MINISTERUL EDUCAȚIEI NAȚIONALE

UNIVERSITATEA TEHNICĂ
DIN CLUJ-NAPOCA

FACULTATEA DE AUTOMATICĂ ȘI CALCULATOARE

HEALTH MONITORING SYSTEM USING WEARABLE DEVICES

DIPLOMA THESIS

Author: **Ovidiu MOLDOVAN**

Supervisor: **Assistant Eng. Dan GOȚA, PhD**

2020

MINISTERUL EDUCAȚIEI NAȚIONALE

UNIVERSITATEA TEHNICĂ
DIN CLUJ-NAPOCA

FACULTATEA DE AUTOMATICĂ ȘI CALCULATOARE

DEAN,
Prof. Eng. Liviu MICLEA, PhD

HEAD OF AUTOMATION DEPARTMENT,
Prof. Eng. Honoriu VĂLEAN, PhD

Author: **Ovidiu MOLDOVAN**

Health monitoring system using wearable devices

1. **Project proposal:** *Implementation of a monitoring system using data from wearable devices' sensors: heartbeat rate, pedometer, GPS, temperature. The data is sent through internet and stored in a database. This project shows the hardware and software implementation of the system.*

2. **Project contents:** *Presentation page, Statement regarding the project's authenticity (in Romanian), Project summary, Table of Contents, Introduction, State of the art, System design, System Implementation, Tools and testing, Conclusions, Bibliography.*

3. **Place of documentation:** *Technical University of Cluj-Napoca*

4. **Consultants:** -

5. **Thesis emission date:** 25.10.2019

6. **Thesis delivery date:** 08.07.2020

Author's signature _____

Supervisor's signature _____

MINISTERUL EDUCAȚIEI NAȚIONALE

UNIVERSITATEA TEHNICĂ
DIN CLUJ-NAPOCA

FACULTATEA DE AUTOMATICĂ ȘI CALCULATOARE

Declaraţie pe proprie răspundere privind

autenticitatea proiectului de diplomă

Subsemnatul(a) **Ovidiu MOLDOVAN**,
legitimat(ă) cu CI seria CJ nr. 255637 , CNP 1970216125792 ,

autorul lucrării:

HEALTH MONITORING SYSTEM USING WEARABLE DEVICES

elaborată în vederea susţinerii examenului de finalizare a studiilor de licenţă la **Facultatea de Automatică şi Calculatoare**, specializarea **Automatică şi Informatică Aplicată (în limba engleză),** din cadrul Universităţii Tehnice din Cluj-Napoca, sesiunea Iulie 2020 a anului universitar 2019-2020, declar pe proprie răspundere, că această lucrare este rezultatul propriei activităţi intelectuale, pe baza cercetărilor mele şi pe baza informaţiilor obţinute din surse care au fost citate, în textul lucrării, şi în bibliografie.

Declar, că această lucrare nu conţine porţiuni plagiate, iar sursele bibliografice au fost folosite cu respectarea legislaţiei române şi a convenţiilor internaţionale privind drepturile de autor.

Declar, de asemenea, că această lucrare nu a mai fost prezentată în faţa unei alte comisii de examen de licenţă.

In cazul constatării ulterioare a unor declaraţii false, voi suporta sancţiunile administrative, respectiv, *anularea examenului de licenţă*.

Data

08.07.2020

Ovidiu MOLDOVAN

MINISTERUL EDUCAȚIEI NAȚIONALE

UNIVERSITATEA TEHNICĂ
DIN CLUJ-NAPOCA

FACULTATEA DE AUTOMATICĂ ȘI CALCULATOARE

Project summary

of the Diploma Thesis:

Health monitoring system using wearable devices

Author: **Ovidiu MOLDOVAN**

Supervisor: **Assistant Eng. Dan GOȚA, PhD**

1. Requirements:
   - Creating a monitoring system capable of reading, sending, and managing health-related data using wearable devices
2. Chosen solutions:
   - Creating an Apple Watch application and an iPhone complementary application using the Swift programming language.
   - Creating an embedded wearable device using an Arduino-compatible development board based on an ESP32 microcontroller.
   - Creating a web API server using C# and .NET Core.
   - Creating a single page web application using ReactJS.
3. Obtained results:
   - The hardware and software solutions were obtained, making it possible to store the user data from the Apple Watch / ESP32 into a database and manage it from the web application.
4. Tests and validations:
   - Each functionality of each component of the system has been manually tested.
   - Other testing techniques: unit testing, Postman.
5. Personal contributions:
   - Building the hardware component based on the ESP32, making its interface look like a smartwatch, making it capable of sending data over the internet
   - Building the Apple Watch and iPhone applications using their native programming language and frameworks; Created a communication session between them.

- Building the web API server that enables the possibility of authentication, authorization, and data manipulation.
- Building a fast and easy to use single page web application that enables data management.
- Creating the system in such a way that it can be easily extended and can be used in various applications.

6. Bibliographical sources: scientific articles, technical books, official documentation, web pages.

Author's signature 

Supervisor's signature 

Table of contents

1 INTRODUCTION ... 3

1.1 General context ... 3
1.2 Objectives ... 3
1.3 Specifications ... 4
1.3.1 Smartwatch and smartphone solution ... 4
1.3.2 API and Web Application ... 5
1.3.3 Embedded solution ... 5

2 STATE OF THE ART ... 6

2.1 Health monitoring systems ... 6
2.1.1 Monitoring during a pandemic ... 6
2.2 Wearable devices ... 7
2.3 A review on sensors ... 8
2.3.1 Heart rate sensors in wearable devices ... 8
2.3.2 Step counter ... 9
2.3.3 Connectivity in wearable devices ... 10
2.3.3.1 Wi-Fi ... 10
2.3.3.2 Cellular ... 10
2.3.3.3 GPS ... 10
2.3.3.4 Bluetooth ... 10
2.3.4 ECG ... 10
2.4 Statistics, privacy, security, GDPR ... 11
2.4.1 Statistics and data privacy ... 11
2.4.2 Security awareness ... 12
2.4.3 Data protection laws ... 13
2.4.3.1 Data protection in the European Union ... 13
2.4.3.2 Data protection in the United States ... 13
2.4.3.3 Defining identifiers ... 13
2.5 Writing software on mobile devices ... 14
2.5.1 Smartphone Applications ... 14
2.5.2 Smartwatch Applications ... 16
2.5.2.1 States ... 16
2.5.2.2 Life cycle ... 17
2.5.2.3 Other design issues ... 17
2.5.3 The Swift Programming Language and the Apple SDK ... 17
2.5.3.1 Swift 5.2 ... 17
2.5.3.2 Swift in Data Science applications ... 18
2.5.3.3 Swift applications on the Apple Watch ... 18
2.5.3.4 SwiftUI ... 19
2.5.3.5 Swift on Android ... 19
2.6 Embedded solutions for wearable devices ... 19
2.7 Integrating Swift development and embedded solutions ... 20
2.8 REST APIs in IoT ... 20
2.9 Writing software for a web application ... 21
2.10 Good practices in software engineering ... 22

3 SYSTEM DESIGN ... 23

3.1 The big picture ... 23

3.2 CLASS DIAGRAM 24
3.3 USE-CASE DIAGRAM 25
3.4 STATE MACHINE MODEL AS A PETRI NET 25

4 SYSTEM IMPLEMENTATION 27

4.1 WATCHOS APPLICATION 27
4.1.1 Swift and WatchKit functionality 27
4.1.2 Getting the data from the sensors 30
4.1.3 Sending data to the API 32
4.1.4 Communication with the phone application 32
4.1.5 Resulting application 33
4.2 IOS APPLICATION 34
4.2.1 SwiftUI Framework and the request forms 34
4.2.2 Communication with the watch application 35
4.3 EMBEDDED APPLICATION 37
4.3.1 Microcontroller and development board 37
4.3.2 Communication with the API 39
4.3.3 Authentication 41
4.3.4 Using the OLED display 41
4.3.5 Displaying user information 41
4.3.6 Sensors 42
4.3.6.1 Temperature 42
4.3.6.2 Heart rate 43
4.3.7 Main program 45
4.4 API SERVER 45
4.4.1 Models 45
4.4.2 Controllers 47
4.4.2.1 General Controller 47
4.4.2.2 User Controller 48
4.4.2.3 Data Item Controller & Project Controller 49
4.5 WEB APPLICATION 50

5 TOOLS AND TESTING 55

5.1 MANUAL TESTING 55
5.2 UNIT TESTING 55
5.3 POSTMAN 56
5.4 VERSION CONTROL – GIT 57

6 CONCLUSIONS 58

6.1 RESULTS 58
6.2 PROBLEMS ENCOUNTERED 58
6.3 GOING FURTHER 58

7 BIBLIOGRAPHY 60

1 Introduction

1.1 General context

In these days, wearable devices are nothing unusual, with fitness trackers and smartwatches becoming more affordable and more powerful. The sensor readings in these devices are becoming more accurate, bringing software developers a lot of possibilities to create great applications.

This diploma project aims to take advantage of the wearable devices' popularity and reliability in order to create a monitoring system based on the sensor readings available. Such a system can bring great advantages in the healthcare system and it can also be a great tool in conducting studies about people's quality of life.

Medical personnel nowadays, even in the COVID-19 pandemic, do not use such tools to monitor patients for basic symptoms like a fever and most of the time they need to have direct contact just to measure one's temperature. Imagine having a system that automatically notifies doctors when a quarantined at home patient has high temperature.

Conducting a study by tracking an individual's or a set of individuals' heart rate in Beats per Minute, BPM) and physical activity, the user's stress level can be deducted and with that information, an employer can determine his employees' well-being while at work. Add the location and one can determine stress zones in the city.

The system presented here provides a complete implementation from one end to another: a server Application User Interface (API) with a database, a smartwatch application with a smartphone companion application, a web application in which the data can be easily seen and managed. Because the data available on a commercial smartwatch is limited (e.g. the one used in this project does not have a temperature sensor), this project also presents an embedded solution that is open for extensions and could be adapted to the needs of a specific use.

In what follows, the next chapter presents studies on this matter, similar solutions, what technologies should such a system use and addresses the security issues such systems encounter. The chapter after that describes the design decisions, the implementation steps with information about the specific technologies used and the test methods for each application. The last chapter concludes what has been done, what this project brings new and further directions for improvements and new features.

1.2 Objectives

The following objectives have been set in order to implement the end-to-end solution of a health monitoring system, with the questions asked here being answered in the following chapters:

- Developing a smartwatch application and its smartphone companion that can access sensor readings and send them to the system:
    - What sensor readings are available to the developer?
    - What programming language should be used in order to achieve that?
    - How will the smartwatch and the smartphone communicate?
- Developing a secure API that can store users' data in a well-organized way using a database:
    - What technologies can be used in order to achieve secure storage and communication?
    - What is the best way for communication over the HTTP protocol?
    - How should the data be organized?
- Developing a web application as an administrative user interface to easily see and manage the data:
    - What technology should be used to develop a fast and responsive web application?
- Developing a proprietary embedded solution to extend the sensor readings possibilities:
    - What microcontroller and development board can be used?
    - What sensors can be used?
    - How will the microcontroller communicate with the API?

1.3 Specifications

To meet the objectives, the health monitoring system's specifications are set as follows.

1.3.1 Smartwatch and smartphone solution

- Hardware and relevant features
    - Apple Watch
        - Wi-Fi, Bluetooth Low Energy connectivity
        - GPS
        - Heart rate monitor
        - Pedometer (steps and distance sensor)
        - 1.65-inch, touch screen, pressure-sensitive, OLED
    - Apple iPhone
    - MacBook
- Software
    - watchOS -> the operating system of the Apple Watch
    - iOS -> the operating system of the Apple iPhone
    - macOS -> the operating system of the MacBook
    - Xcode -> the IDE used to write the applications
    - Swift -> the programming language used natively by Apple products

The specific versions that the system has been developed on are: Apple Watch Series 3 (2017), Apple iPhone X (2017), MacBook Pro (2014). It has also been tested on other versions of hardware (detailed in *Chapter 3*).

1.3.2 API and Web Application

- Hardware
    - the MacBook the server runs onto
- Software
    - C# & .NET Core -> the programming language and framework used for the API
    - Entity Framework Core -> the framework used for the database connection
    - SQLite -> the database technology
    - ReactJS -> the JavaScript framework used to create the web application interface
    - Visual Studio 2019 for Mac, Visual Studio Code -> IDEs

1.3.3 Embedded solution

- Hardware
    - ESP32 microcontroller on a development board by Heltec Automation
        - Wi-Fi, Bluetooth Low Energy connectivity
        - 0.96-inch OLED screen
        - dual core, 32bit, 240MHz
        - 12 bits ADC, 8 bits DAC
        - 4MB flash memory
    - Heart rate sensor
    - Temperature sensor
- Software
    - Arduino IDE

2 State of the art

2.1 Health monitoring systems

As [1] states, using smart devices as health monitoring systems that can monitor an individual's health parameters, such as heart rate, temperature or blood pressure, can lead to a real advancement in an individual's quality of life and in healthcare services.

By providing such a system with personal data, software algorithms can assist and advise people regarding possible health issues. More than that, patients suffering from chronic diseases can live a more independent life while medical doctors can still monitor their conditions. Telemedicine has been around for a while now and smart devices are improving that, especially in poorly developed countries, as well as in crisis situations.

One of the big challenges of healthcare services is represented by patient prioritization. With real-time data gathered from monitoring systems, this process can be optimized which ultimately leads to saving more lives.

This is possible due to the advancements in the Internet of Things. As [2] concludes, such systems cannot exist without HTTPS web servers and web services, e.g. RESTful. The connection between the sensors and a main device is done over a Low-Power Wireless Network (LPWN) such as IEE 802.15.4 (ZigBee) or IEE 802.15.1 (Bluetooth). The main device must connect to the internet, ideally through a wireless network using Wi-Fi or a cellular network.

[3] exemplifies a real-time monitoring system, motivating that telemedicine has already been adopted for patients with chronic diseases, such as asthma, diabetes, or heart failures because these conditions require long-term, continuous monitoring in order to minimize the threat. More than that, the patients can often be located far away from medical facilities and going to the doctor on a regular basis is something that takes time and can have a financial impact. This paper presents a system where various sensors, e.g. heart rate sensors, are connected to an Android smartphone over a Bluetooth Low Energy (BLE) connection. An application for the smartphone is created in Android Studio which receives the data, adds GPS data using the built-in sensor of the device and sends it to a PHP web application that provides a REST API and stores the data received on a SQL database and allows authorized doctors to view patients' data. This system also presents a fuzzy logic approach to interpret data from a heart rate sensor, a blood pressure sensor and a body temperature sensor in order to automatically warn the user about possible diseases: hypothermia, bradycardia, tachycardia, hypotension/hypertension or just a simple fever. This system has been tested on 40 cardiac patients and found that, depending on the area and the networks, the delays can be up to 6 minutes which is acceptable, but certainly can be improved. Also, this system generated false alarms.

2.1.1 Monitoring during a pandemic

According to [4], monitoring of people exposed to confirmed cases of patients suffering of COVID-19 in the United States is done by daily telephone or text with the

medical personnel about fever or other symptoms. This is not an efficient way. People are asked to check their own body temperature regularly and report back to the medic. This monitoring technique is inefficient in so many ways because it relies on individuals' behavior and takes medics' precious time in such hard times. As of March 2020, the United States do not make use of any automated health monitoring system to fight the pandemic. This is even more important when talking about more difficult situations, such as pregnancy, as [5] recommends future mothers to regularly check their heart rate, body temperature and blood pressure. Symptoms spotted early can prevent complications and doctors can decide a premature delivery. Also, the study shows that the virus does not transmit to the fetus in late stages of the pregnancy, but the mother must be prepared for a temporary separation from the baby, so doctors can suggest psychological consultations. Real-time health monitoring systems can spot symptoms early in the development of the disease and can help everyone, from patients to medical staff, to act properly.

2.2 Wearable devices

As [6] stated in 1999, researchers always focused on making the components of a computer smaller, allowing its user to carry it around. The authors defined the relationship between a human and the computer sitting on a desk as a weak one. Humans can often feel in an adversarial relationship to the desktop PC and, in a similar way, the computer alone does not know much about the user either. Even portable devices such as laptops and smartphones, which since then have replaced the personal digital assistants, still have a gap between the computer and its user. This paper predicted that wearable devices would create a symbiosis relationship between them and the human, using the data gathered from a daily worn computer to improve an individual's life.

Fast-forwarding to 2015, we start seeing news reports, for example [7], where the Apple Watch, a commercially available wearable computer, starts saving lives of its users based on heart rate sensor readings. These readings can detect symptoms of cardiovascular diseases. In this case, the person had a resting heart rate of 145 beats per minute which is a symptom of rhabdomyolysis, a condition that causes severe muscle pain and weakness [8]. When the smartwatch detected that, it suggested the user a visit to a doctor. That visit turned out to be lifesaving.

Besides the critical situations that can be prevented with the help of this kind of technology, other benefits should, nevertheless, be considered. [9] introduces the "quantified self" term, where a wearable device that gathers information about physical activity can motivate and educate an individual to gain better health and making better habits. With more and more jobs leading to a sedimentary lifestyle, such widely available devices can improve population health. This is done by setting goals and sending reminders. For example, as the World Health Organization advises, smartwatches usually set a goal of 30 minutes exercise per day. While they do not give a specific definition of exercise, this can usually be anything that involves effort, from a fast-paced walk to a real workout. The watch uses sensors and algorithms to determine that. As the day goes by, it gives different notifications to the user if the goal is not met, with motivating messages

telling that there is still time to be on track with a healthy lifestyle. This is the feedback loop in which the user is constantly motivated. Another part of this loop is the reward, a virtual medal given by the smartwatch. The rewards are used to motivate the user, but their functionality is quite different than one would expect. They are based on anticipated regret, someone's fear of not winning something.

Unfortunately, the same paper as before concludes that the current feedback loop is not enough to get everyone on board with this technology, as hardware manufacturers and software developers need to improve their engagement strategies. The authors back this statement up by numbers, exemplifying a survey conducted on 6223 people. It concluded that more than half of the people who have bought a wearable device have stopped using them after a while.

2.3 A review on sensors

2.3.1 Heart rate sensors in wearable devices

According to [10] , the majority of heart rate sensors in today's fitness trackers or smartwatches use a cost effective and non-invasive measurement method called photoplethysmography (PPG).

An example of a PPG algorithm is explained in detail in [11]. Basically, this method uses 2 green LEDs as light sources and a photodetector in the immediate proximity of the skin (on the user's wrist) in order to quantify the volumetric alternations in blood flow. The data received by the photodetector goes through an analog to digital converter (ADC) to the microcontroller. The signal obtained can be seen in *Figure 2.1*, with each heartbeat given by each Gaussian distribution. A low pass filter is used to filter out the noise.

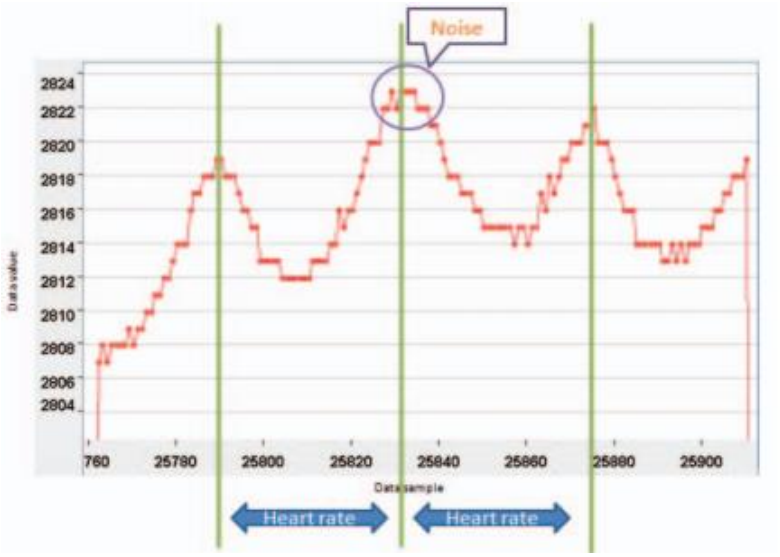


Figure 2.1 Obtained discrete signal obtained from a PPG sensor [11]

The color of the LEDs is important because, according to [12], colors of longer wavelengths have a deeper skin tissue penetration, but as we go to higher wavelengths (such as an infrared light), we encounter more motion artifacts.

