -5.27   Fall detected
0.69   -0.62   1.60   -0.13   -2.18   Fall detected
0.10   0.16   0.41   0.42   -3.55   Fall detected
1.11   1.10   1.11   1.01   1.08   1.01   1.15   1.09   0.99   1.13   1.02
1.07   9.84   Fall detected
0.78   0.96   1.23
```

Figure 10-10: MPU fall detection with 2 as threshold.

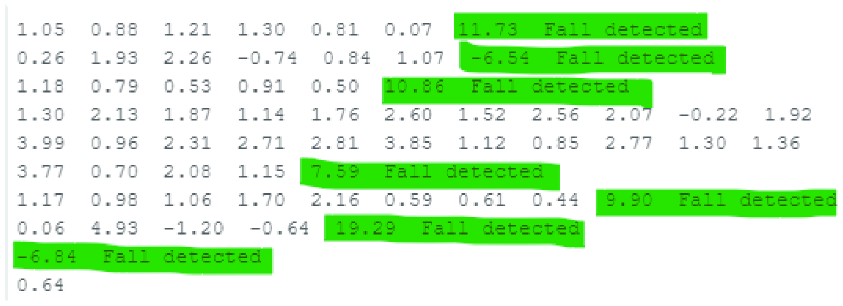
Following that, when testing within the range of [-4, +4], which gave us a similar result to the previous range but with less false alarms.



```
0.39   0.43
0.37   0.41   0.34   0.40   0.41   0.40   0.42   0.43   0.39   0.41   -0.27
-0.47   0.43   0.90   -0.59   30.74   Fall detected
0.11   -0.40   -4.33   Fall detected
-2.07   -1.40   16.54   Fall detected
-0.48   4.90   Fall detected
-0.48   0.04   -1.62   -0.62   -0.50   4.45   Fall detected
-3.56   -1.68   -1.17   0.40   0.29
```

Figure 10-11: MPU fall detection with 4 as threshold.

In the end, using the range [-6, +6], we achieved more satisfactory results. This adjusted range significantly decreased the occurrence of false alarms while effectively detecting most falls.



```
1.05   0.88   1.21   1.30   0.81   0.07   11.73   Fall detected
0.26   1.93   2.26   -0.74   0.84   1.07   -6.54   Fall detected
1.18   0.79   0.53   0.91   0.50   10.86   Fall detected
1.30   2.13   1.87   1.14   1.76   2.60   1.52   2.56   2.07   -0.22   1.92
3.99   0.96   2.31   2.71   2.81   3.85   1.12   0.85   2.77   1.30   1.36
3.77   0.70   2.08   1.15   7.59   Fall detected
1.17   0.98   1.06   1.70   2.16   0.59   0.61   0.44   9.90   Fall detected
0.06   4.93   -1.20   -0.64   19.29   Fall detected
-6.84   Fall detected
0.64
```

Figure 10-12: MPU fall detection with 6 as threshold

10.4 Airbag after fall detection:

In our project the loop for detecting a fall has a duration of 500 milliseconds. In the worst-case scenario, if the fall starts at the starting of the loop, we have a 500-millisecond delay adding to that the delay of airbag deployment that will be discussed next. For testing the airbag is replaced with a buzzer that takes the same signal of an airbag, but it produces a noise.

The delay of deployment of an airbag is around 30 to 50 milliseconds [21]. Adding that to the 500 milliseconds we find that the total fall detection and protection system has a delay of 550 milliseconds.

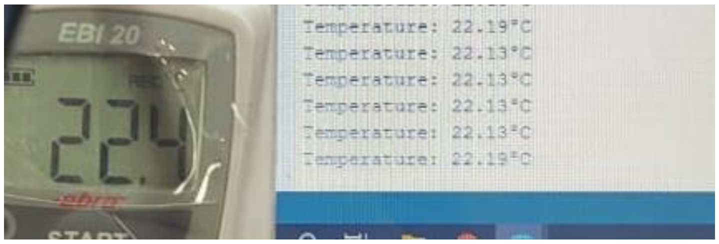
To find if the delay is acceptable or not let's take a person if height 165cm and take the worst-case scenario. That is if the person collapses spontaneously, and collapses on the ground directly without the resistance of the feet. We will use the following formula to find the time required before directly hitting the ground.

$$time = \sqrt{\frac{2h}{g}} = 580\ milliseconds$$

This value is greater than the time required to initiate the fall detection and protection system. In addition, the feet play an important role in slowing down the falling prosses which can get it up to 1 second, adding to that is the person is not fully collapse can help delay the impact.

10.5 Room Temperature using MPU-6050:

In this section, we utilized a room thermometer to accurately measure the ambient room temperature. Following this, the MPU-6050, which functions as the sensor responsible for collecting room temperature data in our project, was tested. In the following image, the values of the room temperature are displayed on both the thermometer and the MPU-6050.



```
Temperature: 22.19°C
Temperature: 22.13°C
Temperature: 22.13°C
Temperature: 22.13°C
Temperature: 22.13°C
Temperature: 22.19°C
```

Figure 10-13: MPU-6050 temperature testing

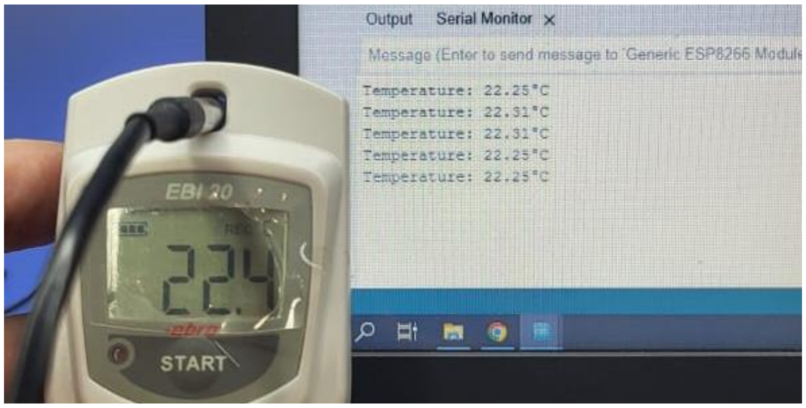
According to the above picture, we can use the values to calculate the error percentage of the sensor:

$$Error = \frac{Sensor\ Reading - Expected\ Value}{Expected\ Value}$$

This formula yields a 1.2% error, which is very acceptable. It indicates that the sensor has a high level of accuracy.

10.6 Body Temperature using DS18B20:

In this section, we employed a body thermometer to precisely determine the temperature of the body. Subsequently, the DS18B20, which serves as the sensor that collects the body temperature in our project, was tested. In the following image, we can see the value of the body temperature on the body thermometer and DS18B20:



```
Temperature: 22.25°C
Temperature: 22.31°C
Temperature: 22.31°C
Temperature: 22.25°C
Temperature: 22.25°C
```

Figure 10-14: DS18B20 testing

According to the above picture, we can use the values to calculate the error percentage of the sensor:

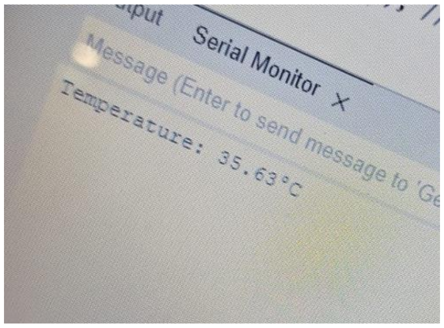
$$Error = \frac{Sensor\ Reading - Expected\ Value}{Expected\ Value}$$

This formula yields a 0.66% error, which is very acceptable. It indicates that the sensor has a high level of accuracy.

10.7 Heating pads:

In this section, we tested the output temperature of the heating pads. We connected the heating. And measured the output temperatures according to the two voltage input modes (9 and 12 Volts). As we have seen before, the heating pads are connected in parallel, so the voltage across each one is 4.5 to 6 volts. The values measured are taken after connecting the power supply for the heating pads for 2 minutes.

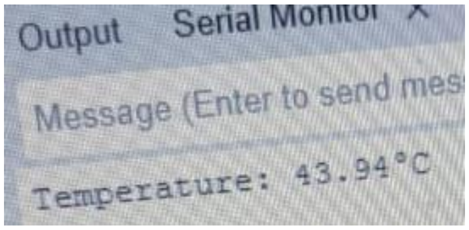
This is the result of connecting the heating pads to a 9-volt power supply:



```
Temperature: 35.63°C
```

Figure 10-15: Heating pad temperature at 9 volts

This is the result of connecting the heating pads to a 12-volt power supply:



```
Temperature: 43.94°C
```

Figure 10-16: Heating pad temperature at 9 volts

According to Cardinal Health [22], the appropriate and safe temperature range is 40-45°C. In the above testing, both values lie within and below this range, indicating that they are safe for use on the body. Additionally, the power supply used can output 9 and 12 volts, providing the user with the flexibility to choose a temperature that they find comfortable.

11 Backend Software Implementation

11.1 Flow Chart:

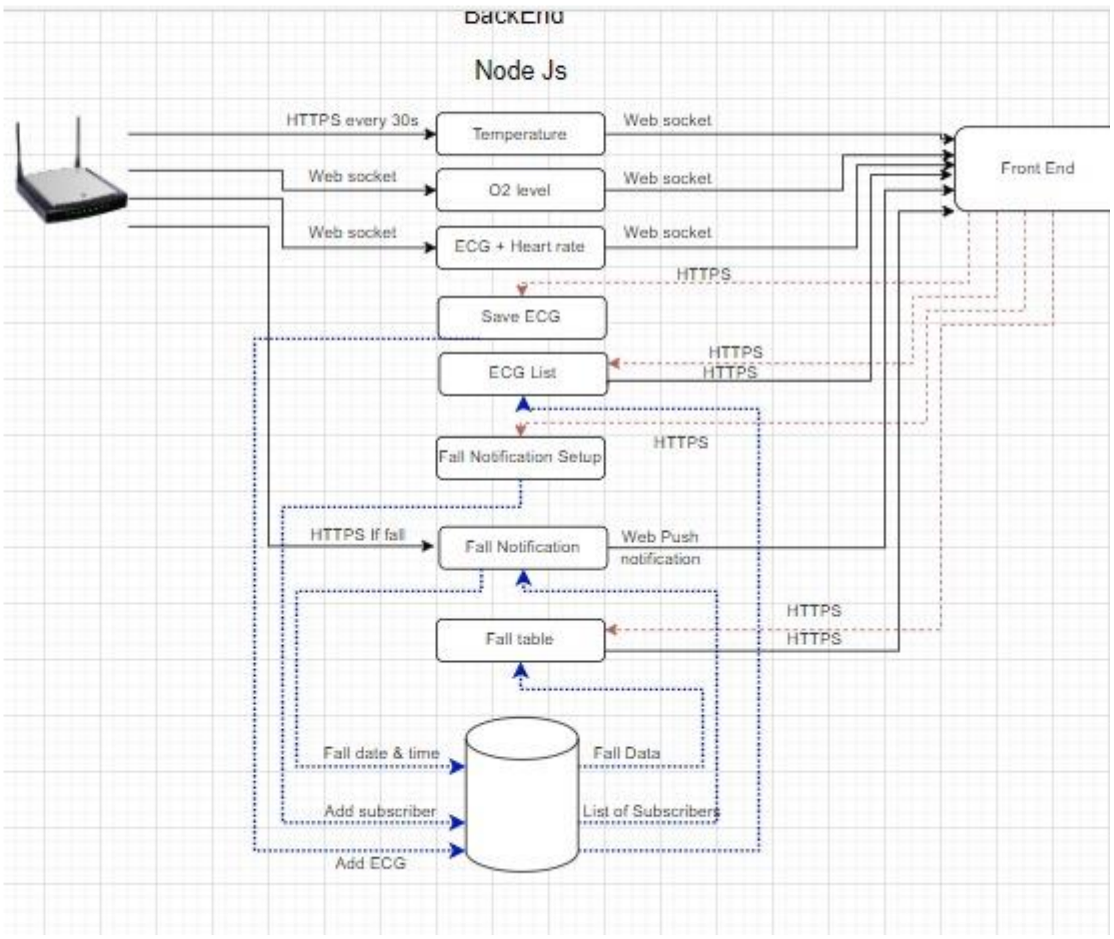


Figure 11-1: Backend flow chart

In the above flow chart, we can see the connections of the backend. Those connections will be explained in detail in the following chapter.

11.2 Initialization:

We created an express server and configured it to use CORS (Cross-Origin Resource Sharing) to allow requests from any origin. Then we initialize a Socket.io connection which is used to communicate with the frontend size. For this server, we are using two different socket libraries:

Socket.io: for a higher-level abstraction with additional features, especially when working with frontend applications or when you need the broader capabilities provided by Socket.IO like sending data over distinct channels.

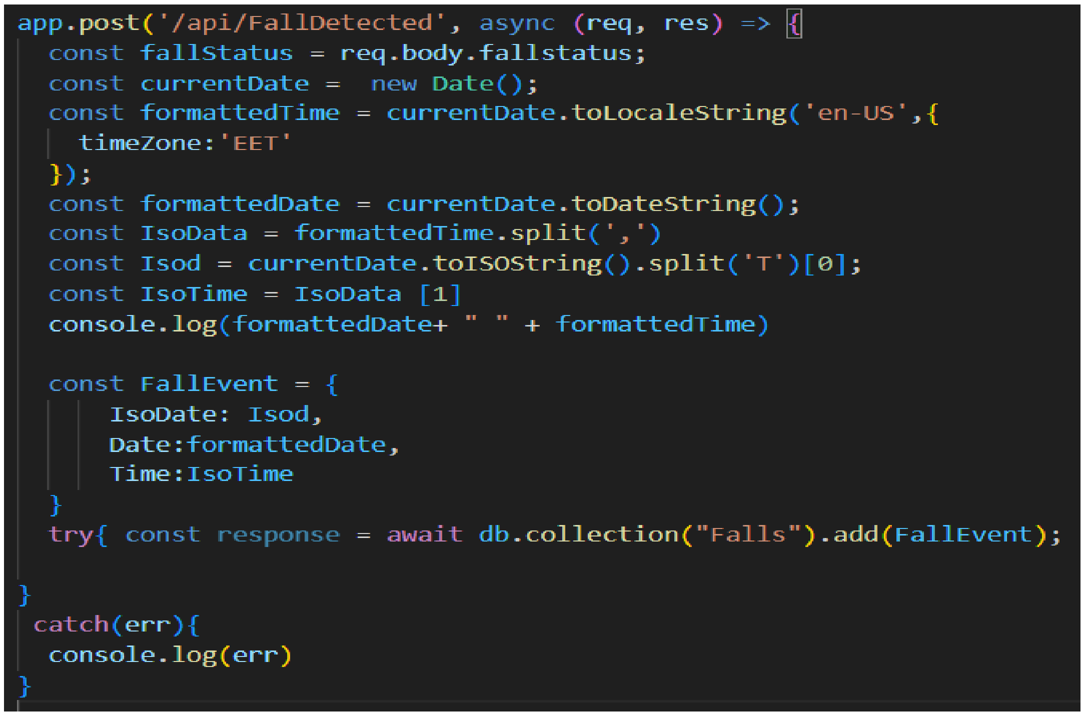
Ws: for lower-level WebSocket communication, suitable for direct communication with devices like Arduino that does not support socket.io.

Notice that io is built on top of ws, it provides a higher-level abstraction for real-time communication. Socket.IO supports various transport, including WebSocket, and it has features like automatic reconnection, broadcasting, and support for different transport mechanisms if WebSocket is not available.

11.3 Implementation:

11.3.1 Fall detection:

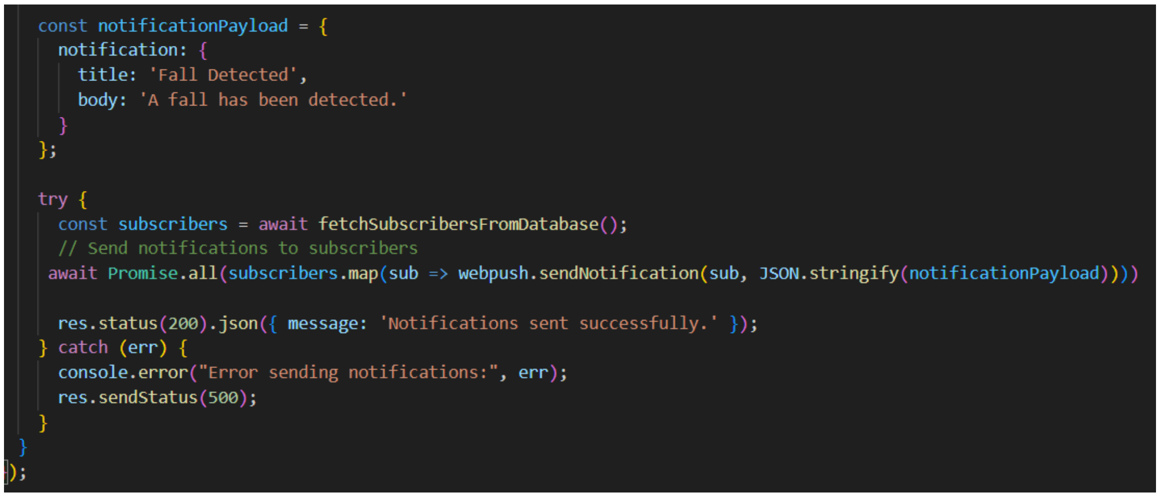
As seen in the flowchart, an HTTPS request is reserved from the Arduino. This post event initiates the following function:



```
app.post('/api/FallDetected', async (req, res) => {
  const fallStatus = req.body.fallstatus;
  const currentDate =  new Date();
  const formattedTime = currentDate.toLocaleString('en-US',{
    timeZone:'EET'
  });
  const formattedDate = currentDate.toDateString();
  const IsoData = formattedTime.split(',')
  const Isod = currentDate.toISOString().split('T')[0];
  const IsoTime = IsoData [1]
  console.log(formattedDate+ " " + formattedTime)

  const FallEvent = {
        IsoDate: Isod,
        Date:formattedDate,
        Time:IsoTime
  }
  try{ const response = await db.collection("Falls").add(FallEvent);

