



Escola d'Enginyeria de Telecomunicació i  
Aeroespacial de Castelldefels

UNIVERSITAT POLITÈCNICA DE CATALUNYA

# TREBALL FINAL DE GRAU

**TÍTOL DEL TFG:** Application for Medical Devices and Data Management

**TITULACIÓ:** Grau en Enginyeria Telemàtica

**AUTOR:** Arnau Mir Hurtado

**DIRECTOR:** Toni Oller Arcas

**DATA:** 8 de febrer de 2023

**Títol:** Aplicació per a Gestió de Dispositius i Dades Mèdiques

**Autor:** Arnau Mir Hurtado

**Director:** Toni Oller Arcas

**Data:** 8 de febrer del 2023

## Resum

Durant els últims 30 anys, el sector sanitari ha vist una sorprenent evolució en molts dels seus camps, especialment pel que fa a la digitalització de dades clíniques, el que ha comportat una gran millora de l'eficiència, no només pel relatiu a diagnòs com a tal, sinó també en termes d'emmagatzematge de dades i el seu subseqüent anàlisi. La capacitat d'emmagatzemar, compartir i accedir a dades mèdiques digitals ha demostrat ser clau quan es tracta de distribuir expedients mèdics de pacients independentment de la seva localització, disposar de dades mèdiques a temps real, portar a terme estudis de mitjà i gran abast, així com per la detecció, diagnòs i potencial prevenció per a futurs casos, entre d'altres.

Malgrat tot, aquesta digitalització està normalment limitada a centres mèdics o ambients hospitalaris, encara que moltes dades mèdiques són també generades fora d'aquests, mitjançant mesures individuals a casa amb aparells mèdics més rutinaris com termòmetres, tensiòmetres, bàscules, etc. Per tant, en la majoria d'aquests casos, els resultats són solament observats i analitzats pel pacient, i no digitalitzats per a posteriors usos o avaluació per part de professionals sanitaris, el que comporta una pèrdua substancial d'informació que podria ser útil per a diagnòs remota o per multitud d'usos beneficiosos en altres àmbits.

L'anterior problema estableix les arrels d'aquest projecte, l'objectiu del qual ha estat crear una aplicació mòbil, desenvolupada en Flutter, que permeti als usuaris connectar els seus dispositius mèdics, prendre mesures, emmagatzemar-les i accedir al seu registre des d'una base de dades FHIR, seguint el format de dades mèdiques estandarditzat HL7 FHIR, per tal que aquestes dades puguin ser utilitzades per un professional sanitari per a seguiment remot de l'estat de salut del pacient i potencial detecció d'anomalies, entre una llista d'altres funcionalitats que també han estat afegides per aportar un valor afegit a l'aplicació.

Aquest document exposa detalladament com s'ha complit aquest objectiu, des de la teoria darrere l'emmagatzematge i gestió de dades mèdiques, fins com el producte final s'ha desenvolupat en l'àmbit tècnic, així com algunes idees pel relatiu a línies de continuïtat que es podrien seguir per a futures millores de l'aplicació.

**Title:** Application for Medical Devices and Data Management

**Author:** Arnau Mir Hurtado

**Director:** Toni Oller Arcas

**Date:** January 8, 2023

## Overview

Over the last 30 years, healthcare has seen an astounding evolution in a vast scope of its fields, especially as far as the digitalization of clinical information is concerned, which has led to a massive improvement of the efficiency, not only relative to diagnosis itself but also in terms of data storage and the subsequent analysis of the latter. The ability of storing, sharing and accessing digital medical data has proven to be key when it comes to distributing patients' health records regardless of location, getting real-time health data, carrying out medium and large-scope studies, as well as premature symptoms detection and the following respective diagnosis and potential prevention for future cases, among others.

However, this digitalization is usually limited to medical centers or hospital environments, yet medical data is often generated outside these as well by means of individual self-measurement and more rutinary medical devices such as thermometers, pressure bracelets, scales, etc. Thus, in most of these cases, the results are just observed, analyzed and stored momentarily by the patient, and not digitalized for further uses or evaluation by health-care professionals, leading to a loss of information that could be useful for remote diagnosis or other beneficial uses in different realms.

The previous problem sets the roots for this project, which aim has been to create a mobile application, developed using Flutter, that allows users to connect their medical devices, take measurements, store and access their previous measurements' data from a FHIR database in the official medical standardized data format HL7 FHIR so that this data can be used by a healthcare professional to remotely keep track of a patient's health state and potentially detect anomalies, among a list of other functionalities that have been added as well to provide an added value to the application.

This document closely covers how this objective has been accomplished, from the background theory of the storage and management of medical data, to how the final product has been achieved technically, and also some ideas on the direction that could be taken for future improvements of its functionalities and usefulness.

To my beloved partner,  
for all these years and all hazelnut coffees while I was working

To my family,  
for the constant support and encouragement

To the Finland Erasmus family,  
for all the memories attached to this thesis that you made me have

## LIST OF ACRONYMS AND ABBREVIATIONS

HL7	Health Level Seven
CDA	Clinical Document Architecture
SPL	Structured Product Labeling
FHIR	Fast Healthcare Interoperability Resources
RIM	Reference Information Model
UML	Unified Modelling Language
OOP	Object-Oriented Programming
HER	Electronic Health Record
PHR	Personal Health Record
HMD	Hierarchical Message Description
SNOMED	Systematized Nomenclature of Medicine
LOINC	Logical Observation Identifier Names and Codes
SDK	Software Development Kit
MVC	Model-View-Controller
BLoC	Business Logic Components
JWT	JSON Web Token
UI	User Interface
API	Application Programming Interface

# INDEX

<b>INTRODUCTION</b> .....	<b>1</b>
<b>CHAPTER 1. MEDICAL DATA MANAGEMENT</b> .....	<b>4</b>
<b>1.1. How medical data is stored: The HL7 FHIR Standards</b> .....	<b>4</b>
1.1.1. HL7v2 and HL7v3.....	5
1.1.2. HL7 FHIR.....	10
1.1.3. International coding systems for observations.....	15
<b>1.2. Data Structures and Analysis</b> .....	<b>17</b>
1.2.1. Data generation and storage.....	17
1.2.2. Data access and analysis.....	31
<b>CHAPTER 2. BEHEALTHAPP</b> .....	<b>36</b>
<b>2.1. General architecture of the solution</b> .....	<b>36</b>
<b>2.2. Introduction to BeHealthApp's functionalities</b> .....	<b>38</b>
2.2.1. Profiles / Roles of the user.....	38
2.2.2. Medical Devices Management.....	39
2.2.3. Measurements and Automatic Anomaly Detection.....	42
2.2.4. Groups and Shared Data.....	44
2.2.5. User credentials and profile management.....	51
<b>2.3. Backend elements</b> .....	<b>53</b>
2.3.1. HAPI FHIR API.....	54
2.3.2. NodeJS API Gateway.....	55
2.3.3. MongoDB.....	59
<b>2.4. User Interface Development</b> .....	<b>61</b>
2.4.1. Technology choice: Flutter.....	61
2.4.2. States Management: BLoC Pattern Architecture.....	65
<b>2.5. User Authorization: JWT</b> .....	<b>69</b>
<b>CONCLUSIONS AND CONTINUITY</b> .....	<b>72</b>
<b>ANNEX A. HL7v3 XML message analysis</b> .....	<b>74</b>
<b>ANNEX B. CODE</b> .....	<b>78</b>
<b>BIBLIOGRAPHY</b> .....	<b>79</b>

## INDEX OF FIGURES

Fig. 1.1 HL7v2 message example (ORU^R01) .....	6
Fig. 1.2 HL7 UML RIM.....	8
Fig. 1.3 FHIR API response given an incorrect resource structure.....	12
Fig. 1.4 List of FHIR Resources classified in layers and use fields .....	13
Fig. 1.5 Human-readable fields in an Observation Resource fragment .....	14
Fig. 1.6 Different coding systems in an Observation Resource fragment.....	16
Fig. 1.7 Unique LOINC coding system in an Observation fragment .....	17
Fig. 1.8 UML Diagram of a Patient Resource .....	18
Fig. 1.9 Complete Data Structure of a Patient Resource.....	19
Fig. 1.10 Example of a Patient Resource's fields used in the project .....	20
Fig. 1.11 UML Diagram of a Metadata Resource .....	21
Fig. 1.12 Human-readable information of the Patient .....	22
Fig. 1.13 HumanName Resource structure .....	23
Fig. 1.14 ContactPoint Resource structure.....	24
Fig. 1.15 Address Resource structure .....	25
Fig. 1.16 UML Diagram of a complete Observation Resource .....	26
Fig. 1.17 Example of an Observation Resource's fields used in the project.....	27
Fig. 1.18 Coding Resource Structure .....	28
Fig. 1.19 List of category codes of an Observation .....	29
Fig. 1.20 Different types of values depending on the Observation context.....	30
Fig. 1.21 Multi-component Observation from a blood pressure measurement .	31
Fig. 1.22 UI visualization of a patient's measurements .....	32
Fig. 1.23 Most relevant requests supported by the FHIR server .....	33
Fig. 1.24 Patient's Observations request.....	33
Fig. 1.25 Request and response from the FHIR server .....	34
Fig. 1.26 Anomaly detection code implementation.....	35
Fig. 2.1 General architecture of the solution.....	37
Fig. 2.2 Role selection in the registration process within the application .....	39
Fig. 2.3 Medical devices management page within the application .....	40
Fig. 2.4 Connection process of a linked device and disconnection alerts.....	41
Fig. 2.5 UML structure of a FHIR Device Resource .....	42
Fig. 2.6 Temperature measurement and daily record with anomalies .....	43

Fig. 2.7 List of anomalies in the profile section.....	44
Fig. 2.8 List of groups created by the manager .....	45
Fig. 2.9 Creation process of a group by a manager .....	46
Fig. 2.10 Group request accepted by the patient.....	47
Fig. 2.11 List of patients of a group from the manager's perspective .....	47
Fig. 2.12 Patient's data from the manager's perspective.....	48
Fig. 2.13 Chat service between manager and patient .....	49
Fig. 2.14 UML diagram of the Extension Resource .....	50
Fig. 2.15 Group's performance and statistics page .....	51
Fig. 2.16 Login (left) and register (right) pages.....	52
Fig. 2.17 Hashed password stored in the MongoDB database.....	52
Fig. 2.18 User's profile (left) and settings (right) pages .....	53
Fig. 2.19 Server's settings and setup file.....	55
Fig. 2.20 Group Schema in the NodeJS server .....	56
Fig. 2.21 Group Request Schema in the NodeJS server.....	56
Fig. 2.22 Mongoose Mapping between NodeJS and MongoDB.....	57
Fig. 2.23 The four beHealthApp collections seen from MongoDB Compass....	57
Fig. 2.24 FHIR Patient request data to the NodeJS server .....	58
Fig. 2.25 User verification before request.....	58
Fig. 2.26 Patient's endpoints in the NodeJS server.....	59
Fig. 2.27 Relational database (SQL) example.....	60
Fig. 2.28 Non-relational database (NoSQL) example.....	60
Fig. 2.29 Real example from the project of a Group document in MongoDB ...	61
Fig. 2.30 Cross-platform demonstration with Android and web .....	62
Fig. 2.31 Flutter widget's structure .....	63
Fig. 2.32 Spanish (left) and English (right) JSON files .....	64
Fig. 2.33 BLoC general structure.....	65
Fig. 2.34 BLoC Hierarchy .....	66
Fig. 2.35 BLoC entities used in the project.....	67
Fig. 2.36 Authorization BLoC states (left) and events (right) definition.....	67
Fig. 2.37 Authorization BLoC logic definition.....	68
Fig. 2.38 Mapping function between states and widgets .....	68
Fig. 2.39 Structure of a JSON Web Token (JWT) .....	69
Fig. 2.40 Signature process of the JWT payload in the NodeJS server .....	70
Fig. 2.41 The application stores the received token from the NodeJS server ..	70



Fig. 2.42 The application attaches the token as an authorization header.....	71
Fig. 2.43 Server verification of the user's permissions .....	71
Fig. A.1 HL7v3 Transmission Wrapper.....	74
Fig. A.2 HL7v3 Trigger Event Control Act Wrapper.....	75
Fig. A.3 Observation Event (I). Measurement results and parameters.....	76
Fig. A.4 Observation Event (II). Author section .....	76
Fig. A.5 Observation Event (III). Record target section .....	76
Fig. A.6 Observation Event (IV). Original order information. ....	77

## INTRODUCTION

The digitalization of medical data has proven to be a catalyst regarding the rate of improvement in most fields of modern medicine, enabling the healthcare industry to reach a high level of automation and data connectivity that greatly reduces the complexity that might initially be associated to certain tasks, such as diagnosis or large population studies. The digitalization of patients' health records grants healthcare professionals instant access to the most up-to-date data, with a complete traceability of previous diseases, treatments, clinical measurements, among others, which directly translates to more accurate future prescriptions and more precise and convenient refill requests, for instance. As a summary, clinical data digitalization enables a more holistic approach as far as decision-making approaches in different fields of medicine is concerned.

The motivation for this project comes from the detection of a loss of potentially useful data coming from healthcare-related measurements carried out outside a medical context or, generalizing, not carried out by healthcare professionals that would put them in the patient's health record afterwards. An example would be an individual taking his body temperature with a thermometer or any other rutinary medical device that is generally owned by most people in a regular basis; the result of the measurement might be particularly useful for him, but it won't be registered anywhere and so, it will be out of reach for his doctor, for example. This might not be a big deal in a considerable percentage of situations, but there are some in which it would, such as when considering individuals with some pathological conditions both physical and mental such as reduced-mobility or agoraphobia, among others, which might lead to the situation where it's very difficult for a doctor to keep track of the health state of his patient. So, in general, it can be stated that the objective of this thesis is to provide a base framework for the utilization of daily health-related data generated by simple medical devices for wider case, such as remote diagnoses or big data studies.

BeHealthApp is a mobile application that contains a set of tools which cover the previously mentioned need, enabling a user to connect medical devices that allow a Bluetooth connection to his phone, and store the results in the official standard for the exchange, integration, sharing and retrieval of electronic health information (HL7). It also includes some features for automatic detection of health anomalies and groups management where a data manager can access to data of his assigned patients to ensure the normalcy of their measurements, allowing a certain degree of remote diagnosis in some particular cases.