As explained in [13], motion artifacts appear due to discontinuous contact of the skin and the sensor. This is a serious problem when talking about wearable devices, with every move of a person's hand slightly moving the device as well. This is even a bigger problem when tracking information during a heavy workout or jogging. This paper presents an algorithm that uses an accelerometer data to cancel errors due to motion. The results have a high accuracy with an error of 2.57% at a maximum speed of 17km/h, with an order of complexity O (N logN) per second and 10Hz sampling rate. It also concludes that green LEDs give the best results. PPG motion efficiency is still major research topic.

Going back to [10], the paper states that heart rate activity has turned out to be one of the most important heart parameters, a device of measuring it being much more accessible to an individual than a device to measure the Electrocardiogram (ECG) which requires electrodes in specific body locations, even though that is subject to change and will be explained in *Chapter 2.3.4*. Using the PPG method, heart rate variations (HRV) can be defined as the peak-to-peak interval. Different HRV values could also be symptoms of various diseases (e.g. cancer). Also, using the second derivative of the PPG signal we get the acceleration photoplethysmogram (APG), the acceleration indicator of the blood. APG is also used in the detection and diagnosis of cardiac abnormalities.

[14] proves that the Apple Watch has a maximum error of 10% when moving, making it just a little bit less accurate than a specialized chest strap that uses ECG signals.

2.3.2 Step counter

Nowadays, a lot of devices come with specialized step counter chips (pedometers) and accelerometers, and the results of each implementation can vary. For example, the authors of [15] built a smartwatch app for the Samsung Galaxy Gear 2 that counts steps based on the raw accelerometer data available on the watch. The result turned out to be better than the built-in application available on the smartwatch, which Samsung claims it uses a pedometer chip. The paper also concludes that the built-in application has a 20% to 30% error in counting steps. [16] which tested the reliability of various accelerometers and pedometers concludes that the Omron pedometer they have tested was more reliable then accelerometer step counters. [17] tests the accuracy of a specialized step counter, the Fitbit Zip, in order to determine if such a device can be used to monitor patients with cardiac disease. The Fitbit Zip was accurate enough only at obvious walking speeds, over 3.6 km/h, which was considered a good result.

[18] presents a study that compares the Apple Watch Series 1 to the Fitbit Charge HR and concludes that the Apple Watch provides a more accurate step counter. [19] is another study that validates the accuracy of the Apple Watch's steps counter at various

walking speeds, for adults with different body-mass indexes and age groups. According to Apple [20], the step counter is calibrated using GPS when the device is outdoors.

2.3.3 Connectivity in wearable devices

Having to fit small cases, wearable devices have small batteries and all the connectivity solutions focus on small energy consumption and size.

2.3.3.1 Wi-Fi

Modern smartwatches usually include the IEEE 802.11b/g/n with a frequency band of 2.4 GHz. While many question the decision of producers' like Apple or Samsung to not include the novel 5 GHz band, [21] concludes that even the simplest applications can have a significant impact on power consumption at 5 GHz compared to a 2.4 GHz connection.

2.3.3.2 Cellular

While there still exist wearable devices that use an usual Subscriber Identity Module (SIM) card, the modern wearable devices use an embedded SIM (eSIM), which, according to [22], is a chip on the device that allows the device owner to change the mobile network operator over the internet. That way, the eSIM chip can occupy much less space, being integrated into the SoC, allowing the manufacturer to optimize the hardware, while operators' and consumers' benefits remain the same.

2.3.3.3 GPS

Most wearable devices use localization data. The ones that do not have a built-in GPS chip will use the paired phone's sensors. But, according to [23], real-time monitoring of a route can discharge a smartwatch in a matter of hours, exemplifying on a Sony Smartwatch 3 that GPS tracking drew 3 times more battery than the display and CPU combined. In order to fix this, the article suggests a smaller sample rate and a path reconstruction algorithm that claims to reduce the battery consumption by 97% while having an error of only 7% related to the actual route.

2.3.3.4 Bluetooth

For the short-range communications, most manufacturers choose to include the latest version of Bluetooth in their devices and that is Bluetooth 5.0. As explained in [24], this version focused on the low energy consumption, while still being faster and having a longer range than the previous one. Its main characteristics are a range of up to 200 meters, a data rate of up to 2 Mb/s and a latency lower than 3 milliseconds, all that while being in some cases even 2 times less power-consuming than Bluetooth 4.0.

2.3.4 ECG

According to [25] , the Apple Watch Series 4 was the first device to introduce an ECG sensor and receive clearance from the Food and Drug Administration (FDA) in the United States for atrial fibrillation (AF) detection. It has since been approved in the European Union as well.

A circuit is created between the watch back, where is a detector, and the watch crown which has built-in electrodes. This creates a single-lead electrocardiogram (iECG) which was as effective as the 12-lead ECG medical services use in detecting AF. Even so,

having only one lead is still a limitation, as typical ECGs can diagnose more heart issues. *Figure 2.2* shows an ECG result from the Apple Watch.

Figure 2.2 ECG on the Apple Watch [26]

According to the official documentation [27], the watch can detect a sinusoidal rhythm (regular heart beat pattern) or atrial fibrillation (irregular heart beat pattern). While it cannot diagnose a disease, it can warn of its symptoms and advise visiting a doctor.

2.4 Statistics, privacy, security, GDPR

2.4.1 Statistics and data privacy

[28] talks about the problems that statistics based on users' locations represent significant danger regarding the location privacy of an individual. Without a doubt, location-based services are of great benefit for the user, and maps applications (Google Maps, Apple Maps, Waze) are great companions in today's life. All these applications use the device's built-in GPS and upload the coordinates that give its position to a server in order to give back to the device its position on a map. More than that, this data is collected and used to create statistics about different locations, such as traffic congestions and average speed on a street. Social networking services take use of that as well, providing approximations of number of people in restaurants or parks. This technique raises many security concerns, as service providers could put together the data labeled by each client, thus violating one's privacy. The privacy concern is not only about companies, but also hackers that can access the databases in which this data is stored. Users have the right to provide only as much data as they want, should always be aware that they are doing that and must have the option to choose not to do that at any moment. The solution suggested by the paper involves using anonymization networks (e.g. Tor) and clients selecting the service which they want to share their data, as well as details such as sample rate and precision of their position. Their server solution also does not keep information about individuals.

Location-based applications are also used a lot in social networking, as presented in [29]. Mobile applications like Foursquare (or Foursquare Swarm) gained 55 million users that actively and willingly publish their location, taking advantage of people wanting to increase in popularity among their friends on social media. Even though that may seem like non-sensitive information, because people only share between their group of friends what places, bars or restaurants, a possible attacker could determine behavioral patterns about an individual, such as going to the church weekly or heavy drinking, then use this information against the person. This article proposes a solution in which the user selects classes of audiences such as friends or employers, then allow the users to create a virtual character which should represent only a part of their personality, for example a professional part, a family part or a social part and only share that part with a class of audience. But in doing so, a lot of location-based services would suffer.

While privacy is an important issue, [30] describes the so-called *privacy paradox*, an inconsistent behavior pattern of people that often claim big concerns about their data privacy, but have no problem to willingly give away personal information if they are receiving even a small reward in return. One explanation given in the paper is that a person who has explicitly given access of personal information to a service would be more careful about her behavior using that service, trying to hide her personality. This study focused on this privacy paradox, trying to prove that people do not care about their personal data on social networking as much as some claim to. Even on a small set of data, 24 out of 38 participants (63%) were willing to sell their Facebook information, with 19 of them (79%) willing to sell it for 10 CHF (approximately 9.5 EUR at the time of writing).

So, we can say that on one hand, people seem to understand the importance of data privacy, [28] even reminding about lawsuits filed against companies like Google and Apple by users who felt their privacy invaded. On the other hand, though, some people will still renounce their right to privacy as long as they get something in return.

2.4.2 Security awareness

Even if every manufacturer will say they use some kind of encryption on their smart devices, nothing is 100% secure and, as the study in [31] presents, some users do not even think about any security issues that a smartwatch may have. Allegedly, this seems to be related with the company producing the device. People using an Apple smartwatch tend to have less security concerns than those who use a Samsung one. The authors of the article put that on the fact that the Apple watch was not found to have any major security issue, while Samsung or FitBit have been proved to have security issues in the past.

More than that, the study concluded that people with no computer science knowledge are unaware of many of their smartwatches' capabilities, and that makes them implicitly unaware of many security threats.

Another thing observed is that culture has a great effect on one's awareness of privacy and security issues. It seems that countries in which cyber-attacks are not seen as a major threat, being practically absent in the media and news, will have more unaware citizens than countries in which you even see advertisements explaining the importance of digital security (the study shares a Greek person's comparison of Greece to the United States).

The article talks about people's "nothing to hide" argument, even if there are a number of studies like [32] or [33] show how gaining access to a smartwatch's sensors could track the hand motion of the user while typing on a keyboard with the help of the gyroscope and the accelerometer, then generate the user's data. Such attacks are still to be considered when talking about the security of wearable devices.

2.4.3 Data protection laws

2.4.3.1 Data protection in the European Union

The General Data Protection Regulation (GDPR) is a set of rules in the European Union that aims to protect the essential rights and freedoms of a person regarding his or her personal data.

According to [34], the GDPR came into force on May 25, 2018 and had an impact on the way every data collecting device works, replacing the old Directive from 1995. GDPR Article 4 defines personal information as any information that can be used to pinpoint and identify an individual. Every service must have the clear consent of the user to collect the specified information. More than that, every service must provide a way for the users to access, modify and delete his data whenever they wish. The IoT industry, which relies on personal data, must invest a lot of money in order to apply the GDPR to every device. There are even firms who changed their business model because of that.

[35] investigates the complexity of IoT devices that comply to the GDPR, saying that the consent part is the most difficult because of the nature of IoT devices which tend to be limited, often without a display or an input device and writing on the device about data collection is not legal because it takes away from the user the right to reject. The authors suggest that the setup of the IoT device should be done via a smartphone or desktop computer.

2.4.3.2 Data protection in the United States

The United States use a different approach regarding to data protection and privacy. While not having a general regulation like the European Union, there are specific laws in different domains that aim to protect the citizens' data. For example, HIPAA (Health Insurance Portability and Accountability Act) is a set of nation-wide laws in the healthcare system and GLBA (Gramm-Leach-Bliley Act) which protects the personal data of customers in financial institutions. [36]

States usually have state-wide laws that furthermore increase the digital security of their citizens. CCPA (California Consumer Privacy Act) is such a law that extends those already in motion like the HIPAA and GLBA. The CCPA is a lot more like the GDPR, providing people the right to access and delete their personal data wherever it is stored. It also gives people the right to not share their data and to opt out of any marketing campaigns. [37]

2.4.3.3 Defining identifiers

The previous mentioned laws use the word *identifier*, a piece of information that can lead to pinpoint to or to identify somebody using his or her personal data. These are bits of information left by users, intentionally or not, on the websites or applications they use.

For example, websites can determine identifiers based on operating system version, IP address from which the user's location can be determined, installed fonts and so on, all these without the user knowing. This data is stored then in the cookies where it can be shared between different apps [38].

But IoT solutions have a lot more data from which they can identify a user, thus posing greater security risks. [39] points that there are 2 fingerprint categories (Figure 2.3): physiological fingerprints, such as a thumb fingerprint, the face or the iris and behavioral fingerprints, such as the voice, the signature, or the keystroke.

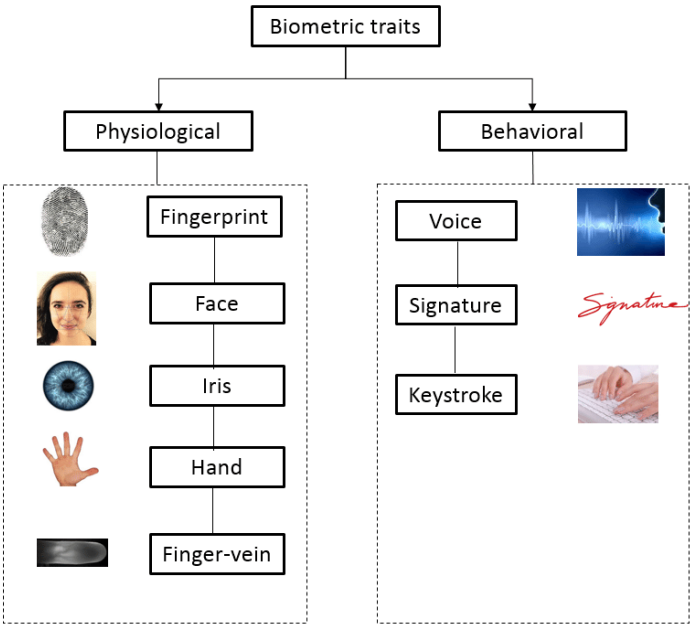


Figure 2.3 – Physiological vs Behavioral fingerprint [39]

Physiological fingerprints are successfully used in authentication and identification of a user in large scale applications (e.g. smartphones), but behavioral fingerprinting can also be used with great benefits in a lot of fields. Even so, they must be handled carefully and only stored in encrypted states, with the user being aware and in control of his or her information.

2.5 Writing software on mobile devices

2.5.1 Smartphone Applications

With Android and iOS, the most important operating systems (OS) in smartphones (both of them have a combined market share of ~99.8%), being very different at core, the first things that should be decided when designing a mobile application is the programming language used in the process, because this will affect many factors such as the application's lifecycle. [40] explains the details of the development processes for each platform.

Android is a Linux-based open-source OS designed by Google. It used to use Java as the primary programming language for app developers, but because developers felt like the development process was 30% slower than on Apple's iOS, Google created Kotlin and, according to [41], as of 2017 it is one of the official languages that Android supports. It appears that developers have already adapted the language, mainly because it runs on the JVM and has Java interoperability, while bringing new features at the same time. Google provides the Android Studio environment, which is based on IntelliJ from JetBrains.

Going back to [40], the paper continues explaining the iOS way. It recognizes iOS superiority over Android in the security aspect, with the App Store on iOS having a more rigorous review process than the Google Play Store on Android. The language of choice for writing iOS software used to be Objective-C, but Apple introduced Swift in 2014 having as the main advantage a shorter development time among the similarities with languages like JavaScript, Python, C#, or Java. Its speed is comparable to C++ and, according to [42], it is backwards compatible with Objective-C, C and C++, which is a reason why it could easily be adopted. Apple provides the Xcode environment, which unfortunately runs only on the macOS operating system, but because Swift is open source, there are Windows and Linux alternatives. Swift is presented in more details in *Chapter 2.5.3*.

But [40] explains a problem most of the mobile developers face: what should a developer do when building an app for both Android and iOS? It classifies the current construction of mobile applications into 3 categories:

- *Native applications* built using the device's officially supported programming language(s), having direct access to system functions and libraries. These give the most efficient results.
- *Web applications* built using web technologies (HTML5, CSS3, JavaScript or Ajax). These have all the limitations a browser has, as they are using the operating system's browser capabilities.
- *Hybrid applications* built using more general-purpose languages like C# (Xamarin), Python (Flutter) or JavaScript (React Native). These can be fast and cost-effective but lack some of the speed the native ones have and the access to the system functions.

[43] says that this makes perfectly sense on iOS, being the way Steve Jobs referred to Web Apps in 2007 when introducing the first iPhone and to Native Apps when introducing its second iteration in 2008. The author suggests a new category should be considered especially when talking about Android, motivating that C/C++ is used in what Google calls the Android Native Development Kit (Android NDK). But both operating systems support C/C++, often seen in game development frameworks such as Unreal Engine or Unity. The new category should be called "System Language App". The paper goes on further, suggesting a new taxonomy:

- *Endemic apps* built using the Source Development Kits (SDK) provided by Apple or Google.
- *Pandemic apps* built using languages the operating systems support but there are no official SDKs (C/C++, HTML, CSS, JS).

- *Ecdemic apps* built using cross-platform frameworks (Flutter, Xamarin, React Native) but are made compilable to the operating system's virtual machine – Low Level Virtual Machine (LLVM) on iOS and Android Runtime (ART) on Android – usually via source code to source code translation.

The author agrees that the naming scheme can be improved.

Of course, every developer will choose the programming language according to the project's requirements and his or her experience with certain programming languages. For example, an experienced web developer might want to use JavaScript with React Native as this would be a very similar experience to writing web applications with ReactJS which is shortly presented in *Chapter 2.8*.

Writing a companion application for a smartwatch is best written in a native or endemic programming language, requiring at least access to the Bluetooth connection. Even if, by compiling an app on the built-in virtual machine (hybrid / ecdemic apps), one can theoretically get access to all the system functions, having access to the official SDK is a clear advantage. Also, the operating system running on the smartwatch is to be considered in case of writing an app for an Apple Watch or an Android Wear device.

2.5.2 Smartwatch Applications

[44] presents as a use-case the design process of an Apple Watch application that uses as the main connection Bluetooth 4.0, also known as Bluetooth Low Energy (BLE). The application is built using Swift and Xcode. The authors present some design issues that are to be taken into consideration and the main concept is simplicity because of the small screen the smartwatch provides. A smartwatch app must have a clean interface and a simple workflow. A good idea is to have multiple screen interfaces that are kept clean and allowing the user to easily navigate through them with basic known gestures (swipes). Also, the developer must integrate into the app the smartwatch specific hardware features, in this case the Digital Crown that can be used for scrolling. Even so, as the study concludes, the developer has a more difficult job in designing an application for a smartwatch than on a smartphone because in a typical environment, users will get the information needed by tapping the screen more times than a typical smartphone app.

2.5.2.1 States

Going further on the design problem, [45] explains that a typical smartwatch app on an Apple Watch has 4 states:

- *The full apps* which behave like smartphone apps.
- *Glances* which are screens that only show information and have no interactive elements.
- *Notifications* which appear when a Push Notification event is triggered.
- *Complications* which are small widgets shown on the Apple Watch's main screen (main watch face) and can show little information (e.g. a weather application can show degrees or icons that resemble the weather outside). They also act like a shortcut, tapping on them leading to the full application.

2.5.2.2 Life cycle

The life cycle of a smartwatch app is slightly different than a smartphone or desktop one. Each state presented in *Chapter 2.5.2.1* launches a different section of the application, and, very similarly to a smartwatch application, a smartwatch application has 5 life cycle states:

- *Not running*, whether the user has not run the app, or the system has killed the process.
- *Inactive*, usually the time spent by the application in loading from the moment it is launched until it gets in the next state.
- *Active*, the app functions in the foreground, receiving input from the user and giving back information.
- *Background*, whether the user has gone back to the main menu or the display went off. In this state, the app can still run some functionalities for a limited amount of time.
- *Suspended*, the app is still loaded in the RAM, but does not run any tasks.

2.5.2.3 Other design issues

As mentioned before, the developer must pay great attention to the fact that a smartwatch screen is very small. While features like the Digital Crown and Force Touch (pressure sensitive screen) add a nice touch, the text input is limited to voice dictation and a very slow method of drawing letters. Going further, smartwatches have significantly less battery capacities, processing power and storage.

2.5.3 The Swift Programming Language and the Apple SDK

Figure 2.4 Swift Logo [46]

2.5.3.1 Swift 5.2

Briefly explained in *Chapter 2.5.1*, Swift (*Figure 2.4*) is the modern way to go when working with Apple devices, but not only. The current version at the time of writing is Swift 5.2 which officially supports macOS and Ubuntu Linux as development platforms, but Swift 5.3 is right around the corner and will provide support for Windows and other Linux distributions [47].

Being a project backed up by one of the biggest companies in the world, it is in a continuous developing stage and there is plenty of documentation available either online [46] or as an eBook [48] that is constantly updated with every new iteration of the language, with a short explanation of changes and new functionalities. The main features and patterns presented are:

- Initialization of variables is done before use

- Automatically check for out of bounds array indices and overflow of integers
- Optional types that explicitly handle null (*nil* in Swift) values
- Automatic memory management

The recommended Integrated Development Environment (IDE) is Xcode, available only on macOS, although there are alternatives on other operating systems as well. iPadOS also supports Swift Playgrounds, an application that offers a Swift interpreter, recommended especially for students to learn the language, but offers other interesting functionalities, like programming and controlling the Adafruit Rover, the Makeblock mBot or other compatible Arduino platforms.