}
 catch(err){
  console.log(err)
}
```

Figure 11-2: NodeJS reception of fall data.

Firstly, the route (/api/FallDetected) is designed to handle the HTTP POST request related to fall detection. Whenever this API is called, it signifies that a fall has been detected. Consequently, we immediately record the date and time into the database. This is achieved by obtaining the current date based on Lebanon's timing (EET) and, after performing some manipulation to extract the desired format, placing the data in an object called FallEvent. Subsequently, we save it into the NoSQL Firestore database collection named "Falls." This data will be utilized later by our UI to present falls in a table.

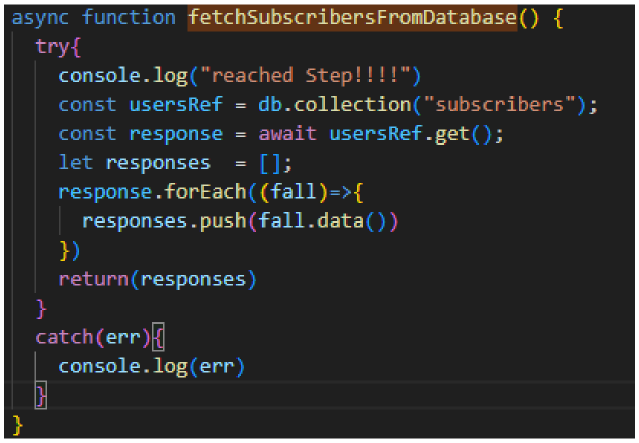


```
    const notificationPayload = {
      notification: {
        title: 'Fall Detected',
        body: 'A fall has been detected.'
      }
    };

    try {
      const subscribers = await fetchSubscribersFromDatabase();
      // Send notifications to subscribers
     await Promise.all(subscribers.map(sub => webpush.sendNotification(sub, JSON.stringify(notificationPayload))))

      res.status(200).json({ message: 'Notifications sent successfully.' });
    } catch (err) {
      console.error("Error sending notifications:", err);
      res.sendStatus(500);
    }
  }
});
```

Figure 11-3: NodeJS saving data and sending notification.

Now it's time to send notifications to those concerned about the detected fall. To identify the recipients, we invoke the function fetchSubscribersFromDatabase(), which we designed to retrieve subscribers list. This allows us to send notifications to the relevant parties. The details of how this data is stored will be elaborated in a subsequent section.

This is the function fetchSubscribersFromDatabase(). It is responsible for retrieving a list of all subscribers form the database.

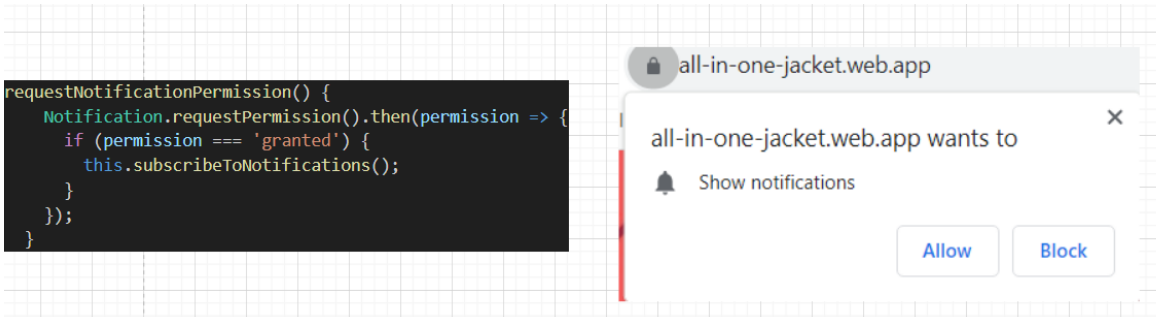


```
async function fetchSubscribersFromDatabase() {
  try{
    console.log("reached Step!!!!")
    const usersRef = db.collection("subscribers");
    const response = await usersRef.get();
    let responses  = [];
    response.forEach((fall)=>{
      responses.push(fall.data())
    })
    return(responses)
  }
  catch(err){
    console.log(err)
  }
}
```

Figure 11-4:NodeJS rasding list of subscribers from database.

11.3.2 Notification List:

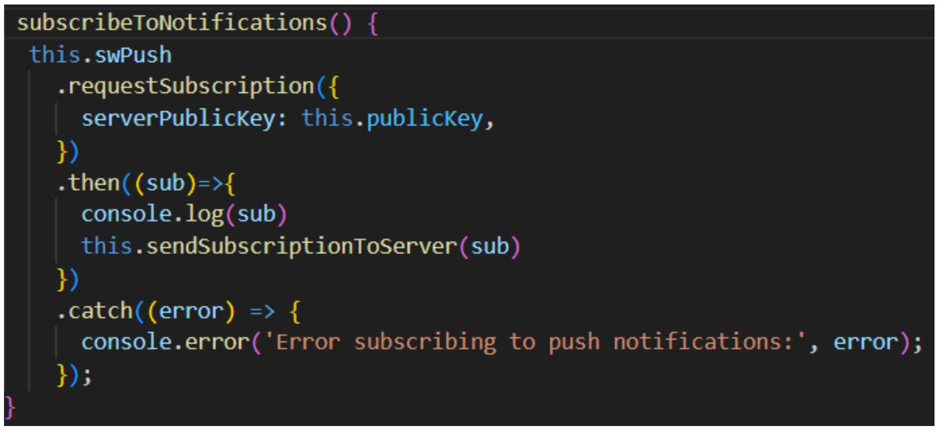
First step is to ask for the user consent, this is done in the Frontend Angular project:



```
requestNotificationPermission() {
    Notification.requestPermission().then(permission => {
      if (permission === 'granted') {
        this.subscribeToNotifications();
      }
    });
  }
```

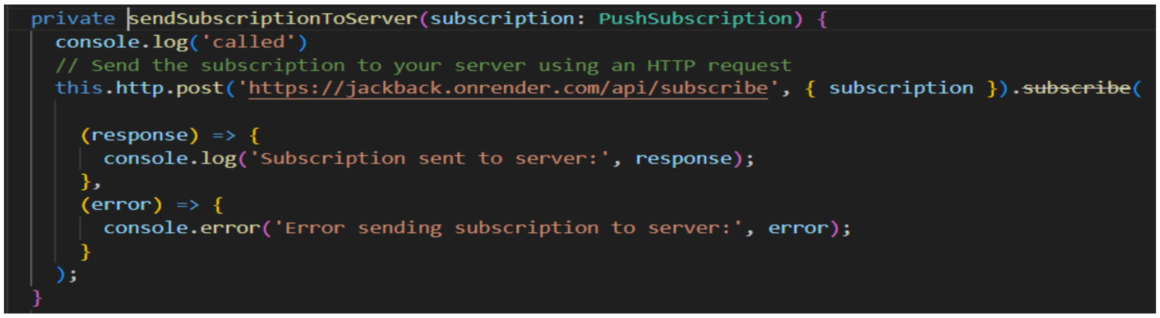
Figure 11-5: Angular asking user about notification.

If the user accepts the notification, a request is sent to the Nodejs server using the following function named subsribeToNotifications(). This function prepares the information to be sent to the backend and invokes the function sendSubscriptionToServer(). This function issues a post request from angular side to https://jackback.onrender.com/api/subscribe which is our node.js backend hosted on the cloud containing the {subscription} object. This object is used as a reference to the destination of the notification, in addition to some keys that are typically used by the server to encrypt and sign the payload of the notification.



```
subscribeToNotifications() {
  this.swPush
    .requestSubscription({
      serverPublicKey: this.publicKey,
    })
    .then((sub)=>{
      console.log(sub)
      this.sendSubscriptionToServer(sub)
    })
    .catch((error) => {
      console.error('Error subscribing to push notifications:', error);
    });
}
```

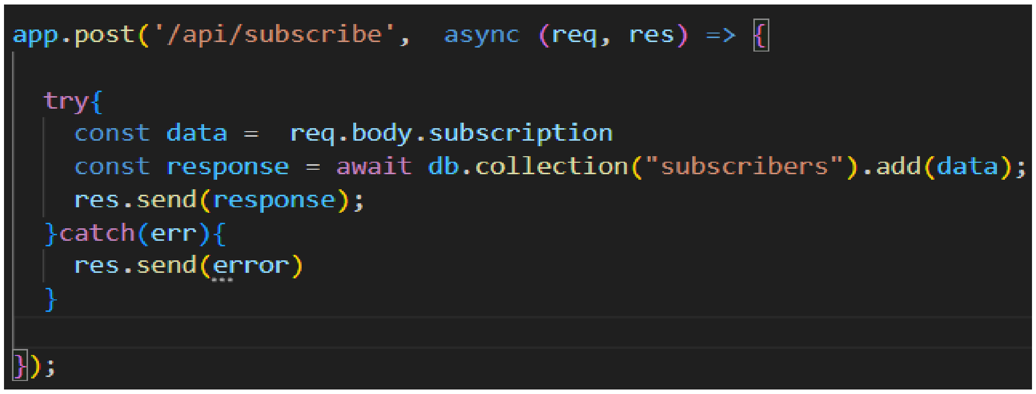
Figure 11-6: Angular processing acceptance.



```
private sendSubscriptionToServer(subscription: PushSubscription) {
  console.log('called')
  // Send the subscription to your server using an HTTP request
  this.http.post('https://jackback.onrender.com/api/subscribe', { subscription }).~~subscribe~~(

    (response) => {
      console.log('Subscription sent to server:', response);
    },
    (error) => {
      console.error('Error sending subscription to server:', error);
    }
  );
}
```

Figure 11-7: Angular sending subscription to NodeJS.

Now back to the NodeJs server. A HTTPS post is detected which will save the data to the database. The function is in the following snap of the code:



```
app.post('/api/subscribe',  async (req, res) => {

  try{
    const data =  req.body.subscription
    const response = await db.collection("subscribers").add(data);
    res.send(response);
  }catch(err){
    res.send(error)
  }

});
```

Figure 11-8: NodeJS saving user subscribe.

In conclusion, we can summarize the process of sending a notification ass follows:

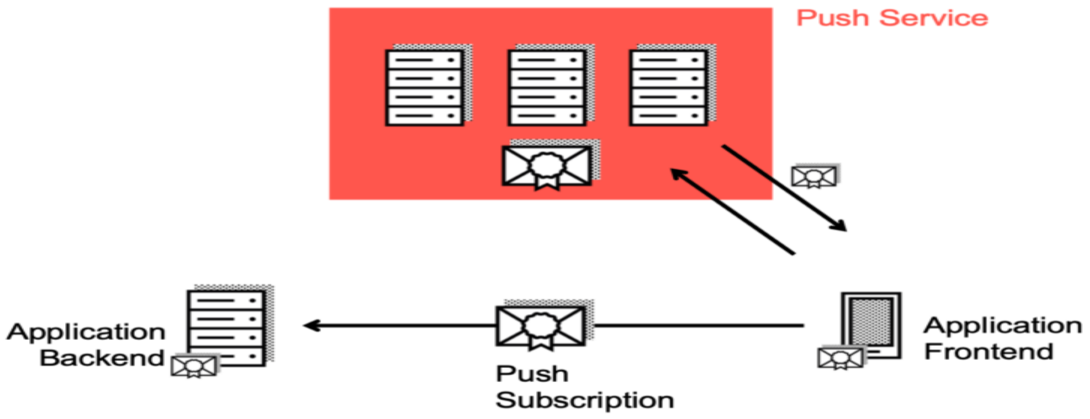
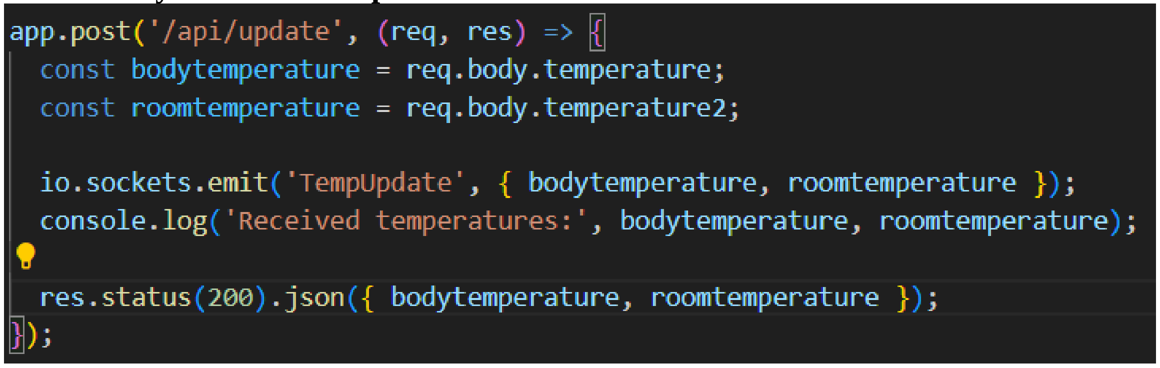


Figure 11-9: Push notification service explanation

The user (frontend) send its subscription to backend, the backend saves them in the database, and whenever a notification should be sent, the backend fetches the list of subscribers from database and send them the notifications on the destinations according to the list.

11.3.3 Body and room temperatures:

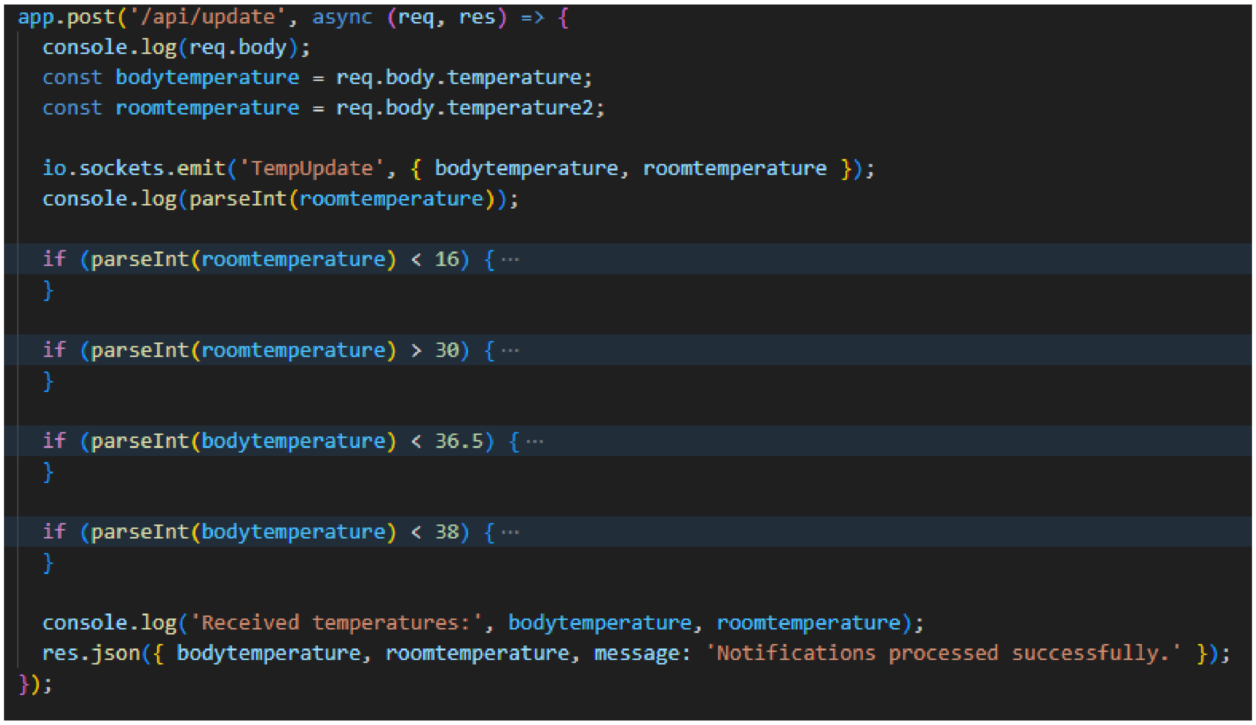


```
app.post('/api/update', (req, res) => {
  const bodytemperature = req.body.temperature;
  const roomtemperature = req.body.temperature2;

  io.sockets.emit('TempUpdate', { bodytemperature, roomtemperature });
  console.log('Received temperatures:', bodytemperature, roomtemperature);

  res.status(200).json({ bodytemperature, roomtemperature });
});
```

Figure 11-10: NodeJS receiving temperature from Arduino and sending it to Angular.

A route is defined to listen for HTTP POST requests at the path '/api/update'. As illustrated in the flowchart, a POST request is received every 30 seconds, expecting data in the request body, specifically temperature and temperature2. Subsequently, the received data is sent to the frontend using Socket.IO.

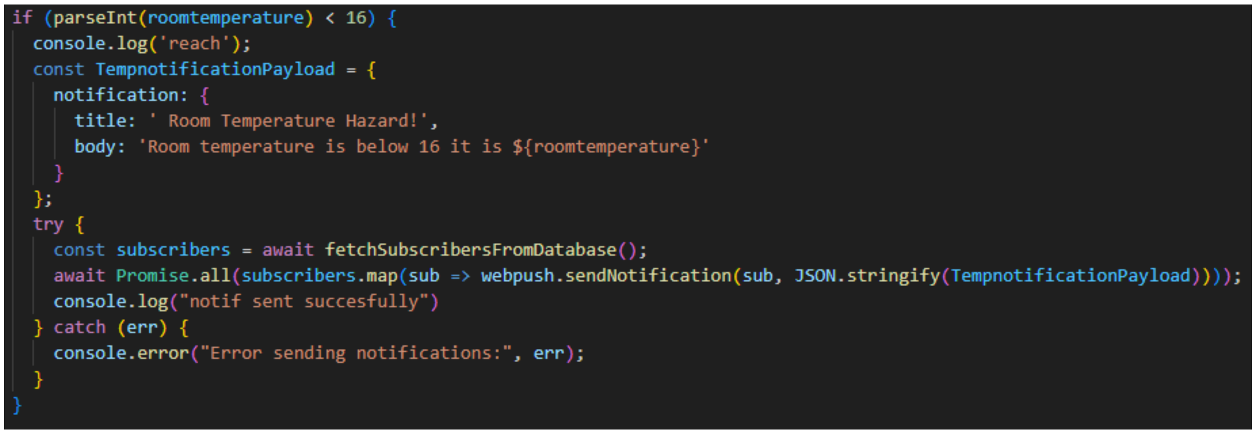
In addition, we added four additional functionalities to this function. A notification is sent if the room temperature is outside the range [16, 30] degree Celsius and if the body temperature is outside the range [36.5, 38] degree Celsius.