To achieve the mentioned product, it has been necessary to develop a backend with NodeJS, that acts like a gateway between the mobile application made with Flutter and an external HAPI FHIR HL7 database that stores the information related with the medical measurements of the patient, and at the same time is connected to a non-relational database based on MongoDB. The latter stores the user's information and other necessary data that cannot be stored in the former. Please, note that more specific information about every single

component regarding the architecture of the solution will be closely approached in separate sections of this document.

The document has been divided in three main chapters approaching the solution from different perspectives.

In the first chapter, the reader will find all the theory behind the storage and management of electronic health information used in the modern industry, especially emphasizing on how individual measurements of a patient are stored as for the standardized structure and the definition of the attributes, especially those that have a remarkable importance for this specific solution.

In the second chapter, a close look to the functionalities of the application together with a deep analysis of the architecture used to carry out the software to achieve the desired results is carried out. All the technical descriptions behind the product can be found in this chapter, including data management information, UI creation and security implementations.

And finally, together with the conclusions, an continuity analysis is carried out, giving some ideas on how future versions of the application should evolve to improve its degree of usefulness.

## CHAPTER 1. MEDICAL DATA MANAGEMENT

So far, the importance of the digitalization of medical data has already been discussed, but one already finds a big challenge in the very first step of the mentioned process; how can this data be stored in an efficient, scalable, traceable and, if possible, simple way so that the given data can be accessed, shared and globally used within the healthcare industry? To begin with, the data must be clearly stored following an international standard to make sure everyone can use it equally without incompatibility problems. A standard is simply a commonly agreed way of carrying out something; in this case, to structure electronic healthcare-related data.

Another immediate question that might come to one's mind, once the data has been generated and stored, is how can this data be analyzed and used for practical cases.

The aim of this first chapter is to carry out a deep analysis of the possibilities that the current standard for medical data storage offers, and to clarify how the solution that this project sets out will use it in this first version.

### 1.1. How medical data is stored: The HL7 FHIR Standards

The set of standards that were created by HL7 International to fulfil the mentioned purpose are the HL7 Standards (Health Level Seven).

HL7 provides a framework for the exchange, integration, sharing and retrieval of electronic health information. By means of this set of international standards, healthcare providers are able to transfer clinical and administrative data between software applications. These standards contain the definition of how information is packaged, setting the language, structure and data types that have to be used to ensure the seamless integration between systems. HL7 standards support, not only clinical practice data, but also the delivery, management and evaluation of an enormous variety of health services. This ability to exchange information should help to minimize the tendency for medical care to be geographically isolated, leading to the possibility of carrying out large-scale studies involving worldwide data.

Furthermore, the HL7 Standards are recognized as the most commonly used in the world. For this reason, the medical data coming from devices measurements that the application developed in this project will follow these standards, giving the solution a large scope of possibilities in terms of interconnection and integration in the healthcare industry. As it will be closely approached in a following section, this project will specifically use the FHIR (Fast Healthcare Interoperability Resources) implementation.

The most commonly used and implemented standards and methodologies within the HL7 set are the following ones:

- HL7v2 Messaging Standard
- HL7v3 Messaging Standard
- Clinical Document Architecture (CDA)
- Continuity of Care Document (CCD)
- Structured Product Labeling (SPL)
- Clinical Context Object Workgroup (CCOW)
- Fast Healthcare Interoperability Resources (FHIR)
- Arden Syntax
- Claims Attachments
- Functional specification of Electronic Health Record (EHR) and Personal Health Record (PHR) systems
- GELLO

To individually define each of the implementations in a close manner is not within the scope of this project given the fact that, as mentioned before, the proposed solution only handles relatively simple data coming from routine medical devices using the HL7 FHIR standard. However, it's especially interesting to mention some of the particularities that some of them offer, considering that they all share the same common purpose of allowing the exchange of electronic medical data.

HL7v2 and HL7v3 will be addressed individually and down to the last detail along the next chapters but, essentially, they are messaging standards that define the basis of an interoperability specification for health and medical transactions, setting the roots for other HL7 standards that handle data in a more specialized manner as far as its type and applications are concerned. Some examples of the latter are the CDA standard, which sets a model based on HL7v3 (consequently, in XML, as it will be seen) for clinical documents, CCD, which consists on a US specification for the very specific aim of exchanging medical summaries, based on CDA, SPL for encoding all the published information relative to a particular medicine, CCOW, used for the visual integration of user applications, or GELLO, which is an interoperability specification for the visual integration of user applications.

So, throughout the following chapters, only HL7v2, HL7v3 and, specially, FHIR will be covered.

### **1.1.1. HL7v2 and HL7v3**

Although the solution proposed in this project opts for using the FHIR implementation for a list of reasons that will be covered in the following section, it is far from being the most worldwide used implementation of FHIR Standards for electronic medical data. The HL7v2 is arguably the most widely implemented so far and, of course, it allows the exchange of clinical data between systems. It is designed to support both a central patient care system

and a more distributed environment where data is stored in departmental systems.

Before HL7v2, every interface between systems had its own specifications as a consequence of the continuous creation of custom designs, constantly requiring programming on the part of sending and receiving application vendors to make them compatible. So, because of the lack of a standard collection of patient attributes or other health-related information that could allow a unification leading to a compatibility between systems, interfaces were expensive.

Note that rarely did commercial teams share proprietary data and information on how their application are built, which was the main reason why it was so difficult for other teams to build compatible applications.

But fortunately, as has already been mentioned, some like-minded healthcare community members finally agreed on creating a volunteer group to make interfacing easier, leading to the creation of HL7 and its first official version HL7v2.

The Version 2 Messaging Standard was first released in October 1987 as an Application Protocol for Electronic Data Exchange in Healthcare Environments. Version 2.7 is the last update to the Version 2 Standard, and it was published in 2011. Generally, all 2.X versions are backward-compatible with earlier versions, as the HL7v2 standard allows applications to ignore message elements they do not expect. HL7v2 targets both healthcare IT vendors and healthcare providers.

Systems using HL7v2 transmit ASCII text-based messages containing information about a great variety of events to one another. Examples of these events could be when the result of a measurement carried out on a patient has to be stored, when a doctor prescribes medication to a patient, or other less directly clinical-related data, such as when a patient is admitted to a hospital.

In Fig. 1.1, the structure of an HL7v2 message can be seen, specifically it represents the particular use case of an ORU^R01 (Observation Result R01) message, which consists on an unsolicited transmission of an observational result. This message was generated after the results of an observation were received and needed to be communicated to the system that ordered that measurement.

```
MSH|^~\&|GHH LAB|ELAB-3|GHH OE|BLDG4|200202150930||ORU^R01|CNTRL-3456|P|2.4<cr>
PID|||555-44-4444||EVERYWOMAN^EVE^E^^^L|JONES|19620320|F|||153 FERNWOOD DR.^
^STATESVILLE^OH^35292|||(206) 3345232|(206) 752-121|||AC555444444||67-A4335^OH^20030520<cr>
OBR|1|845439^GHH OE|1045813^GHH LAB|15545^GLUCOSE|||200202150730|||
555-55-5555^PRIMARY^PATRICIA P^^^^MD^^^|F|||444-44-4444^HIPPOCRATES^HOWARD H^^^^MD<cr>
OBX|1|SN|1554-5^GLUCOSE^POST 12H CFST:MCNC:PT:SER/PLAS:QN|^182|mg/dl|70_105|H||F<cr>
```

**Fig. 1.1** HL7v2 message example (ORU^R01)

The syntax encoding in HL7v2 is commonly referred to as the vertical-bar syntax. As it can be seen, the message contains 4 main fields:

- **MSH (Message Header):** In this section, one can find the sender, the receiver, the generation date and time, as well as the message type. In the example of Fig. 1.1, the sender is GHH LAB, located in ELAB-3, the receiver is GHH OE, located in BLDG4, the message was generated on 15/02/2002 at 09:30, and the message type, as already mentioned, is ORU^R01.
- **PID (Patient Identification):** This segment contains demographic information of the patient. In the example, the patient was Eve E. Everywoman, she was born on 20/03/1962, and she lives in Statesville OH. Her hospital ID can also be found in this section, and it is 555-44-4444.
- **OBR (Observation Request):** It identifies the observation itself in the realm of its original request, meaning how it was originally ordered. The measurement request was 15545^GLUCOSE, and it was ordered by Patricia Primary MD, and later carried out by Howard Hippocrates MD.
- **OBX (Observation):** This last segment simply contains the result of the measurement, which in this case is 182 mg/dl.

HL7v2 provides a lot of benefits, given that it supports the majority of common interfaces used in the healthcare industry globally, leading to a reduction of implementation costs, and it even provides a framework for negotiations of what is not in the standard. Also, another benefit for new organizations that decide to implement HL7v2 as their messaging standard is that 95% of US healthcare organizations and more than 35 countries implement it.

So, due to its widespread use, the presence of HL7v2 protocol will still be largely present in healthcare messaging in the near future, despite the creation of new implementations that show significant improvements over the original version. The latter were created as a consequence of a series of limitations. Some examples of these limitations regarding HL7v2 are its lack of traceability between messages, events and fields, the fact that it doesn't support object-oriented technologies, not featuring "Plug and Play" capabilities to detect the addition of a new input or output device and automatically activate the appropriate control software, having an implicit information model, not explicit, leading to inconsistency across message types, it requires controlled vocabularies, it's limited to a single encoding syntax, there isn't an explicit support for security functions, it has long implementation times and no one-to-many data exchange capabilities are supported, reducing the interoperability on an organizational level. Thus, HL7v2 definitely needed to be improved, leading to the creation of HL7v3.

HL7v3 was first released in late 2005, and its creation was strongly influenced by governments and medical information users rather than clinical interface specialists. In general terms, the goal of this new version was to increase the worldwide adoption of the standard, define a consistent data model and, above

all, to provide a more precise and solid standard by the creation of a completely new implementation that wouldn't be hindered by legacy issues, and so, that has to be completely disengaged of the original HL7v2 protocol. Of course, the latter leads to one of the first problems of this new version, yet the main reason why it hasn't been widely adopted, as to adopt HL7v3, users would need to deploy a whole new infrastructure of HL7v3-based applications and to implement interfaces between the new ones and the old HL7v2 interfaces that are still in use. In other words, not only HL7v2 couldn't be completely replaced by its evolved version, because of the vast usage of the former in the healthcare industry context, but also, if new HL7v3 interfaces wanted to be implemented, an extra work of interfacing to make the older ones able to communicate with the newest ones and vice versa would also be required.

Despite the complexity, as far as its integration in the healthcare industry is concerned, HL7v3 provides important improvements that will set the basis for the creation of the FHIR implementation (addressed in the next chapter). Among the characteristics of HL7v3, one can find the following ones:

- It introduces a Reference Information Model (RIM) for the first time, which is a structured specification of the information within the healthcare scope. RIM uses Unified Modelling Language (UML) to graphically represent a classes model that allows the contextualization of any event that occurs in any healthcare service. From the RIM, the specifications for the generation of specific messages for the different fields are built (Figure 1.2). The RIM has around 70 classes that come from a main core with 6 fundamental classes. It represents the business-logic of any healthcare context.

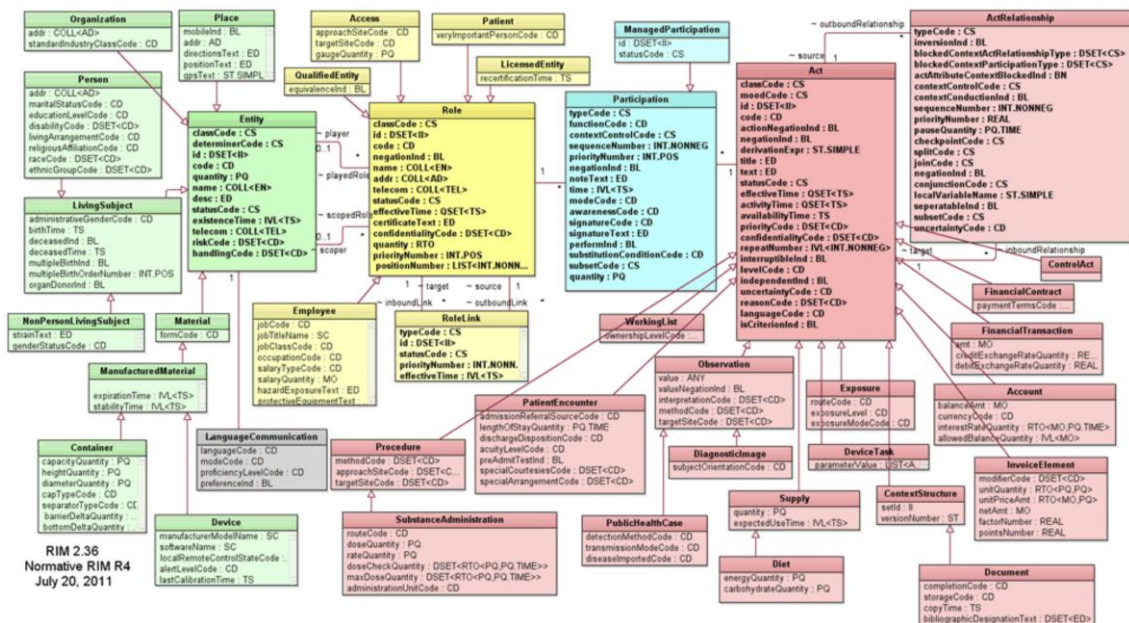


Fig. 1.2 HL7 UML RIM



- Implementation of XML syntax to encode the messages, following the international tendency as a language to exchange structured information between different platforms. This leads to a more human readable, but also more bloated messages, as it will be seen in a moment.
- The utilization of Object-Oriented Programming (OOP) and UML provide a formal methodology, contributing to a higher detail, clarity and accuracy concerning the specifications of the standard, as well as a greater control over the final designs of generated messages.
- It's not limited to the application layer of the OSI Model. After many years implementing HL7v2, HL7 realized the need of creating an understandable standard that included other OSI layers. That's the reason why HL7v3 includes specifications about XML, security, vocabulary, modelling, methodology, etc.
- HL7v3 opted for controlled vocabulary, intensively using international coding systems to classify healthcare-related data (see 1.1.3). Additionally, HL7v3 also offers a list of own codifications for specific cases that might not be covered by the international coding system.
- The standard was created in order to support all healthcare workflows, not only limited to the exchange of data between organizations. Differently to HL7v2, HL7v3 has the capacity to support the exchange of high complexity medical data which, as a whole, can lead to the implementation of data-assisted decision making, EHR or clinical research, among others.