2.5.3.2 Swift in Data Science applications

Swift is an open source project and that made other companies, like Google, to explore its capabilities and even contribute to the project on GitHub. [49] studies why Google has been trying to replace Python with Swift in Tensorflow, its Machine Learning solution, and other Data Science applications, the main reason being that its speed is close to C, while Python is slower and not so good in parallel applications. Swift also allows writing high level code, almost like Python, and it allows C and Python interoperability.

This study goes on with comparisons. Speed-wise, an almost identical algorithm that took Python 360μs was executed in Swift in 14μs, and a little memory optimization made the Swift one to execute in 3μs. The same algorithm run in C was executed in 1.5μs. With one more change in the Swift algorithm, the author made it run as fast as the C one. In conclusion, the same amount of work put in a Python algorithm can run faster and allows more room for optimization in Swift.

Another great thing in Swift is code readability, further explained in *Chapter 2.10*, with features like adding brief explanations called labels in a function's arguments that can make code sound more like English and reduce the need for comments.

2.5.3.3 Swift applications on the Apple Watch

[45] talks about WatchKit, the Swift framework used in watchOS applications. It is provided by Apple in the latest versions of Xcode and shares some similarities with UIKit, the framework used in iOS apps. It provides basic functionality for buttons, labels, images and so on.

Another important framework is WatchConnectivity which provides a communication session between the watch and the phone that it is paired with. It also allows sending data between one device to another and creates event triggers and event listeners for that.

Talking about monitoring, it is important to note the HealthKit framework which provides a secure way to access health data and sensors such as the heart rate sensor. Along with that, CoreMotion is another framework that allows access to the watch's accelerometer, gyroscope, and pedometer.

Frameworks built by the open source community are also a very important part of Swift programming on the Apple Watch, with SwiftifyJSON making the JSON creation easy and Alamofire for HTTP requests and authorization.

2.5.3.4 SwiftUI

In 2019, Apple introduced a new framework that aimed to facilitate a developer's work regarding the application's user interface. [50] describes it as a declarative UI, meaning that the programmer gets to focus on what he wants to implement in the user interface and not how. It does some things by default, such as spacing or the use of the operating system's dark mode, making it great for applications with simple user interfaces.

For simplicity reasons, objects like text fields and text labels are grouped into vertical stacks or horizontal stacks. Also, the UI is changed whenever the data is updated. Xcode also provides real time preview of the changes in the code.

It is backwards compatible with the old frameworks and the developers can update their interfaces gradually. However, SwiftUI requires having an operating system released in 2019 or newer: macOS 15, iOS 13, iPadOS13 or watchOS 6.

2.5.3.5 Swift on Android

Google has not kept its interest in Swift a secret, and part of that is because the Android NDK uses the same compiler (LLVM) as Swift does, meaning that Swift runs natively in Android, any Swift app being treated as a C/C++ one. And [51] proves that yes, this is possible, but the lack of official support means there are not many frameworks available. The SwiftFoundation library offers only basic functionality like network requests, JSON parsing or data storage, but the UI must be written from scratch. The authors of the project also provide a Github repository with a sample Android project written in Swift. Apple also covers compiling Swift for Android in the Swift official documentation, but provides no framework that would make the process easy [52].

The beauty of open source projects is that the community can provide such frameworks. For example, SCADE [53] provides an IDE and an SDK called ScadeKit that aims to provide native cross-platform functionality between iOS and Android. It still has some limitations and the project is still in development.

2.6 Embedded solutions for wearable devices

[54] explains that smartwatches can be built using simple components and proposes a solution based on an Adafruit Feather M0 Bluefruit BLE microcontroller to connect the device via an Android or iOS app, the BLE protocol (Bluetooth 4.0+) being required for iOS devices. An Adafruit FeatherWing monochrome OLED display with a 128 by 32 pixels resolution is connected via I2C. A simple PPG sensor is also used. The microcontroller is programmed in the Arduino environment using C/C++. The data is collected via an Android Java-based app. The study's conclusion is that, while this can be a solid smartwatch experience, heart monitoring noise during motion (as explained in *Chapter 2.3.1*) is very visible and more complex algorithms are needed to filter that.

If there is a need for a more powerful microcontroller, an ARM powered microcontroller can be used, as explained in [55]. An example would be the LPC1768 that uses a 32-bit ARM Cortex M3 core. ARM provides the *mbed* platform specifically designed for IoT devices, providing tools and libraries that make development much faster than traditional embedded C/C++ programming. It makes embedded programming on an ARM processor look more like the Arduino style programming, rather than manually programming bits in registers. This facilitates the concept of Clean Code (see *Chapter 2.10*), thus making the software easily maintaining in time.

2.7 Integrating Swift development and embedded solutions

[56] presents very detailed projects that integrate embedded solutions, IoT applications, with iOS applications written in Swift. The book has a chapter dedicated to building a companion app for a Bluetooth Low Energy capable development board that has an ESP32 microcontroller programmed in Arduino. It explores the possibility of getting data from sensors to an iPhone application using the Bluetooth connectivity available in these devices. More than that, it advises on best practices to make a good user experience on an application that depends on Bluetooth devices.

Another chapter is dedicated to building an application that communicates over Bluetooth with a Raspberry Pi. This one takes advantage of the Raspberry Pi's more powerful hardware and uses it to create a HomeKit bridge.

HomeKit is a framework provided by Apple to standardize IoT-enabled devices throughout the Apple operating systems, giving the end user similar experience on various devices built by different manufacturers.

Next, a Node.js server is created to provide a JavaScript-based API that runs on the Raspberry Pi to deal with the communication over the internet, as such a server provides the possibility to deal with multiple IoT devices. The data is sent as JSON objects. Postman is presented as a testing tool. Security issues (SSL and HTTPS) are also detailed here.

The book's last chapters expand the possibility of IoT devices in the Apple ecosystem with connectivity to smart TVs running tvOS as the operating system and Apple Watch running watchOS. It also addresses more security issues and suggests protecting parts of the applications using passwords, fingerprints (Touch ID) or face unlock (Face ID) technologies.

2.8 REST APIs in IoT

As exemplified in *Chapter 2.1*, [3] uses a Representational State Transfer (REST) application programming interface (API) in its IoT implementation. The study in [57] shows that data transfer in the modern web technology, sending a JSON or an XML over HTTP, is a secure way to standardize the Create, Read, Update, Delete (CRUD) operations over the internet, allowing authentication and authorization in the process. Securing the data transfer is very important, as IoT devices tend to have sensitive data. The authorization is done with encrypted access tokens given by the API to the client that

expire after a certain amount of time (e.g. 1 hour), when a new authorization request must be done. The token is the only authorization parameter that is sent, as sending roles or identifiers is not secure. These values are decrypted from the token. [58] makes use of JSON Web Tokens (JWT) in authentication, motivating that it is rather efficient and does not make frequent calls to the server or the database.

APIs are usually written in C#, Java, node.js or PHP. [59] exemplifies the C# ASP.NET Core framework as one of the most popular used in industry. It uses the Model-View-Controller pattern: the models are the data structures that are used in the application, such as users and items; the views are the user interfaces, be it a mobile app, a smartwatch app or a web app; the controllers define the logic and API calls.

APIs use either SQL databases such as MySQL, Microsoft SQL Server and SQLite, or NoSQL databases such as MongoDB. The .NET framework includes Entity Framework, a framework that allows creation of the database automatically based on the models defined in the MVC pattern and provides a way to write C# code that executes most of the SQL queries. The functionalities of Entity Framework are limited to Microsoft SQL Server and SQLite.

Security-wise, modern frameworks offer basic protection against SQL Injection and the Secure Hyper Text Transfer Protocol (HTTPS) offers encrypted data transfer via internet.

2.9 Writing software for a web application

Web technology has seen quite an advance lately with the popularization of Single Page Applications (SPA). [60] defines a SPA as a web site that does not require a refresh or a reload of the entire application when the content is changed, providing a fast and smooth experience, making the application behave like a desktop one. The primary content like the template and images are downloaded by the browser when first entering the page, then certain resources are being downloaded only when needed, usually as a response to users' actions. The communication with the back-end server or API is done asynchronously in the background.

The same paper exemplifies the most used open source JavaScript SPA frameworks:

- *AngularJS* created by Google and used in web applications such as Gmail or YouTube.
- *ReactJS* created by Facebook and used in Facebook, Facebook Messenger, Instagram. It is not exactly a framework, but a library.
- *VueJS* which intends to be one of the easiest ways to build a SPA.

SPAs have become in a lot of cases as powerful as desktop applications. A very good example is MATLAB for which Mathworks has created a fully functional web application [61], providing an experience that is similar to the desktop version. By using the "Inspect Element" functionality of Google Chrome, we can see that this was built using ReactJS.

ReactJS presents another advantage: the similarity with React Native and the possibility to easily create hybrid mobile applications, as explained in *Chapter 2.5.1*.

From a security perspective, [59] points out that modern front-end frameworks provide basic protection against known vulnerabilities in web applications.

2.10 Good practices in software engineering

Being a software engineer does not mean just writing code and that is it. The development process begins with the design of the application. As [62], by analyzing and evaluating performance properties of an application before the implementation phase, the developer will find bugs and performance issues from the design phase. The paper presents a use-case application, a web-based music streaming service using a server and multiple clients. It is designed using Unified Model Language (UML) diagrams for the application model itself and Petri Nets (PN) to model its performance on the server. As a conclusion, using both UML and PN to evaluate the software and the hardware in the design phase was very effective in this case, contributing to a good performance prediction.

When it comes to implementation, regardless the programming language, the code must be written in a readable way, as Robert C. Martin (Uncle Bob) explains in [63]. The book compares software engineers to authors, as it will certainly be read by someone, be it the same developer or others that may come, motivating that working code is not sufficient. The author estimates that we spend 10 times more time looking at our old code than writing new one and this process must be as enjoyable as it can be. The book continues with a detailed list of advice, such as:

- **Naming** variables, classes, objects, methods, and functions should be always clear, easy to read, respect conventions (classes are nouns, methods are verbs). The reader should always get the idea of what is happening in every piece of code.
- **Functions** should have one responsibility and only one, with no effect about other functionalities than the one they are supposed to have. They should also be as small as possible and not have a lot of arguments.
- **Comments** should be used as a last resort. Naming functions properly makes comments redundant. Comments are hard to maintain, and they tend to remain the same if the code they should explain changes, leading to misunderstandings, confusion, and errors.
- **Formatting**: one should remember that code is read as a book, from left to right and from top to bottom, meaning that functions that depend on each other are placed vertically close. This way, the time spent scrolling within files is smaller. Also, proper indentation is recommended even in small statements.
- **Testing** is as important as the code itself. Unit testing should be a part of the programming process.

3 System design

3.1 The big picture

Figure 3.1 presents a block diagram of the data flow in this health monitoring system.

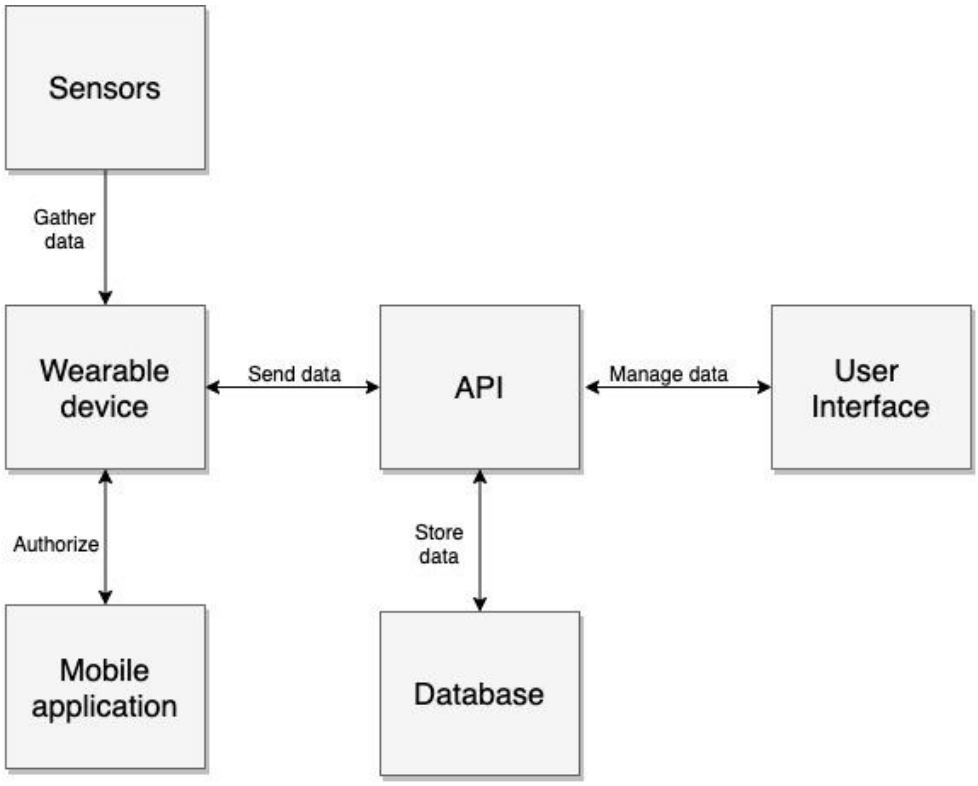


Figure 3.1 Data flow in the system

The data is being collected from the various sensors available: temperature, heart rate, GPS coordinates, pedometer.

The wearable device (the Apple Watch or the embedded solution) gets the data, converts it into strings, creates a JSON object and then sends it to the API as a POST request.

The API sends back a response to let the sender know if the message was transmitted successfully or not. It also sends to the embedded solution the date and time and the outside weather information. Every successful message is immediately recorded into the database.

The user interface gives a simple user access to his data and the option to delete it and gives an administrator the possibility to visualize and manage all the data. It is also the way to set up a new account.

Due to limited input possibilities on the wearable devices, a mobile application is used to log in into a user's account. After logging in, the wearable device receives an authorization token needed to access the API.

3.2 Class diagram

The class diagram is presented in Figure 3.2.

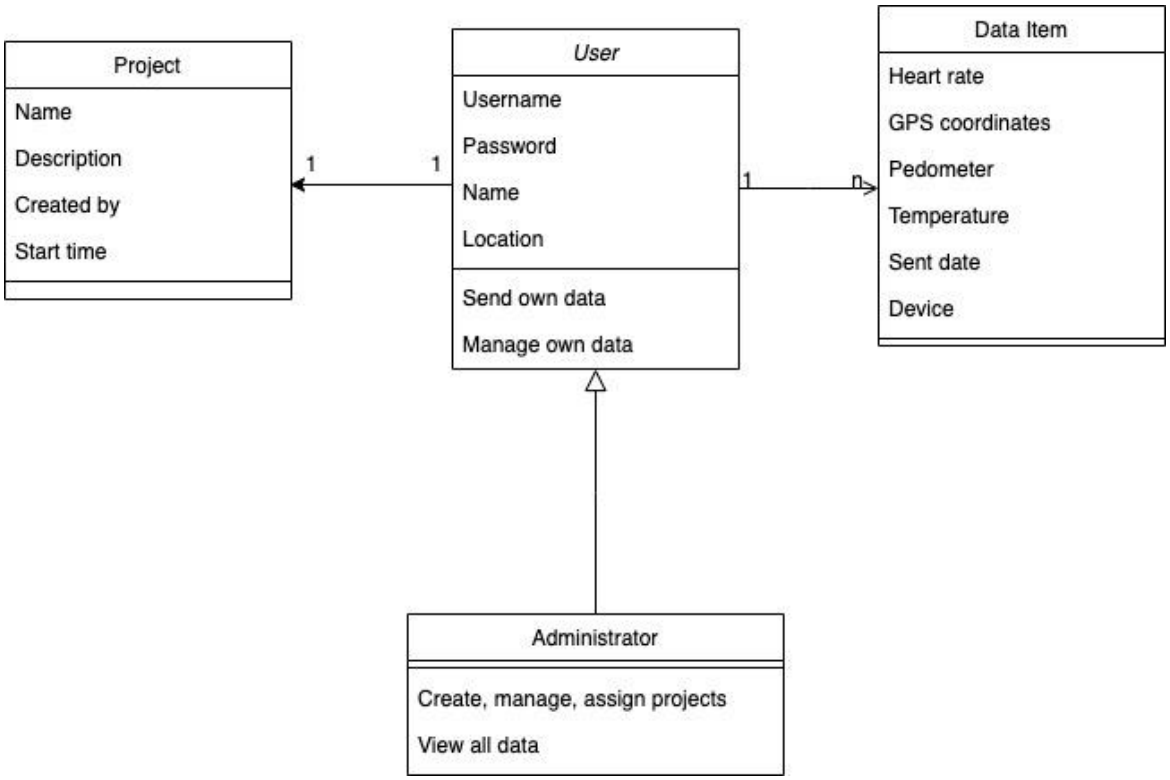


Figure 3.2 Class diagram

The User has a username, a password, a display name, and a location that are set at register. The User can send and manage only his own data.

The Administrator is a User that can also create, manage, and assign projects, as well as view all the data in the application.

Each User is assigned to one project. If not, by default, the application sets each user to be assigned to a project with the ID 0, called "No Project". The projects have a name, a description, a field with the ID of the Administrator that created it and a start time, the moment of time the project has been created.

Each User has a list of Data Items sent from a wearable device. The heart rate, GPS coordinates, pedometer, temperature data and device name are sent from the wearable device. The sent date field is automatically completed in the API when the sensor readings are sent.

3.3 Use-case diagram

The use-case diagram in Figure 3.3 presents the 2 main roles user have in this system: a user and an administrator, with their specific available actions.

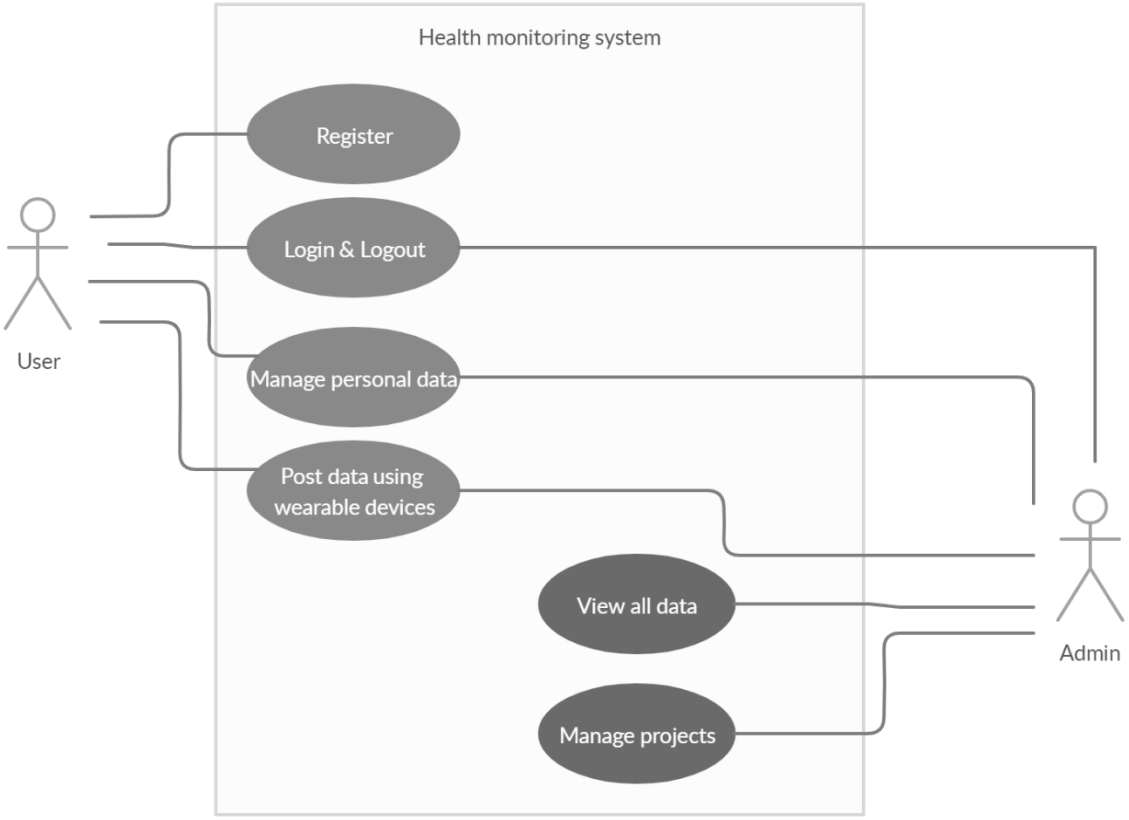


Figure 3.3 The use-case diagram of the system

3.4 State machine model as a Petri Net

The states of the system, from the wearable devices' perspective, are presented in the Petri Net model in Figure 3.4. It is presented as a discrete events system. This figure has been developed with the help of the PIPE2 tool [64].