```
app.post('/api/update', async (req, res) => {
  console.log(req.body);
  const bodytemperature = req.body.temperature;
  const roomtemperature = req.body.temperature2;

  io.sockets.emit('TempUpdate', { bodytemperature, roomtemperature });
  console.log(parseInt(roomtemperature));

  if (parseInt(roomtemperature) < 16) {···
  }

  if (parseInt(roomtemperature) > 30) {···
  }

  if (parseInt(bodytemperature) < 36.5) {···
  }

  if (parseInt(bodytemperature) < 38) {···
  }

  console.log('Received temperatures:', bodytemperature, roomtemperature);
  res.json({ bodytemperature, roomtemperature, message: 'Notifications processed successfully.' });
});
```

Figure 11-11: NodeJS using temperature to send notification.

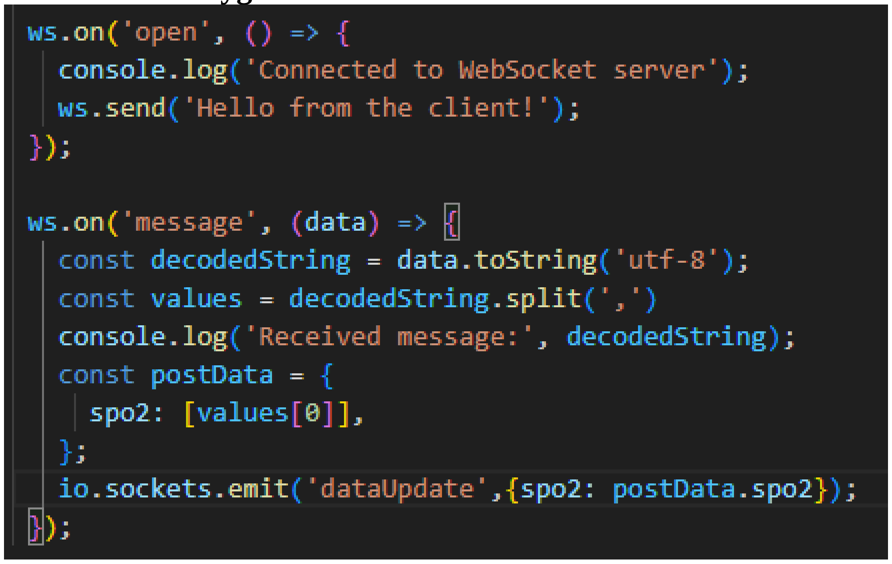
For the four cases the following function is invoked, which send the notification with the appropriate message accordingly. And the same process is done for the other three cases, but each with its appropriate message.



```
if (parseInt(roomtemperature) < 16) {
  console.log('reach');
  const TempnotificationPayload = {
    notification: {
      title: ' Room Temperature Hazard!',
      body: 'Room temperature is below 16 it is ${roomtemperature}'
    }
  };
  try {
    const subscribers = await fetchSubscribersFromDatabase();
    await Promise.all(subscribers.map(sub => webpush.sendNotification(sub, JSON.stringify(TempnotificationPayload))));
    console.log("notif sent succesfully")
  } catch (err) {
    console.error("Error sending notifications:", err);
  }
}
```

Figure 11-12:NodeJS in depth notification.

11.3.4 Blood oxygen level data:

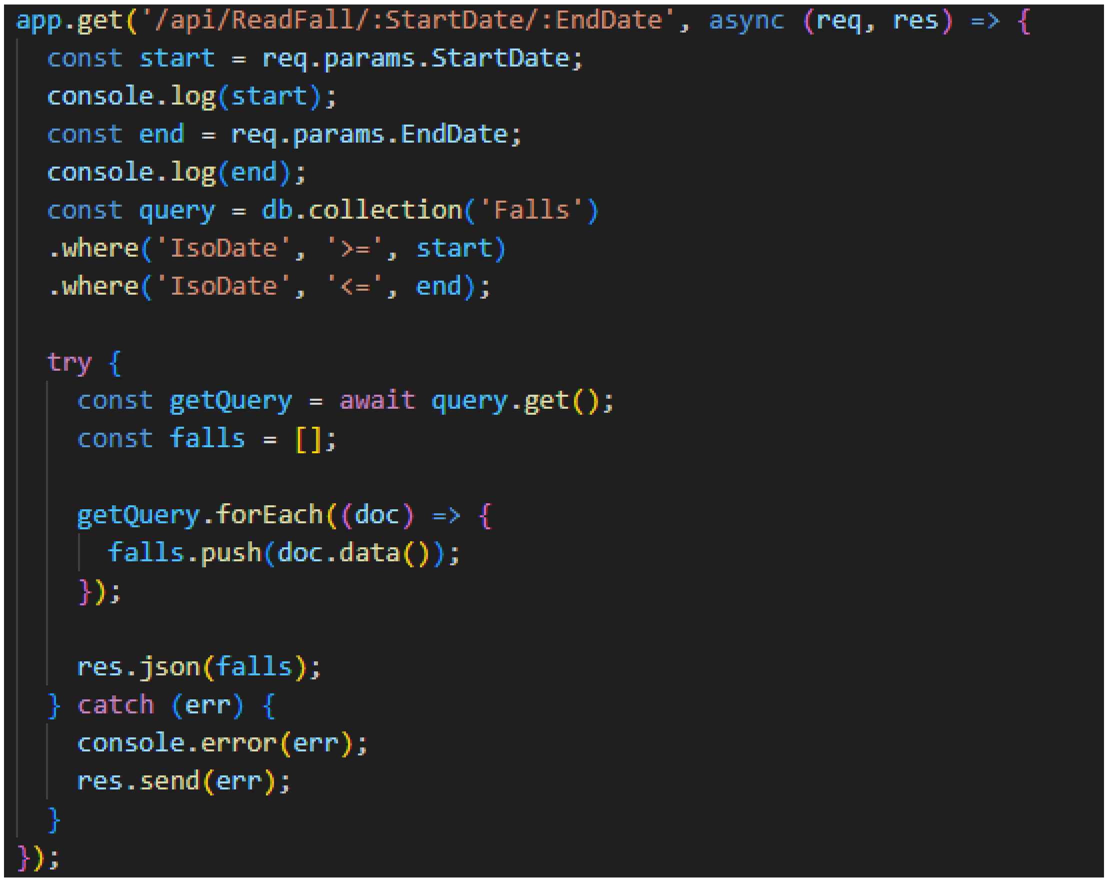


```
ws.on('open', () => {
  console.log('Connected to WebSocket server');
  ws.send('Hello from the client!');
});

ws.on('message', (data) => {
  const decodedString = data.toString('utf-8');
  const values = decodedString.split(',')
  console.log('Received message:', decodedString);
  const postData = {
    spo2: [values[0]],
  };
  io.sockets.emit('dataUpdate',{spo2: postData.spo2});
});
```

Figure 11-13:NodeJS receiving oxygen level from Arduino and sending it to Angular.

As in the flowchart, the data is received using WsSocket from the Arduino side immediately after it detects that our node.js server has connected to its channel, and it logs Hello from the client on the Arduino side (this is for debug purpose). So now, our socket connection from the node.js server is ready to listen for what it sends, and therefore receives the Spo2 (Oxygen level) parse it, save it do some manipulation to extract it in good form and then use our second socket library (socket.io) to forward the data to the frontend.

11.3.5 Fall table:

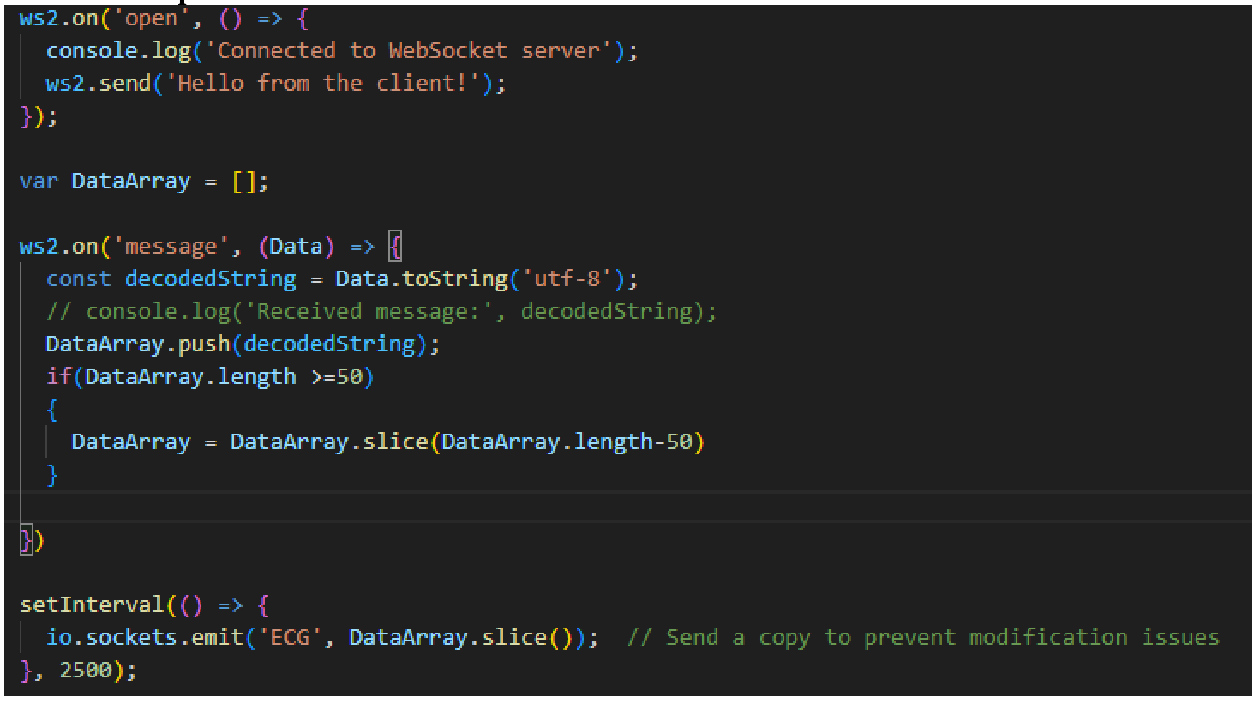


```
app.get('/api/ReadFall/:StartDate/:EndDate', async (req, res) => {
  const start = req.params.StartDate;
  console.log(start);
  const end = req.params.EndDate;
  console.log(end);
  const query = db.collection('Falls')
  .where('IsoDate', '>=', start)
  .where('IsoDate', '<=', end);

  try {
    const getQuery = await query.get();
    const falls = [];

    getQuery.forEach((doc) => {
      falls.push(doc.data());
    });

    res.json(falls);
  } catch (err) {
    console.error(err);
    res.send(err);
  }
});
```

Figure 11-14:NodeJS sending Fall data to Angular.

This API is made to be called from the frontend, when the user selects the start date and the end date using the calendar UI we provided, we extract those dates and call this api using a GET HTTPS request. As you can see in the figure above, first we take those start and end dates using the req.params and then we build the database query to fetch all the falls within this specified date range, using the query.get() we get all the falls data, put them in an array and send for the frontend to display them in a UI. (check front end section).

11.3.6 ECG plot:

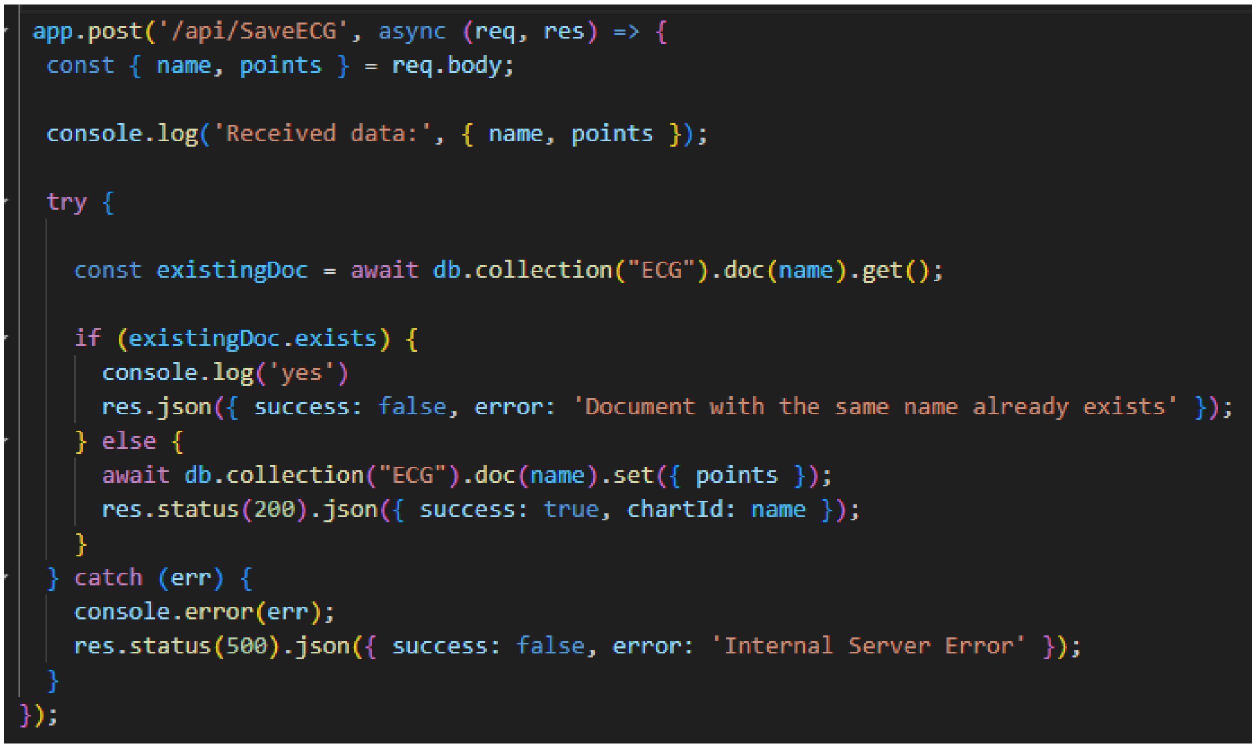


```
ws2.on('open', () => {
  console.log('Connected to WebSocket server');
  ws2.send('Hello from the client!');
});

var DataArray = [];

ws2.on('message', (Data) => {
  const decodedString = Data.toString('utf-8');
  // console.log('Received message:', decodedString);
  DataArray.push(decodedString);
  if(DataArray.length >=50)
  {
    DataArray = DataArray.slice(DataArray.length-50)
  }

})

setInterval(() => {
  io.sockets.emit('ECG', DataArray.slice());  // Send a copy to prevent modification issues
}, 2500);
```

Figure 11-15: NodeJS receiving ECG from Arduino and sending it to Angular.

As you can see in the picture above, the data is retrieved from the Arduino using WsSocket just like the case before and constantly pushed to an array. Now, when the array has more than 50 points, we remove the oldest point and push the new point, with this, we always display the last 50 points. (we can choose any number of points we wish to plot). Now, in order not to overload the connection between the backend server and node.js (since we are using a free cloud hosting server tool) we are sending those 50 points every 2.5 seconds (see the code above, the setInterval() function) we are emitting to the frontend every 2.5 seconds.

11.3.7 ECG save data:

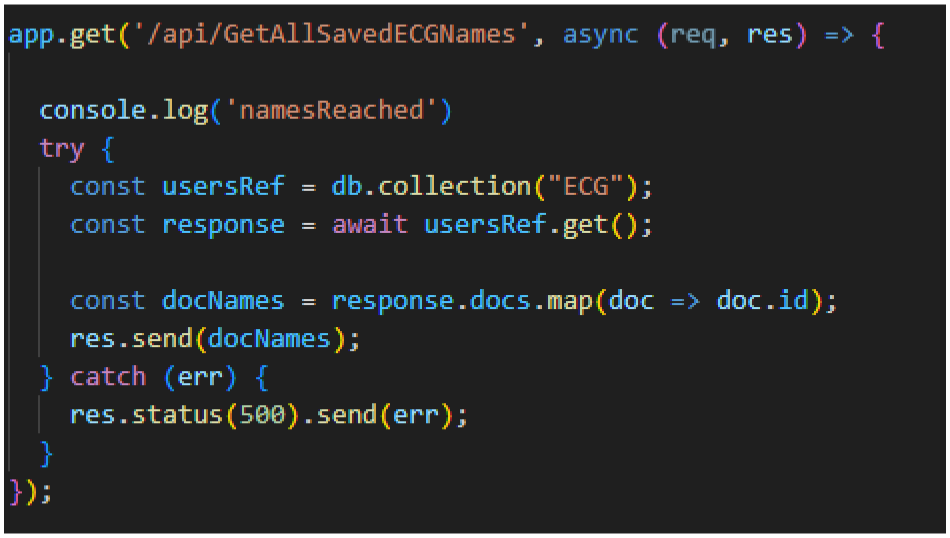