In order to clarify the differences between HL7v2 and HL7v3 as far as the structure of the messages are concerned, let's take a close look to the same example in Fig. 1.1, but now encoded using HL7v3.

As it is about to be seen, HL7v3 messages contain a list of wrappers that contain a series of tags, where a diverse variety of information is transported. The function of these wrappers is to provide a structured and standardized context for the measurement in terms of its identification, type, sender, receiver, interpretation, patient, performer of the measurement, etc. The exact specification of the different fields, groups, sequences and cardinality that the message contains is defined by a Hierarchical Message Description (HMD).

In order to be able to encompass all the necessary information that defines a specific message, three main pillars have to be formalized. The first one is the different possible Application Roles within the system, defining the responsibilities of both emitter and receiver systems. The second one is the definition of the Activation Events, also referred as trigger events, which define the reason why the message has been generated and sent. And the last one is the definition of the different Scenarios, also known as storyboards, which, in general, define the use case that defines the stream of events of the interaction and declares the associated preconditions and postconditions.

In Annex A, some examples of these contextual wrappers will be presented from a lower to a higher level, including the section containing the information about the patient itself, and the result of the observation.

As a conclusion, HL7v3 is full model driven, and all models are derived from the RIM. This model serves to promote a high degree of consistency between messages. The ability to derive/specialize segments of the message leads to the use of properly constrained models for specific use cases.

One of the most widely implemented standard based on HL7v3 is the already mentioned CDA, also known as the e-Document standard. CDA ensures both human readability as well as software processability.

So, one can state that, to a certain degree, HL7v3 can be considered a next-generation HL7v2, as it is based on reusable structures, an underlying data type definition, and it uses an industry standard as the syntax encoding method. HL7v3, however, has the capacity to comprehend complex medical data and allows the data exchange in an interoperability context and between organizations. Of course, these improvements are a consequence of a significant increase of complexity in the message structure.

But, would it be possible to create a next generation standard that combines the best features of HL7v2 and HL7v3 while leveraging the latest web service technologies? This is actually the motivation behind the creation of the FHIR standard, which will be closely described in the next section.

### **1.1.2. HL7 FHIR**

In 2014, the Fast Healthcare Interoperability Resources (FHIR) standard was introduced as an important alternative to HL7v2 and HL7v3 standards. FHIR is built on the other HL7 previous standards, but unlike them, it employs RESTful web services and open web technologies, including XML, RDF and JSON formats. These technologies are vastly used in nowadays software development, leading to a reduction of the learning curve of this standard compared with its antecessors. It's already worth mentioning that this project, as will be seen more closely in future chapters, uses the latter to exchange data with a FHIR API.

FHIR also offers multiple options for exchanging data among systems, among which one can find messaging (sharing similarities to HL7v2) and documents, as well as RESTful API approach. The latter can simplify data sharing as it allows the replacement of point-to-point interfaces, used in the previous standards, with one-to-many interfaces, notably increasing the potential for greater interoperability not only among systems and devices within organizational IT systems, but also mobile apps, medical devices and wearables.

The FHIR specification is targeted to individuals and organizations developing software and architecting interoperable solutions that will be using FHIR. The FHIR specification does not attempt to define good or best clinical practices, and so, because of its focus on implementation, many aspects of the

specification deal with the technical underpinnings of the exchange of clinical information between electronic systems.

The FHIR standard defines “Resources”, which are self-describable pieces of data as generic templates that are stored or exchanged (the equivalence within the context of HL7v2 messages would be the “segments”). Some examples of these Resources will be seen later, especially the ones that have been used for this project, but essentially a Resource can include metadata, text or bundled collections of information that form clinical documents. The resource instances are not limited to patient-related information, but also can refer to administrative information (such as practitioners, organizations and locations) as well. In fact, some Resources are infrastructure components used to support the technical exchange of information by describing what systems are able to do. This means that some systems, such as clinical decision support engines, that have to use these data for carrying out its function, may expose FHIR interfaces, even though they don’t actually store any patient or administrative information themselves. The previous example is just to clarify that FHIR has an extended scope of usages other than just storing medical data.

Note that each Resource defines a relatively small amount of simple highly-focused data, and thus, a single Resource might not provide enough data for some specific use cases. However, a collection of Resources taken together creates a useful clinical record. As will be seen throughout the own Resources implementations that make up this project in future chapters, it’s worth mentioning that Resources also have the ability to reference one another (an observation referring a certain patient, for instance), so that the information can be kept stored in two different data structures or Resources yet being somehow related to provide a wider context to the data.


One of the features that makes accessing data an easy task is the presence of an identifying tag that uniquely identifies the Resource. The RESTful API approach also enables Resources to be updated and deleted.

Furthermore, the FHIR Server also provides an extra feature consisting on an automatic Resource validation mechanism providing overall syntax and semantic validations, including data types, attributes and business-related rules to ensure FHIR Resources conform to the predefined structure of the standard. If a request is carried out with an incorrect Resource structure, the API will return an error with status code 400 (see Fig. 1.3).

400  
Undocumented Error: response status is 400

Response body

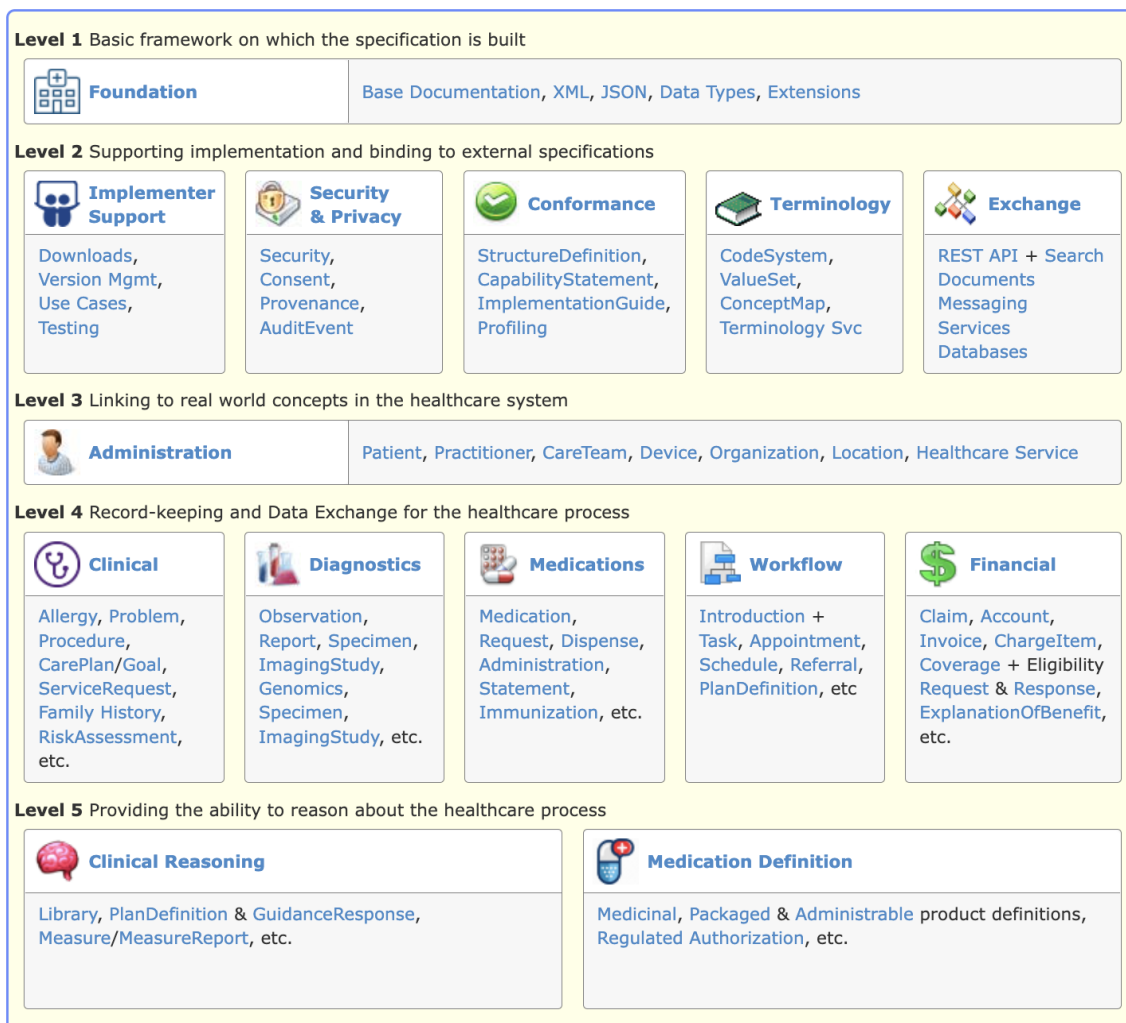
```
{
  "resourceType": "OperationOutcome",
  "text": {
    "status": "generated",
    "div": "<div xmlns=\"http://www.w3.org/1999/xhtml\"><h1>Operation Outcome</h1><table border
=\"0\"><tr><td style=\"font-weight: bold;\">ERROR</td><td></td><td><pre>HAPI-0450: Failed to
parse request body as JSON resource. Error was: HAPI-1814: Incorrect resource type found, expec
ted \"Observation\" but found \"Wrong estructure\"</pre></td>\n\t\t\t</tr>\n\t
\t</table>\n\t</div>"
  },
  "issue": [
    {
      "severity": "error",
      "code": "processing",
      "diagnostics": "HAPI-0450: Failed to parse request body as JSON resource. Error was: HAPI
-1814: Incorrect resource type found, expected \"Observation\" but found \"Wrong estructure\""
    }
  ]
}
```

 Download

**Fig. 1.3** FHIR API response given an incorrect resource structure

There is another interesting feature provided by FHIR in terms of extensibility. Given that Resources are somehow generic and they have to be usable in a very large variety of different contexts, there is the possibility that there's the need to store some data that is not strictly predefined by the standard. To overcome that, FHIR provides the ability to adjust the Resources to be able to handle the needs of different implementation spaces by defining "extensions" as well as enforcing constraints. Resources are designed so that the addition of these changes doesn't affect the normal functioning of data exchanges, enabling any system to consume completed forms even if they have additional elements added, whether or not these are used or even recognized by the receiving system, giving an enormous flexibility as far as new utilizations of data are concerned.

As homologously happened in HL7v3, FHIR has a relatively large number of Resources (see Fig. 1.4), and so, to keep the numbers reasonable, some of them are fairly broad. For example, the Observation resource (which will be largely used in this project), is used for vital signs (such as blood pressure, oxygen levels, temperature, etc.), lab results, psychological assessments and a variety of other things.



**Fig. 1.4** List of FHIR Resources classified in layers and use fields

One of the main problems of HL7v2 was the lack of human-readable messages, and this new version of the standard also deals with this issue. FHIR is intended to support sharing data in a computable manner to allow a certain degree of automatism in the form of computer-mediated processes such as decision support, rules triggering, trend analysis, etc. However, there are a lot of scenarios where certain data cannot be captured in a numerical and discrete manner. As a consequence of this, FHIR Resources support sharing, not only discrete data for computational purposes, but also a human-readable view (see Fig 1.5) so that humans on each end of a healthcare information exchange can still get a full picture of the context in which the data is being handled.

```

"code": {
  "coding": [
    {
      "system": "http://loinc.org",
      "code": "15074-8",
      "display": "Glucose [Moles/volume] in Blood"
    }
  ]
},
"subject": {
  "reference": "Patient/f001",
  "display": "P. van de Heuvel"
},

```

**Fig. 1.5** Human-readable fields in an Observation Resource fragment

So, as a conclusion, among the most important characteristics that position the HL7 FHIR as an improved version of the previous HL7 standards, making it the most optimal solution for this project, one can find the following ones:

- Best accessibility leading to an improvement of healthcare coordination, as it allows the electronic medical records of all patients to be accessed by multiple users enabling easy access to medical data and seamless communication.
- Easy data sharing, as every Resource is linked with a unique identifier, quite similar to a URL. FHIR makes it feasible and easy to get access to the right set of data from any device or application. Note that as a consequence of this identifiers' assignment, FHIR eliminates and cuts down the long process of exchanging data back and forth between systems as it happened with the previous standards.
- FHIR Specifications are free to use, with no restrictions. Until 2013, the FHIR specifications were not freely available and it might have been one of the main reasons why the healthcare industry has lagged on interoperability until the last decade.
- FHIR can be easily set up and accessed by mobile phones, given its simplicity and used technologies. Most mobile devices support programs with HTTP and JSON. So, it sets the perfect scenario for new healthcare-related software development, as the one proposed by this project.
- FHIR supports RESTful architecture, making healthcare data manipulation easier than ever.

A FHIR-based system's capabilities are defined by what the Resources can say and, from a clinical perspective, these things define the clinical record:

- What kind of Resources are defined within the system?
- Which are their data contents and which are the rules regarding what terminology codes (see section 1.1.3) are supported and/or required?
- How are Resources referenced to one another?
- How can this information be accessed?

All these questions, in the domain of the solution developed in this project, will be answered in future chapters.

Also, the close analysis of some FHIR Resources as for its internal structure will be left for future chapters, when the specific use cases of this project will be addressed (see Chapter 1.2).

### **1.1.3. International coding systems for observations**

So far, a general approach around HL7 FHIR standards (together with its predecessors) has been carried out in an unrelated way as far as the specific solution for this project is concerned. However, from now on, the focus will be progressively put on more practical subjects that directly affect the use case of this project. This will be attested already in this section where the Resource “Observation” (vastly used in the project, together with the Resource “Patient”, as will be seen in the next chapter) will be partially addressed.

It has already been mentioned in the previous section that, in order to try to reduce the complexity of the FHIR standard as far as diversity of data is concerned, some Resources might be a little bit generic. This is the case of the Observation Resource, which, among other uses, it serves as the structure to store the result of a measurement carried out on a certain patient. Moreover, within the specific use case of this Resource, the same Observation is used to digitalize different types of measurements’ results, regardless the type of the vital sign that is being measured (pressure, weight, pulse rate, etc.).