The input places are *Username*, *Password* and *Sensors* and the output places are *Display* and *DataJSON*.

The Username and Password are passed to the Authenticate transition which will store the authentication data in the InitialState place. From there, the Start transition initializes the InternalData state. InternalState receives information from Sensors through the sendSensorData transition. After that, the buildJSON transition creates an InternalJSON object with all the data gathered until this point. The sendDataToServer transition outputs the results in *Display* and sends the POST request with the *DataJSON*. The feedback is generated by the *ResponseFromServer*, usually a status code sent by the server to indicate that the data has been sent correctly or not. These steps are repeated in a loop every 5 seconds.

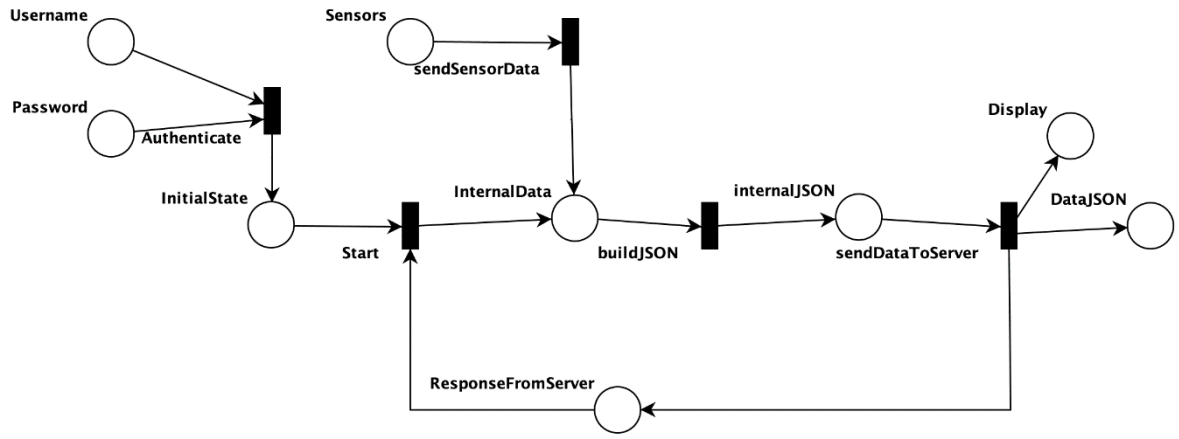


Figure 3.4 The Petri Net of the system as a feedback loop

3.5 Sequence diagram

Figure 3.5 shows the sequence diagram of the devices and the communication between them and the API server.

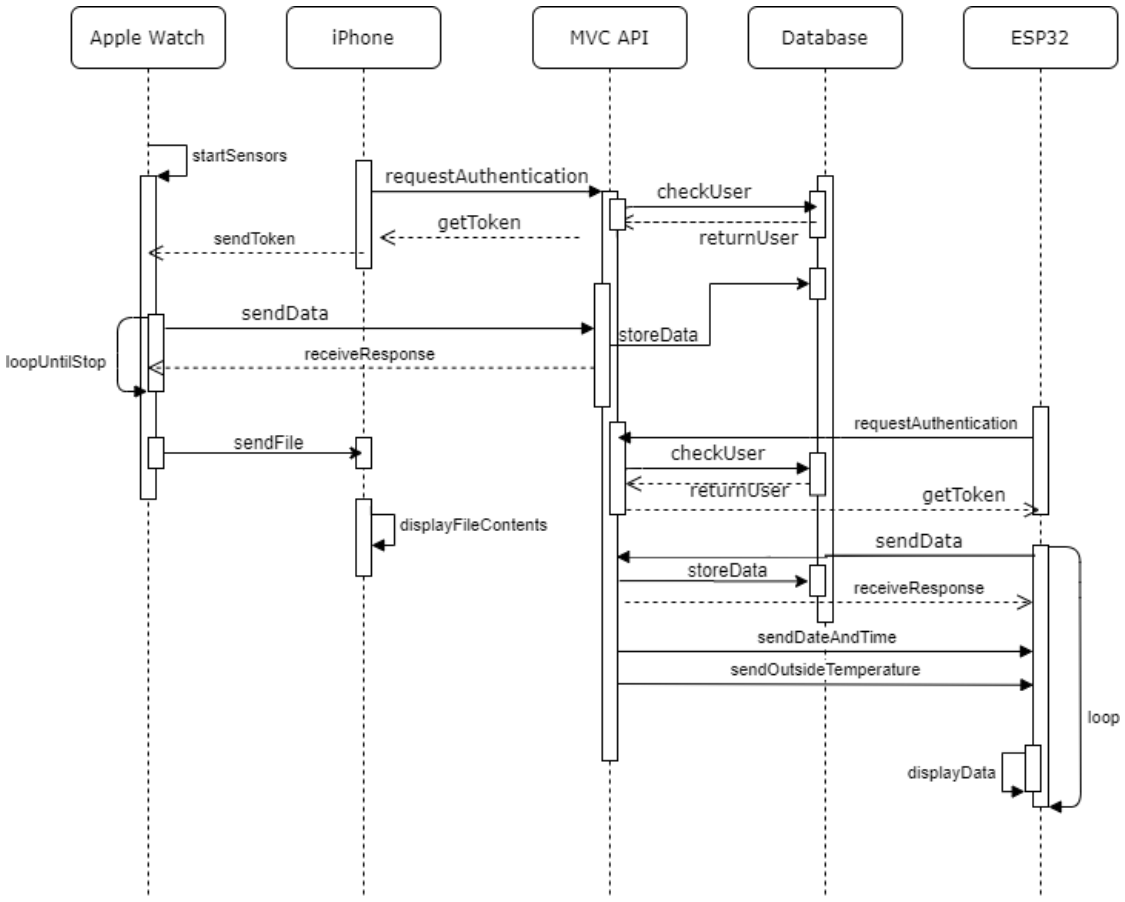


Figure 3.5 The sequence diagram of the system

4 System implementation

4.1 watchOS Application

The watchOS application is written in Swift using the Xcode IDE. Initially intended to be a standalone Apple Watch application, a mobile companion application is useful especially for the login process, so the applications are created in the same project, but more on the mobile one in Chapter 4.2.

Apple products are known for being more locked systems than the Android counterparts, mainly for maintaining a high security profile. So, naturally, the first thing I had in mind when I decided to take on this theme was that building an app that wants to access data from sensors and sensitive health data is going to be a challenge. And I was not wrong.

4.1.1 Swift and WatchKit functionality

WatchKit is the framework provided by Apple that allows the development of basic Apple Watch functionalities.

Figure 4.1 shows a screenshot of the project in Xcode. The folder structure can be seen on the left side, with the *watchConnectivitySample WatchKit App* being the project that deals with the user interface of the watchOS app and the *watchConnectivitySample WatchKit Extension* being the project that deals with the internal logic. This is a similar design approach to the front-end / back-end design of many other applications.

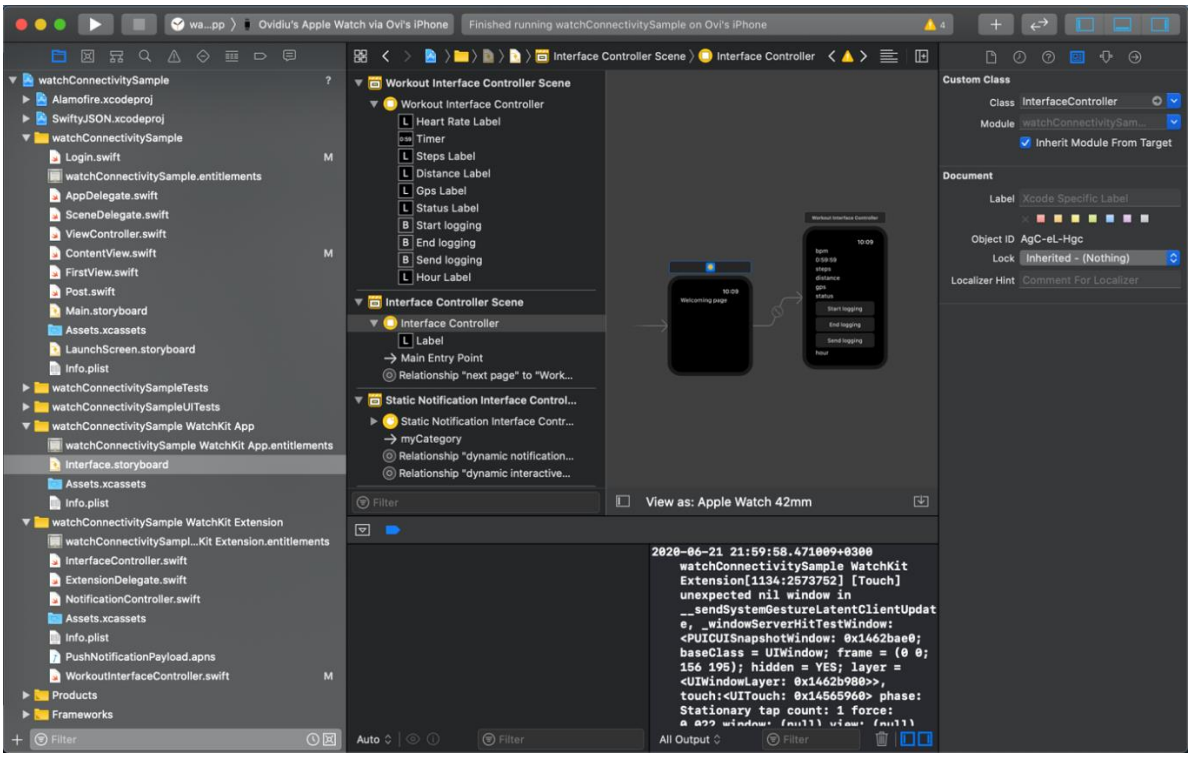


Figure 4.1 The project in Xcode, with an Interface storyboard

Figure 4.1 also presents the Interface storyboard in the middle, where the design elements can be defined. The *Interface Controller* is the main entry point of the application, the first screen the user encounters, which has a welcoming text. The second screen is the *Workout Interface Controller*, a controller that has more information about the sensors in the application and some control buttons. These 2 elements are binded with a "next page" relationship, allowing the user to go through them with swipe gestures.

One of the important configuration files is the *.entitlements* file (Figure 4.2.) in which the developer specifies the framework permissions needed by the app if needs any unusual ones. In this case, *HealthKit* and *HealthKit Capabilities* are required by the *HealthKit* framework, the one that allows getting the heart rate of the user. *App Groups* is needed by the *WatchConnectivity* framework which allows communication between the iPhone and the Apple Watch that are in the same group.

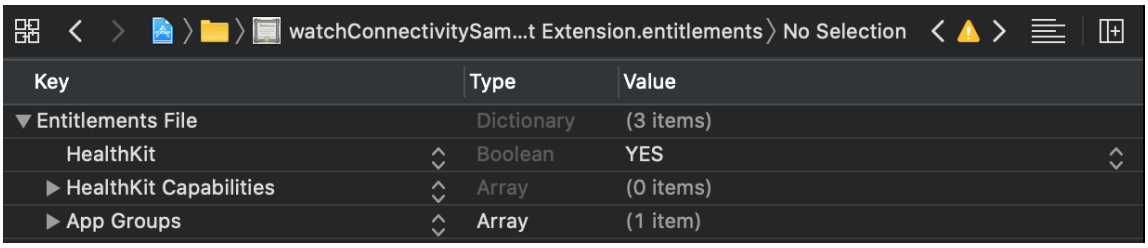


Figure 4.2 .entitlements file

One other important configuration file is the *Info.plist* (Figure 4.3). Here, each permission needed by the app is specified, along with other configuration settings like the ability to install this app without a companion smartphone app.

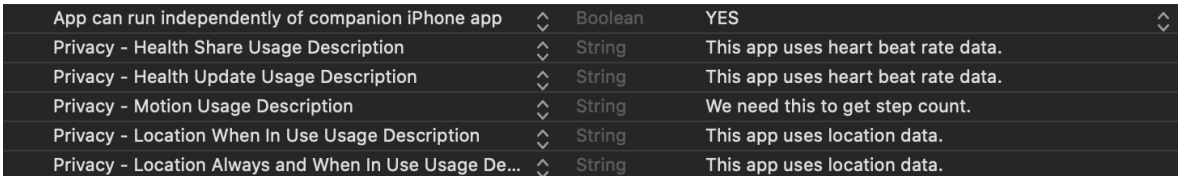


Figure 4.3 Info.plist file

Going to some code examples, Figure 4.4 shows how variables are declared and used in Swift. The *@IBOutlet* marker is used usually in front of a label, allowing the changes to that variable to appear in the user interface. A full dot instead of a line number indicates that, indeed, this variable is connected to an object in the Interface Controller.

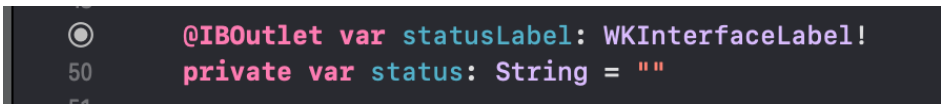


Figure 4.4 Variable declaration and interface label usage in Swift

Figure 4.5 gives an example of an action button with the *@IBAction*. Its purpose is to write some text in a file, preparing it for logging the data gathered from sensors, using

methods from the *FileHandle* class like *seekToEndOfFile* or *write*. Also, some labels are modified to let the user know the current status of the application.

```
@IBAction func startLogging() {
    if(status == "stopped") {
        resumeWorkout()
    }

    self.distance = "0"
    self.steps = "0"

    status = "running"

    do{
        if let fileUpdater = try? FileHandle(forUpdating:
            self.filePath){
            fileUpdater.seekToEndOfFile()
            let string = "\n\nStart sesiune noua"
            print(string)
            fileUpdater.write(string.data(using: .utf8)!)
            fileUpdater.write("\n".data(using: .utf8)!)
            fileUpdater.closeFile()
        }
    }

    statusLabel.setText("Status: running")
    timer.start()

    startPedometerUpdating()

    setupLocation()
}
```

Figure 4.5 The startLogging action function

Another way to interact with the Apple Watch is through its pressure sensitive display called Force Touch. If available, a menu will be displayed when a force touch is detected. This menu can be modified with the help of the Interface storyboard or with the *addMenuItem* function (Figure 4.6) of the WatchKit framework. The first argument is the icon, the second is the name of the item and the third is a function call.

```
addMenuItem(with: .decline, title: "End", action: #selector(endWorkoutAction))
```

Figure 4.6 Adding Menu Items on a Force Touch

The resulting menu is presented in Figure 4.7.

Figure 4.7 Resulting Force Touch menu

4.1.2 Getting the data from the sensors

First of all, because of dealing with very sensitive data, specific permissions have to be requested. The *requestAuthorization* method of a *HealthStore* object (Figure 4.6), called when the app is first launched in the *didAppear* method of the controller, creates a screen (Figure 4.7) that asks the user what health-related data it wants to share with the app. The app explanation is specified as a string in the *Info.plist* file.

```
healthStore.requestAuthorization(toShare: typesToShare, read: typesToRead)
```

Figure 4.6 Requesting Authorization method

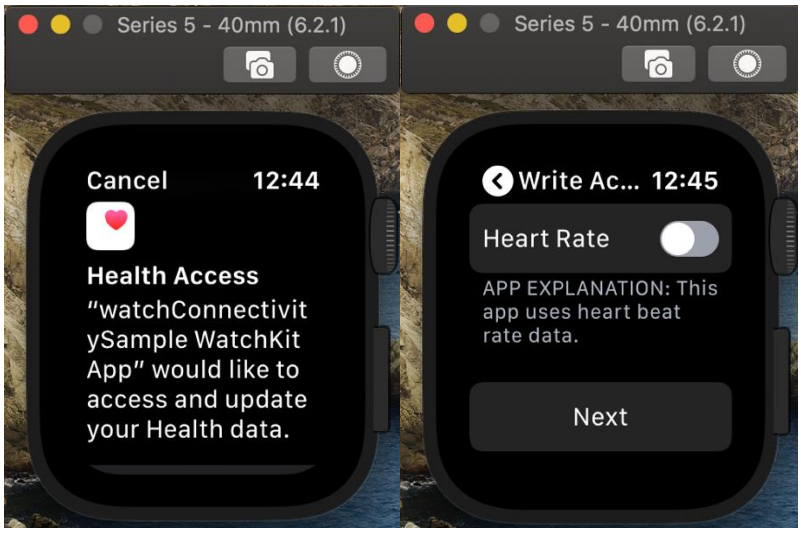


Figure 4.7 Requesting Authorization screen

For the heart rate sensor, I found 2 ways of getting its data. The first one was instantiating a *HKUnit* object from the *HealthKit* framework and that worked only as long

as the screen was on but stopped whenever the screen turned off. This makes it not so useful, as the Apple Watch screen turns off when it is flipped in the opposite direction. Keeping the screen on would also not be a viable solution due to the limited battery capacity a smartwatch has.

The working solution instantiates a Workout Session (Figure 4.8) from the *HKWorkoutSession* class. Workout sessions are, at the moment of writing, the only way the sensors are allowed to record data when the application is running in the background.

```
//Workout session
do {
    workoutSession = try HKWorkoutSession(healthStore: healthStore, configuration:
        configuration)
    builder = workoutSession.associatedWorkoutBuilder()
} catch {
    dismiss()
    return
}
```

Figure 4.8 Starting a Workout Session

This way, the heartrate is continuously measured and a HKUnit object can access the its value at any time by calling the *count* method (Figure 4.9).

```
let heartRateUnit = HKUnit.count().unitDivided(by: HKUnit.minute())
```

Figure 4.9 Getting the heart rate BPM value

To get the pedometer (steps and distance) data, a *CMPedometer* is instantiated from the *CoreMotion* framework. The *countSteps* function is created, which starts the pedometer logging with the startUpdates method, then this data can be accessed through the distance and numberOfSteps attributes (Figure 4.10).

```
private let pedometer = CMPedometer()
private func countSteps() {
  pedometer.startUpdates(from: Date()) {
      [weak self] pedometerData, error in
      guard let pedometerData = pedometerData, error == nil else { return }

      DispatchQueue.main.async {
        self?.stepsLabel.setText(pedometerData.numberOfSteps.stringValue)
        print(pedometerData.numberOfSteps.stringValue)
        self?.distanceLabel.setText(pedometerData.distance?.stringValue)
        self?.distance = pedometerData.distance?.stringValue ?? "0"
        self?.steps = pedometerData.numberOfSteps.stringValue
      }
  }
}
```

Figure 4.10 Pedometer function that counts steps

The GPS coordinates are obtained in a similar way, using a *CLLocationManager* object from the *CoreLocation* framework. Then, the latitude and longitude attributes can be accessed (Figure 4.11).

```
self.gpsCoords = String(currentLocation.coordinate.latitude) + " " +
    String(currentLocation.coordinate.longitude)
```

Figure 4.11 Getting the GPS coordinates

4.1.3 Sending data to the API

Sending the data to the API is done through a POST request on the watch (Figure 4.12). After the headers are set, a JSON object is created as a dictionary with a String value as the key. Then, the dictionary is serialized to a real JSON and the request is sent.

```
var request = URLRequest(url: self.url!)
request.setValue("application/json; charset=utf-8", forHTTPHeaderField: "Content-Type")
request.setValue("application/json; charset=utf-8", forHTTPHeaderField: "Accept")
request.setValue("Bearer " + self.token, forHTTPHeaderField: "Authorization")
request.httpMethod = "POST"

let json = [
    "heartBpm": lastHeartRate as NSNumber,
    "gpsCoordinates": (self.gpsCoords) as NSString,
    "steps": (steps as NSString).integerValue as NSNumber,
    "distance": (distance as NSString).integerValue as NSNumber,
    "device": WKInterfaceDevice.current().model
    ] as [String : Any]

if let jsonData = try? JSONSerialization.data(withJSONObject: json, options: []){
    URLSession.shared.uploadTask(with: request, from: jsonData){ data, response, error in
        print("print action")
        if let httpResponse = response as? HTTPURLResponse{
            print(httpResponse.statusCode)
            print(httpResponse.allHeaderFields)
        }
    }.resume()
}
```

Figure 4.12 POST request on the API

4.1.4 Communication with the phone application

To communicate with the phone, a *WCSession* object is instantiated from the *WatchConnectivity* framework. This creates a communication session between the devices.