```
app.post('/api/SaveECG', async (req, res) => {
  const { name, points } = req.body;

  console.log('Received data:', { name, points });

  try {

    const existingDoc = await db.collection("ECG").doc(name).get();

    if (existingDoc.exists) {
      console.log('yes')
      res.json({ success: false, error: 'Document with the same name already exists' });
    } else {
      await db.collection("ECG").doc(name).set({ points });
      res.status(200).json({ success: true, chartId: name });
    }
  } catch (err) {
    console.error(err);
    res.status(500).json({ success: false, error: 'Internal Server Error' });
  }
});
```

Figure 11-16: NodeJS saving specific plot in database.

This POST API gets called, whenever the user on the frontend presses the button save to cloud, immediately the ECG points on the UI will be sent alongside a name that the user chooses (see the frontend section). As you can see in the figure above, we first extract the name and the points, then we go to the DB and check if the name was already assigned to another plot, if not we save the points with the name into the database.

11.3.8 Get names of ECG saved data:

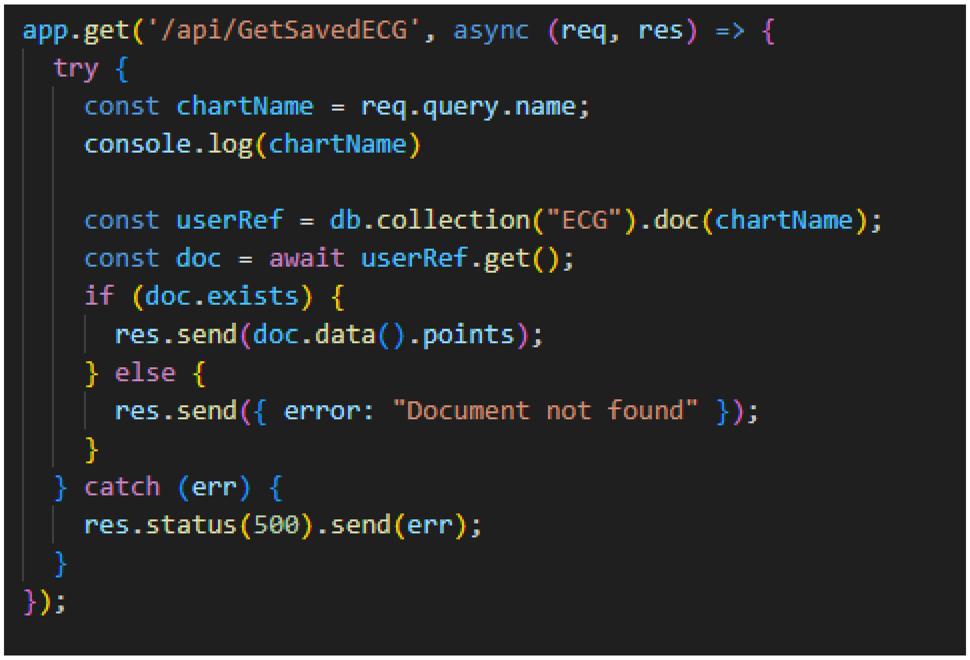


```
app.get('/api/GetAllSavedECGNames', async (req, res) => {

  console.log('namesReached')
  try {
    const usersRef = db.collection("ECG");
    const response = await usersRef.get();

    const docNames = response.docs.map(doc => doc.id);
    res.send(docNames);
  } catch (err) {
    res.status(500).send(err);
  }
});
```

Figure 11-17: NodeJS sending list of names of saved ECGs to Angular.

This figure shows a get API that aims to get all the names of all ECGs available in database. It is used by our front end, in the plots section when the user wants to visualize a saved ECG.

11.3.9 Get ECG saved data:



```
app.get('/api/GetSavedECG', async (req, res) => {
  try {
    const chartName = req.query.name;
    console.log(chartName)

    const userRef = db.collection("ECG").doc(chartName);
    const doc = await userRef.get();
    if (doc.exists) {
      res.send(doc.data().points);
    } else {
      res.send({ error: "Document not found" });
    }
  } catch (err) {
    res.status(500).send(err);
  }
});
```

Figure 11-18: NodeJS sending a selected saved ECG to angular.

In the previous figure, the api was used to show all the names of ECGS. Now when the user press on a certain ECG name, the plot should appear in the UI, so this API, extracts the name requested, go to the database, return the ECG points associated with this name, then the fronted (angular) uses those points and plots them.

12Frontend Software Implementation

12.1Flow chart:

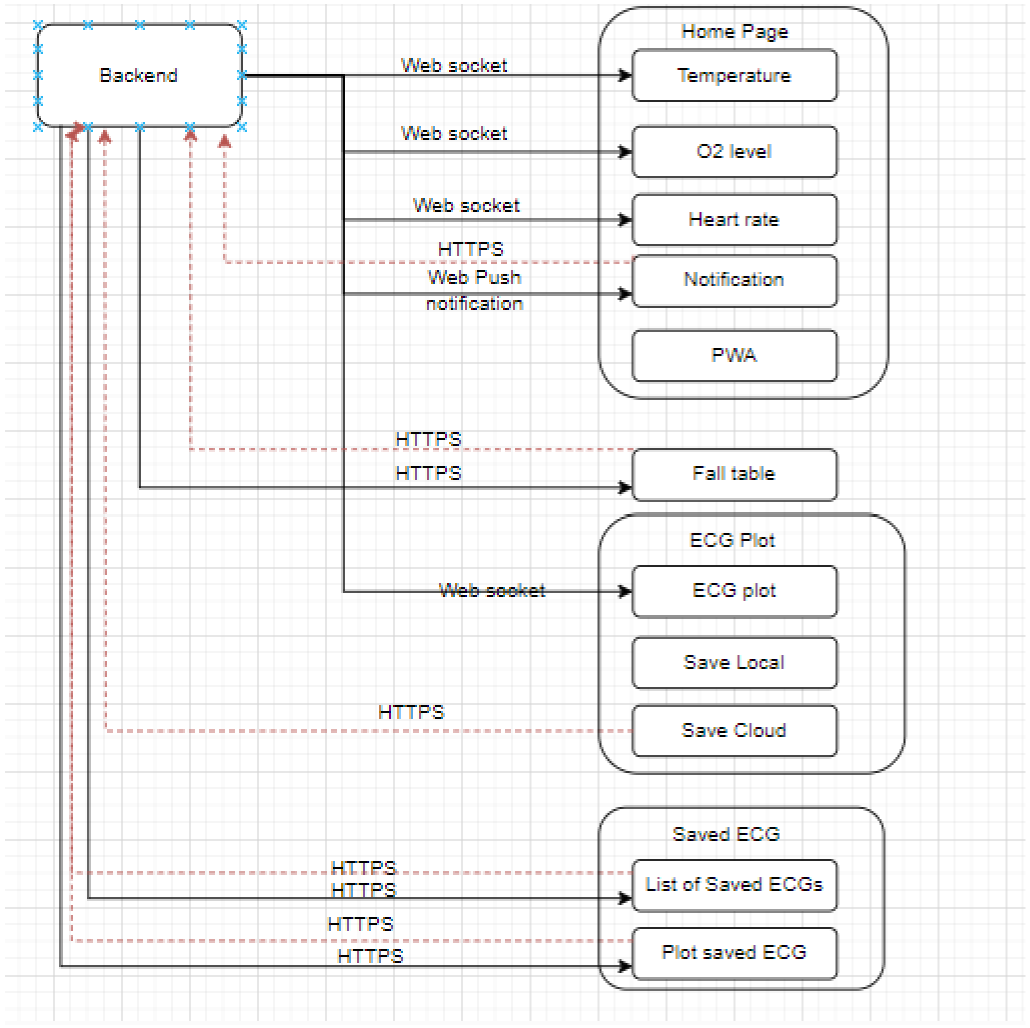


Figure 12-1: Front end flow chart

In this section, we will follow the flow chart above and explain each component and its subcomponents. We will delve into the logic (TypeScript) more than the HTML and CSS, which are used for markup and design. But before we begin, let's start with a brief explanation about Angular:

Angular is a popular framework for building web applications using TypeScript, HTML, and CSS. It is based on the concept of components, which are reusable pieces of UI that can interact with data and logic. Components can be nested inside other components to create complex and dynamic web pages.

Each component consists of:

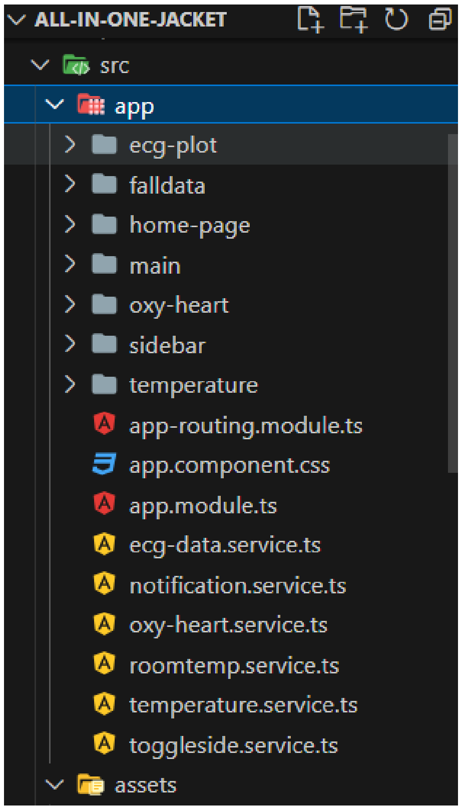
- An HTML template that declares what renders on the page
- A TypeScript class that defines behavior
- A CSS selector that defines how the component is used in a template

https://angular.io/guide/component-overview

In our project, we have the following components, as seen in the flow chart:

1. Home page
2. Fall data
3. ECG plotter
4. ECG saved data
5. Notification

As we can see in the following snippet of the file repository, we have multiple components in addition to some other files.



```
ALL-IN-ONE-JACKET
  src
    app
      ecg-plot
      falldata
      home-page
      main
      oxy-heart
      sidebar
      temperature
      app-routing.module.ts
      app.component.css
      app.module.ts
      ecg-data.service.ts
      notification.service.ts
      oxy-heart.service.ts
      roomtemp.service.ts
      temperature.service.ts
      toggleside.service.ts
    assets
```

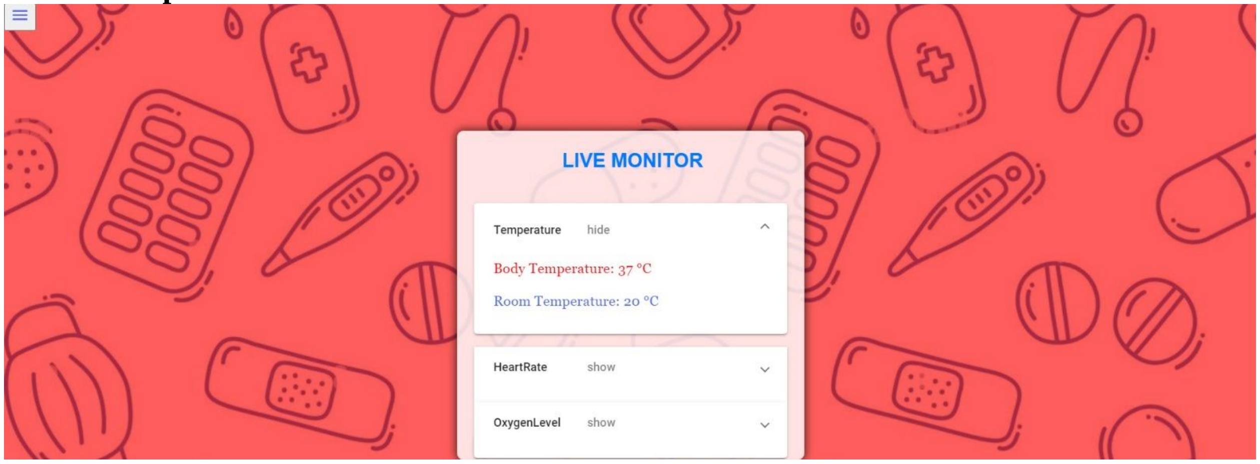
Figure 12-2: Filles of angular

This picture, shows the main components of our application, the app-routing.module.ts manage the routing between our different components like to go from home-page to the ecg-plot section… In addition, we have several services created to serve the purpose of listening to the data being monitored by the sensors.

Each component can have multiple elements. For example, the home page includes temperature, oxygen level, and heart rate.

12.2Home page component:

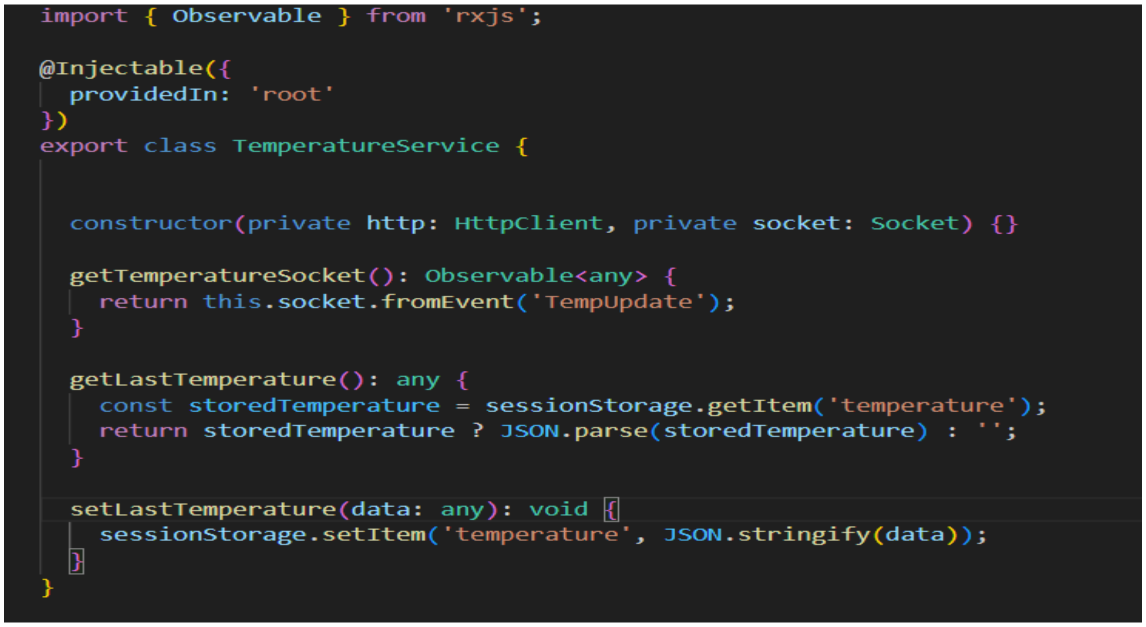
12.2.1 Temperature:



Body Temperature: 37 °C
Room Temperature: 20 °C

Figure 12-3: Temperature webpage

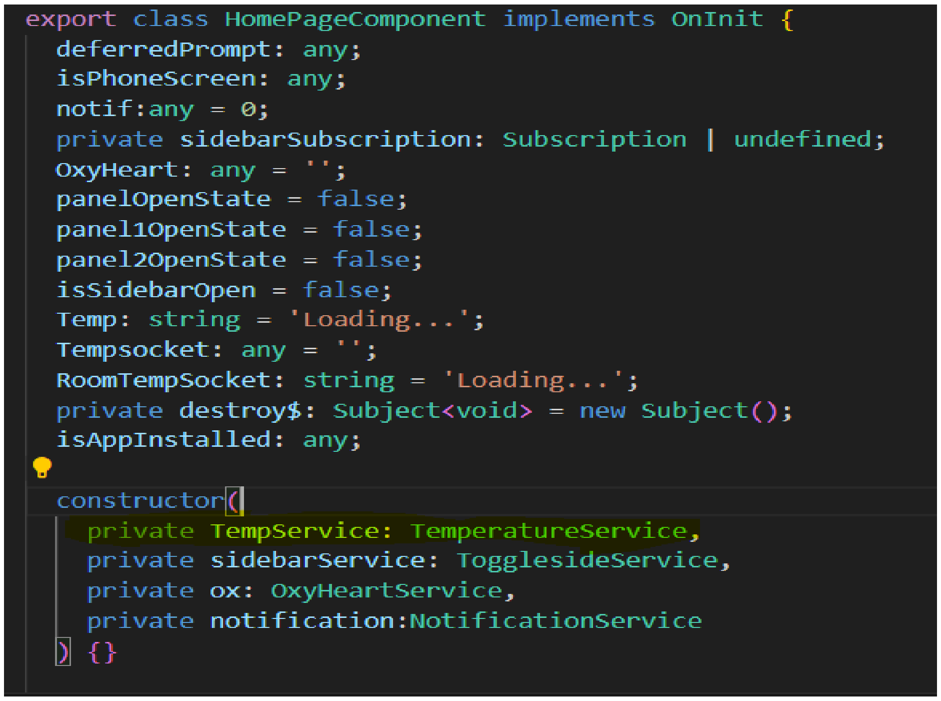
Let’s start with the temperature. We start by getting the temperature data in the temperature service according to the following:



```
import { Observable } from 'rxjs';

@Injectable({
  providedIn: 'root'
})
export class TemperatureService {


  constructor(private http: HttpClient, private socket: Socket) {}

  getTemperatureSocket(): Observable<any> {
    return this.socket.fromEvent('TempUpdate');
  }

  getLastTemperature(): any {
    const storedTemperature = sessionStorage.getItem('temperature');
    return storedTemperature ? JSON.parse(storedTemperature) : '';
  }

  setLastTemperature(data: any): void {
    sessionStorage.setItem('temperature', JSON.stringify(data));
  }
}
```

Figure 12-4: Angular code for temperature service.

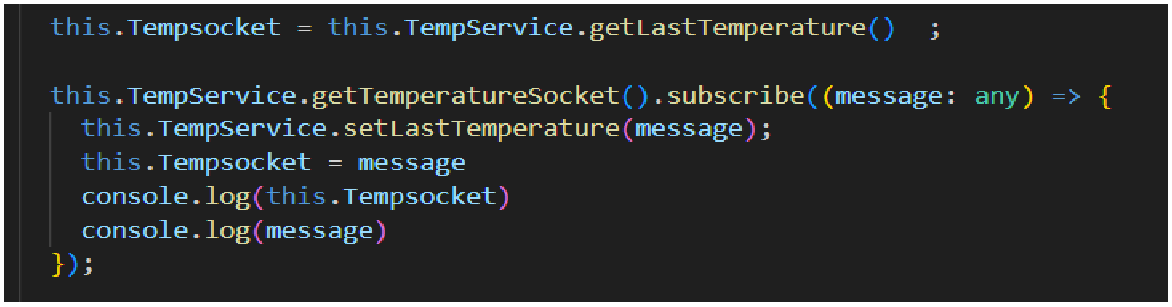
In the above code, we are listening to the WebSocket event named "TempUpdate," as explained in the backend section. This event continuously listens for any temperature update coming from the Arduino and passing by our server. Once the data is received in the service, it is directly saved to the session storage of the user, since as we know the data is sent ever 30 seconds, and if the user left the homepage and came back, we don't want the temperature data to disappear and wait for another 30 secs for it to show back. Now this temperature service is ready to be called and subscribed to by any component in the project. We will show how we benefited from it in the "homepage.ts" file which is the home component. The following snippet of code illustrates the data handling process.