So, there’s obviously a need to find out how to integrate these variety of measurements in the general and reusable Observation Resource.

The proposed solution is to rely on a standard (or multiple standards) that maps every single use case of measurement to a code that defines what is being measured. This is exactly what International Coding Systems do.

There are multiple standards that aim to provide a solution for this mapping, being the most known “SNOMED (Systematized Nomenclature of Medicine) Clinical Terms” and “Logical Observation Identifier Names and Codes” (LOINC). For the implementation of this project, the latter has been chosen to map the data, but giving the system the possibility to understand other coding standards would be within the scope of future versions.

LOINC is an international standard that provides universal code names and identifiers for laboratory tests and other medical terminology that can be used in medical health records, facilitating interoperability and communication within healthcare networks. Given the increasingly connected nature of healthcare, standardization to a universal coding system allows for the smooth aggregation of laboratory (or more routine medical devices) results from multiple facilities for clinical care, research, and outcome management. LOINC is committed to a healthcare ecosystem where data is available with open standards that unlock

the potential for information systems and applications to improve health decision-making and care.

In Fig. 1.6, a fragment of an Observation can be seen, specifically the section where the information relative to the coding systems can be found. Please, note how the Observation Resource allows to support multiple international codes, three in this particular case (LOINC, SNOMED and ACME), each one with its individual and unique code mapping.

If the reader analyzes the structure closely, he will see a couple of things that might shock him at first glance; the fact that in this example there are two LOINC codes for the same observation, and the presence of a field named “display”, where a human-readable description of the code is provided. As for the former, an Observation instance can make reference to more than a single code within the standard. This example leads with a body weight measurement, which can be linked with both codes 29463-7 and 3141-9 (the reasons why in this example both codes are added are outside the scope of this discussion, as it depends on contextual factors that are ignored). And regarding the latter, another example of the implementation of an increased degree of human-readability in the FHIR standards can be seen. The key point is that systems will interpretate the codes regardless the human-readable description, allowing both a global interoperability between systems and human understanding of the data within the same Resource.

```
"code": {
  "coding": [
    {
      "system": "http://loinc.org",
      "code": "29463-7",
      "display": "Body Weight"
    },
    {
      "system": "http://loinc.org",
      "code": "3141-9",
      "display": "Body weight Measured"
    },
    {
      "system": "http://snomed.info/sct",
      "code": "27113001",
      "display": "Body weight"
    },
    {
      "system": "http://acme.org/devices/clinical-codes",
      "code": "body-weight",
      "display": "Body Weight"
    }
  ]
},
```

**Fig. 1.6** Different coding systems in an Observation Resource fragment



Again, however, in this project only LOINC has been considered as international coding standard, leading to a structure like the one that can be seen in Fig. 1.7, which shows a fragment a blood pressure observation (specifically the section regarding the systolic measurement).

```
"code": {
  "coding": [
    {
      "system": "http://loinc.org",
      "code": "8480-6",
      "display": "Systolic blood pressure"
    }
  ]
},
```

**Fig. 1.7** Unique LOINC coding system in an Observation fragment

## 1.2. Data Structures and Analysis

Now that the reader has a clearer understanding on how medical data is stored and, specially, how the FHIR standard works, a closer analysis of the data structures that have been used to develop the solution presented in this project can be carried out in a more direct and practical manner, focusing more on the practical objectives to be achieved rather than on a theoretical level.

This chapter is divided into two different sections, which correspond to the two main actions carried out by the system as far as medical data management is concerned, storing data to make it persistent in a FHIR server and accessing it for several reasons such as UI visualization, analysis, basic diagnosis, etc.

Note that this chapter only addresses data in the context of the FHIR standard. So, all the generated data within the application defined for internal uses (such as roles, groups, group requests, devices, etc.) that is completely independent on the FHIR side will be addressed in future chapters.

### 1.2.1. Data generation and storage

All the details regarding the functionalities of the application will be covered in future chapters (see Chapter 2.1) but essentially, within the scope of data generation, the use case can be summarized in a user generating data by carrying out measurements with his medical devices. This data will be structured following the FHIR standard, stored in a FHIR server, and later accessed by the same user or other users (which a special role that would allow them to access the data from other patients, as will be seen later on).

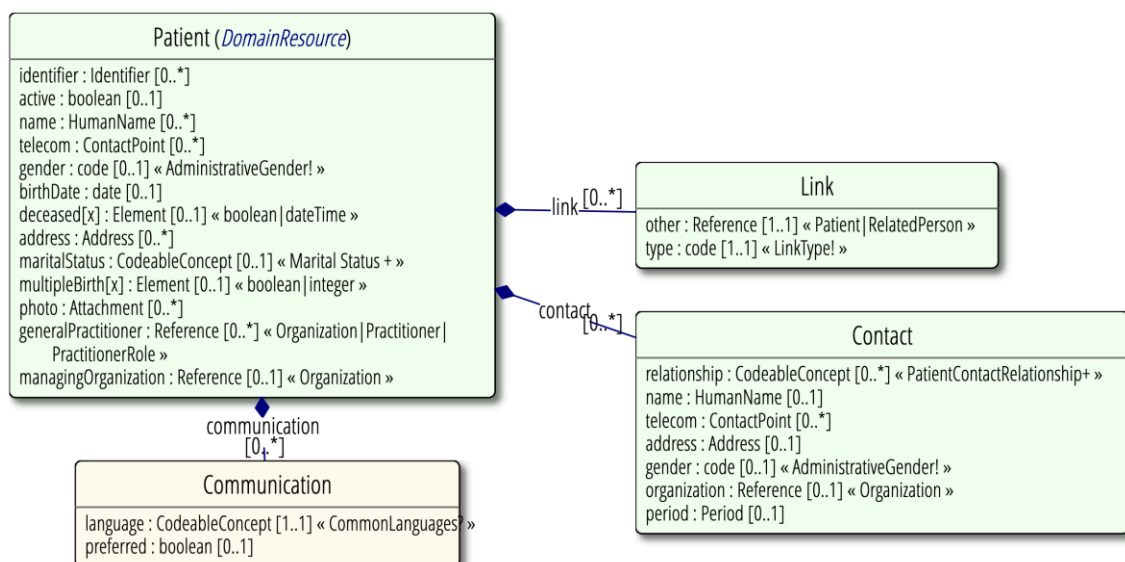
Note that, in this scenario, the user will adopt the role of a patient, and so, the FHIR Resource "Patient" will be used to store their personal data and demographics. As for the results of the measurements, as has already been

suggested in previous chapters, the FHIR resource “Observation” will be used. In this chapter, a close analysis of both Resources will be made, as well as the justification of their choice and utilization as for the application context.

The Resource Patient contains all the necessary attributes to store the demographic information that is needed to support administrative, financial and logistic procedures. A Patient record is generally created and maintained by each organization providing healthcare services for a patient, and thus, a patient receiving care at multiple organizations may have its information present in multiple Patient Resources. This justifies the creation of a Patient Resource for each user within the realm of this project.

In Fig. 1.8, the UML Diagram for a Patient Resource can be seen. Note that in order to provide the maximum flexibility to the Resource, a large variety of fields are available, but not strictly required to generate a Patient Resource instance. In Fig 1.9, the complete data structure of the resource can be found, together with a short description of each field. Please, note that most of its fields are FHIR Resources themselves, with their respective sub-fields, which are not displayed in Fig. 1.9.

The very nature of this project, as far as its objectives are concerned, leads to the possibility (and often to the need because of the lack of data) of using a more simplified schema, where only the essential parameters in the scope of the application are included. So, in this section, a close analysis of the fields that have been used to generate instances of Patients Resources in this project will be carried out. If the reader is still interested in knowing all the specific information about all the other fields, they will be able to find it in the official HL7 FHIR documentation attached in the references.



**Fig. 1.8** UML Diagram of a Patient Resource

Name	Flags	Card.	Type	Description & Constraints
Patient	<b>N</b>		DomainResource	Information about an individual or animal receiving health care services Elements defined in Ancestors: <code>id</code> , <code>meta</code> , <code>implicitRules</code> , <code>language</code> , <code>text</code> , <code>contained</code> , <code>extension</code> , <code>modifierExtension</code>
identifier	Σ	0..*	Identifier	An identifier for this patient
active	?! Σ	0..1	boolean	Whether this patient's record is in active use
name	Σ	0..*	HumanName	A name associated with the patient
telecom	Σ	0..*	ContactPoint	A contact detail for the individual
gender	Σ	0..1	code	male   female   other   unknown <i>AdministrativeGender (Required)</i>
birthDate	Σ	0..1	date	The date of birth for the individual
deceased[x]	?! Σ	0..1		Indicates if the individual is deceased or not
deceasedBoolean			boolean	
deceasedDateTime			dateTime	
address	Σ	0..*	Address	An address for the individual
maritalStatus		0..1	CodeableConcept	Marital (civil) status of a patient <i>MaritalStatus (Extensible)</i>
multipleBirth[x]		0..1		Whether patient is part of a multiple birth
multipleBirthBoolean			boolean	
multipleBirthInteger			integer	
photo		0..*	Attachment	Image of the patient
contact	I	0..*	BackboneElement	A contact party (e.g. guardian, partner, friend) for the patient <i>+ Rule: SHALL at least contain a contact's details or a reference to an organization</i>
relationship		0..*	CodeableConcept	The kind of relationship <i>Patient Contact Relationship (Extensible)</i>
name		0..1	HumanName	A name associated with the contact person
telecom		0..*	ContactPoint	A contact detail for the person
address		0..1	Address	Address for the contact person
gender		0..1	code	male   female   other   unknown <i>AdministrativeGender (Required)</i>
organization	I	0..1	Reference(Organization)	Organization that is associated with the contact
period		0..1	Period	The period during which this contact person or organization is valid to be contacted relating to this patient
communication		0..*	BackboneElement	A language which may be used to communicate with the patient about his or her health
language		1..1	CodeableConcept	The language which can be used to communicate with the patient about his or her health <i>Common Languages (Preferred but limited to AllLanguages)</i>
preferred		0..1	boolean	Language preference indicator
generalPractitioner		0..*	Reference(Organization   Practitioner   PractitionerRole)	Patient's nominated primary care provider
managingOrganization	Σ	0..1	Reference(Organization)	Organization that is the custodian of the patient record
link	?! Σ	0..*	BackboneElement	Link to another patient resource that concerns the same actual person
other	Σ	1..1	Reference(Patient   RelatedPerson)	The other patient or related person resource that the link refers to
type	Σ	1..1	code	replaced-by   replaces   refer   seealso <i>LinkType (Required)</i>

**Fig. 1.9** Complete Data Structure of a Patient Resource

In Fig. 1.10, the final structure used to store patient's data, derived from the complete Patient Resource and only providing the necessary data to ensure the correct functioning of the application, can be seen. This data, as will be seen in future chapters, is collected when the user registers in the application, and can be modified afterwards from a configuration section within the profile page of the application.

It's worth mentioning that, even though FHIR supports both XML and JSON formats, the latter is the one that have been selected to encode the data in this project.

```

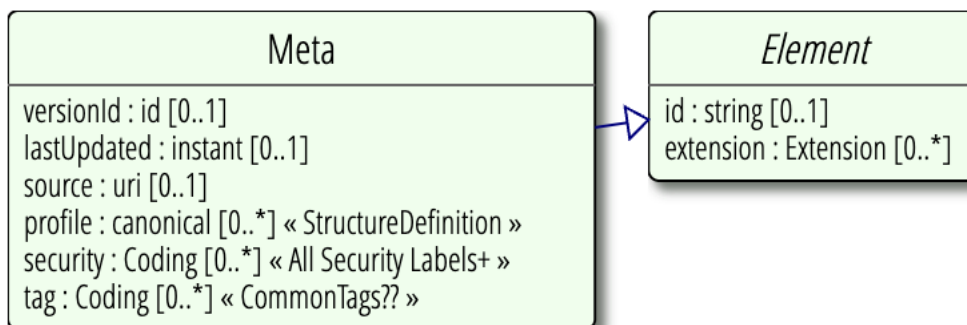
{
  "resourceType": "Patient",
  "id": "502", // Resource ID
  "meta": {
    "versionId": "1",
    "lastUpdated": "2023-01-20T08:40:15.950+00:00",
    "source": "#8kG6pBeRwIVwQJOf"
  },
  "text": {
    "status": "generated",
    "div": "<div xmlns=\"http://www.w3.org/1999/xhtml\"><div class=\"hapiHeaderText\"><b>MIR HURTADO</b></div><table class=\"hapiPropertyTable\"><tbody><tr><td>Identifier</td><td>63ca536f40f3d3de539e9ee4</td></tr><tr><td>Address</td><td><span>C/Prat de la Riba, 45</span><br/><span>Les Borges Blanques</span><span>Spain</span></td></tr><tr><td>Date of birth</td><td><span>15 November 2000</span></td></tr></tbody></table></div>"
  },
  "identifier": [ // List of identifiers
    {
      "use": "usual",
      "system": "http://behealthapp/patients.com",
      "value": "63ca536f40f3d3de539e9ee4" // Identifier within the context of the project
    }
  ],
  "active": true, // Current status of the patient
  "name": [
    {
      "use": "usual",
      "text": "Arнау Mir Hurtado",
      "family": "Mir Hurtado"
    }
  ],
  "telecom": [ // List of methods to contact the patient
    {
      "system": "phone",
      "value": "+34608764523"
    },
    {
      "system": "email",
      "value": "arnau@gmail.com"
    }
  ],
  "gender": "male",
  "birthDate": "2000-11-15",
  "deceasedBoolean": false,
  "address": [ // List of addresses of the patient
    {
      "use": "home",
      "line": [
        "C/Example, 78"
      ],
      "city": "Les Borges Blanques",
      "postalCode": "25400",
      "country": "Spain"
    }
  ],
  "communication": [ // List of languages of the patient
    {
      "language": {
        "coding": [
          {
            "system": "urn:ietf:bcp:47",
            "code": "en"
          }
        ]
      }
    }
  ]
}

```

**Fig. 1.10** Example of a Patient Resource's fields used in the project

So, the fields that have been selected to make up this specific implementation of a the FHIR Patient Resource are the following ones:

- **resourceType:** A mandatory field that simply indicates the type of resource that is being sent.
- **id:** It's the identification number for this particular Patient Resource within the FHIR server. Note that there can be multiple Resources with the same id, but they must refer to different Resource types. So, within the context of Patient Resources, this is a unique identifier. This identifier will later be used in Observation Resources instances to link the latter with the corresponding Patient instance.
- **meta:** This field is generated automatically when the instance is created, and it simply shows some potentially useful information regarding traceability of the data and provides technical and workflow context to the resource. In fact, this is a Resource itself, called Metadata (see Fig 1.11), that extends from the Element Resource, which is essentially the base definition for all elements contained in a resource, as they have an internal id together with a differentiated extension. As can be seen, this Resource contains more attributes than the ones that can be seen in Fig 1.10, and this is because only the required (that is to say mandatory) fields are generated. Among the latter, the reader will find a "versionId" field, which changes each time the content of the resource changes. So, on receiving a write operation, the FHIR server shall update this item to the current value. This can be used to ensure that updates are based on the last version of the resource and, again, for traceability purposes. The "lastUpdated" attribute just contains the date when the last modification of the resource was carried out. And finally, the "source" attribute is a Uniform Resource Identifier (URI) that identifies the source system of the resource, which can be used to form assessments about its quality, reliability, trustworthiness, or to provide pointers for where to go further investigate the origins of the resource and the information in it.