First, the watch needs to receive the authentication token from the API which will be further used as an identifier. This requires an overload of the session function which receives as an argument an event when a message is being sent from the phone to the watch (Figure 4.13). The message is saved as a string on the watch.

```
func session(_ session: WCSession, didReceiveMessage message: [String : Any]) {
    print("received message: \(message)")
    DispatchQueue.main.async {
      if let value = message["Bearer "] as? String {
        print(value)
        self.token = value
      }
    }
  }
```

Figure 4.13 Overloading of the session function with a receiving message event

The application also saves all the logged data into a file, independent of the login or API status. To access this file, it is sent to the phone using the *transferFile* method of the session object (Figure 4.14).

```
    @IBAction func sendLogging() {
114         session.transferFile(self.filePath, metadata: nil)
115     }
```

Figure 4.14 Button that transfers a file from the watch to the phone through a Watch Connectivity Session

4.1.5 Resulting application

The resulting application running on the Apple Watch Series 3 is presented in Figure 4.15.

Figure 4.15 Watch Application. Left: labels showing the data. Right: scrolling down to see the action buttons

4.2 iOS Application

The iOS application is built primarily to make a login request on the API and send back to the watch the authentication token. It is also used to receive the data file from the watch and display its contents or send it to someone. Also, there is the possibility for the user to manually post his or her heartrate BPM.

The iOS application is also written in Swift in a similar way to the watchOS one. In terms of permissions, the *.entitlements* file and *Info.plist* file need to have the same permissions as the ones the watch application has. Figure 4.16 shows the Xcode environment in the Main storyboard (the entry point) of the application.

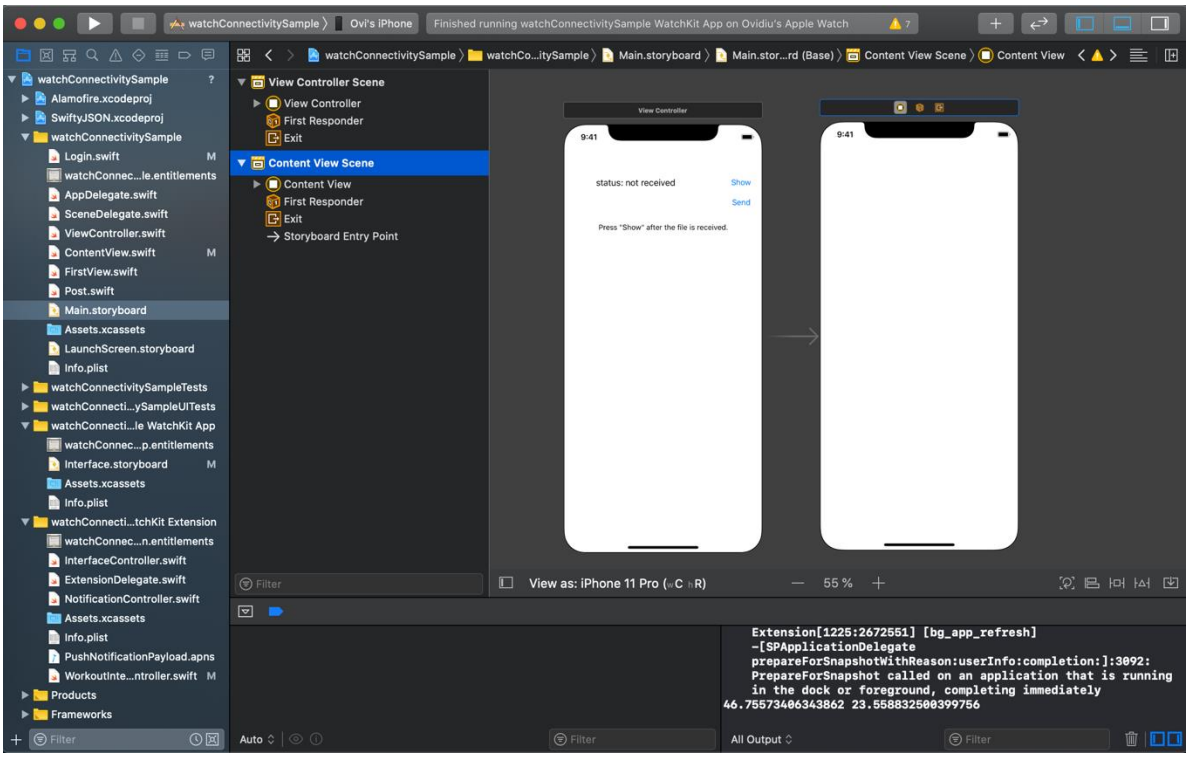


Figure 4.16 Xcode environment. iOS App main storyboard

4.2.1 SwiftUI Framework and the request forms

In the case of iOS applications, there are multiple design frameworks. SwiftUI is the newest one, with support for fluid animations, simple menus creation and automatically supports Dark Mode / Light Mode settings of the iOS interface.

With the help of SwiftUI, the login form can be written in a declarative way as seen in Figure 4.17. The Login View is declared as a struct which has a *body* member inside which the elements that will be rendered are declared. The changing variables inside the View are marked with *@State*, thus letting the compiler know it has to be aware of updates to those objects. The *@EnvironmentObject* marker is used to allow objects to be used in other views as well. Here, after request is done, its response will be saved in the *userSettings* object.

```
struct Login: View {
    @EnvironmentObject var userSettings: UserSettings
    @State var username = ""
    @State var password = ""
    @State var name = "not logged in"
    let url = URL(string: "http://192.168.0.105:5000/Api/User/login")
    var body: some View {
        NavigationView{
            Form{
                TextField("Username", text: $username)
                SecureField("Password", text: $password)
                Button(action: {
                    let loginRequest = [
                        "username" : self.username,
                        "password" : self.password
                    ]
                    let header: HTTPHeaders = [
                    "Content-Type": "application/json"
                    ]
                    AF.request(self.url!, method: .post, parameters: loginRequest, encoding: JSONEncoding.default, headers: header)
```

Figure 4.17 SwiftUI View containing a login form and a POST request

The other request forms are done in a similar way. The result is presented in Figure 4.18.

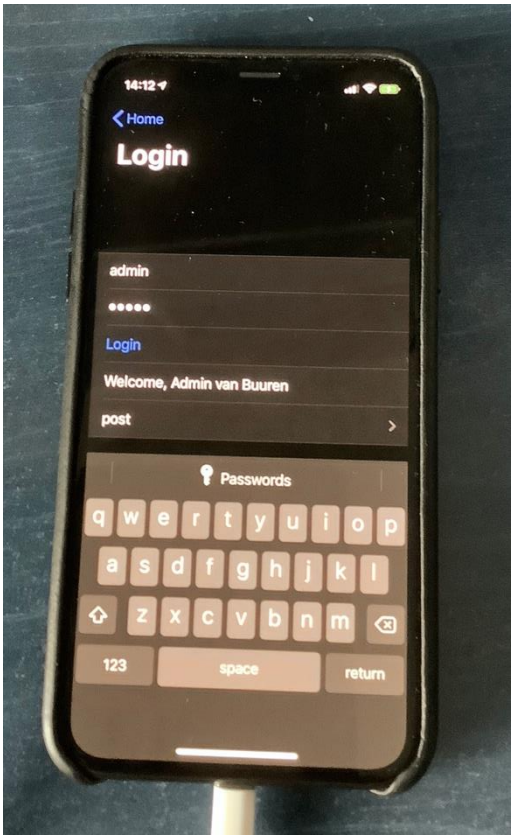


Figure 4.18 The login form on the iPhone

4.2.2 Communication with the watch application

The communication with the watch application is done in a very similar way with what has been presented in Chapter 4.1.4: a session object is instantiated, and the specific methods are overloaded. The file received is converted into a string and its contents are

shown in a List component. The button that sends the token to the watch is defined in Figure 4.19 and the Home page of the application is shown in Figure 4.20.

```
Button(action:{
    self.wcProvider.token = self.userSettings.token
    self.wcProvider.sendTokenToWatch()
}){
    Text("Send token to watch")
}

func sendTokenToWatch(){
    let data: [String: Any] = ["Bearer ": self.token as Any]
    session.sendMessage(data, replyHandler: nil, errorHandler: nil)
}
```

Figure 4.19 SwiftUI Button that sends data to the watch

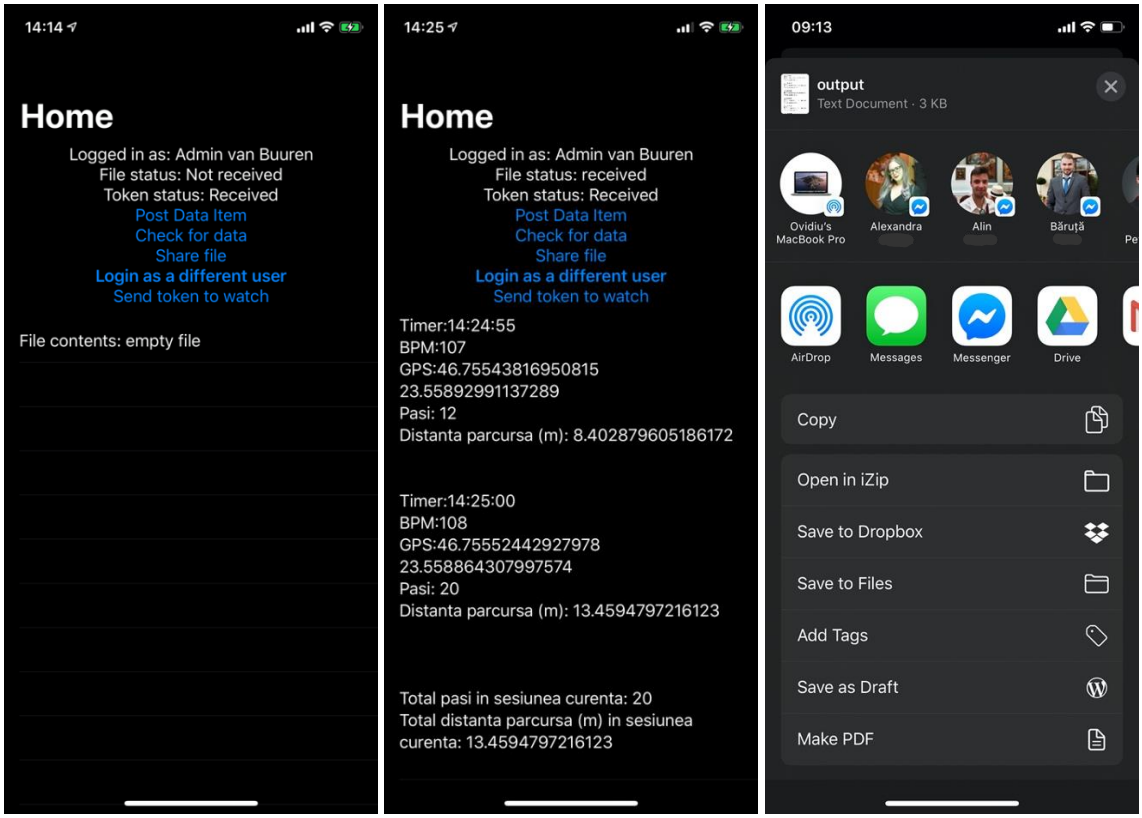


Figure 4.20 Home page of the application.

Left: The application before receiving the file.

Middle: The application after receiving the file.

Right: The share sheet of the received file

The share sheet (Figure 4.20 in the right) opens a sharing menu which allows the file to be sent via usual methods like e-mail or other supported applications. AirDrop (the standard sharing method between Apple devices) was used in this case.

4.3 Embedded Application

4.3.1 Microcontroller and development board

The implementation of the embedded application started with implementing the basic internet functionalities over Wi-Fi on a low cost ESP8266 module with an OLED of 128x32 pixel resolution (Figure 4.21).

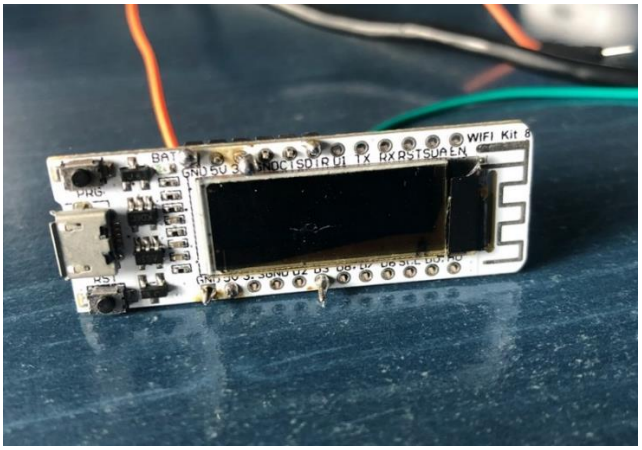


Figure 4.21 ESP8266

After a short time, its Wi-Fi chip stopped working, maybe due to bad soldering or other factors, so I decided to go with another, similar but better, microcontroller and chose the ESP32 with an OLED of 128x64 pixel resolution (Figure 4.22). Both development boards are produced by Heltec Automation. This allowed for an easy transition of the current progress.

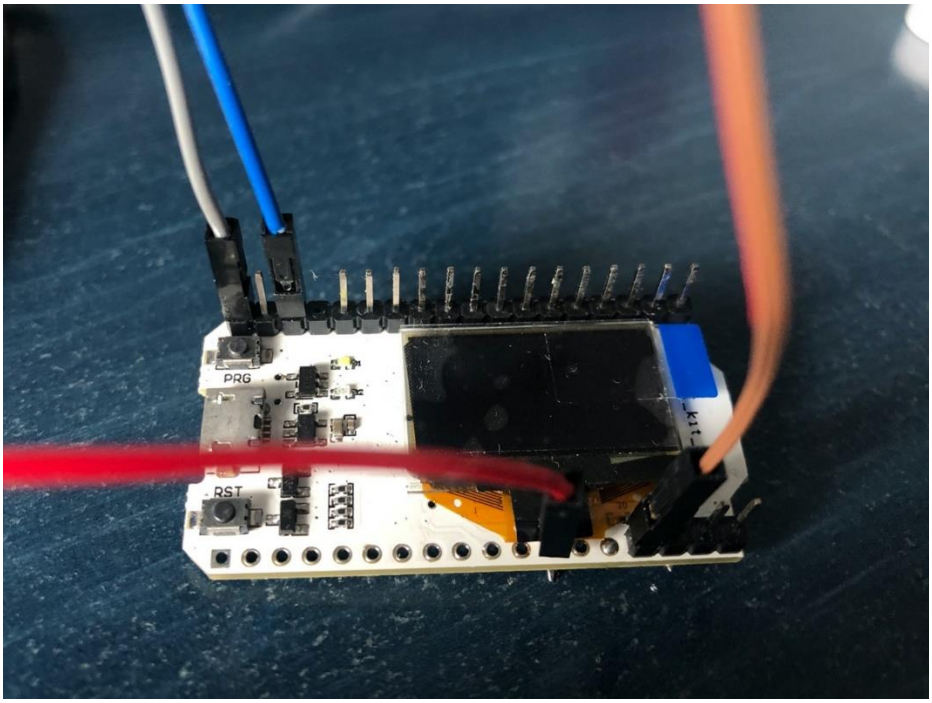


Figure 4.22 ESP32

The pinout diagram for the ESP32 is presented in Figure 4.23.

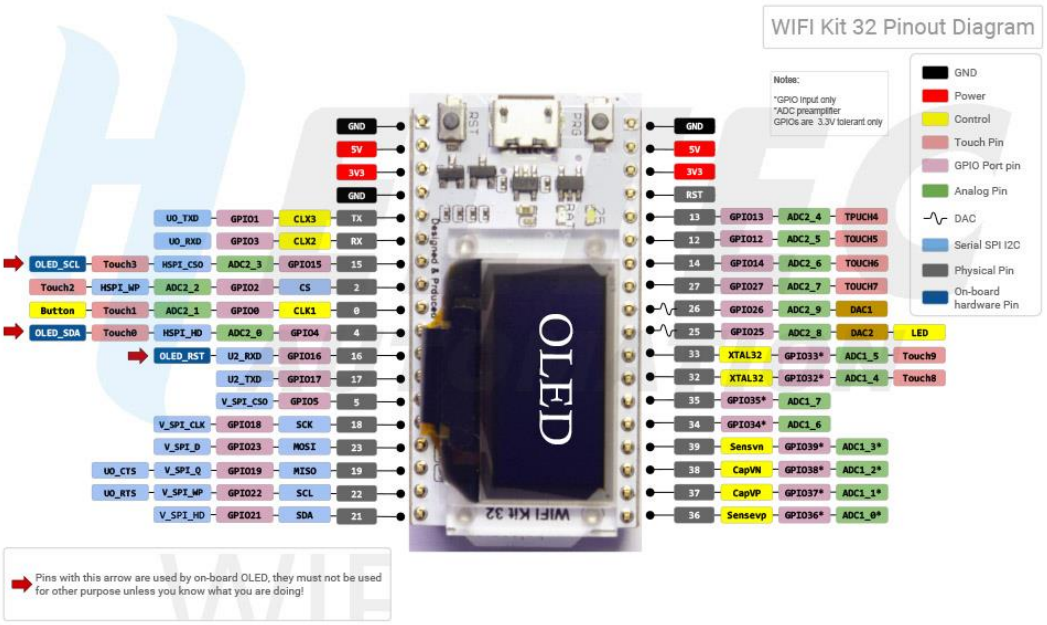


Figure 4.23 ESP32 pinout diagram [65]

An approximation of the electric circuit is represented in Figure 4.24, made with the circuito.io tool.

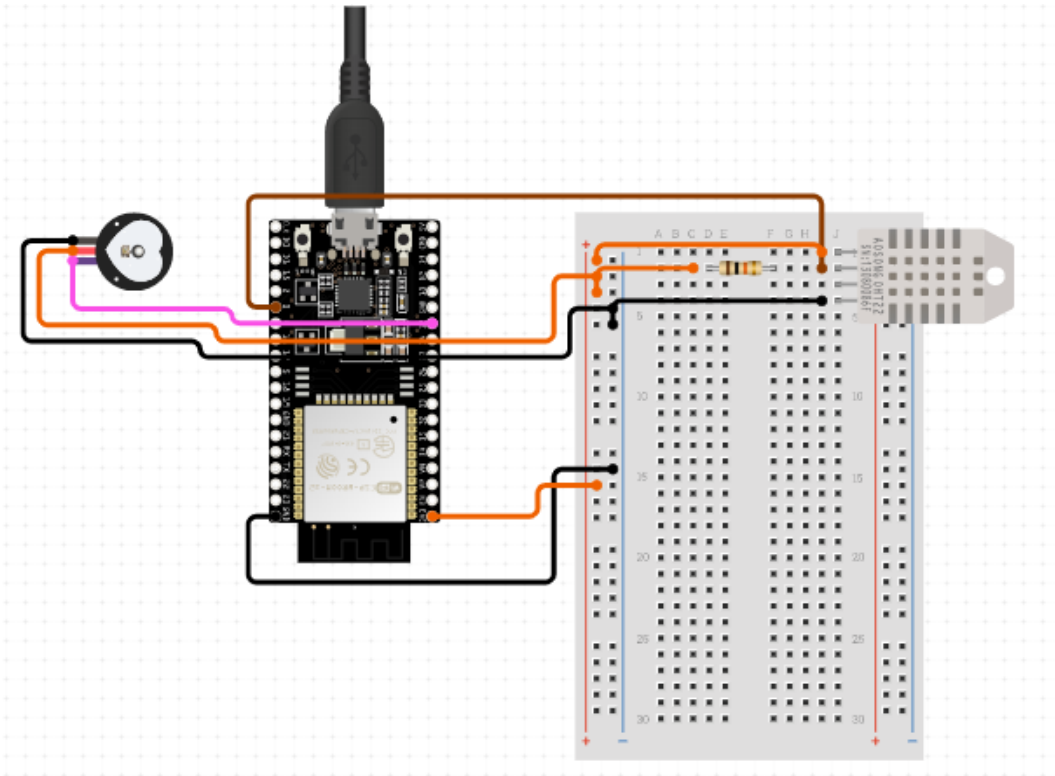


Figure 4.24 Electric circuit

The final solution is shown in Figure 4.25. The used temperature sensor is a slightly different version (DHT22 in Figure 4.24 and DHT11 in Figure 4.25). Also, the GPIO connections are not all the same.