```
export class HomePageComponent implements OnInit {
  deferredPrompt: any;
  isPhoneScreen: any;
  notif:any = 0;
  private sidebarSubscription: Subscription | undefined;
  OxyHeart: any = '';
  panelOpenState = false;
  panel1OpenState = false;
  panel2OpenState = false;
  isSidebarOpen = false;
  Temp: string = 'Loading...';
  Tempsocket: any = '';
  RoomTempSocket: string = 'Loading...';
  private destroy$: Subject<void> = new Subject();
  isAppInstalled: any;

  constructor(
    private TempService: TemperatureService,
    private sidebarService: TogglesideService,
    private ox: OxyHeartService,
    private notification:NotificationService
  ) {}
```

Figure 12-5: Angular code of initializing services.

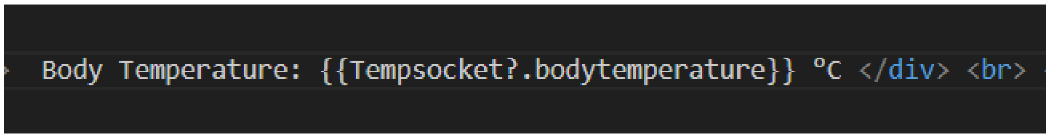
So, as you can see, highlighted in yellow, in the constructor, is how we first injected our TempService into the homepage component we want to use (of course after we have imported it). Now it's time to use this service to show the temperature.



```
this.Tempsocket = this.TempService.getLastTemperature()  ;

this.TempService.getTemperatureSocket().subscribe((message: any) => {
  this.TempService.setLastTemperature(message);
  this.Tempsocket = message
  console.log(this.Tempsocket)
  console.log(message)
});
```

Figure 12-6: Angular code to get temperature from service and make it available for HTML.

Clearly, we are using the getLastTemperature() to save the object it returns into the variable Tempsocket. (Remember, the socket from node.js sent the temperature as an object containing { bodytemperature, roomtemperature }) Now thanks to angular's super feature, we can bind these variable changes to html so whenever the value is updated, it gets reflected on the UI. For brevity we will only show how we bind it in html in the figure below:

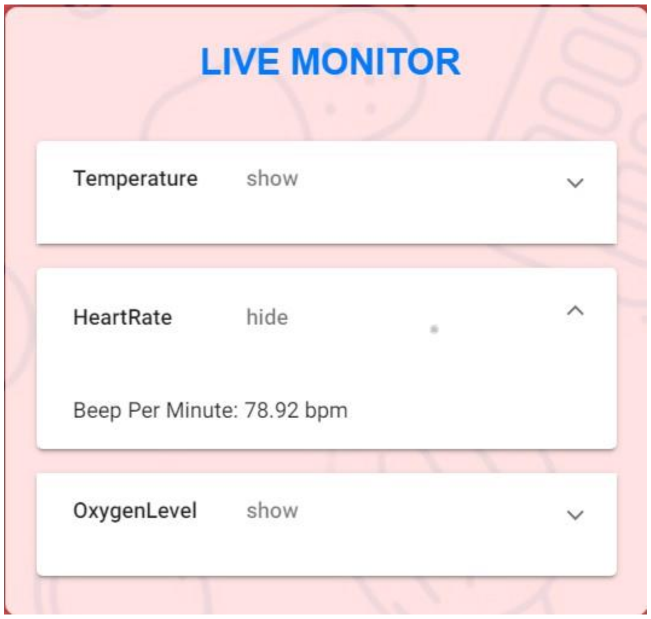


```
Body Temperature: {{Tempsocket?.bodytemperature}} °C </div> <br>
```

Figure 12-7: Angular html code for temperature.

Where here we are showing the bodytemperature in the same manner we are showing the roomtemperature, oxygen level …..

12.2.2 Oxygen level and heart rate:


LIVE MONITOR

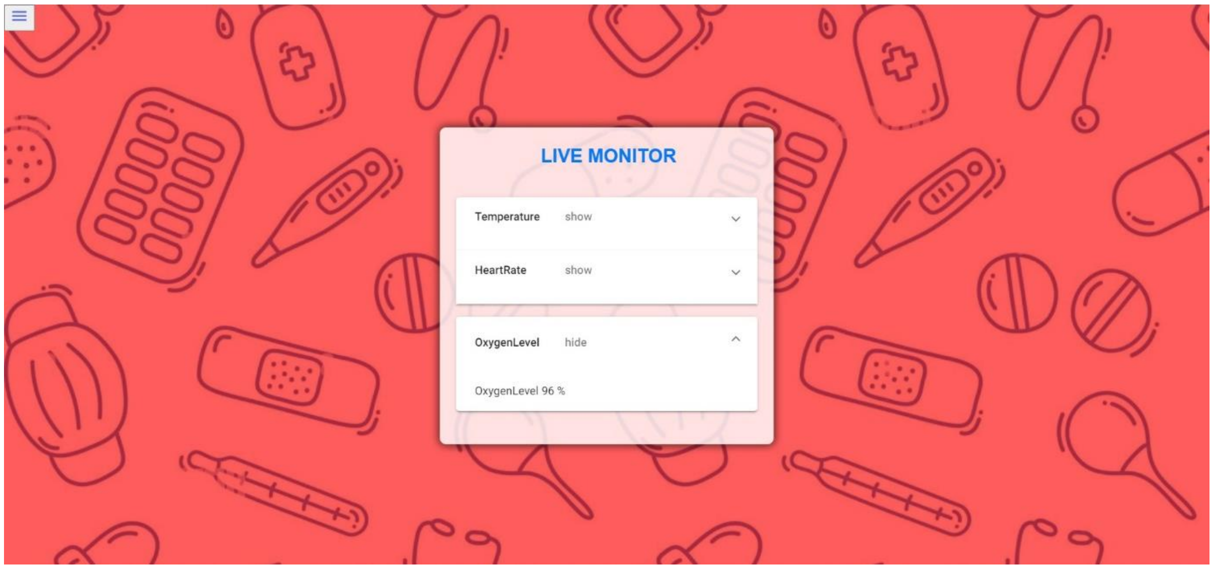
Temperature show

HeartRate hide

Beep Per Minute: 78.92 bpm

OxygenLevel show

Figure 12-8: Heart Rate webpage


LIVE MONITOR

Temperature show

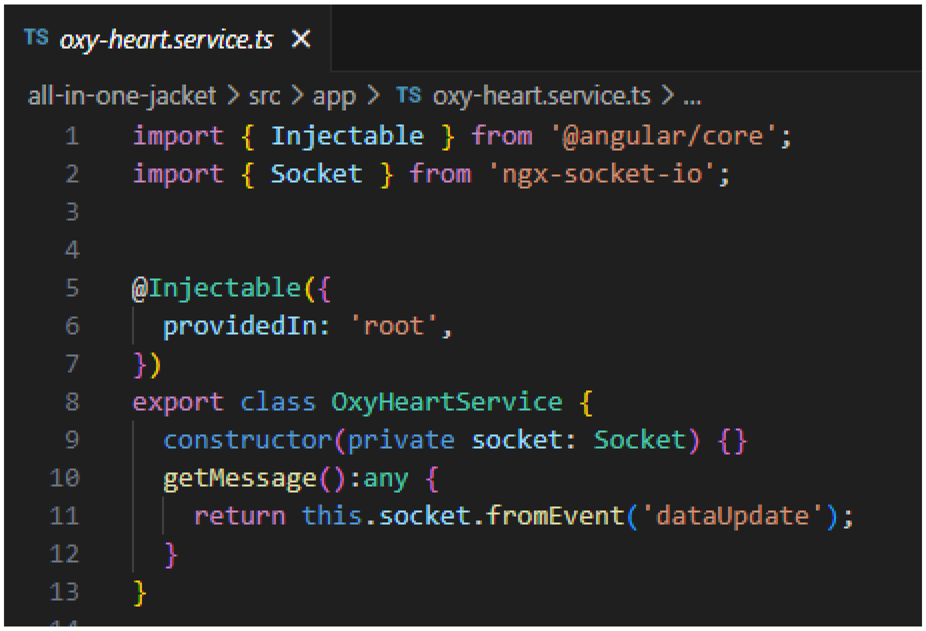
HeartRate show

OxygenLevel hide

OxygenLevel 96 %

Figure 12-9: Oxygen level webpage

In the same way, the oxygen level and heart rate are handled. The service receives the data from the web socket, and then it is passed to the homepage.ts where it is visualized. For the sake of brevity, we will only include the code for the oxygen level. The entire code will be available in the GitHub repository link.

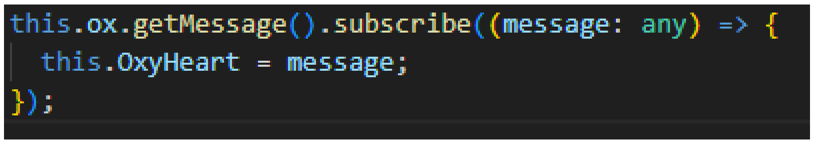


oxy-heart.service.ts

all-in-one-jacket > src > app > oxy-heart.service.ts > ...

```
import { Injectable } from '@angular/core';
import { Socket } from 'ngx-socket-io';


@Injectable({
  providedIn: 'root',
})
export class OxyHeartService {
  constructor(private socket: Socket) {}
  getMessage():any {
    return this.socket.fromEvent('dataUpdate');
  }
}
```

Figure 12-10: Angular code for oxygen level service.



```
this.ox.getMessage().subscribe((message: any) => {
  this.OxyHeart = message;
});
```

Figure 12-11: Angular code to get oxygen level from service.

And then we visualize the OxygenLevel and heart Rates on the UI.

12.2.3 Notification:

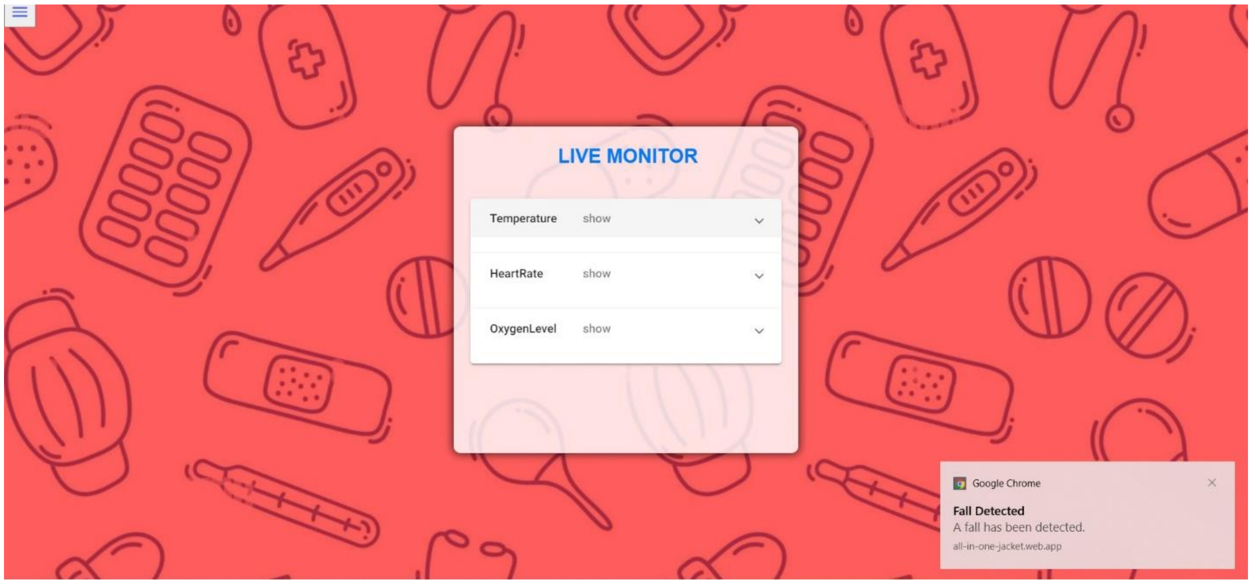


Figure 12-12: Notification

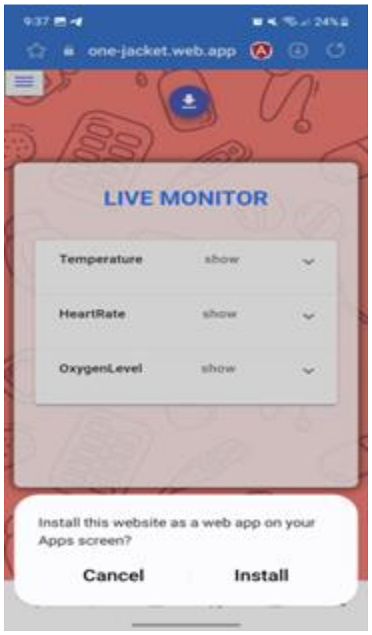
This part is already explained in the chapter of backend.

12.2.4 Angular Progressive web App (PWA):

Quick overview about PWA: Progressive Web Apps (PWAs) are like a new kind of app that uses modern web technology to make websites feel more like the apps you use on your phone. They let you do things on websites that usually only apps can do, such as getting notifications, using the site even when you're offline, and connecting with your device's hardware. As more and more companies start making PWAs, it's important to keep up with what's happening in this technology.

Research indicates that the market for progressive web applications (PWAs) is predicted to be worth around 10.77 billion dollars by 2027, showing a remarkable yearly growth rate of over 30%. Additionally, Google reports a significant increase of 270% in desktop installations of PWAs from the beginning of 2021.

We highly prioritize user experience on mobile devices, and we understand the convenience and accessibility that a dedicated app can offer. To enhance this experience, we've developed a Progressive Web App (PWA) using Angular. PWAs provide a seamless and responsive experience, ensuring users can access our services effortlessly on their phones. One of the key features of our PWA is the implementation of service workers, allowing users to enjoy a reliable and fast experience, even in offline mode. Additionally, to make it even more convenient for our users, we've added an install button icon, making it easy to add our PWA to phone's home screen for quick access. As you can see in the figure below:



Install this website as a web app on your Apps screen?

Figure 12-13: Installing app webpage.

When we open the app, there is an install button, if installed the PWA will be installed and appear exactly like any other app, and this can be done on both ANDROID and IOS that’s its power.

12.3Fall data table:

12.3.1 Select start date:

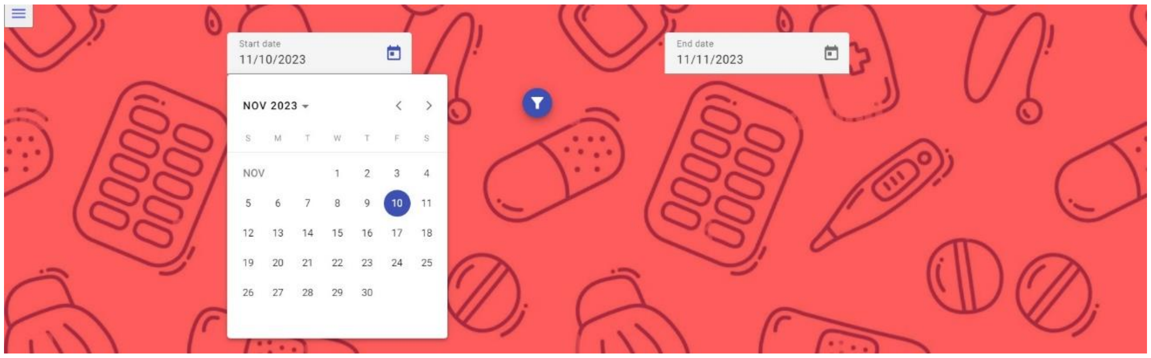
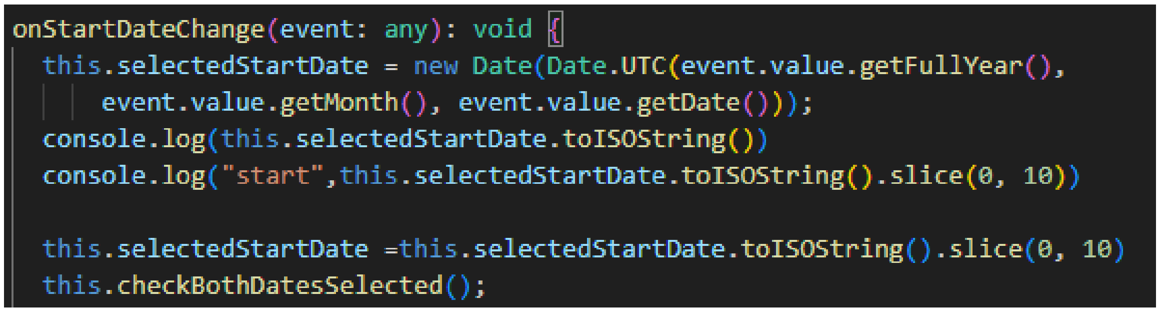


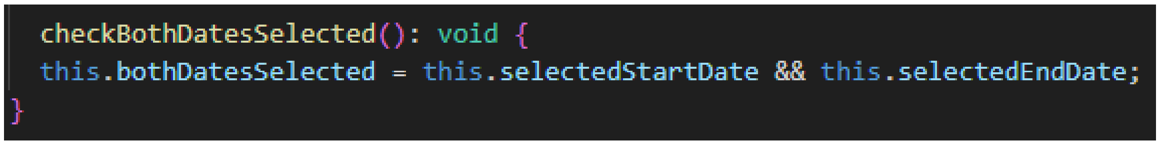
Figure 12-14: Select start date webpage.