**Fig. 1.11** UML Diagram of a Metadata Resource

- **text:** This is one of the elements in charge of providing the already mentioned human-readability that characterizes the FHIR standard. It includes an HTML narrative that contains a summary of the resource and may be used to represent its content to a human. In Fig 1.12, the reader

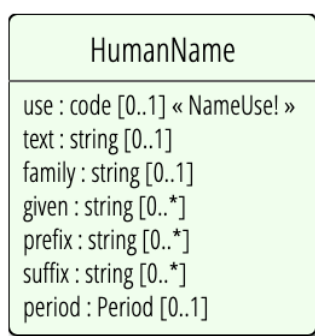
will find the HTML visualization of the same example in Fig. 1.10. Note that only the most basic information is shown by this human-readable representation of the Patient Resource, but a more extended description could be generated if the “status” field within this “text” element was set to “extensions”, instead of “generated”.

<b>MIR HURTADO</b>	
Identifier	63ca536f40f3d3de539e9ee4
Address	C/Prat de la Riba, 45 Les Borges Blanques Spain
Date of birth	15 November 2000

**Fig. 1.12** Human-readable information of the Patient

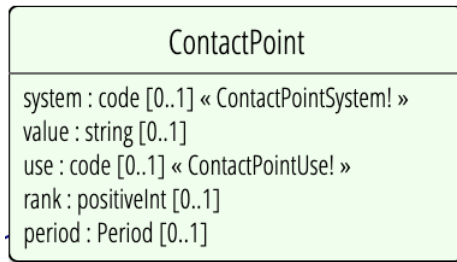
- **identifier:** Differently from the already addressed “id” attribute of the Patient Resource, this is a business identifier of the patient. This means that, although the logical id of a resource changes as it moves from server to server (given that “id” must uniquely identify a Patient Resource within the server), the “identifier” value never changes, as all copies of the resource refer to the same underlying person. In other words, two servers can have the exact same instance of a Patient and, while they will individually assign an “id” to uniquely identify that Resource within their own context, both copies will share the same “identifier” field, as it references the same physical patient. The value of this identifier is generated within a context, which means that the patient will be assigned a unique identifier. The context that this identifier references is indicated in the sub-field “system” of the “identifier” element. Moreover, note that this “identifier” is actually defined as an array of identifiers, which is completely logical, as a single patient can be part of multiple contexts (the patient exists with a different identifier in the public health system, one or multiple private healthcare companies, a medical study, this application, etc.), and the more identifiers are provided, the more potential regarding interoperability that Patient Resource will have. For the scope of this project, a can be seen in Fig. 1.10, only the identifier of the “beHealthApp” application is provided, and thus, so far this Patient instance is only meaningful within the project scope, but it has the potential of being extended to other contexts in the future.
- **active:** It simply indicates whether this patient record is in active use. Many systems use this property to mark as non-current patients, such as those that have not been seen for a period of time based on an organization’s business rules. For the scope of this project, a patient is active as long as it has a created profile within the application.
- **name:** It consists on a HumanName Resource (see Fig 1.13) that contains the most basic information about the patient’s name. Some examples of the different possible values of the “use” field can reference to are “usual”, known as the conventional name that a patient normally

uses, “official”, which is the formal name as registers in an official government registry, but which name might not be commonly used (may be called “legal name”), “temp”, which would be a temporary name assigned at birth or in emergency situations, “nickname”, which would be used to address the person in an informal manner, “anonymous”, used to protect a person’s identity for privacy reasons, “old”, for names that are no longer in use (or were never correct, but retained for records), or “maiden”, which covers the case of names changes for marriage, displaying the name used prior to changing it. Within the scope of the first version of this project, in order to avoid problems related with concreteness, the “usual” value has been used in all cases. The reader will also note that the full name has been stored in the “text” attribute of the “name” element, and also the family name separately in the “family” attribute for particular uses within the application. As it happened with the “identification” field, the “name” field is defined as an array of HumanName Resources, enabling the possibility to save more multiple names, which might have a different “use” among the possibilities that were mentioned before.



**Fig. 1.13** HumanName Resource structure

- **telecom:** Contains a list with the details of different contact sources of the Patient under the structure of another FHIR Resource called ContactPoint (see Fig. 1.14). Within the context of this project, only two contact sources are asked to the user registers in the application; the phone number and an email address. For this particular implementation, each of them only contains the two essential fields “system” and “value”, where the former indicates the type of source it’s being referenced, and the latter contains the information itself.



**Fig. 1.14** ContactPoint Resource structure

- **gender:** It simply indicates the gender of the patient.
- **birthDate:** It indicates the birth date of the patient in a standardized format.
- **deceasedBoolean:** A Boolean that indicates if the patient is deceased or not. The importance of this element's presence within the basic objectives of the application might be indeed debatable, but there's always the possibility that some of the measurement's data that the patient generated within the application might be useful regarding some kind of subsequent investigation of the decease causes, and so, if the information of deceases patients is kept, as well as the Observation Resources that they generated, some potential binding information can be extracted for future implementations of preventing services.
- **address:** It contains a list of Address Resources (see Fig. 1.15), storing data relative to the patient's addresses. As can be seen, the Address Resource is a relatively large Resource, but only some fields are used in the application (as can be seen in Fig. 1.10). As it happened with the "use" field of the HumanName Resource, an Address also contains this same attribute, which in this case can adopt the values "home", "work", "temp", for temporary addresses, "old", for addresses that are no longer in use or were never correct but retained for records, and "billing", which references an address to be used to send bills, invoices, receipts, etc. In this case, the application allows more flexibility offering the possibility to select three different uses when the user registers; "home", "work" and "temp".



Name	Flags	Card.	Type	Description & Constraints
Address	Σ N		Element	An address expressed using postal conventions (as opposed to GPS or other location definition formats) Elements defined in Ancestors: <a href="#">id</a> , <a href="#">extension</a>
use	?! Σ	0..1	code	home   work   temp   old   billing - purpose of this address <a href="#">AddressUse (Required)</a>
type	Σ	0..1	code	postal   physical   both <a href="#">AddressType (Required)</a>
text	Σ	0..1	string	Text representation of the address
line	Σ	0..*	string	Street name, number, direction & P.O. Box etc. This repeating element order: The order in which lines should appear in an address label
city	Σ	0..1	string	Name of city, town etc.
district	Σ	0..1	string	District name (aka county)
state	Σ	0..1	string	Sub-unit of country (abbreviations ok)
postalCode	Σ	0..1	string	Postal code for area
country	Σ	0..1	string	Country (e.g. can be ISO 3166 2 or 3 letter code)
period	Σ	0..1	Period	Time period when address was/is in use

Fig. 1.15 Address Resource structure

- communication:** This last field just indicates a list of languages which may be used to communicate with the patient about his or her health. It can be seen in Fig. 1.10 that the language code (in that case, English), is accompanied by a “system” value, that simply maps the code with the IETF BCP language tag standard, which is a standard that is used to identify human languages in the Internet. Within the scope of this first version of the application, the user is offered to choose among a list of two available languages (English and Spanish), which will be the language that will be used to display the contents of the application. Of course, in future versions, the number of available languages should increase to ensure the possibility of a global utilization.

After addressing how a patient’s data will be generated and stored, the second type of data that this project will deal with is medical data in the form of measurement’s results coming from the user’s medical devices that will be connected to the application.

Among all the existing FHIR Resources, the one that is aimed to address a measurement report is the Observation Resource. This Resource addresses measurements and simple assertions made about a patient, device or another subject.

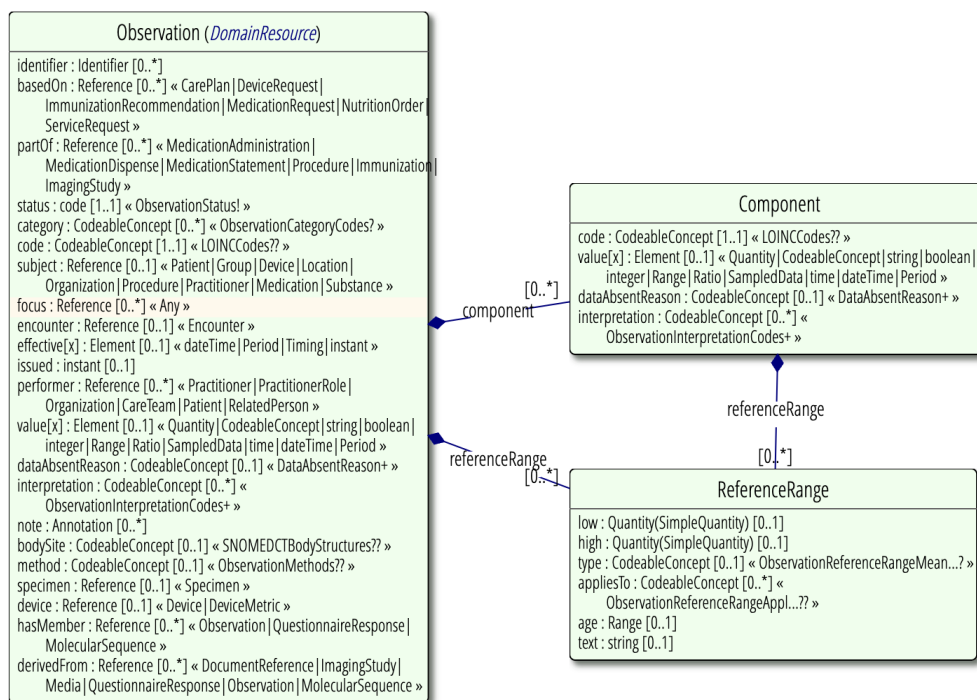
Observations are a central element in healthcare, used to support diagnosis, monitor progress, determine baselines and patterns, and even capture demographic characteristics. Most observations are relatively simple assertions with some metadata, but some others group other observations together logically, or they might even be multi-component Observations. An example of this latter will be covered later with a blood pressure Observation, which is made of two components (systolic and diastolic pressure).

There are other super-Resources such as the DiagnosticReport Resource, which provides a clinical or workflow context for a set of Observation Resources, which are obviously referred in the former, and which are used to complement the full report.

Some uses for the Observation Resource include vital signs such as body weight, blood pressure or temperature (note that this is the block that concerns the use case of this project), laboratory data, imaging results, personal characteristics (as eye-color), social history (as tobacco use, cognitive status, etc.), or some core characteristics (like pregnancy status).

The Observation Resource is intended for capturing measurements and subjective point-in-time assessments. Note that is not intended for specific cases where there are already other Resources that specifically address those, such as the AllergyIntolerance Resource, among many others. The HL7 FHIR official documentation has a section to consult whether the Observation Resource is appropriate for a specific usage.

The Observation Resource can be used for a large list of situations that imply reporting some kind of information, which allows great flexibility as for its usage, but also leads to an accordingly enormous Resource regarding the number of fields involved in its complete form (see Fig 1.16). Nonetheless, as it happened with the Patient Resource, a highly simplified version of the Observation Resource has been used for this project, including only the necessary fields to ensure that that the objectives of the application are met.



**Fig. 1.16** UML Diagram of a complete Observation Resource

In Fig. 1.17, the reader will be able to see a real example of a sample measurement carried out within the context of this project. In this particular case, the result of a body temperature measurement is represented using the Observation Resource, but only the strictly essential fields for this first version of the application.







```
{
  "resourceType": "Observation",
  "id": "504", // Resource ID
  "meta": {
    "versionId": "1",
    "lastUpdated": "2023-01-20T08:49:46.489+00:00",
    "source": "#APgP8aZGCEbuBd2Q"
  },
  "status": "final",
  "category": [
    {
      "coding": [
        {
          "system": "http://terminology.hl7.org/CodeSystem/observation-category",
          "code": "vital-signs",
          "display": "Vital Signs"
        }
      ]
    }
  ],
  "code": {
    "coding": [
      {
        "system": "http://loinc.org",
        "code": "8310-5",
        "display": "Body temperature"
      }
    ]
  },
  "text": "Temperature"
},
"subject": {
  "reference": "Patient/502"
},
"effectiveDateTime": "2023-01-20",
"valueQuantity": {
  "value": 34,
  "unit": "degrees C",
  "system": "http://unitsofmeasure.org",
  "code": "Cel"
},
"referenceRange": [
  {
    "low": {
      "value": 36.1,
      "unit": "degrees C"
    },
    "high": {
      "value": 37.2,
      "unit": "degrees C"
    }
  }
]
}
```

**Fig. 1.17** Example of an Observation Resource's fields used in the project

Note that a measurement involving temperature is a single-component Observation. If it was multi-component, as would be the case for blood pressure

measurements, as will be seen in a moment, the Observation Resource would contain a “component” attribute. Also, note that the Observation Resource also contains the fields “resourceType”, “id” and “meta”, which have been already closely addressed in the Patient Resource analysis and, consequently, will be omitted here. The rest of fields are the following ones:

- status:** This field indicates the current status of the result value, and it can cover multitude of cases, adopting the values “registered”, meaning that the observation is registered, but there is no result available yet, “preliminary”, for incomplete or unverified observations, “final”, for observations that have been completed and there are no further actions needed (note that this will always be the case for the Observation instances generated within the context of this project), “amended”, meaning subsequent to being final, of observations that have been modified, “corrected”, for observations that have been modified to correct an error in the result, “cancelled”, for operations that are unavailable because the measurement was not started or not completed, “entered-in-error”, for withdrawn operations, and “unknown”, for potential specific cases where none of the previous one seems to fit the situation.
- category:** This field indicates the type of observation that is being considered by indicating a code (see the list of possible codes in Fig. 1.19) mapped by a standard system indicated in the parameter “system”. There is also a “display” field which, essentially, introduces a human-readable version of the code. Note that all this information is encoded within a sub-Resource called Coding, which is a representation of a defined concept using a symbol from a defined code system (see Fig. 1.18).