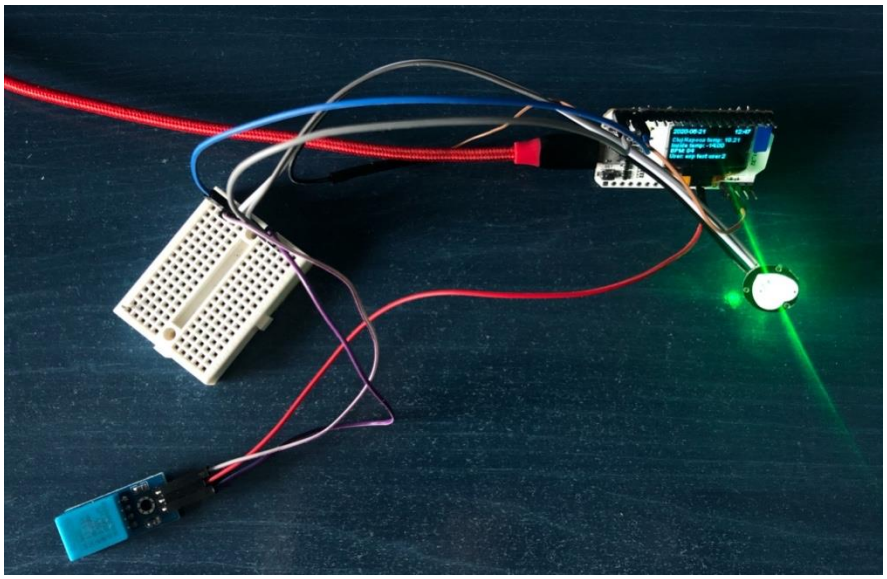


Figure 4.25 Final embedded solution

The breadboard was used only to multiplicate Vcc and Ground signals and the data pins are connected directly onto the board.

4.3.2 Communication with the API

The Wi-Fi connectivity is done using the *WiFi* and *WiFiMulti* libraries of the Arduino IDE (Figure 4.26), then by instantiating a *WiFiMulti* object and calling its method to add as an access point (AP) the local Wi-Fi network (Figure 4.27).

```
#include <WiFi.h>
#include <WiFiMulti.h>

#include <ArduinoJson.h>
#include <HTTPClient.h>
```

Figure 4.26 Arduino Wi-Fi communication libraries

```
WiFiMulti wifiMulti;
wifiMulti.addAP("MyWifiRouter", "MyWifiPassword");
```

Figure 4.27 Connecting the microcontroller to local Wi-Fi network

Getting data from the server is done using the *HTTPClient* library and the *ArduinoJson* helper library (Figure 4.26). In Figure 4.28 an *HTTPClient* object is

instantiated, with its *begin* method being called, having the API URL as an argument. Calling its *GET* method sends the HTTP GET request to the server. Calling the *getString* method converts the JSON response to a string.

```
HTTPClient http;

http.begin("http://192.168.0.105:5000/api/general/hour");

int httpCode = http.GET();

if (httpCode == HTTP_CODE_OK) {
    String payload = http.getString();
    serialCon.println(payload);
    CurrentDate = payload;
}
```

Figure 4.28 Getting the date from the API

By using a *DynamicJsonDocument* object and calling the *deserializaJson* function, an easy to access array is created (Figure 4.29).

```
DynamicJsonDocument doc(capacity);
deserializeJson(doc, payload);

float weatherTemperature = doc["main"]["temp"].as<float>();
Temp = weatherTemperature;
```

Figure 4.29 Deserializing a JSON in Arduino

Sending HTTP POST requests (Figure 4.30) is similar to the HTTP GET method. Notice the *addHeader* method responsible for letting the API know it is sending a JSON and an authorization token. The *POST* method receives as an argument a string in a JSON format.

```
HTTPClient http;

http.begin("http://192.168.0.105:5000/Api/DataItem");
http.addHeader("Content-Type", "application/json");
http.addHeader("Authorization", "Bearer " + authToken);

int httpResponseCode = http.POST("{\"heartBpm\":" + String(BPM) + ",\"

String response = http.getString();
```

Figure 4.30 Posting data to the API

4.3.3 Authentication

As seen in Figure 4.30, the *authToken* variable is sent to the API for authentication and authorization purposes. *authToken* is a globally declared String object. The *authenticate* function receives the Username and Password which are initialized in the source code, in the setup process. Then, a POST request is done, and the response object contains the authentication token generated by the API, as well as the user's display name and location from the database. Those are being deserialized and stored for later use.

4.3.4 Using the OLED display

For displaying information on the screen, Heltec Automation provides a library and some basic methods (Figure 4.31).

```
Heltec.display -> clear();
Heltec.display -> drawString(0, 35, "BPM: " + String(BPM));
Heltec.display -> display();
```

Figure 4.31 Heltec Automation display methods

The display should be cleared to turn off the pixels before turning any more pixels on. Then, the *drawString* method creates the matrix with information about which pixel should be turned on. It receives as arguments the X and Y position from where to start drawing, with the starting point (0,0) being the top left pixel and the ending point (123, 63) being the bottom right pixel. The third argument is the string which the method will draw. The *display* method needs to be called in order to turn on any of the pixels on the screen.

4.3.5 Displaying user information

From the authentication method, the application knows the user's name and location, which are displayed on the screen. The date and time are known from an API call.

The outside weather information uses the Open Weather API to request information about the local weather (the location the user set at register).

Heart BPM and inside temperature are known from the installed sensors.

The result can be seen in Figure 4.32.

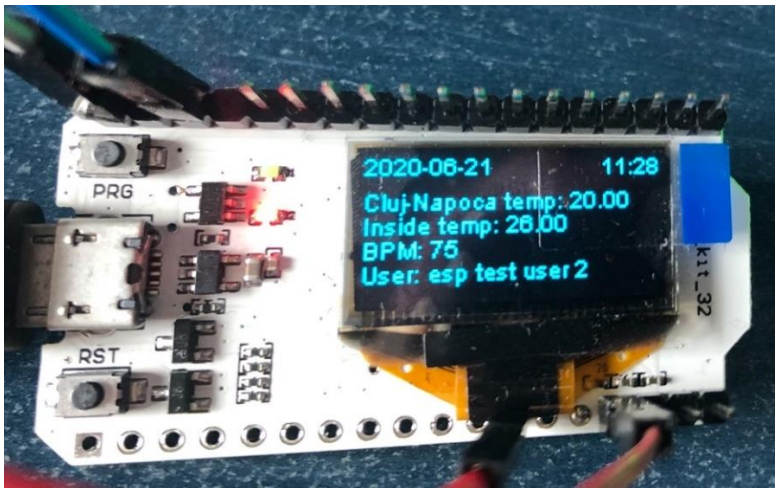


Figure 4.32 Embedded solution's display

4.3.6 Sensors

4.3.6.1 Temperature

The temperature sensor used is the DHT11 (Figure 4.33). The + pin is connected to 3.3Vcc, the - pin to Ground and the data pin to GPIO32.

Figure 4.33 DHT11 temperature sensor

Accessing its data is done with the help of the library provided by DHT, creating a *DHT* object and using its *readTemperature* method (Figure 4.34).

```
void getInsideTemp(){
  DHT dht(DHTPIN, DHTTYPE);
  if(!isnan(dht.readTemperature())) {
    insideTemp = dht.readTemperature();
  }
}
```

Figure 4.34 Reading the temperature from the sensor

4.3.6.2 Heart rate

The heart rate sensor (PPG sensor) used is a PulseSensor (Figure 4.35). It contains a green LED and a photodiode to measure light intensity. The + pin is connected to 3.3Vcc, the - pin is connected to Ground and the data pin is connected to GPIO34.

Figure 4.35 PulseSensor

Figure 4.36 presents how the *PulseSensorPlayground* object (defined in the library provided by PulseSensor) is used to get the current heart rate BPM of the user. A threshold is used to manually tune the detection of a heartbeat. The *sawStartOfBeat* method returns true when the detected signal goes over the threshold, then the *getBeatsPerMinute* method computes an estimation of number of heartbeats in a minute.

```
PulseSensorPlayground bpmSensor;
bpmSensor.analogInput(bpmPIN);
bpmSensor.setThreshold(bpmThreshold);
if(bpmSensor.sawStartOfBeat()){
  BPM = bpmSensor.getBeatsPerMinute();
}
```

Figure 4.36 Reading the heart rate

Figure 4.37 shows the output of the sensor over a period of time, using the Serial Plotter tool of the Arduino IDE.

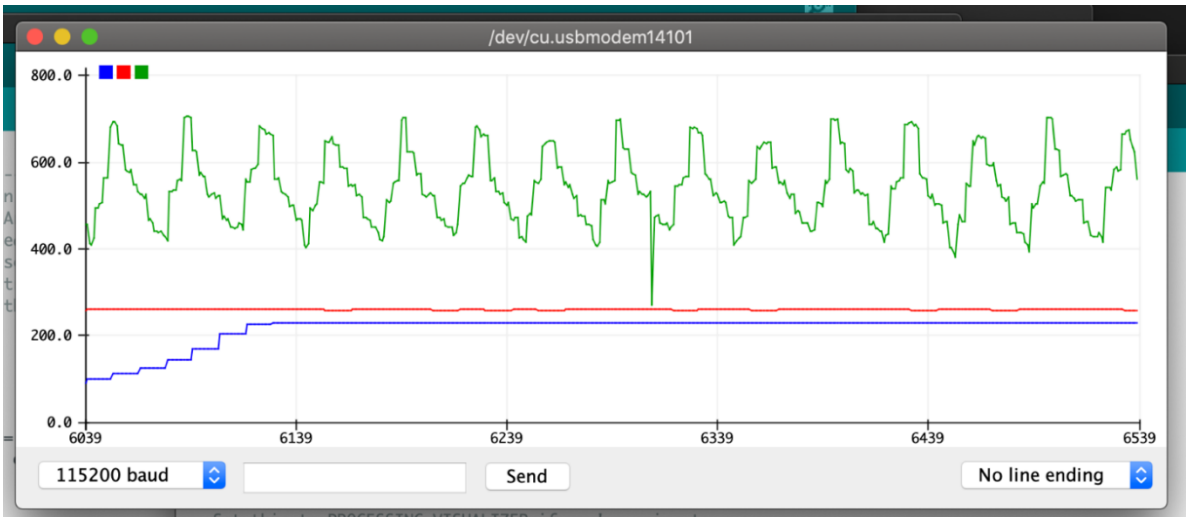


Figure 4.37 PulseSensor analog output

Unfortunately, the data can appear very corrupt sometimes, with users complaining on forums that, because it is a cheap development board, the Wi-Fi connection can interfere with the Analog to Digital Converter (ADC). This phenomenon can be seen in Figure 4.38.

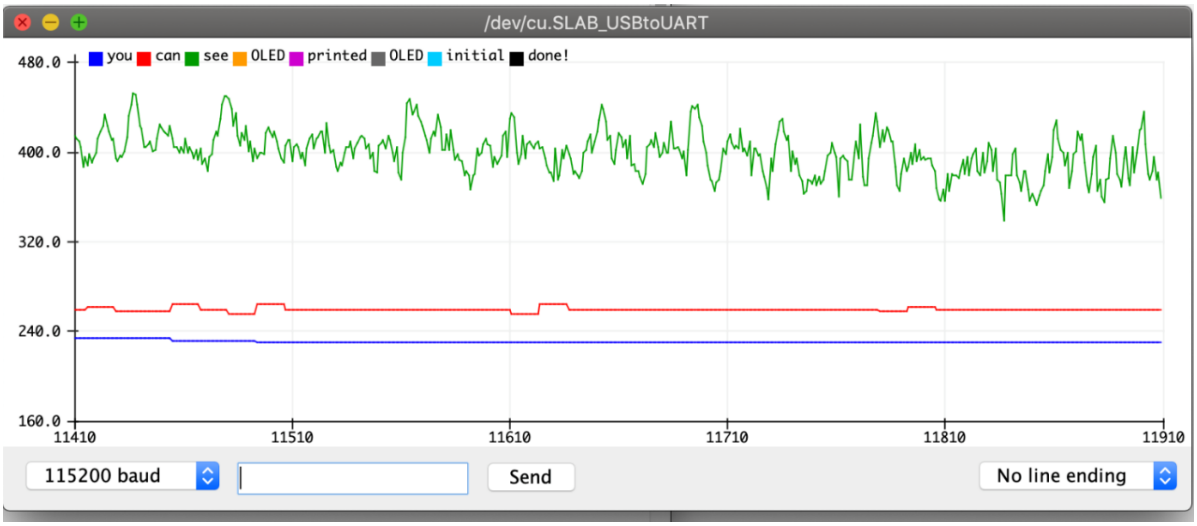


Figure 4.38 ADC disturbances caused by the Wi-Fi signal

To solve this issue, a secondary microcontroller (an Arduino Nano) has been considered to work in a master-slave configuration and send data through the I2C protocol. While this has worked very well between 2 Arduinos, the ESP32 does not support at the time such a configuration. Github user *stickbreaker* announced that he is working on a solution for that [66], but it was not available at the time of writing. So, either a different kind of sensor or a different development board should be considered to improve this functionality.

4.3.7 Main program

The main code puts together everything above: checks if there is a Wi-Fi connection, then proceeds to ask for authentication data and the values required to display information, calls the sensors functions. After that, if the authentication is successful, it sends the data as JSON objects to the API. This code runs in a loop, with a delay of 5 seconds between loops.

4.4 API server

The API server is written using the C# programming language and the .NET Core Framework. The architecture used is Model-View-Controller. Because the Views are all the other applications, only the Models and the Controllers are presented in this chapter. Data is being received and sent using RESTful services.

For the development of the API, the IDE used is Visual Studio for Mac (2019, Community License).

4.4.1 Models

Entity Framework Core is the framework used for database integration. The approach used in this application is Code First, meaning that the models are defined as classes in the code, then a migration is created. A migration (Figure 4.39) contains is an automatically generated class with 2 methods: *Up* and *Down*. The *Up* method contains information about what is being changed or added in the database structure with this migration and the *Down* method contains information about how the table was before, in case there is need to revert that at some point. This approach enables easy creation and modification of the database. Everything is structured in the Models folder.

```
public partial class AddTemperature : Migration
{
    protected override void Up(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.AddColumn<double>(
            name: "Temperature",
            table: "DataItems",
            nullable: false,
            defaultValue: 0.0);
    }

    protected override void Down(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.DropColumn(
            name: "Temperature",
            table: "DataItems");
    }
}
```

Figure 4.39 Example of a migration that adds a Temperature column in the DataItems table

To create a migration, the model class is created or modified, then the following command is used in the terminal:

```
dotnet ef migrations add {MigrationClassName}
```

Then, to apply these changes to the database:

```
dotnet ef database update
```

Models of this API are described in the Class Diagram (Figure 3.2): User, Data Item, Project. They are created here, Figure 4.40 exemplifying how the User model is defined.

```
public class User
{
    public long Id { get; set; }

    public string Username { get; set; }
    public string Password { get; set; }
    public string Name { get; set; }
    public string Location { get; set; }

    public List<DataItem> DataItems { get; set; }

    public int ProjectId { get; set; }

    public string Role { get; set; }
}
```

Figure 4.40 Example of the User class

The result is the SQLite database containing these tables (Figure 4.41).

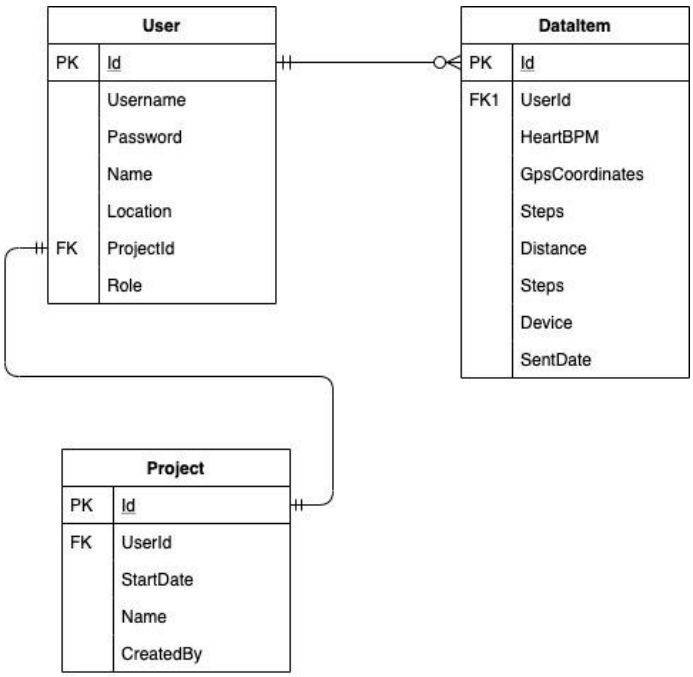


Figure 4.41 Database tables

SQLite is the database technology used because of its compatibility with Entity Framework Core and the fact that it is cross-platform, running on macOS with no issues, unlike Microsoft SQL Server, the usual database management system of choice for C# developers, which is not natively supported on non-Windows operating systems.

To access the database, the application used is DB Browser for SQLite (Figure 4.42), a free and lightweight application that also allows modifications and SQL queries execution.

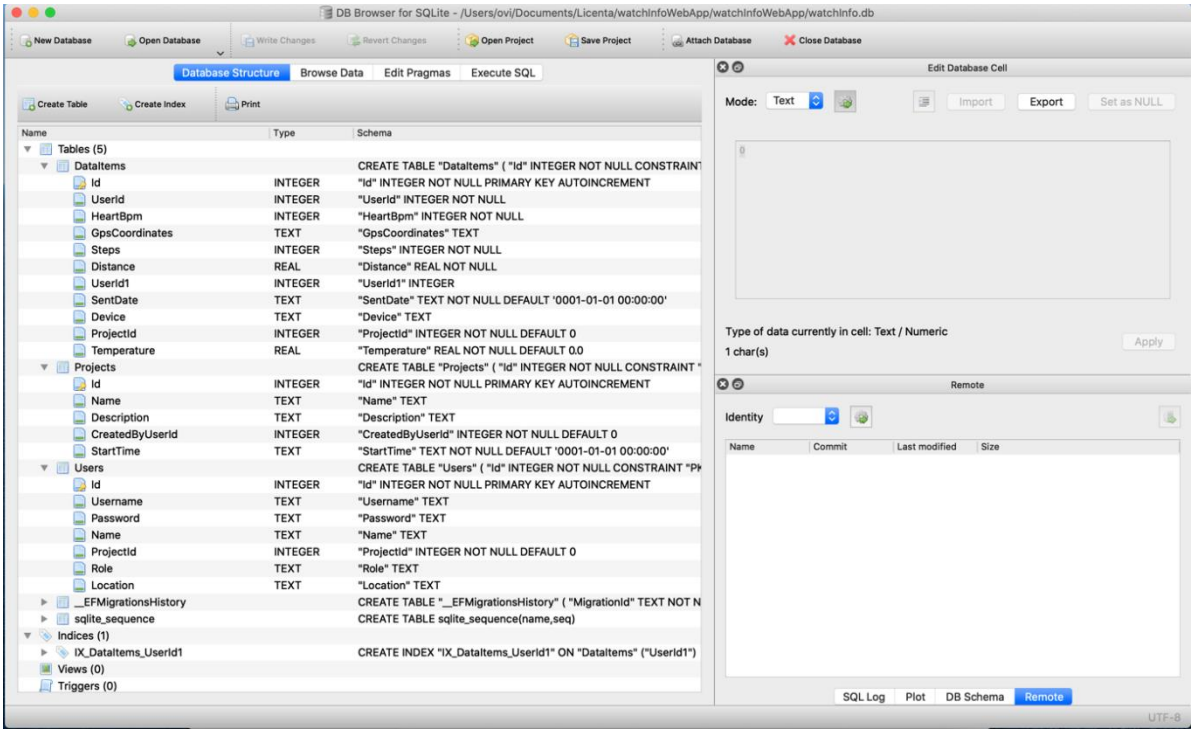


Figure 4.42DB Browser for SQLite

4.4.2 Controllers

The controllers define the logic of the API. There are 4 controllers: A General Controller, a User Controller, a Data Items Controller, and a Projects Controller, each one having its own responsibility.