```
onStartDateChange(event: any): void {
  this.selectedStartDate = new Date(Date.UTC(event.value.getFullYear(),
      event.value.getMonth(), event.value.getDate()));
  console.log(this.selectedStartDate.toISOString())
  console.log("start",this.selectedStartDate.toISOString().slice(0, 10))

  this.selectedStartDate =this.selectedStartDate.toISOString().slice(0, 10)
  this.checkBothDatesSelected();
```

Figure 12-15: Angular code to select start date.

In the above function, when the start date is selected, a manipulation is done to transform the format of the selected data to the proper format needed to fetch the database. We start by getting the date using the Date.UTC(event.value.getFullYear(), event.value.getMonth(), vent.value.getDate()), this line of code reads the value of the event that is triggered when a date is selected. Then the date is saved into a variable named selectedStartDate. After that a call is done to the function checkBothDatesSelected() which has a role of saving the start date and end date in a variable:

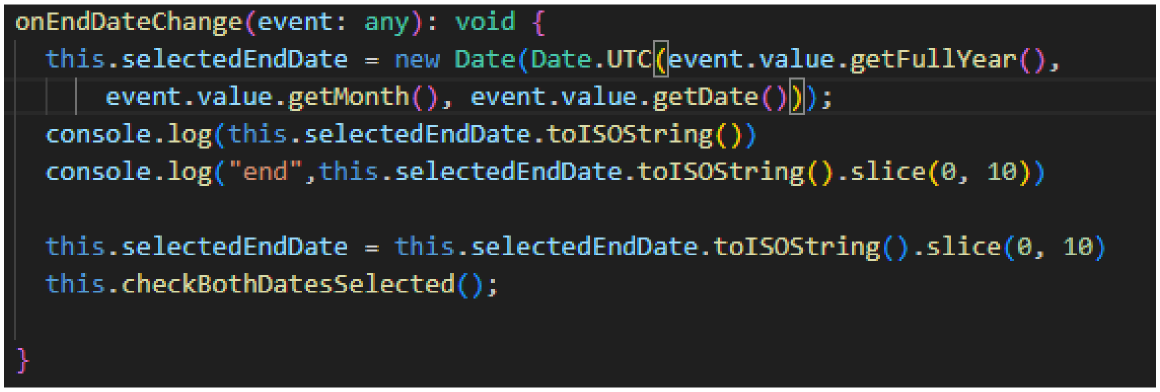


```
  checkBothDatesSelected(): void {
  this.bothDatesSelected = this.selectedStartDate && this.selectedEndDate;
}
```

Figure 12-16: Angular code to check selected dates.

12.3.2 Select end date:

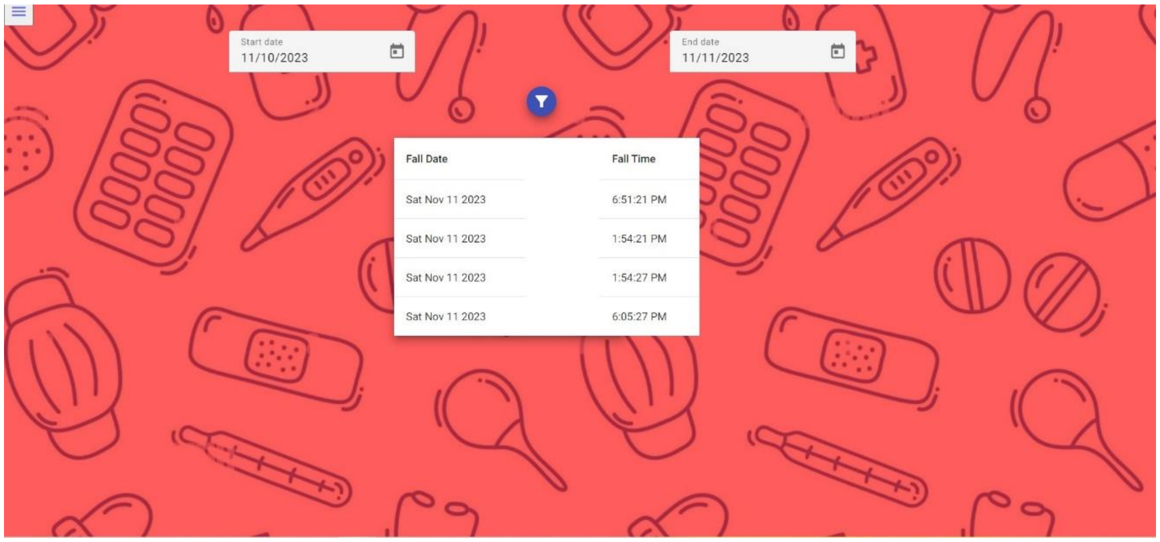
In the same exact way the end date is selected, and saved using the function checkBothDatesSelected().



```
onEndDateChange(event: any): void {
  this.selectedEndDate = new Date(Date.UTC(event.value.getFullYear(),
      event.value.getMonth(), event.value.getDate()));
  console.log(this.selectedEndDate.toISOString())
  console.log("end",this.selectedEndDate.toISOString().slice(0, 10))

  this.selectedEndDate = this.selectedEndDate.toISOString().slice(0, 10)
  this.checkBothDatesSelected();

}
```

Figure 12-17: Angular code to select end date.

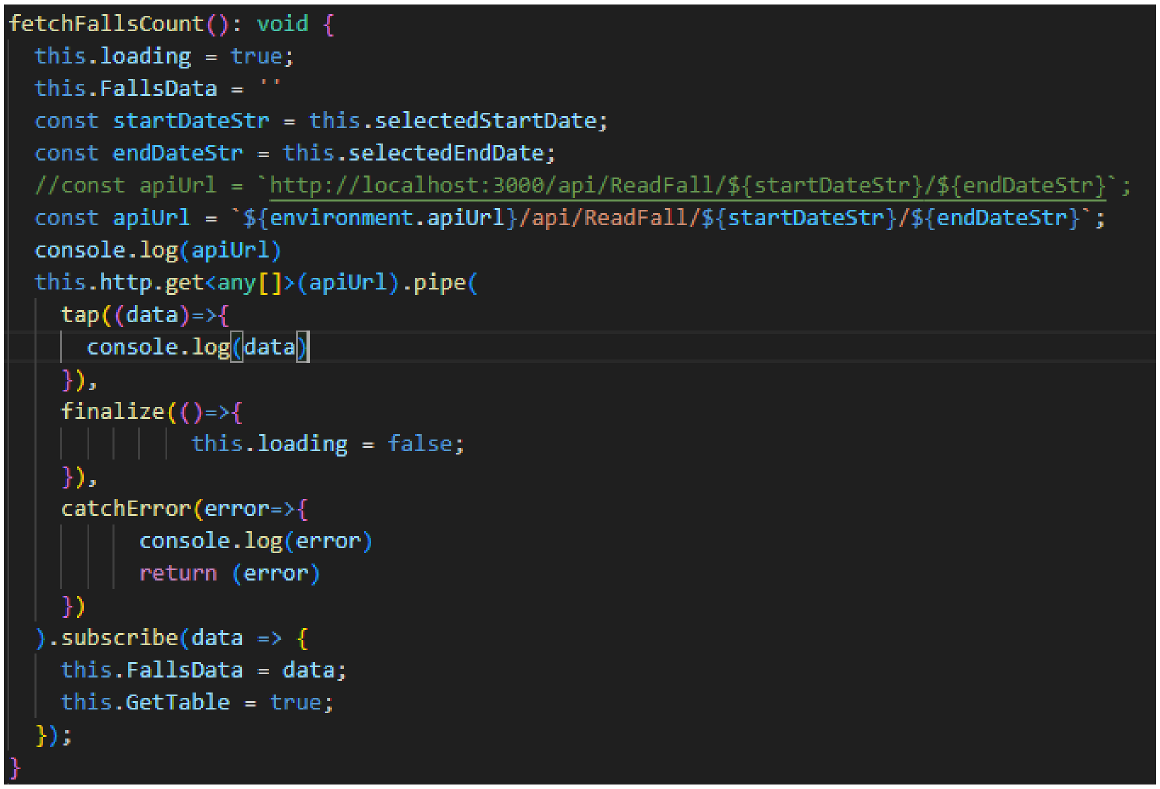
12.3.3 Fetch Falls data:



Fall Date	Fall Time
Sat Nov 11 2023	6:51:21 PM
Sat Nov 11 2023	1:54:21 PM
Sat Nov 11 2023	1:54:27 PM
Sat Nov 11 2023	6:05:27 PM

Figure 12-18: Fall table webpage.

After the user click on the button to get the data, a function called fetchFallsCount() is called:



```
fetchFallsCount(): void {
  this.loading = true;
  this.FallsData = ''
  const startDateStr = this.selectedStartDate;
  const endDateStr = this.selectedEndDate;
  //const apiUrl = `http://localhost:3000/api/ReadFall/${startDateStr}/${endDateStr}`;
  const apiUrl = `${environment.apiUrl}/api/ReadFall/${startDateStr}/${endDateStr}`;
  console.log(apiUrl)
  this.http.get<any[]>(apiUrl).pipe(
    tap((data)=>{
      console.log(data)
    }),
    finalize(()=>{
              this.loading = false;
    }),
    catchError(error=>{
          console.log(error)
          return (error)
    })
  ).subscribe(data => {
    this.FallsData = data;
    this.GetTable = true;
  });
}
```

Figure 12-19: Angular code to get list of falls.

As explained in the backend, an HTTPS request is sent to the server with the start and end dates, and a response with the table is received. In the above function, we pass the parameters to the API URL responsible for the HTTPS request:

```
const apiUrl = `${environment.apiUrl}/api/ReadFall/${startDateStr}/${endDateStr}`;
```

After filling the data, the HTTPS GET request is sent using this.http.get<any[]>(apiUrl). Upon sending the request, we wait for your response. The response is read using the subscribe(data…) method, and the data is stored in an array. The flag GetTable is used so that when the data is received, it is outputted on the website.

12.4ECG plot:

12.4.1 Plot ECG:

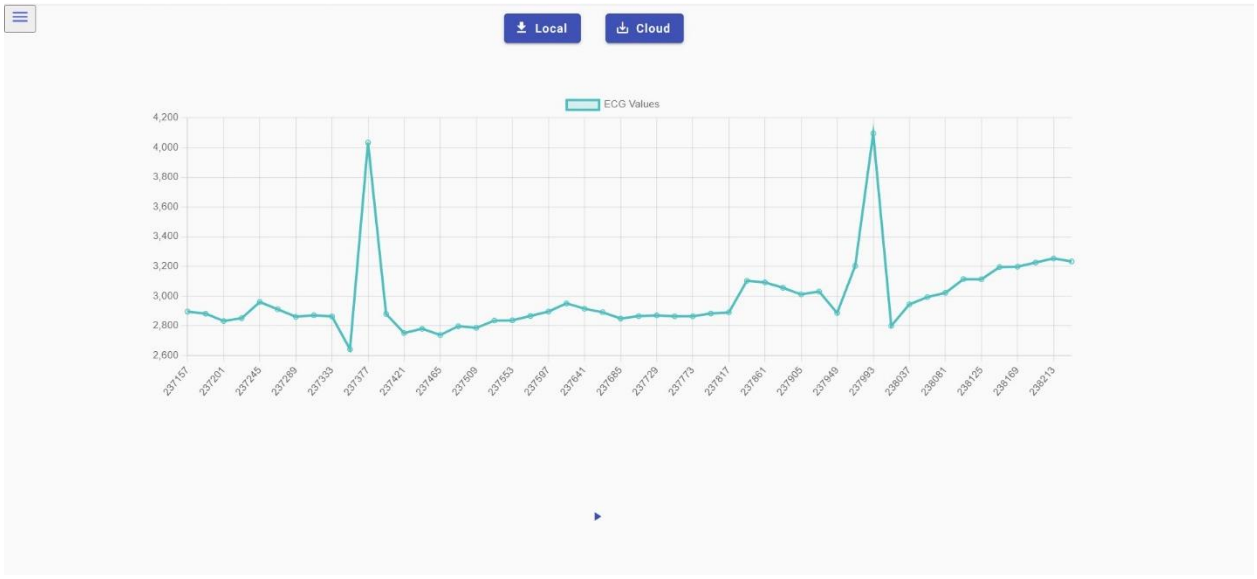
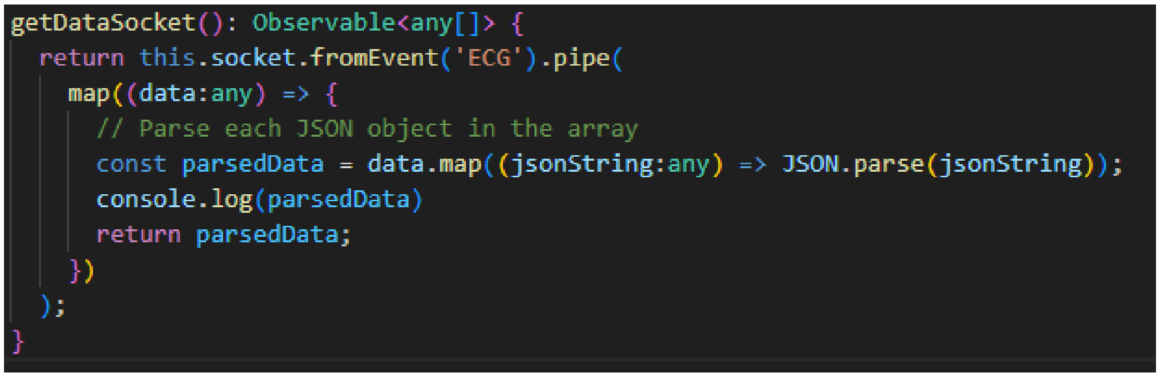


Figure 12-20: ECG Plot webpage.

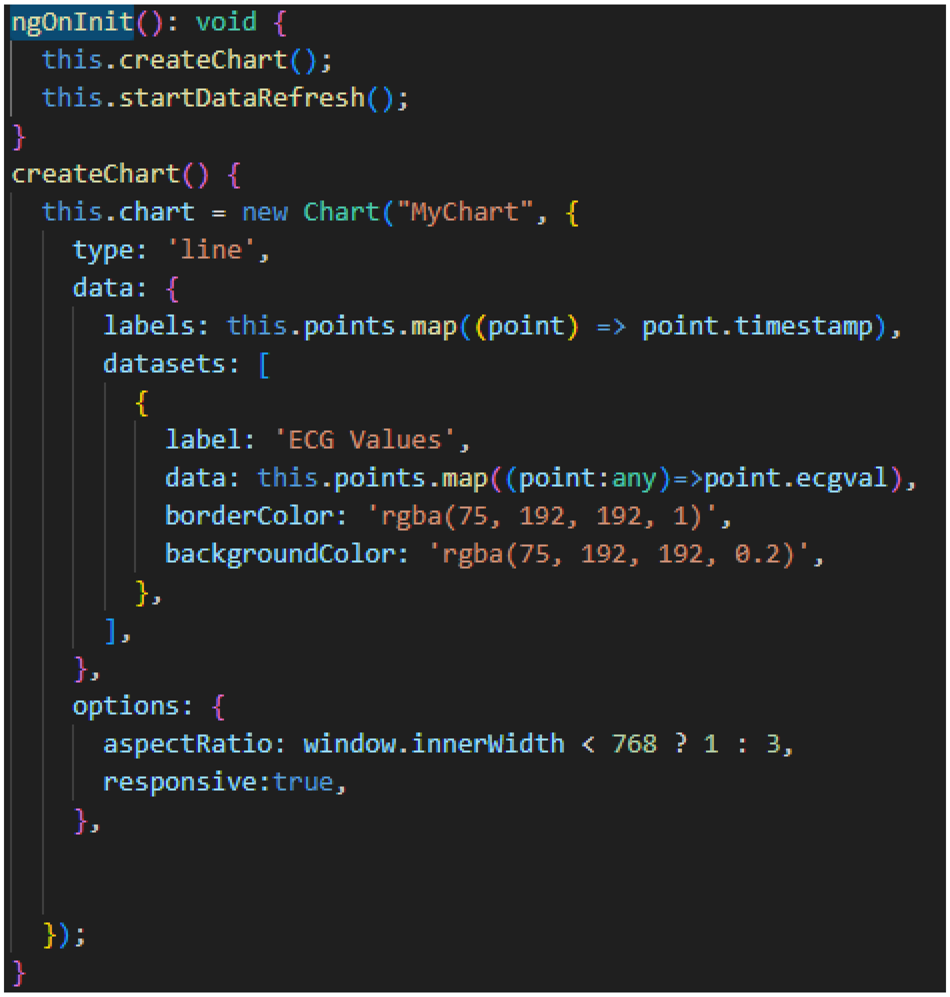
In this component, data is received using a socket connection, as seen in the backend chapter. The process begins in the ecg-data.service.ts file, where the data is received in the form of JSON. Subsequently, it is returned to the ecg.component.ts.