Name	Flags	Card.	Type	Description & Constraints
 Coding	$\Sigma$ <b>N</b>		Element	A reference to a code defined by a terminology system Elements defined in Ancestors: <a href="#">id</a> , <a href="#">extension</a>
 system	$\Sigma$	0..1	uri	Identity of the terminology system
 version	$\Sigma$	0..1	string	Version of the system - if relevant
 code	$\Sigma$	0..1	code	Symbol in syntax defined by the system
 display	$\Sigma$	0..1	string	Representation defined by the system
 userSelected	$\Sigma$	0..1	boolean	If this coding was chosen directly by the user

**Fig. 1.18** Coding Resource Structure

Code	Display	Definition
social-history	Social History	Social History Observations define the patient's occupational, personal (e.g., lifestyle), social, familial, and environmental history and health risk factors that may impact the patient's health.
vital-signs	Vital Signs	Clinical observations measure the body's basic functions such as blood pressure, heart rate, respiratory rate, height, weight, body mass index, head circumference, pulse oximetry, temperature, and body surface area.
imaging	Imaging	Observations generated by imaging. The scope includes observations regarding plain x-ray, ultrasound, CT, MRI, angiography, echocardiography, and nuclear medicine.
laboratory	Laboratory	The results of observations generated by laboratories. Laboratory results are typically generated by laboratories providing analytic services in areas such as chemistry, hematology, serology, histology, cytology, anatomic pathology (including digital pathology), microbiology, and/or virology. These observations are based on analysis of specimens obtained from the patient and submitted to the laboratory.
procedure	Procedure	Observations generated by other procedures. This category includes observations resulting from interventional and non-interventional procedures excluding laboratory and imaging (e.g., cardiology catheterization, endoscopy, electrodiagnostics, etc.). Procedure results are typically generated by a clinician to provide more granular information about component observations made during a procedure. An example would be when a gastroenterologist reports the size of a polyp observed during a colonoscopy.
survey	Survey	Assessment tool/survey instrument observations (e.g., Apgar Scores, Montreal Cognitive Assessment (MoCA)).
exam	Exam	Observations generated by physical exam findings including direct observations made by a clinician and use of simple instruments and the result of simple maneuvers performed directly on the patient's body.
therapy	Therapy	Observations generated by non-interventional treatment protocols (e.g. occupational, physical, radiation, nutritional and medication therapy)
activity	Activity	Observations that measure or record any bodily activity that enhances or maintains physical fitness and overall health and wellness. Not under direct supervision of practitioner such as a physical therapist. (e.g., laps swum, steps, sleep data)

**Fig. 1.19** List of category codes of an Observation

- **code:** This attribute is also a Coding Resource that describes the type of observation that was carried out. This is where the LOINC codes that were described in section 1.1.3 are used. In the particular example of Fig. 1.17, the LOINC code corresponding to a body temperature measurement is 8310-5. As was already mentioned in section 1.1.3, some observations might be mapped to more than a single code and, for this reason, this field is defined as a list of Coding Resources.
- **subject:** In this field is where the binding with the Patient takes place. For the use case of this project, the subject will always be a Patient Resource, but in other contexts, this could be a reference to other Resources such as Group, Organization, Location, Practitioner, Medication, etc.
- **effectiveDateTime:** It simply indicates the date (and optionally the time) when the Observation was created or, in the realm of this project, when the measurement was done by the patient.
- **valueQuantity:** In this field is where the measurement result itself is stored. However, as can be seen in Fig. 1.20, “valueQuantity” is just one of the possible fields that can be used to do so. Given the generality associated to the use of the Observation Resource, several types of results can be stored, both quantitative and qualitative data, and so, the most convenient “value” attribute has to be used. In the particular case of medical devices’ measurements, a quantitative result is provided, and so,

the “valueQuantity” attribute is used, which is a FHIR Resource itself called Quantity. Within this latter, as can be seen in Fig. 1.17, the result itself can be found in the “value” attribute but, of course, the unit that accompanies the result also has to be provided in the field “unit”, which itself is globally mapped to a code, which can be found in the “code” field, by the Unified Code for Units of Measure (UCUM), which is a code system intended to include all unites of measure being contemporarily used in international science, engineering and business.

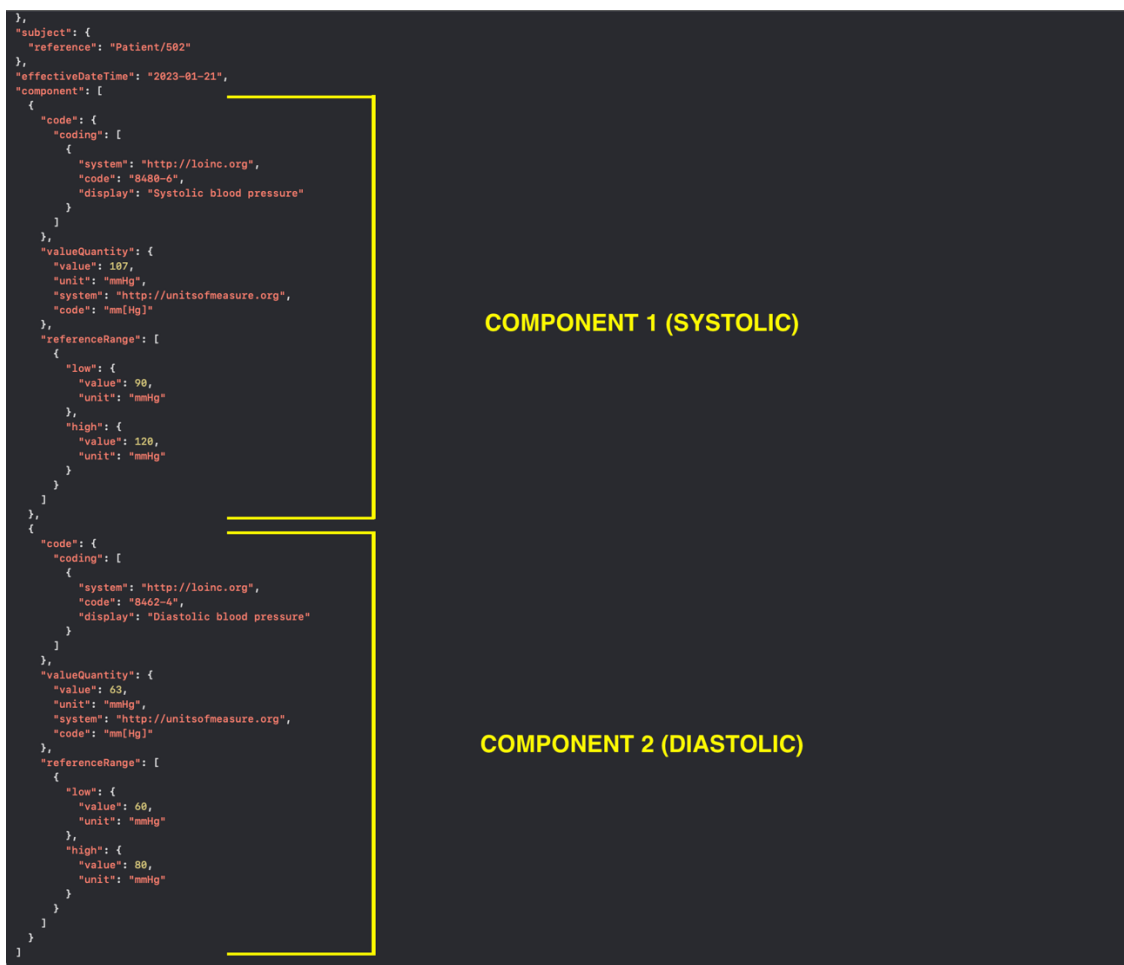
Attribute	Type
value[x]	Σ 0..1
valueQuantity	Quantity
valueCodeableConcept	CodeableConcept
valueString	string
valueBoolean	boolean
valueInteger	integer
valueRange	Range
valueRatio	Ratio
valueSampledData	SampledData
valueTime	time
valueDateTime	dateTime
valuePeriod	Period

**Fig. 1.20** Different types of values depending on the Observation context

- **referenceRange:** This field offers guidance on how to interpret the result of the measurement by the comparison to a normal or recommended range. FHIR offers the possibility to provide some context to the given reference range by indicating the target population this range applies to, the age at which is applicable, among others. In this first version of the application, this context is not provided, and only a range that is generally considered as normal is provided without further consideration on more specific personal information of the patient.

As has already been mentioned, the example described by Fig. 1.17 addresses a temperature measurement, which is a single-component Observation, as body temperature is not a composite value. However, if the patient carries out a blood pressure measurement, then both systolic and diastolic values have to be included within the same Observation Resource instance, leading to a multi-component Observation.

Multi-component Observations have the exact same structure as the one that has recently been described as far as some common attributes are concerned, but those which are specifically related with the result itself are separated into two different components encompassed in the attribute “component”. This is the case for “code”, “valueQuantity” and “referenceRange” fields (see Fig. 1.21).

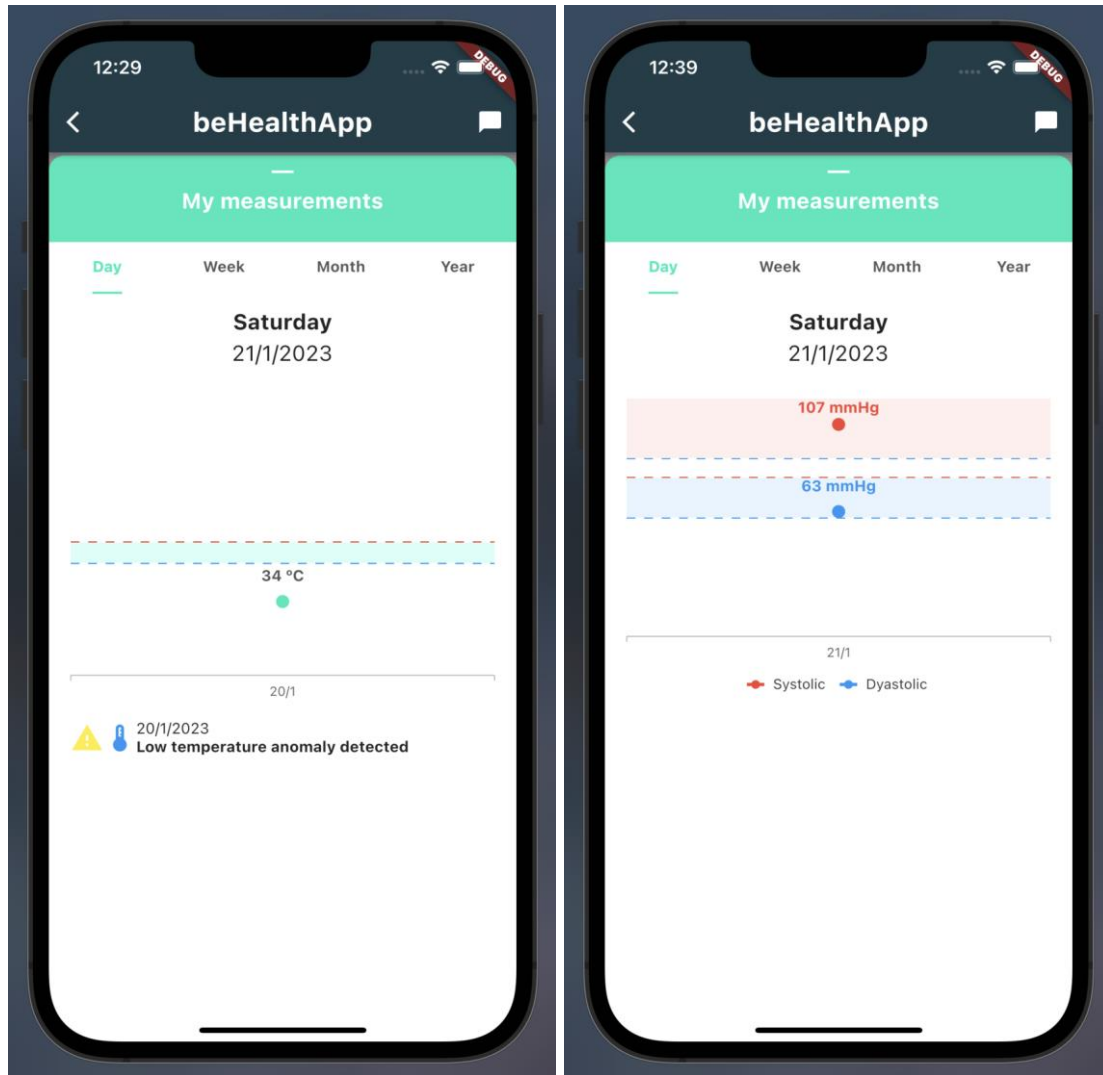


**Fig. 1.21** Multi-component Observation from a blood pressure measurement

## 1.2.2. Data access and analysis

Once the data has been generated and stored in the FHIR server, then the user must be able to access this data, visualize it in the UI of the mobile application and be able to see if there's any anomaly with regards to the obtained value. More detailed information about the functionalities of "beHealthApp" will be covered in Chapter 2, but this section aims to address how the medical data stored in the FHIR server is accessed and give some practical examples from the user's perspective.