One particular aspect of Entity Framework Core is that it allows the database to be referred to as an *ApplicationDbContext* object, providing methods for the communication with the database, thus allowing the developer to write C# code most of the time, that code being automatically translated into SQL commands. This also provides basic security against SQL Injection.

The most relevant functionalities of the controllers are presented in what follows.

4.4.2.1 General Controller

The General Controller (Figure 4.43) is the simplest one, having only one purpose at this time: to give the date and time whenever it receives a GET request on */Api/General/hour*. It is mostly intended so that the embedded solution automatically gets the time.

```
[ApiController]
[Route("Api/[controller]")]
public class GeneralController
{
    [HttpGet("hour")]
    public ActionResult<DateTime> GetTime()
    {
        return DateTime.Now;
    }
}
```

Figure 4.43 General Controller Date & Time functionality

4.4.2.2 User Controller

The User Controller mostly deals with the user registration (Figure 4.4.2.2.1) and user authentication (Figure 4.44).

```
[AllowAnonymous]
[HttpPost("register")]
public async Task<ActionResult<User>> CreateUser(User user)
{
    user.Password = ComputeSha256Hash(user.Password);
    _context.Users.Add(user);
    await _context.SaveChangesAsync();
    user.Password = null;

    return CreatedAtAction(nameof(GetUser), new { username = user.Username }, user);
}
```

Figure 4.44 User registration

User registration is done by sending a POST request containing a JSON object with at least a Username and a Password on */Api/User/register*. The password is then encrypted using a SHA256 encryption algorithm. The user is added to the database using the *Add* and *SaveChangesAsync* methods provided by Entity Framework Core. The password field of the User object is then made null and the user information, without the password, is sent back as a response.

The use of the AllowAnonymous tag is required, as the whole UserController class is marked as Authorized and requires an access token if anonymous requests are not explicitly allowed.

The method is marked as asynchronous, thus allowing the task to not block the entire application until it is finished, resulting in shorter waiting times.

```
[AllowAnonymous]
[HttpPost("login")]
public async Task<IActionResult> Authenticate([FromBody]User userParam)
{
    var user = await _userService.Authenticate(userParam.Username, ComputeSha256Hash(userParam.Password));

    if (user == null)
        return BadRequest(new { message = "Username or password is incorrect" });

    return Ok(user);
}
```

Figure 4.45 User authentication

The authentication (Figure 4.45) is done via a POST request on */Api/User/login*. The Authenticate method searches for the user then assigns it a token containing encoded information about its role and its identifier, data that can be extracted later from that token as Claims. This is done for a better security design, so that the user identifiers are not public. Then, the method returns a Login View Model, a class created without a password field to be sent back as a response.

```
[Authorize(Roles = "Admin")]
[HttpPut("changeProject/{newProjectId}")]
public async Task<IActionResult> ChangeMyProject(int newProjectId)
{
    var oldProjectId = claimsGetter.ProjectId(User?.Claims);
    var userId = claimsGetter.UserId(User?.Claims);
    var user = await _context.Users.SingleAsync(x => x.Id == userId);
    user.ProjectId = newProjectId;
    await _context.SaveChangesAsync();
    return NoContent();
}
```

Figure 4.46 Example of update request

Figure 4.46 presents an example of an Update (PUT) request on */Api/User/changeProject/{newProjectId}*, allowing the user to change its current project. As defined in the Class Diagram (Figure 3.2), only the Administrator should be able to change a project, so the Authorize attribute checks the token claims for the *Admin* role. Also, the user identifier and the current project identifier are taken from the token claims as well.

4.4.2.3 Data Item Controller & Project Controller

The Data Item Controller provides basic functionality to the application with methods similar to the ones above, so that each user can send data and view only his own data. More than that, to be GDPR compliant and protect the privacy of the user, anyone has the ability to delete his own Data Items (Figure 4.47), no questions asked. This is done by a DELETE request on */Api/DataItem/{id}*.

The Project Controller also provides basic functionality using the same kind of methods.

```
[Authorize]
[HttpDelete("{id}")]
public async Task<ActionResult<List<DataItem>>> DeleteMyDataItem(long id)
{
    var healthDataItem = new DataItem();
    var userId = claimsGetter.UserId(User?.Claims);
    var userRole = claimsGetter.UserRole(User?.Claims);

    //The user can delete only his data unless he is admin
    healthDataItem = await _context.DataItems.Where(x => x.Id == id && (x.UserId == userId || userRole == "Admin")).FirstOrDefaultAsync();

    if(healthDataItem != null)
    {
        _context.DataItems.Remove(healthDataItem);
        await _context.SaveChangesAsync();
    }

    return NoContent();
}
```

Figure 4.47 Example of Delete request

For more complicated queries, the Linq library can be used. Figure 4.48 exemplifies a join query between the DataItems table and the Users table, used to get information about the user knowing a specific data item.

```
[HttpGet("getProjectDataWithUser/{id}")]
public async Task<ActionResult<DataItemWithUserDto>> GetProjectDataItemWithUser(long id)
{

    var healthDataItems = from dataItem in _context.DataItems
                          join user in _context.Users
                            on dataItem.UserId equals user.Id into DataItemWithUserDto
                          from defaultValue in DataItemWithUserDto.DefaultIfEmpty()
                          where dataItem.Id == id
                          select new DataItemWithUserDto
                          {
                              Id = dataItem.Id,
                              UserId = dataItem.UserId,
                              HeartBpm = dataItem.HeartBpm,
                              GpsCoordinates = dataItem.GpsCoordinates,
                              Steps = dataItem.Steps,
                              Distance = dataItem.Distance,
                              SentDate = dataItem.SentDate,
                              Device = dataItem.Device,
                              Name = defaultValue.Name
                          };


    return await healthDataItems.FirstOrDefaultAsync();
}
```

Figure 4.48 Linq query joining 2 tables

4.5 Web Application

To create a modern, fast, responsive, single page web application, ReactJS is chosen. For the rendering part, JavaScript XML (JSX) is used as the scripting language and the Bootstrap library helps with the design elements: buttons, tables, the navigation bar. The IDE used is Visual Studio Code.

The most important functionalities of the web application are:

- Login/logout
- Register
- Home page
- Admin Panel
- Position map

Figure 4.49 shows a preview of the application, on the Home page, where user data is presented in a table with the possibility of deleting items. The menu is visible on the top right.

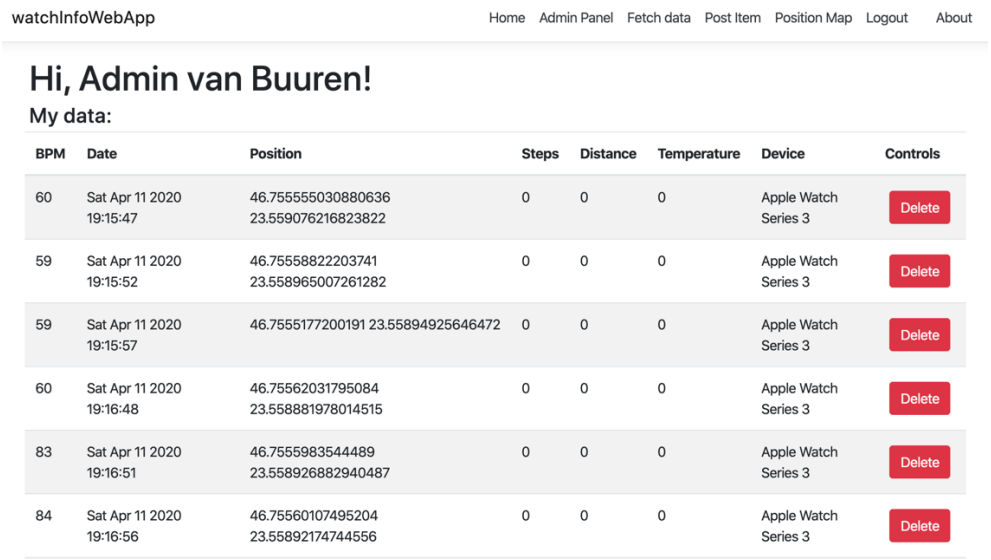


Figure 4.49 Preview of the web application

The application provides a basic user interface that allows API calls. An example of such a call is given in Figure 4.50, with a function that gets all the current user's Data Items. Notice that no arguments are given, as the user's id is known from the authentication token included in *authHeader*.

```
function getAll() {
    const requestOptions = {
        method: "GET",
        headers: authHeader(),
    };

    return fetch(`Api/DataItem/`, requestOptions).then(handleResponse);
}
```

Figure 4.50 GET request for the Data Items

Another noteworthy example is the deletion of a Data Item call presented in Figure 4.51.

```
function deleteDataItemById(id) {
  const requestOptions = {
    method: "DELETE",
    headers: authHeader(),
  };

  return fetch(`Api/DataItem/` + id, requestOptions).then(handleResponse);
}
```

Figure 4.51 DELETE request for a Data Item

After calling the *getAll* function in the initialization of the page, the table seen in Figure 4.5.1 is defined in the *render* method of the Home component using JSX in a very similar way to HTML, as in Figure 4.52.

```
<tbody>
  {dataItems.map((dataItems, index) => (
    <tr key={dataItems.id}>
      <td>{dataItems.heartBpm}</td>
      <td>{this.formatDate(dataItems.sentDate)}</td>
      <td>{dataItems.gpsCoordinates}</td>
      <td>{dataItems.steps}</td>
      <td>{dataItems.distance}</td>
      <td>{dataItems.temperature}</td>
      <td>{dataItems.device}</td>
      <td>
        <button
          className="btn btn-danger"
          onClick={() => this.deleteDataItemById(dataItems.id)}
        >
          Delete
        </button>
      </td>
    </tr>
  ))}
</tbody>
```

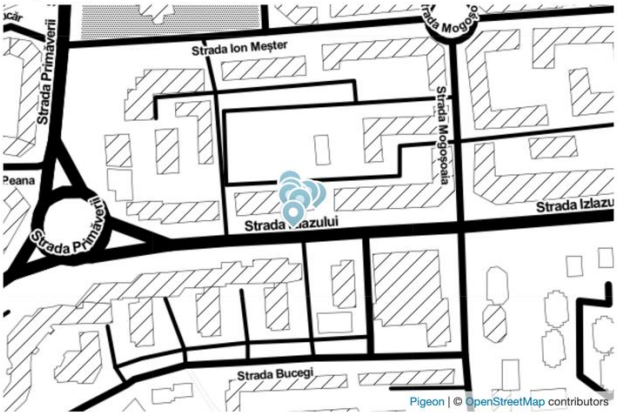
Figure 4.52 Render method of a React component

The authentication is done through the Login page. If the request is done successfully, the token received from the API is stored in the browser's local storage. After login, the user is redirected to the Home page where he or she can see the Data Items table. If the user is an administrator, the Admin Panel page will also appear in the menu.

The input forms in the application are being checked for validation in the *handleChange* method, so the user can get real-time feedback about the input.

The Position Data page (Figure 4.53) uses a Pigeon Maps [67] component to put a marker on every Data Item that has GPS coordinates. With a click action on the marker, the user can see details about that specific data item.

My data:



BPM	Date	Steps	Distance	Device
79	Sat Apr 11 2020 19:31:37	26	20	Apple Watch Series 3

Figure 4.53 Map component

The usage of the Map component is exemplified in Figure 4.54. The *gpsDataSplit* object contains the latitude, longitude, and id of the data items. If there is at least one data item, the map will be automatically centered to that point. If there are more, the first one is chosen as the center point. When a Marker component is clicked, the *handleMarkerClick* method receives as the payload the identifier of the data item and populates the details table.

```
<Map
  center={
    gpsDataSplit[0] != null
      ? [gpsDataSplit[0][0], gpsDataSplit[0][1]]
      : [0, 0]
  }
  zoom={17}
  width={600}
  height={400}
>
  {gpsDataSplit[0] != null &&
    gpsDataSplit.map(function (elem, index) {
      return (
        <Marker
          anchor={[elem[0], elem[1]]}
          payload={elem[2]}
          onClick={({ event, anchor, payload }) => {
            _this.handleMarkerClick(payload, anchor);
          }}
        />
      );
    })}
</Map>
```

Figure 4.54 Map component code example

The Admin Panel page (Figure 4.55) allows an administrator to create new projects, assign projects and view all the user data in a project.

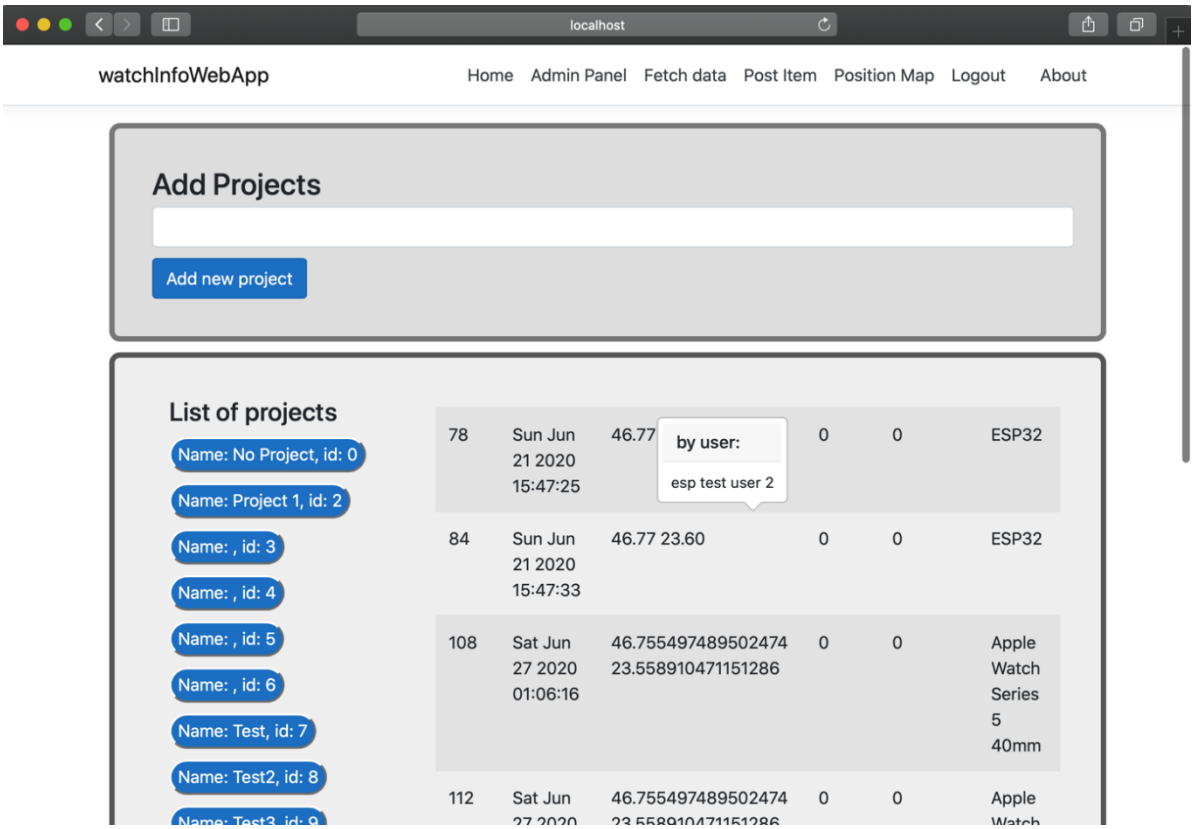


Figure 4.55 Admin Panel

The administrator can select a project and view all the data in that project. When selecting a project, the table in the right populates with data. By clicking a data row, a popup appears showing the user who posted that item. An administrator can also assign users to projects, as shown in Figure 4.56. Project id is 0 by default, the first project being automatically created as a "No project" one to have as a default.

Figure 4.56 Assigning a user to a project

5 Tools and testing

5.1 Manual testing

The components of the system were manually tested with each feature implementation. After that, their integration in the system as a whole was also tested to find and fix the various issues that had appeared. Every functionality turned out to be working as presented.

The Apple Watch application was tested in sessions of up to 30 minutes to ensure no problem would be encountered because of the limited resources and the fact that it is storing all the data in a file. The application was tested on 2 different Apple Watch Series 3 and an Apple Watch Series 5 as real devices, as well as Apple Watch Series 3, 4 and 5 in the Xcode Simulator. The iPhone companion app was tested on iPhone versions 7, 8, X, Xs and also iPhone 11 in the Xcode Simulator.

The embedded ESP32 solution was also tested in sessions of up to 30 minutes, mainly to check the Wi-Fi connection stability on the low-cost development board, as well as the application itself. Sensor functionality was also tested on an Arduino Uno and the ESP8266, without Wi-Fi connectivity.

The web application has been verified so that each function works at it should. The database has also been checked so that no invalid information or duplicate data is added by mistake. It has been tested on different screen sizes, including phones.

The API was tested with 4 clients sending data simultaneously, which it handled without any issue. It has been tested on the Macbook used for development and on a Windows desktop PC.

5.2 Unit testing

For the API, a series of unit tests have been written to test the functionalities of the controllers. The framework used is xUnit and the Moq library has been used to mock the functionality of the database. An example of a unit test is shown in Figure 5.1 and the running tests in Figure 5.2. Also, some functionalities of the API were implemented using a Test Driven Development (TDD) approach: the test was written first, then the method was implemented to make that test pass.

```
[Fact]
public void AddUser_Password_IsEncrypted()
{
    //Arrange
    User user = new User
    {
        Username = "username",
        Password = "test",
        Name = "Name"
    };

    //Act

    var result = userController.CreateUser(user);

    //Assert
    var addedUser = dbContext.Users.First(x => x.Username == user.Username);
    Assert.True(addedUser.Password == ComputeSha256HashTest("test"));
}
```

Figure 5.1 Unit test for the password encryption

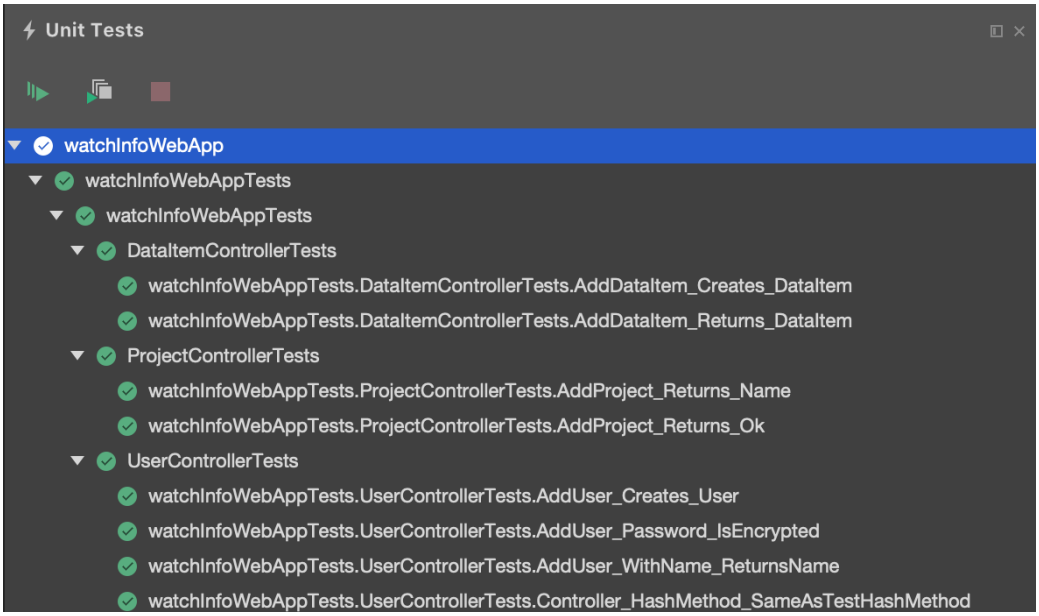


Figure 5.2 Running unit tests

5.3 Postman

Postman is the tool used to check the API calls and their responses before they are implemented in an application. By giving clear messages for each call, every RESTful method can be easily tested, with or without authorization. An example of such a test is given in Figure 5.3.