```
getDataSocket(): Observable<any[]> {
  return this.socket.fromEvent('ECG').pipe(
    map((data:any) => {
      // Parse each JSON object in the array
      const parsedData = data.map((jsonString:any) => JSON.parse(jsonString));
      console.log(parsedData)
      return parsedData;
    })
  );
}
```

Figure 12-21: angular code to receive ECG data.

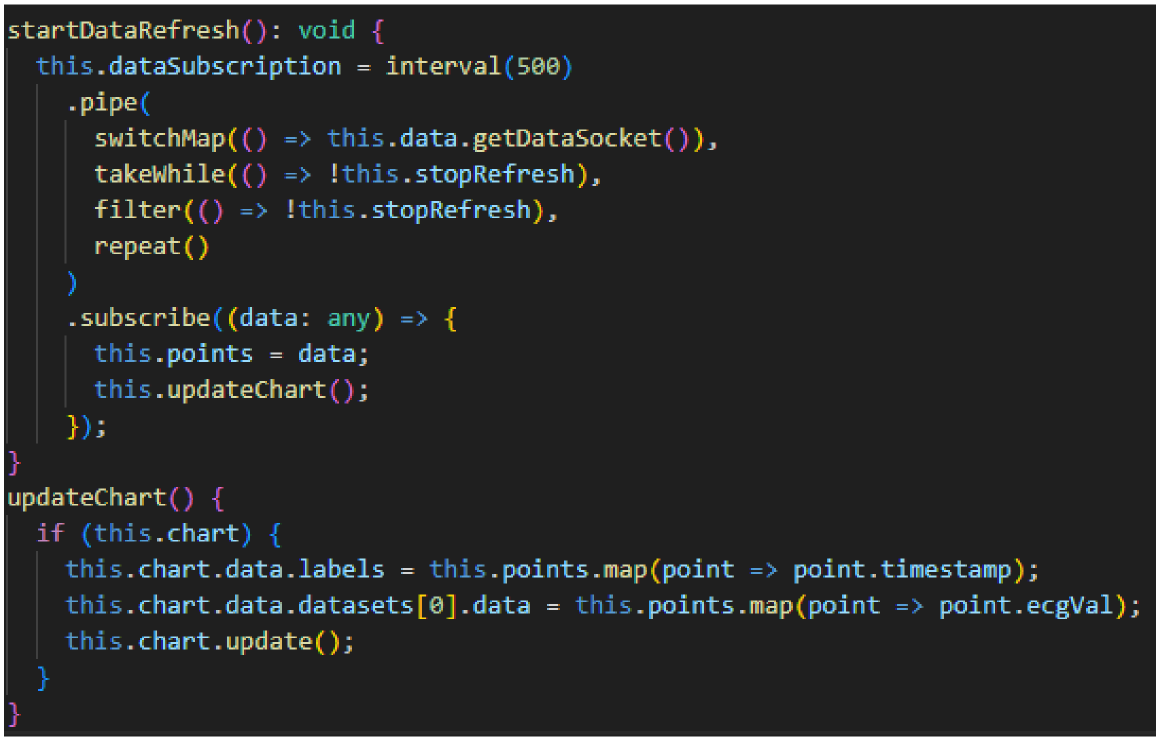
In the component, the ngOnInit() function is invoked when the page is opened, serving as the constructor of every component. This function, in turn, calls the createChart() and startDataRefresh() functions. The createChart() function utilizes a built-in chart library to facilitate the visualization of the plot.



```
ngOnInit(): void {
  this.createChart();
  this.startDataRefresh();
}
createChart() {
  this.chart = new Chart("MyChart", {
    type: 'line',
    data: {
      labels: this.points.map((point) => point.timestamp),
      datasets: [
        {
          label: 'ECG Values',
          data: this.points.map((point:any)=>point.ecgval),
          borderColor: 'rgba(75, 192, 192, 1)',
          backgroundColor: 'rgba(75, 192, 192, 0.2)',
        },
      ],
    },
    options: {
      aspectRatio: window.innerWidth < 768 ? 1 : 3,
      responsive:true,
    },


  });
}
```

Figure 12-22: Angular code to plot data.

Following that, the startDataRefresh() function is invoked. This function serves the purpose of updating the data used for visualizing the chart. It waits for 500 milliseconds and then utilizes the subscription to the Socket.IO, previously explained, to retrieve the data. Subsequently updating the array. The presence of the flag stopRefresh will be elucidated in a later section. Afterward, the function calls updateChart(), which is responsible for updating the chart using the newly acquired data and then recalling startDataRefresh() function. Thus, entering an infinite loop of updating the chart data and the chart for every 500 milliseconds.

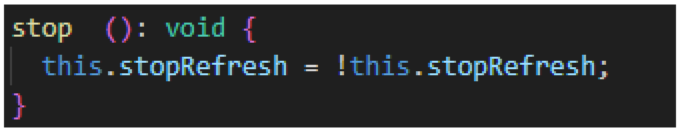


```
startDataRefresh(): void {
  this.dataSubscription = interval(500)
    .pipe(
      switchMap(() => this.data.getDataSocket()),
      takeWhile(() => !this.stopRefresh),
      filter(() => !this.stopRefresh),
      repeat()
    )
    .subscribe((data: any) => {
      this.points = data;
      this.updateChart();
    });
}
updateChart() {
  if (this.chart) {
    this.chart.data.labels = this.points.map(point => point.timestamp);
    this.chart.data.datasets[0].data = this.points.map(point => point.ecgVal);
    this.chart.update();
  }
}
```

Figure 12-23: Angular code to check for new data.

12.4.2 Stop refresh:

Stop or pause button, when clicked, is used to pause the refresh of the chart. This is done to be able to see the chart carefully, download the current chart, or save the chart to the database for later use.



```
stop  (): void {
  this.stopRefresh = !this.stopRefresh;
}
```

Figure 12-24: Angular code to pause refresh.

12.4.3 Download chart:

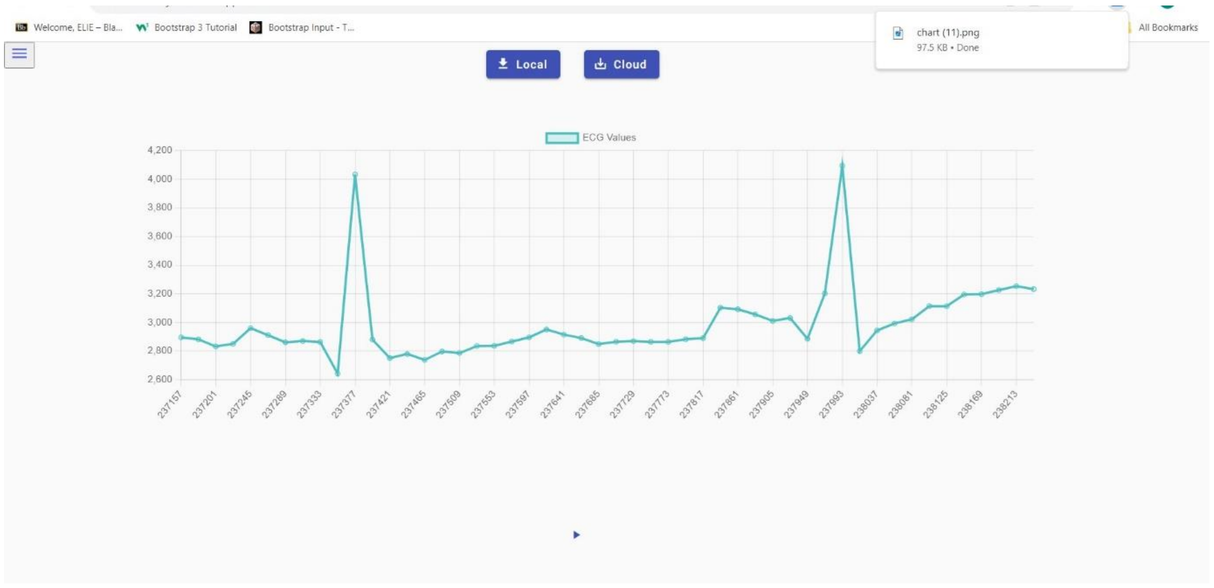
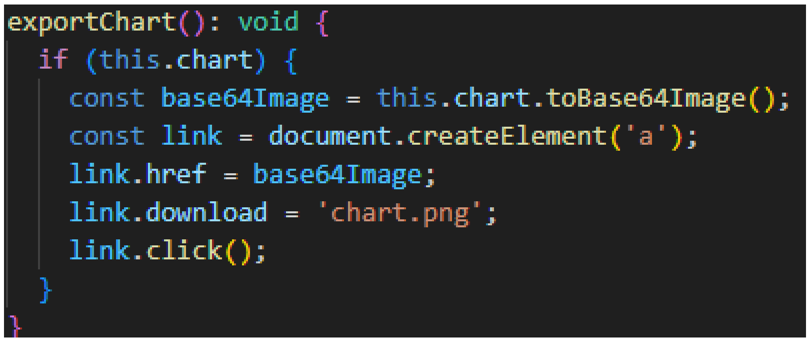


Figure 12-25: Download chart

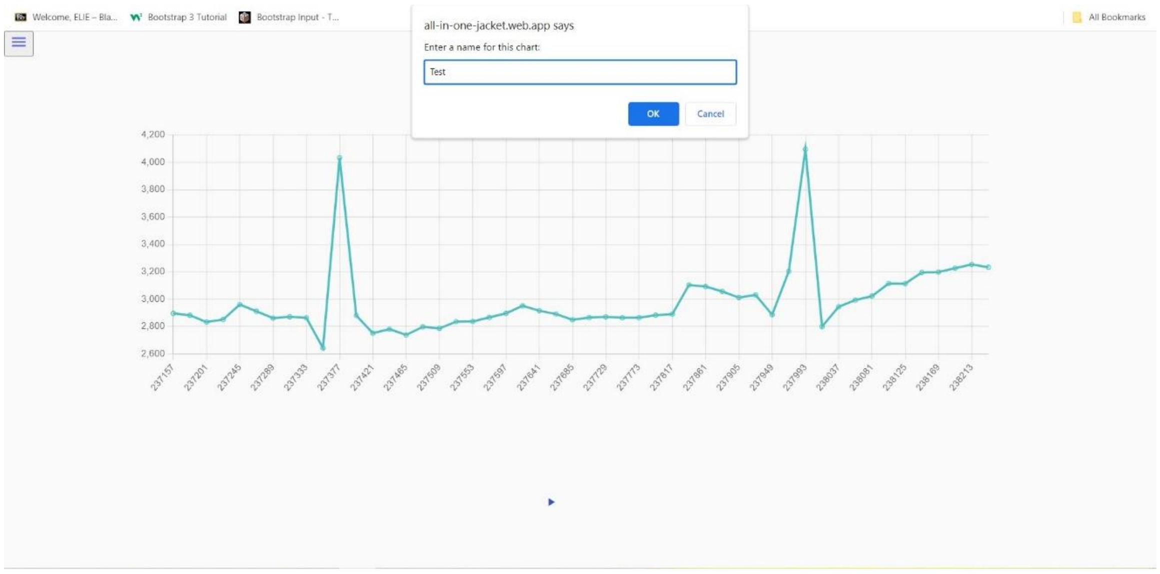
This function is called when the download local button is clicked. The function transforms the chart to an base64Image which is an image of type png. After that the download function is used to download the image to the download folder in this device.



```
exportChart(): void {
  if (this.chart) {
    const base64Image = this.chart.toBase64Image();
    const link = document.createElement('a');
    link.href = base64Image;
    link.download = 'chart.png';
    link.click();
  }
}
```

Figure 12-26: Angular code to download plot.

12.5 Save chart to database:



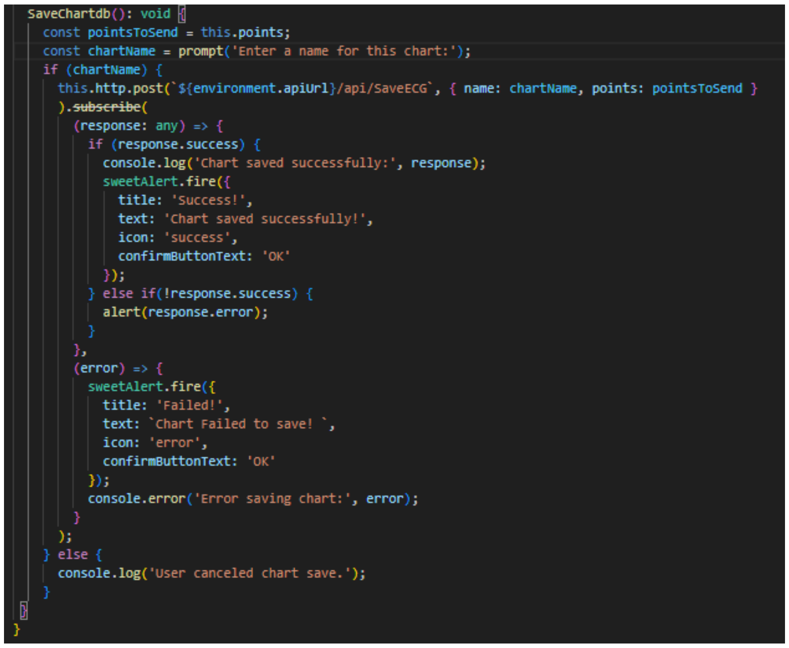
all-in-one-jacket.web.app says

Enter a name for this chart:

Test

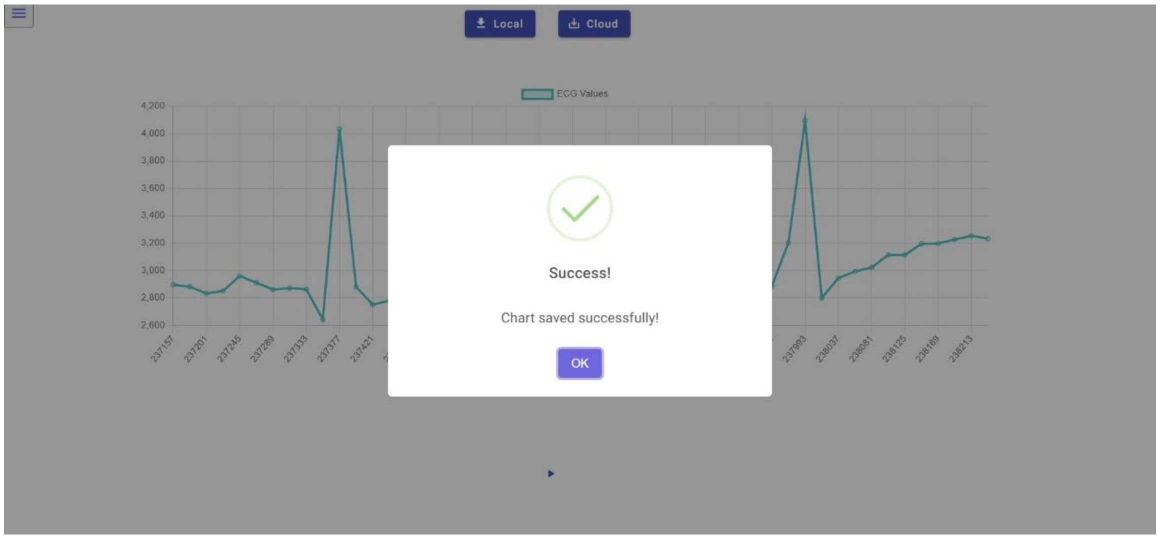
Figure 12-27: Select name for ECG to save

This function is called when the save to cloud button is clicked. The function gets the points to be sent. In addition, a prompt is displayed so the user can enter the name of the chart. Then, using the https post request the data is sent to the server, the server handles the data, explained in chapter backend. After handling the data, the reply to the frontend with a status code. After that, using the sweetAlert library an alert is done to display if the data is stored successfully or not, according to the status code.



```
  SaveChartdb(): void {
    const pointsToSend = this.points;
    const chartName = prompt('Enter a name for this chart:');
    if (chartName) {
      this.http.post(`${environment.apiUrl}/api/SaveECG`, { name: chartName, points: pointsToSend }
      ).~~subscribe~~(
        (response: any) => {
          if (response.success) {
            console.log('Chart saved successfully:', response);
            sweetAlert.fire({
              title: 'Success!',
              text: 'Chart saved successfully!',
              icon: 'success',
              confirmButtonText: 'OK'
            });
          } else if(!response.success) {
            alert(response.error);
          }
        },
        (error) => {
          sweetAlert.fire({
            title: 'Failed!',
            text: `Chart Failed to save! `,
            icon: 'error',
            confirmButtonText: 'OK'
          });
          console.error('Error saving chart:', error);
        }
      );
    } else {
      console.log('User canceled chart save.');
    }
  }
}
```

Figure 12-28: Angular code to save plot to cloud.



Success!

Chart saved successfully!

Figure 12-29: sweetAlert when success

12.6ECG saved plot:

12.6.1 List of ECG saved plots:

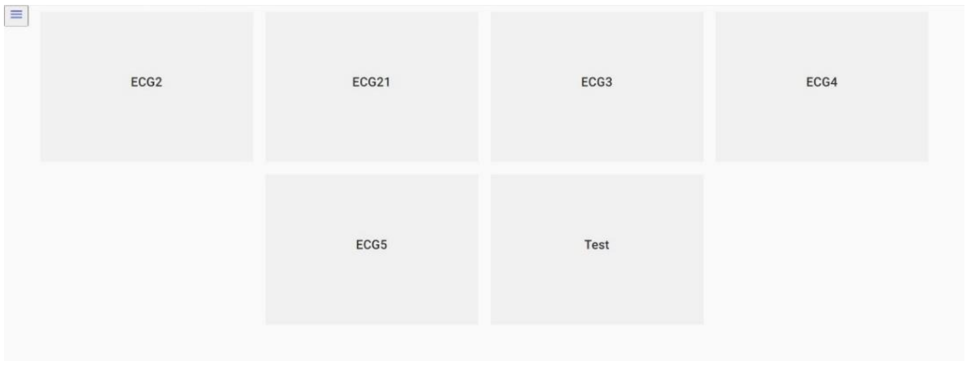
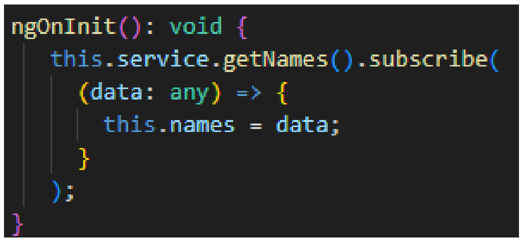


Figure 12-30: List of ECG saved plots

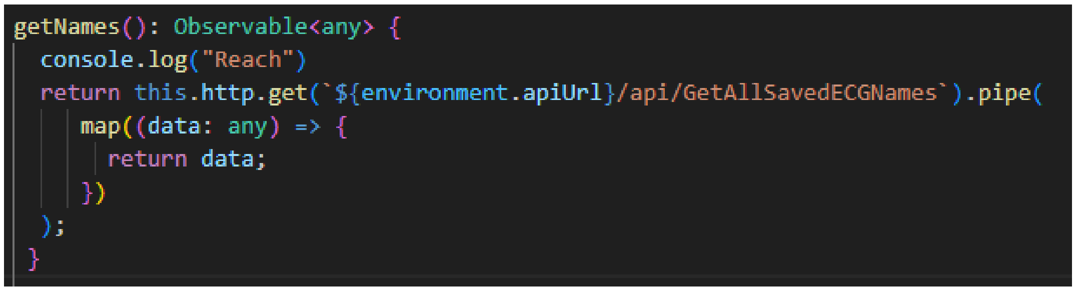
When the page is open an HTTPS Get request is sent to get the list of all the saved ECGs. This is done in the ecgplots.ts using the following code that request the data using a service.