The UI when the user accesses his measurements record within a specific medical device, he'll be able to see a page that looks like Fig. 1.22, which are exactly the Observation Resources analyzed in the previous section (Fig. 1.10 and Fig. 1.17 on the right and on the left, respectively), to give an example of how a multi-unit-component Observation affects the UI.



**Fig. 1.22** UI visualization of a patient's measurements (single-component and multi-component Observations on left and on the right respectively)

So, let's go back a few steps to superficially understand how can an Observation Resource stored in the FHIR server be accessed and utilized for UI visualization purposes as well as to detect medical anomalies.

The first thing that must be done is to request a patient's Observations to the FHIR server. This latter has an extended list of supported requests, but the most relevant one can be seen in Fig. 1.23, where the relative URLs that will be used to access the different methods can also be seen.



Observation		The Observation FHIR resource type	^
GET	/Observation/{id}	read-instance: Read Observation instance	∨
PUT	/Observation/{id}	update-instance: Update an existing Observation instance, or create using a client-assigned ID	∨
DELETE	/Observation/{id}	instance-delete: Perform a logical delete on a resource instance	∨
PATCH	/Observation/{id}	instance-patch: Patch a resource instance of type Observation by ID	∨
GET	/Observation/{id}/_history/{version_id}	vread-instance: Read Observation instance with specific version	∨
GET	/Observation	search-type: Search for Observation instances	∨
POST	/Observation	create-type: Create a new Observation instance	∨
GET	/Observation/_history	type-history: Fetch the resource change history for all resources of type Observation	∨
GET	/Observation/{id}/_history	instance-history: Fetch the resource change history for all resources of type Observation	∨

**Fig. 1.23** Most relevant requests supported by the FHIR server

Please note that the project was carried out in a development environment, and so, the FHIR database is just running in a Docker Container built from a Docker Image. Consequently, all absolute URLs that will appear within this document contain localhost and port 8080 as the gateway to access the FHIR server. This can already be seen in Fig. 1.24, where the reader will be able to find the code executed from the NodeJS API gateway (addressed later in 2.3.2) to make a GET request to receive all Observation instances of the patient by means of proving their id (the Patient Resource's id) as a path parameter.

```
var urlFhir = "http://localhost:8080/fhir/Observation?patient=" + patientFound.fhir_id;
const response = await fetch(urlFhir, {
  method: 'get',
  headers: {'Accept': 'application/fhir+json'} });
const responseJson = await response.json();
```

**Fig. 1.24** Patient's Observations request

This petition is received by the FHIR server, which returns a JSON with some metadata, the number of Observation instances found for that specific Patient, and the Observations themselves within the "entry" field, as can be seen in Fig. 1.25. The reader might recall that FHIR supports both JSON and XML, and given that JSON is the selected one for this project, it has been indicated in the "Accept" header of the request, so that the server also responds with the same format.

Curl

```
curl -X 'GET' \
  'http://localhost:8080/fhir/Observation?patient=502' \
  -H 'accept: application/fhir+json'
```

Request URL

```
http://localhost:8080/fhir/Observation?patient=502
```

Server response

Code	Details
200	<p>Response body</p> <pre>{   "resourceType": "Bundle",   "id": "47c2c441-458b-4dc1-8f79-837276775ea8",   "meta": {     "lastUpdated": "2023-01-21T12:45:39.354+00:00"   },   "type": "searchset",   "total": 2,   "link": [     {       "relation": "self",       "url": "http://localhost:8080/fhir/Observation?patient=502"     }   ],   "entry": [     {       "fullUrl": "http://localhost:8080/fhir/Observation/504",       "resource": {         "resourceType": "Observation",         "id": "504",         "meta": {           "versionId": "1",           "lastUpdated": "2023-01-20T08:49:46.489+00:00",           "source": "#APgP8aZGCEbuBd2Q"         }       }     }   ] }</pre> <p>Response headers</p> <pre>connection: keep-alive content-encoding: gzip content-type: application/fhir+json;charset=UTF-8 date: Sat, 21 Jan 2023 12:45:39 GMT keep-alive: timeout=60 last-modified: Sat, 21 Jan 2023 12:45:39 GMT transfer-encoding: chunked x-powered-by: HAPI FHIR 6.1.0 REST Server (FHIR Server; FHIR 4.0.1/R4) x-request-id: naOzPFf7SXs7CCmt</pre>

Responses

Code	Description	Links
200	Success	No links

**Fig. 1.25** Request and response from the FHIR server

The server response indicates that the patient has 2 Observation instances stored (the temperature and the blood pressure measurements). So, the NodeJS API gateway will send this information to the mobile application, and by parsing the different fields of the response and, of course some extra coding regarding the UI, the user will be able to see the data of his measurements.

As far as anomaly detections are concerned, a simple check is carried out; a normal scenario would be one in which the result was within the range specified by the Observation's "referenceRange" or, on the contrary, if the result is below the minimum or above the maximum, it would mean that there's an anomaly in that measurement, and the user should take the necessary actions.

In Fig. 1.26, the reader will be able to see the code implementation for detecting anomalies in the NodeJS gateway and, again, the results will be sent to the application and the user will be able to see if there's any anomaly. For the continuous example dragged throughout this document, as can be clearly seen in Fig. 1.22, there's an anomaly in the temperature measurement, as the obtained value is below the minimum reference, while everything is correct in the blood pressure measurement.

```
function checkAnomaly (minRef : Number, maxRef : Number, value : Number) : Number {
  if (value < minRef){
    return 1;
  }
  else if (value > maxRef){
    return 2
  }
  else{
    return 0;
  }
}

var value = currentObservation['valueQuantity']['value'];
var highReference = currentObservation['referenceRange'][0]['high']['value'];
var lowReference = currentObservation['referenceRange'][0]['low']['value'];
var anomalyResult = checkAnomaly(lowReference, highReference, value);
```

**Fig. 1.26** Anomaly detection code implementation

## CHAPTER 2. BEHEALTHAPP

Now that the reader is already familiar with all the theory behind how medical data is stored using the HL7 FHIR standard, as well as how this data is generated, stored, accessed and used within the context of this project, it's time to adopt a practical approach and start addressing how the mobile application "beHealthApp", together with the rest of elements that allow its correct functioning, has been carried out.

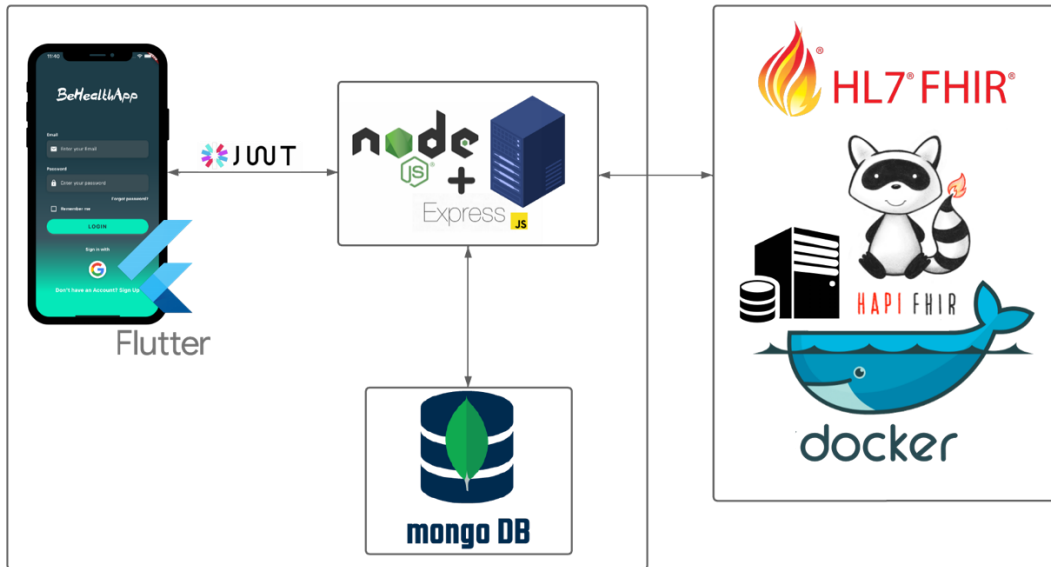
In this chapter, the reader will be able to find all the information regarding the solution itself, from the general architecture together with a detailed description of each of its elements, to a close analysis of all functionalities that "beHealthApp" offers.

Please note that this chapter will have a descriptive approach, which aim is to present all the features of the application in the context of their potential to serve as an important complementary tool for remote tracking of the patient's health state.

### 2.1. General architecture of the solution

Before going into details, it's important to address how the ecosystem that shapes the project has been structured. As the reader will be able to see in Fig. 2.1, there are two main blocks to consider. The first one is the HL7 FHIR API (the right block in Fig. 2.1) which, essentially, has already been covered in the first chapter. However, there are some extra details regarding the technologies used to set up this block that will be introduced in this second one (see section 2.3.1). The second one (the left block), is the proprietary ecosystem that has been built from scratch for this project.

The elements within these blocks will be individually addressed in separated sections of this second chapter, but in this section, the focus will be put on their relations, how they are interconnected, and a general overview of the data flows between one another.



**Fig. 2.1** General architecture of the solution

As has been seen in the first chapter, the HAPI FHIR server (which is an implementation of an HL7 FHIR server, as will be seen in future sections) is in charge to store the medical information following the HL7 FHIR standards, however, there's other information that is also needed to enable the correct functioning of the "beHealthApp" application. This extra information can be, for instance, the medical devices of the user, groups, group requests, the user's password to access the application, conversations, among others. This information will be stored in a separate non-relational database based on MongoDB.

The ecosystem has been built so that the communication with the FHIR server is carried out by a server in the middle, making it transparent for the mobile application. This server has been built using NodeJS and Express, and it acts as a bridge between the mobile application, the MongoDB server, and the HAPI FHIR server. It also handles all the security behind the connections, making sure that the user has the required roles to be authorized for certain operations.

So, when the user wants to carry out some action within the application (register, login, store a result from a measurement, access his measurement's record, send a message to his doctor, etc.), firstly an HTTP request with the pertinent Authorization headers will be sent to the NodeJS server. Let's say the user has just made a measurement, and an Observation Resource has been created and has to be sent to the HAPI FHIR server to be stored. The patient's FHIR "id" is stored in the MongoDB server, together with other information with regards to the patient, and so, the NodeJS server first has to access the MongoDB server to obtain this "id", and then it can send an HTTP request to store the new Observation instance with the correct "id" of the user. Please note

that this description does not fit exactly how the system works, but it's a superficial representation of the whole logic of the ecosystem.

## **2.2. Introduction to BeHealthApp's functionalities**

Now that the reader has already gone through all the necessary background information to understand the pillars of this project, let's now jump into practical matters as far as the application is concerned.

By being able to store medical data in the official HL7 FHIR standard, not only will it be able to persist and be useful for several applications, but other collateral implementations now become possible. An example of the latter is enabling a service for doctors so that they are able to create groups of patients within the application, and to access their measurements records for remote diagnosis or periodical checks in a very simple and direct manner, as well as providing a chat service so that the doctor and the patient can chat if necessary. So, as a consequence of making this medical data persistent with a standardized format, now the application itself can act as a platform where doctors and patients can share data and communicate between each other.

This chapter will address all elements and functionalities within the mobile application, and the reader will be able to discover the features that position it as a potent tool to complement and facilitate remote health tracking.

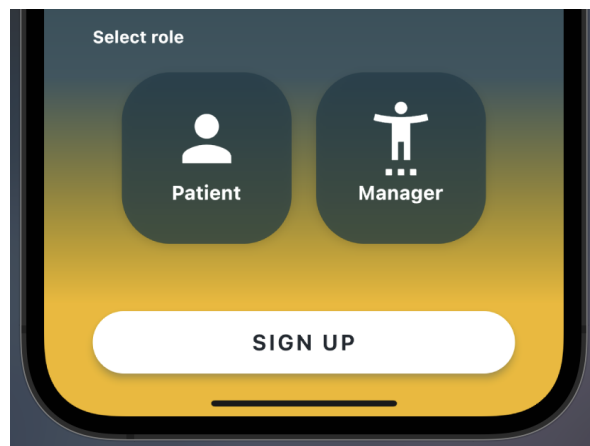
### **2.2.1. Profiles / Roles of the user**

As can be deduced from the previous introduction, two different types of user profiles will be needed in order to be able to provide a doctor-patient interaction within the application. With the aim of generalizing the former, this specific profile simulating the role of a doctor has been called "Manager", while the latter remains as "Patient".

Users under the role "Patient" and "Manager" (which, from now on, will be just referred as a patients and managers, respectively), share a considerable amount of features regarding the available functionalities within the application. Both have full access to Medical Devices Management functionalities (see section 2.2.2), Measurements and Automatic Anomaly Detection (described in section 2.2.3), as well as to Profile Settings features (section 2.2.5).

The main difference can be found in the Groups and Shared Data block of functionalities. The general idea is that managers are able to access all measurement's results from his patients, as well as other relevant data, while patients can only access their own. The specific differences will be exposed naturally throughout the close description of this given block carried out in section 2.2.4, and so, it would be redundant to mention them here.

This differentiation is carried out at the very beginning, already when the user is carrying out the registration process (see Fig. 2.2). In this first version of the application, the user can choose any of both just by pressing on one of the options, without any further steps. Please, note that in future versions, some kind of authentication and subsequent authorization processes should also be carried out in order to ensure that only healthcare professionals can sign up as managers.