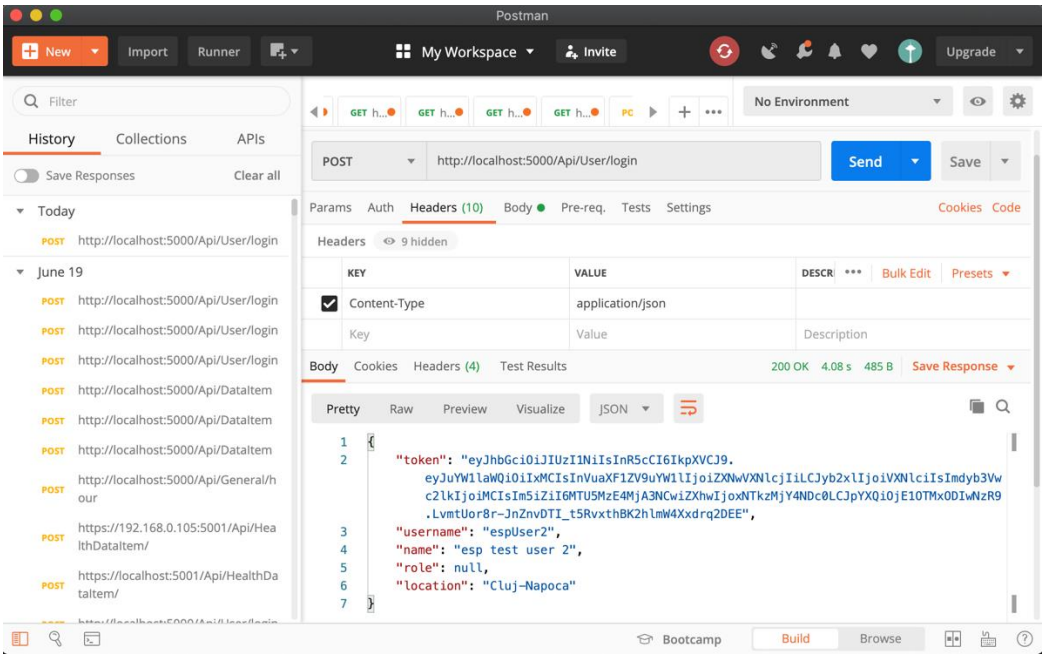


Figure 5.3 Response of a Postman login test

5.4 Version control – Git

Version control in this system was introduced after the first version of the Apple Watch application was working and has been updated ever since. It has approximately 50 commits and 2 branches and has been used a couple of times to revert modifications that created more bugs than they solved. The tool used is Github, with a free-for-students *PRO* account providing the ability to upload this project in a private repository. It also provides an interesting graph with the programming languages used (Figure 5.4).

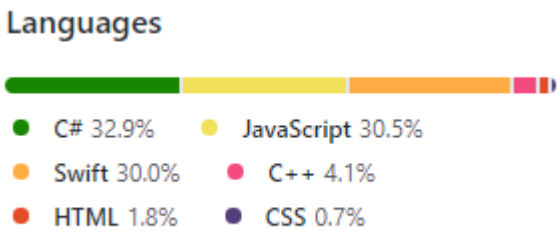


Figure 5.4 Programming languages used in the system

6 Conclusions

6.1 Results

The Apple Watch application showed good results, with accurate enough data to get a good quality set of data. During 30 minutes of running with other normal processes running as well in the background, it had drained approximately 5% the battery from an Apple Watch Series 3 which is almost the same as a usual workout application. This data may vary due to a lot of factors: the battery used in this test was more than 2 years old, the screen brightness is automatically set by the watch and so on. Anyway, the app can run for at least 8 hours of continuously monitoring and sending data.

The embedded solution was not tested for power consumption. However, running the device for more than 5 minutes made it run at very high temperatures, to the point it was too hot to touch. This has been partially fixed by switching the clock frequency in the Arduino IDE from 240MHz (default for the ESP32) to 80MHz (the minimum available).

The heart rate sensor in the embedded solution turned out to be highly unreliable.

With both solutions gathering data for approximately 30 minutes, each one sent more than 250 data items, making them a viable source for big data algorithms.

6.2 Problems encountered

As explained in *Chapter 4.3*, the embedded solution started on a development board and ended on another. The first board was ordered from China before the pandemic, but luckily enough I was able to find a similar one produced by the same company, and there were very little changes to make the code function properly. Also, there were problems with the heartrate sensor.

The Apple Watch development was troublesome. After learning the Swift programming language, it took a long time understanding the permissions imposed by Apple systems. Without properly creating the screen that requests Health data permissions on the watch, the application was crashing without any message. I also needed to exchange emails with someone from Apple and file a bug report.

At one point in March 2020, I had an issue with a framework used in the project, and it was a bug known by Apple, but it was solved only in April 2020. During this time, iOS and watchOS development were impossible for these applications.

6.3 Going further

Going further, the application is open to extensions in various ways.

The embedded solution can be extended to have more sensors, like a pedometer, an accelerometer, a GPS module and so on. Finding a viable kit to make this application a usable wearable device would also be an improvement. A better heart rate sensor should

be considered. The (minimum) 80MHz of the ESP32 may be a higher than necessary clock frequency, so a lower speed processor should be considered to optimize energy consumption. Also, a mobile app companion can be created.

Every year, a new Apple Watch is released with improved or even new sensors, which this system could take advantage of. Also, Android versions can be developed.

The system could be improved with a machine learning algorithm that can predict areas of stress at a specific time, as the solution is able to generate a good quality set of training data.

One use of the system could be selling it to government institutions that can then pay people to gather sets of data in various cities during a day. With that much data, a model can be created that can be used to see stress levels during a day in a specific area (most likely some of the crowded intersections) and send police teams to ensure people remain calm and civilized.

Another use could be selling the application to a private company which has employees that are always on the move, so the employer can make sure the employees are handling or not the various situations they are in.

7 Bibliography

[1] N. Kalid, A. Zaidan, B. Zaidan, O. H. Salman, M. Hashim and H. Muzammil, "Based Real Time Remote Health Monitoring Systems: A Review on Patients Prioritization and Related "Big Data" Using Body Sensors information and Communication Technology," *Journal of Medical Systems,* vol. 42, no. 30, 2018.

[2] M. U. Ahmed, M. Björkman, A. Čaušević, H. Fotouhi and M. Lindén, "An Overview on the Internet of Things for Health Monitoring Systems," in *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, vol 169*, Springer, Cham, 2016.

[3] P. Kakria, N. K. Tripathi and P. Kitipawang, "A Real-Time Health Monitoring System for Remote Cardiac Patients Using Smartphone and Wearable Sensors," *International Journal of Telemedicine and Applications,* 2015.

[4] R. Burke, C. Midgley and A. Dratch, "Active Monitoring of Persons Exposed to Patients with Confirmed COVID-19," *Morbidity and Mortality Weekly Report,* vol. 69, no. 9, pp. 245-246, 2020.

[5] H. Liang and G. Acharya, "Novel corona virus disease (COVID-19) in pregnancy: What clinical recommendations to follow?," *Acta Obstet Gynecol Scand,* vol. 99, pp. 439-442, 2020.

[6] M. Billinghurst and T. Starner, "Wearable devices: new ways to manage information," *Computer,* vol. 32, no. 1, pp. 57-64, 1999.

[7] J. Block, "How An Apple Watch Helped Save A High School Football Player's Life," 22 September 2015. [Online]. Available: https://www.huffpost.com/entry/paul-houle-apple-watch_n_5601878de4b08820d91a4688. [Accessed 22 April 2020].

[8] J. R. Nance and A. L. Mammen, "Diagnostic evaluation of rhabdomyolysis," *Muscle Nerve,* vol. 51, no. 6, pp. 793-810, 2015.

[9] M. S. Patel, D. A. Asch and a. K. G. Volpp, "Wearable devices as facilitators, not drivers, of health behavior change.," *Jama,* vol. 313, no. 5, pp. 459-460, 2015.

[10] D. Castaneda, A. Esparza, M. Ghamari, C. Soltanpur and H. Nazeran, "A review on wearable photoplethysmography sensors and their potential future applications in health care," *International Journal of Biosensors & Bioelectronics,* no. 4, pp. 195-202, 2018.

[11] C. Chu, H. Chiang and J. Hung, "Dynamic heart rate monitors algorithm for reflection green light wearable device," *International Conference on Intelligent Informatics and Biomedical Sciences,* pp. 438-445, 2015.

[12] J. Spigulis, L. Gailite, A. Lihachev and R. Erts, "Simultaneous recording of skin blood pulsations at different vascular depths by multiwavelength photoplethysmography," *Applied Optics,* vol. 46, no. 10, pp. 1754-1759, 2007.

[13] P.-H. Lai and I. Kim, "Lightweight wrist photoplethysmography for heavy exercise: motion robust heart rate monitoring algorithm," *Healthcare Technology Letters,* vol. 2, no. 1, pp. 6-11, 2015.

[14] Z. Ge, P. W. C. Prasad, N. Costadopoulos, A. Alsadoon, A. K. Singh and A. Elchouemi, "Evaluating the accuracy of wearable heart rate monitors," *2016 2nd International Conference on Advances in Computing, Communication, & Automation (ICACCA) (Fall), Bareilly, 2016, pp. 1-6. .*

[15] V. Genovese, A. Mannini and A. M. Sabatini, "A Smartwatch Step Counter for Slow and Intermittent Ambulation," *IEEE Access,* vol. 5, pp. 13028-13037, 2017.

[16] J. A. Lee, S. M. Williams and K. R. L. Dale D. Brown, "Concurrent validation of the Actigraph gt3x+, Polar Active accelerometer, Omron HJ-720 and Yamax Digiwalker SW-701 pedometer step counts in lab-based and free-living settings," *Journal of Sports Sciences,* vol. 33, no. 10, pp. 991-1000, 2015.

[17] T. CB, A. JJ and S. EE, "Accuracy of a step counter during treadmill and daily life walking by healthy adults and patients with cardiac disease," *BMJ Open,* vol. 7, no. 3, 2017.

[18] Y. Bai, P. Hibbing, C. Mantis and G. J. Welk, "Comparative evaluation of heart rate-based monitors: Apple Watch vs Fitbit Charge HR," *Journal of Sports Sciences,* vol. 36, no. 15, pp. 1734-1741, 2018.

[19] Veerabhadrappa, M. M. P. and M. Renninger, "Tracking Steps on Apple Watch at Different Walking Speeds," *J GEN INTERN MED,* vol. 33, pp. 795-796, 2018.

[20] "Calibrating your Apple Watch for improved Workout and Activity accuracy," Apple Inc., 2019 October 2017. [Online]. Available: https://support.apple.com/en-us/HT204516. [Accessed 25 April 2020].

[21] K. Kuo and F. Wu, "A 2.4-GHz/5-GHz Low Power Pulse Swallow Counter in 0.18-μm CMOS Technology," *IEEE Asia Pacific Conference on Circuits and Systems,* pp. 214-217, 2006.

[22] B. A. Abdou, "Commercializing eSIM for Network Operators," *IEEE 5th World Forum on Internet of Things (WF-IoT),* pp. 616-621, 2019.

[23] K. W. Nixon, X. Chen and Y. Chen, "Footfall - GPS polling scheduler for power saving on wearable devices," *21st Asia and South Pacific Design Automation Conference (ASP-DAC),* pp. 563-568, 2016.

[24] M. Collotta, G. Pau, T. Talty and O. K. Tonguz, "Bluetooth 5: A Concrete Step Forward toward the IoT," *IEEE Communications Magazine,* vol. 56, no. 7, pp. 125-131, 2018.

[25] S. S. M. Nino Isakadze, "How useful is the smartwatch ECG?," *Trends in Cardiovascular Medicine,* 2019.

[26] "Apple Watch Series 5 - Health," Apple Inc., [Online]. Available: https://www.apple.com/apple-watch-series-5/health/. [Accessed 01 06 2020].

[27] "Taking an ECG with the ECG app on Apple Watch Series 4 or later," Apple Inc., [Online]. Available: https://support.apple.com/en-us/HT208955. [Accessed 01 06 2020].

[28] R. A. Popa, A. J. Blumberg, H. Balakrishnan and F. H. Li, "Privacy and accountability for location-based aggregate statistics," *Proceedings of the 18th ACM conference on Computer and communications security,* pp. 653-666, 2011.

[29] Z. Riaz, F. Durr and K. Rothermel, "On the Privacy of Frequently Visited User Locations," *17th IEEE International Conference on Mobile Data Management (MDM),* pp. 282-291, 2016.

[30] R. Schubert and I. Marinica, "Facebook Data: Sharing, Caring, and Selling," *International Conference on Cyber Situational Awareness, Data Analytics And Assessment (Cyber SA),* pp. 1-3, 2019.

[31] E. S. Udoh and A. Alkharashi, "Privacy risk awareness and the behavior of smartwatch users: A case study of Indiana University students," *2016 Future Technologies Conference (FTC),* pp. 926-931, 2016.

[32] C. Xu, P. H. Pathak and P. Mohapatra, "Finger-writing with smartwatch: A case for finger and hand gesture recognition using smartwatch," *Proceedings of the 16th International Workshop on Mobile Computing Systems and Applications. ACM,* 2015.

[33] R. Darbara, P. K. Senb, D. P. and D. Samanta, "Using Hall Effect Sensors for 3d space Text Entry on Smartwatches," *Procedia Computer Science,* vol. 84, pp. 79-85, 2016.

[34] J. Seo, K. Kim, M. Park, M. Park and K. Lee, "An analysis of economic impact on IoT under GDPR," *International Conference on Information and Communication Technology Convergence (ICTC),* pp. 879-881, 2017.

[35] G. Y. Lee, K. J. Cha and H. J. Kim, "Designing the GDPR Compliant Consent Procedure for Personal Information Collection in the IoT Environment," *2019 IEEE International Congress on Internet of Things (ICIOT),* pp. 79-81, 2019.

[36] A. Coos, "EU vs US: How Do Their Data Privacy Regulations Square Off?," CoSoSys Ltd., 17 01 2018. [Online]. Available: https://www.endpointprotector.com/blog/eu-vs-us-how-do-their-data-protection-regulations-square-off. [Accessed 03 05 2020].

[37] L. F. d. l. Torre, "A guide to the California Consumer Privacy Act (CCPA)," 15 10 2019. [Online]. Available: https://medium.com/golden-data/a-guide-to-the-california-consumer-privacy-act-ccpa-3a916756ed36. [Accessed 03 05 2020].

[38] K. Boda, A. M. Foldes, G. G. Gulyas and S. Imre, "User Tracking on the Web via Cross-Browser," *Lecture Notes in Computer Science,* pp. 31-46, 2012.

[39] W. Yang, J. Hu, S. Wang and Z. Guanglou, "Securing Mobile Healthcare Data: A Smart Card based Cancelable Finger-vein Bio-Cryptosystem," *IEEE Access,* 5 Jun 2018.

[40] H. Brito, A. Gomes, Á. Santos and J. Bernardino, "JavaScript in mobile applications: React native vs ionic vs NativeScript vs native development," *2018 13th Iberian Conference on Information Systems and Technologies (CISTI),* pp. 1-6, 2018.

[41] V. Oliveira, L. Teixeira and F. Ebert, "On the Adoption of Kotlin on Android Development: A Triangulation Study," *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER),* pp. 206-2016, 2020.

[42] C. G. García, J. P. Espada, B. C. P. García-Bustelo and J. M. C. Lovelle, "Swift vs. Objective-C: A New Programming Language," *International Journal of Interactive Multimedia and Artificial Intelligence,* vol. 3, no. 3, pp. 74-81, 2015.

[43] R. Nunkesser, "Beyond Web/Native/Hybrid: A New Taxonomy for Mobile App," *2018 IEEE/ACM 5th International Conference on Mobile Software Engineering and Systems (MOBILESoft),* pp. 214-218, 2018.

[44] W. K. Bodin, D. Jaramillo, S. K. Marimekala and M. Ganis, "Security challenges and data implications by using smartwatch devices in the enterprise," *2015 12th International Conference & Expo on Emerging Technologies for a Smarter World (CEWIT),* pp. 1-5, 2015.

[45] J. Manning and P. Buttfield-Addison, Swift Development for the Apple Watch, O'Reilly, 2016.

[46] "The Swift Programming Language - Swift 5.2," Apple Inc., [Online]. Available: https://docs.swift.org/swift-book. [Accessed 03 May 2020].

[47] N. Jacque, "Swift 5.3 Release Process," Apple Inc., 2020 March 25. [Online]. Available: https://swift.org/blog/5-3-release-process/. [Accessed 2020 May 03].

[48] A. Inc., The Swift Programming Language (Swift 5.2), Apple Inc., 2014.

[49] J. Alori, "Swift: Google's bet on differentiable programming," Tryolabs | Machine learning and data science consulting, 2020 Apr 2. [Online]. Available: https://tryolabs.com/blog/2020/04/02/swift-googles-bet-on-differentiable-programming/. [Accessed 2020 May 03].

[50] J. Varma, SwiftUI for Absolute Beginners: Program Controls and Views for iPhone, iPad and Mac Apps, Melbourne, VIC, Australia: Apress, 2019.

[51] Readdle, "Swift for Android: Our Experience and Tools," Readdle, 1 June 2018. [Online]. Available: https://blog.readdle.com/why-we-use-swift-for-android-db449feeacaf. [Accessed 2020 05 03].

[52] A. Inc., "Getting Started with Swift on Android," 2020 Jan 25. [Online]. Available: https://github.com/apple/swift/blob/master/docs/Android.md. [Accessed 03 May 2020].

[53] SCADE, "SCADE Documentation," 2019. [Online]. Available: https://docs.scade.io/docs. [Accessed 2020 May 03].

[54] C. Manuel, E. Morocho, W. Lim and D. Kwon, "Using body-measurement indices and wrist-type photoplethysmography signals to categorize consumer electronic users' health state through a smartwatch application," *2018 International Conference on Electronics, Information, and Communication (ICEIC),* pp. 1-4, 2018.

[55] R. Toulson and T. Wilmshurst, Fast and Effective Embedded Systems Design: Applying the ARM mbed, Second Edition, Newnes, 2016.

[56] A. Bakir, Program the Internet of Things with Swift for iOS, Tokyo, Japan: Apress, 2018.

[57] H. Garg and M. Dave, "Securing IoT Devices and SecurelyConnecting the Dots Using REST API and Middleware," *2019 4th International Conference on Internet of Things: Smart Innovation and Usages (IoT-SIU),* pp. 1-6, 14-19 April 2019.

[58] S. Ahmed and Q. Mahmood, "An authentication based scheme for applications using JSON web token," *2019 22nd International Multitopic Conference (INMIC),* pp. 1-6, 2019.

[59] Z. Liu and B. Gupta, "Study of Secured Full-Stack Web Development," *Proceedings of 34th International Conference on Computers and Their Applications,* vol. 58, pp. 317-324, 2019.

[60] S. Deshmukh, D. Mane and A. Retawade, "Building a Single Page Application Web Front-end for E-Learning site," *2019 3rd International Conference on Computing Methodologies and Communication (ICCMC),* pp. 985-987, 27-29 March 2019.

[61] "MATLAB Online," Mathworks, [Online]. Available: https://www.mathworks.com/products/matlab-online.html. [Accessed 03 05 2020].

[62] S. Distefano, M. Scarpa and A. Puliafito, "From UML to Petri nets: The PCM-based methodology," *IEEE Transactions on Software Engineering,* vol. 37, no. 1, pp. 65-79, 2011.

[63] R. C. Martin, Clean Code - A handbook of Agile Software Craftsmanship, Pearson Higher Education, 2008.

[64] N. J. Dingle, W. J. Knottenbelt and T. Suto, "PIPE2: A Tool for the Performance Evaluation of Generalised Stochastic Petri Nets," *ACM SIGMETRICS Performance Evaluation Review (Special Issue on Tools for Computer Performance Modelling and Reliability Analysis),* vol. 36, no. 4, pp. 34-39, 2009.

[65] H. Automation, " WiFi Kit 32 Documentation," [Online]. Available: https://heltec.org/project/wifi-kit-32/. [Accessed 21 06 2020].

[66] C. Todd, "Github Repository - Arduino-Esp32," [Online]. Available: https://github.com/stickbreaker/arduino-esp32/tree/master/libraries/Wire. [Accessed 21 06 2020].

[67] M. Andra, "Github Repository - Pigeon-Maps," [Online]. Available: https://github.com/mariusandra/pigeon-maps. [Accessed 21 06 2020].