```
ngOnInit(): void {
  this.service.getNames().subscribe(
    (data: any) => {
      this.names = data;
    }
  );
}
```

Figure 12-31: Angular ngOnInit() Function.

In the ecgplots.service.ts, a request is sent to the database using a same method as for get the fall table data.



```
getNames(): Observable<any> {
  console.log("Reach")
  return this.http.get(`${environment.apiUrl}/api/GetAllSavedECGNames`).pipe(
    map((data: any) => {
      return data;
    })
  );
}
```

Figure 12-32: Angular code to get all list of saved data.

The request is done using the http.get method. After getting the response, the data is returned to the function in the ecgplot component. Where the data is visualized using HTML and CSS.

12.6.2 ECG Saved Plot:

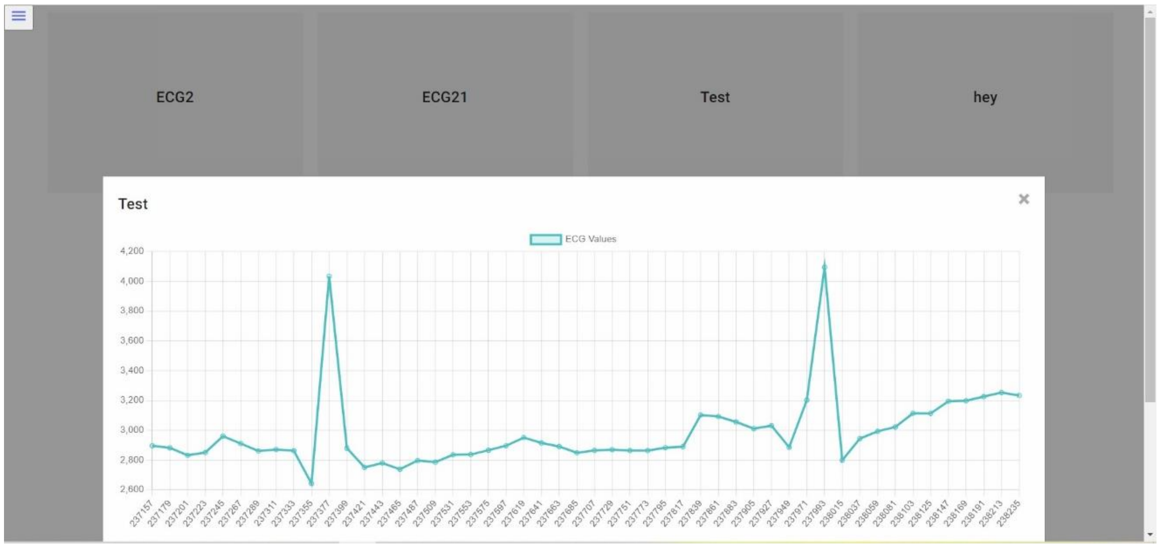
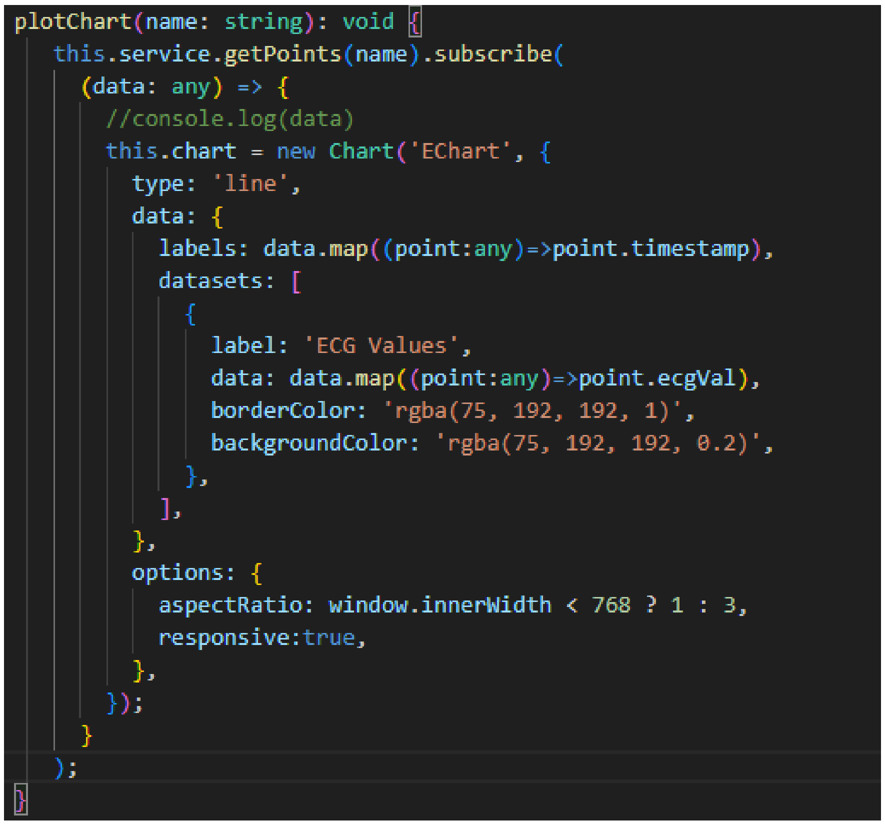


Figure 12-33: ECG saved plot.

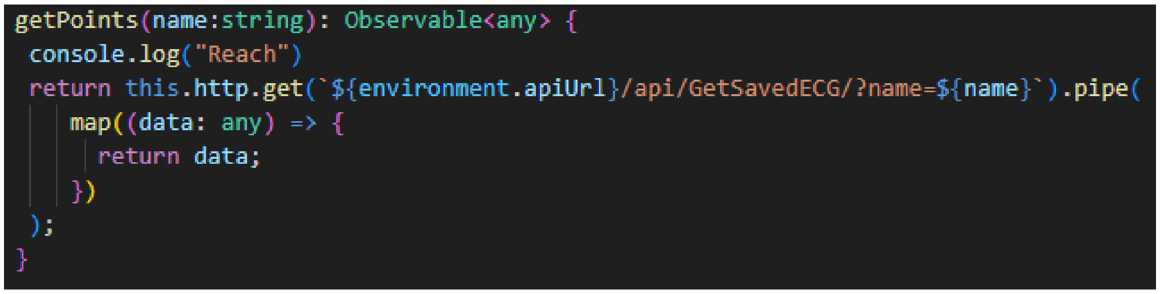
In case the user press on the component to see the saved data, this function is called in the ecgplots component:



```
plotChart(name: string): void {
    this.service.getPoints(name).subscribe(
      (data: any) => {
        //console.log(data)
        this.chart = new Chart('EChart', {
          type: 'line',
          data: {
            labels: data.map((point:any)=>point.timestamp),
            datasets: [
              {
                label: 'ECG Values',
                data: data.map((point:any)=>point.ecgVal),
                borderColor: 'rgba(75, 192, 192, 1)',
                backgroundColor: 'rgba(75, 192, 192, 0.2)',
              },
            ],
          },
          options: {
            aspectRatio: window.innerWidth < 768 ? 1 : 3,
            responsive:true,
          },
        });
      }
    );
}
```

Figure 12-34: Angular code to plot saved data.

In the above function the name of the chart to be plot is used to initiate the call to the service. The service will send the name to the backend using an HTTPS get request. As explained before, the backend uses the name to get the data from the data base and return the result in the HTTPs request.



```
getPoints(name:string): Observable<any> {
 console.log("Reach")
 return this.http.get(`${environment.apiUrl}/api/GetSavedECG/?name=${name}`).pipe(
    map((data: any) => {
      return data;
    })
 );
}
```

After the retuning of the data form the service to the component, an instance of the chart class is created, this is a library used to visualize charts. It takes the data and plot them.

13 **Conclusion**

In conclusion, the IoT was a journey we decided to take on during our senior project. The reasons were that currently the IoT is an extremely promising field expected to have around 500 billion devices connected with the Internet of Things (IoT) by 2030 as indicated by Cisco [23]. In addition, on average, about 25 to 30 percent of people aged 65 years or older experience at least one fall per year, which makes fall detection an important issue that needs to be adhered [24].

As engineers, we succeeded to integrate our hardware and software knowledge to develop a solution that can detect Fall in real time. Initially, the problem was clearly stated, and the scope of the solution was defined. The design phase began by studying of the possible IoT sensors that can be used, and this was thoroughly done during the first semester. In addition, we performed a literature review about the topic. Then, during the second semester, we began programming on Arduino, connecting the sensors, designing a prototype, and building a web application.

In conclusion, the IoT jacket project represents a significant advance in the usage of technology to enhance the safety and well-being of our elderly population. Through innovative combination of multiple IoT sensors and real-time monitoring, this project not only addresses the critical issue of fall detection but also showcases the potential of IoT in revolutionizing healthcare for seniors. By the integration of technology into everyday life, we pave the way for proactive and immediate response, allowing seniors to live independently with an added layer of security and peace of mind for them, for their relatives, and their healthcare professionals.

14 References

[1] Centers for Disease Control and Prevention, "Hip Fractures Among Older Adults," [Online]. Available: https://www.cdc.gov/falls/hip-fractures.html.

[2]A. J. Burge et al., "The cost of hip fracture in Canada: a population-based analysis of health care utilization and outcomes," Canadian Journal of Surgery, vol. 62, no. 2, pp. 113-120, Apr. 2019. [Online]. Available: https://www.mybib.com/tools/ieee-citation-generator

[3] D. C. Schairer et al., "The Association of Daily Physical Activity and Mortality After Hip Fracture," American Journal of Epidemiology, vol. 170, no. 10, pp. 1290-1296, Nov. 2009. [Online]. Available: https://www.scribbr.com/citation/generator/ieee/

[4] S. Scutti, "Hip fracture risk may be heightened by partner's death," CNN, May 13, 2019. [Online]. Available: https://edition.cnn.com/2019/05/13/health/hip-fracture-death-partner/index.html.

[5] Components101, "NodeMCU ESP8266 Pinout, Features and Datasheet," [Online]. Available: https://components101.com/development-boards/nodemcu-esp8266-pinout-features-and-datasheet.

[6] Components101, "ESP32 DevKitC Pinout, Overview, Features & Datasheet," [Online]. Available: https://components101.com/microcontrollers/esp32-devkitc.

[7] Tricog, "12-Lead ECG vs Single Lead ECG," [Online]. Available: https://www.tricog.com/12-lead-ecg-vs-single-lead-ecg

[8] ElProCus, "AD8232 ECG Sensor Working and Its Applications," [Online]. Available: https://www.elprocus.com/ad8232-ecg-sensor-working-and-its-applications/.

[9] Analog Devices, "AD8232 Heart Rate Monitor Front End Data Sheet," [Online]. Available: https://www.analog.com/media/en/technical-documentation/data-sheets/ad8232.pdf.

[10] Circuit Digest, "Understanding ECG Sensor and Program AD8232 ECG Sensor with Arduino to Diagnose Various Medical Conditions," [Online]. Available: https://circuitdigest.com/microcontroller-projects/understanding-ecg-sensor-and-program-ad8232-ecg-sensor-with-arduino-to-diagnose-various-medical-conditions.

[11] The Physiologist, "The ECG Leads, Polarity and Einthoven’s Triangle," [Online]. Available: https://thephysiologist.org/study-materials/the-ecg-leads-polarity-and-einthovens-triangle

[12] Analog Devices, "AD8232 Heart Rate Monitor Front End Product Overview," [Online]. Available: https://www.analog.com/en/products/ad8232.html#product-overview

[13] NHS Blood and Transplant, "Functions of blood: transport around the body," [Online]. Available: https://www.blood.co.uk/news-and-campaigns/the-donor/latest-stories/functions-of-blood-transport-around-the-body/

[14] Cleveland Clinic, "Hypoxemia," [Online]. Available: https://my.clevelandclinic.org/health/diseases/17727-hypoxemia.

[15] Robocraze, "How to use MAX30100 Arduino as Heart Rate Sensor," [Online]. Available: https://robocraze.com/blogs/post/how-to-use-max30100-arduino-as-heart-rate-sensor.

[16] Last Minute Engineers, "MAX30100 Pulse Oximeter and Heart-Rate Sensor Module for Arduino," [Online]. Available: https://lastminuteengineers.com/max30100-pulse-oximeter-heart-rate-sensor-arduino-tutorial/.

[17] Joy-IT, "SEN-MPU6050," [Online]. Available: https://joy-it.net/en/products/SEN-MPU6050.

[18] Dallas Semiconductor, "DS18B20 Programmable Resolution 1-Wire Digital Thermometer Data Sheet," [Online]. Available: https://www.alldatasheet.com/datasheet-pdf/pdf/58557/DALLAS/DS18B20.html.

[19] Katranji, "Katranji Hand Center," [Online]. Available: https://www.katranji.com/item/448739.

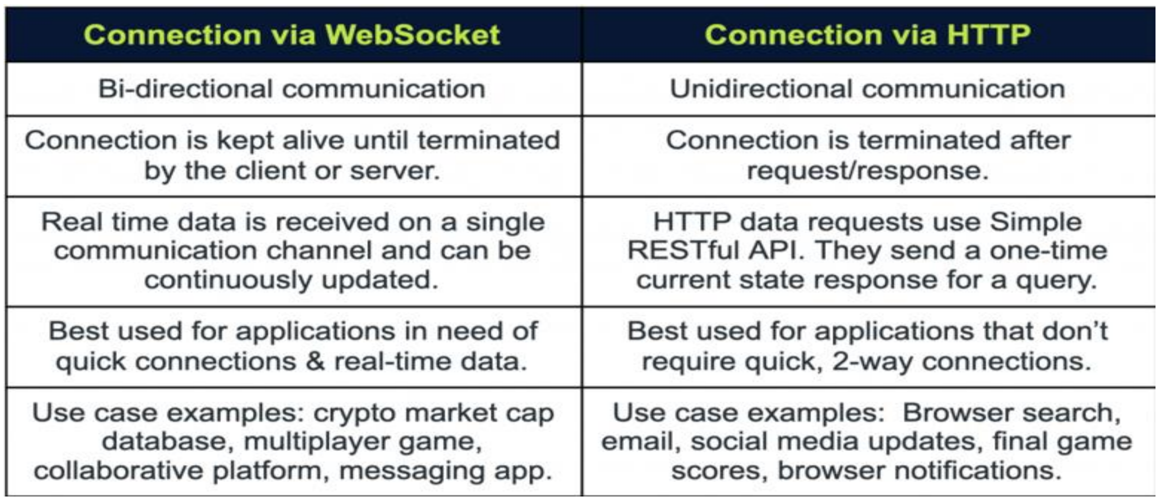
[20] Apidog, "Socket.IO vs WebSocket," [Online]. Available: https://apidog.com/articles/socket-io-vs-websocket/.

[21] Ritza, "How fast does an airbag deploy and other questions about airbags," [Online]. Available: https://ritza.co/articles/gen-articles/how-fast-does-an-airbag-deploy-and-other-questions-about-airbags/.

[22] Cardinal Health, "Localized Temperature Therapy White Paper," [Online]. Available: https://www.cardinalhealth.com/content/dam/corp/web/documents/whitepaper/cardinal-health-localized-temperature-therapy%20White%20Paper.pdf

[23] [ASUS and Innodisk: A Partnership for the Future of AIoT](^1^), InnoNews, vol. 3, no. 2, Nov. 2020. [Online]. Available: https://www.innodisk.com/epaper/innonews/202011-asus-and-innodisk/eng.html
[24] Bergen, Gwen, Mark R. Stevens, and Elizabeth R. Burns. "Falls and fall injuries among adults aged≥ 65 years—United States, 2014." Morbidity and Mortality Weekly Report 65.37 (2016): 993-998.

Appendix:



Connection via WebSocket	Connection via HTTP
Bi-directional communication	Unidirectional communication
Connection is kept alive until terminated by the client or server.	Connection is terminated after request/response.
Real time data is received on a single communication channel and can be continuously updated.	HTTP data requests use Simple RESTful API. They send a one-time current state response for a query.
Best used for applications in need of quick connections & real-time data.	Best used for applications that don't require quick, 2-way connections.
Use case examples: crypto market cap database, multiplayer game, collaborative platform, messaging app.	Use case examples: Browser search, email, social media updates, final game scores, browser notifications.