**Fig. 2.2** Role selection in the registration process within the application

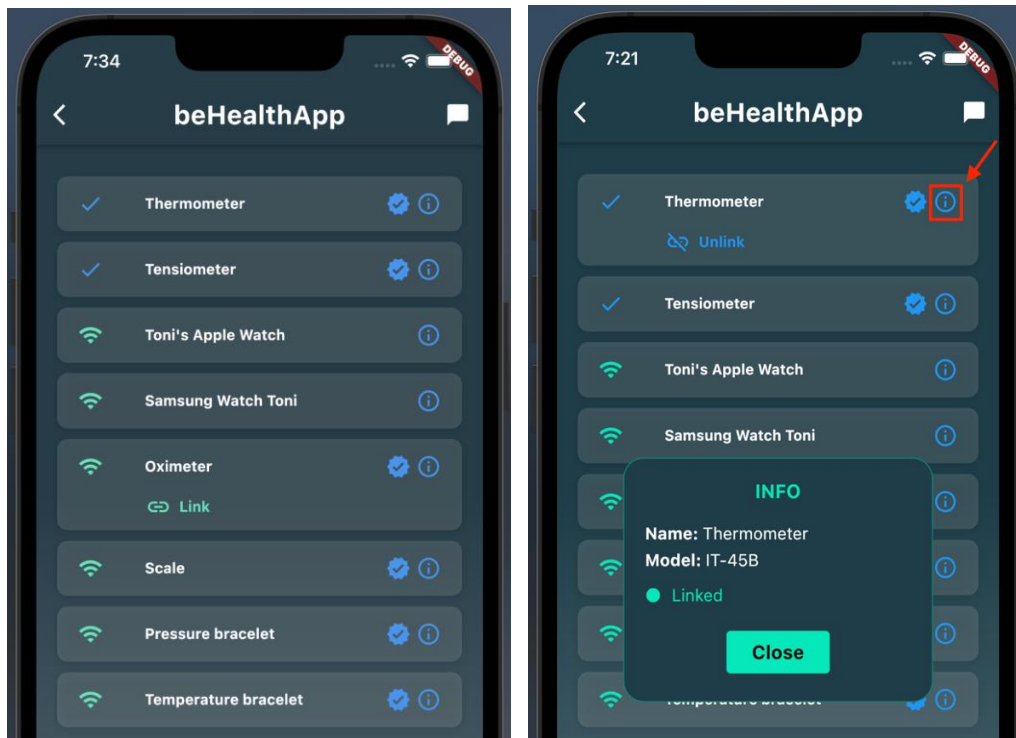
Finally, maybe the reader will have correctly noticed that any role-related information appears in the FHIR Patient Resource (in Fig. 1.10, for instance), because, in fact, this is a parameter that is stored within the MongoDB database and managed by the NodeJS API, being totally independent of the FHIR database.

### **2.2.2. Medical Devices Management**

One of the main features that the application provides is the possibility to connect Bluetooth medical devices in order to be able to carry out measurements and store the resulting data using the HL7 FHIR standard.

To begin with, the connection and synchronization process of the medical devices with the application is not within the scope of this project, which is limited to their management, as well as the management of the data that they generate.

The application offers a very user-friendly interface where the user will be able to see all the nearby medical devices detected, as well as some of their basic information such as their name, model and if the device is currently linked to the application or not (see Fig. 2.3). The user will also be able to link or unlink a device by pressing the "Link/Unlink" button that appears when the device is selected.

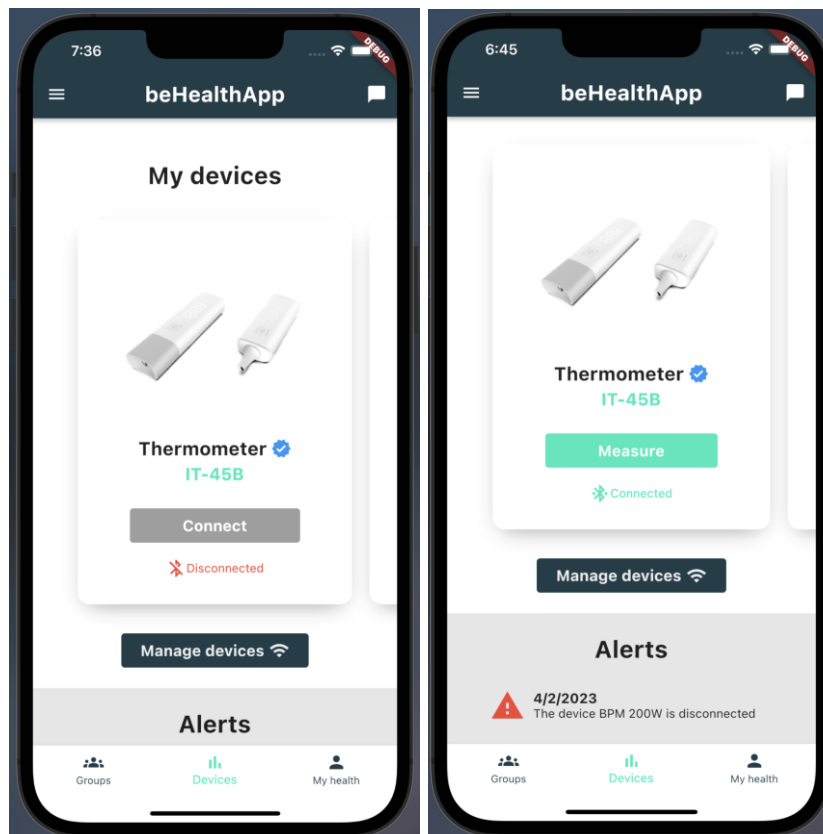


**Fig. 2.3** Medical devices management page within the application

Note that a blue check symbol is present only in some devices. When this symbol is present, it means that the device is verified and, consequently, it complies with the current legislation in order to be considered as clinically certified medical equipment to monitor vital signs. This feature has been introduced in order to remove the limitation that the exclusion of non-certified devices would entail, so that the user can also connect other devices (such as sport or smart watches, among others) to monitor his vital signs, while being aware of this lack of certification.

Once a device has been linked to the application, the user will always be able to find it in the “Devices” section. However, he will need to carry out the connection process to prepare the device to carry out measurements, just by pressing a simple button, as can be seen in Fig. 2.4. Moreover, the application shows a list of alerts indicating the devices that are currently disconnected.





**Fig. 2.4** Connection process of a linked device and disconnection alerts.

The list of linked devices of the patient is stored in the MongoDB database, which again means that devices management is completely independent of the FHIR server. Please note that the FHIR standard does offer a Resource to store Devices (see Fig. 2.5) but, as has already been mentioned, the physical bonding process between devices and the application is outside the scope of this project, which means that all the intrinsic information of the device remains unknown so far. In this first version, devices are stored with a very simple structure, and so, it would be completely pointless to generate a FHIR Device instance with so little information. However, in a future version where this connection has already been integrated to the ecosystem of the project, then the migration to Device Resources could be considered.

Therefore, so far, the FHIR server remains strictly independent of anything other than the user's information as a Patient Resource instance and the measurements that he carries out as Observation Resource instances.

## UML Diagram (Legend)

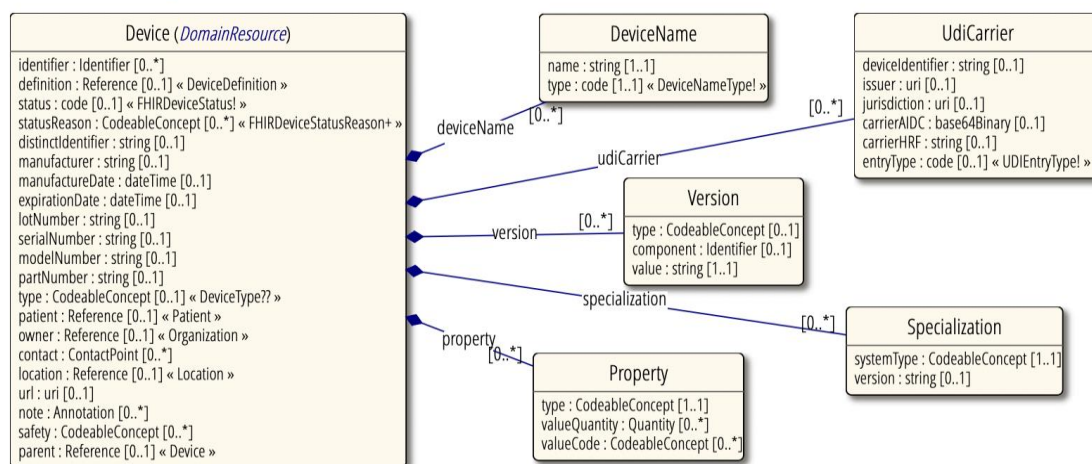


Fig. 2.5 UML structure of a FHIR Device Resource

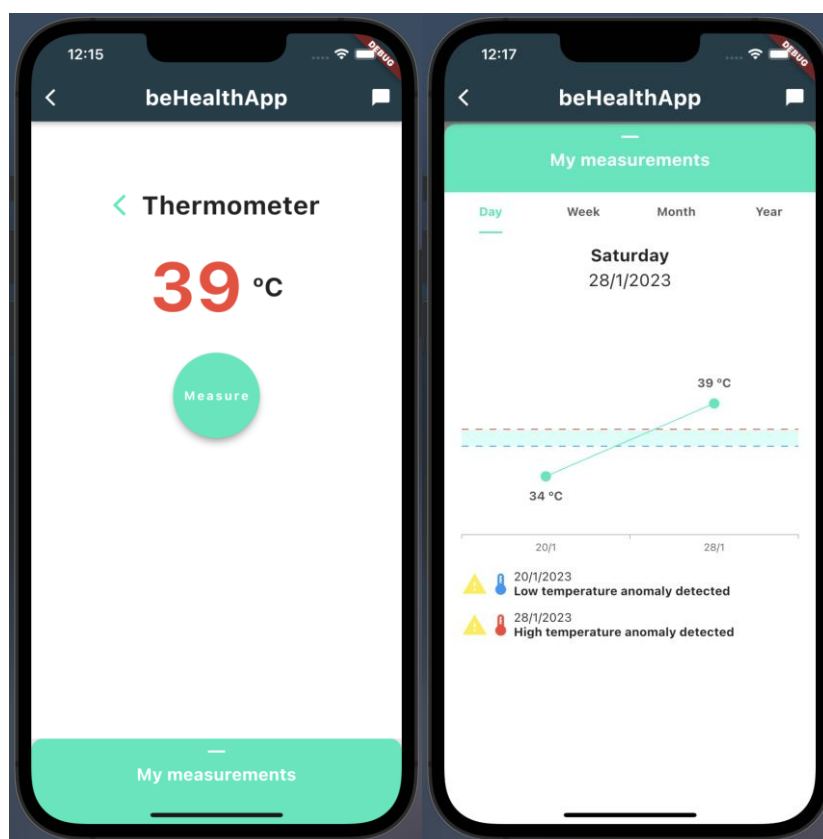
### 2.2.3. Measurements and Automatic Anomaly Detection

Once the user has connected one of their devices, then a measurement can be carried out. Given that within the scope of this first version the physical Bluetooth connection with the medical device is simulated, all measurements values are pseudo-randomly generated between corresponding low and high values that adequately adjust within normal ranges in humans.

In Fig. 2.6, the reader will be able to visualize the generation of a new measurement, together with the measurements' record of that given device.

After the measurement has been done, the application will send the result to the NodeJS API gateway, where a FHIR Observation Resource will be generated using the received data and the FHIR id of the user, which is requested from the MongoDB database and then sent to the FHIR server, where it will be stored referencing the respective Patient Resource instance. And finally, then the user will be able to ask for that information and visualize it in the application.

Please, note that this whole process, as well as any other process that implies communication with the FHIR server, is completely transparent for the application, and so, any future changes with regards to the former will be just neglected by the latter, as well as the potential migration of some data to the FHIR server, as commented in the previous section regarding Device Resources.

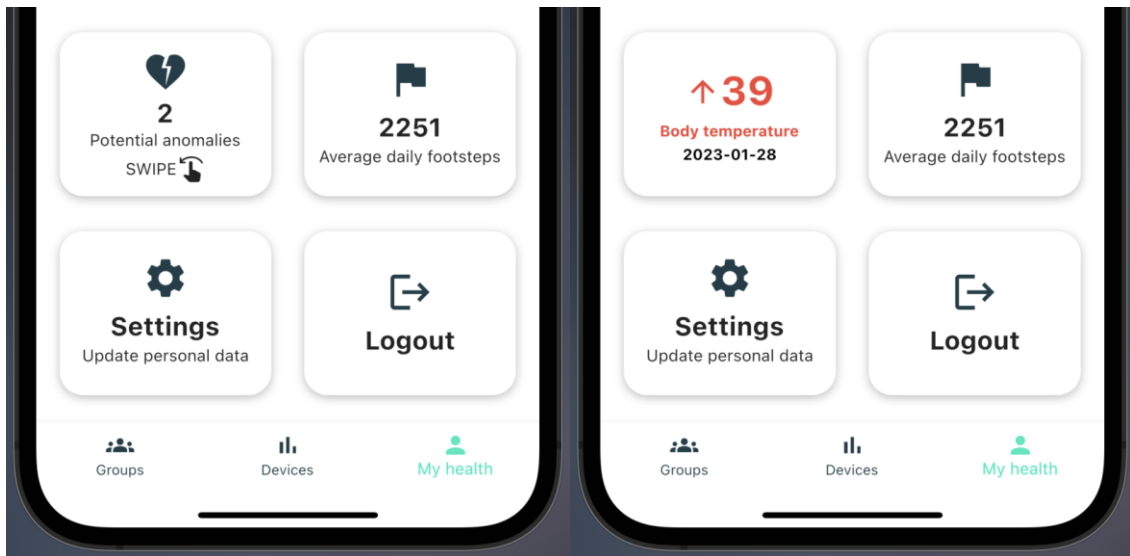


**Fig. 2.6** Temperature measurement (left) and daily record with anomalies (right)

Moreover, the user will be able to see if there's any anomaly in one or various measurements in the same page where the measurements' record for that device can be found. The decision on whether a measurement is considered as an anomaly or not is also transparent for the application, as this is a backend calculation carried out in the NodeJS API gateway, and the decision is based on the reference range indicated by the Observation Resource instance (see Fig. 1.17).

In the particular example of Fig. 2.6, the most recent measurement turns out to be above range and the first one (show in Fig. 1.17) below range. Consequently, two notifications are displayed under the chart. As will be seen in the next section, not only will this record be accessible by the patient itself, but also for a potential manager supervising the health state of the former, and so, these alerts will be especially useful in this scenario to easily detect any anomaly and contact the patient, if the manager finds it necessary.

However, sometimes it might be unpractical to go through every single device and access its record in order to check if there's any anomaly, and therefore, the application also offers the possibility to see a summary of the user's anomalies in the profile section, as can be seen in Fig. 2.7, where the two same anomalies displayed in Fig. 2.6 can be also found.



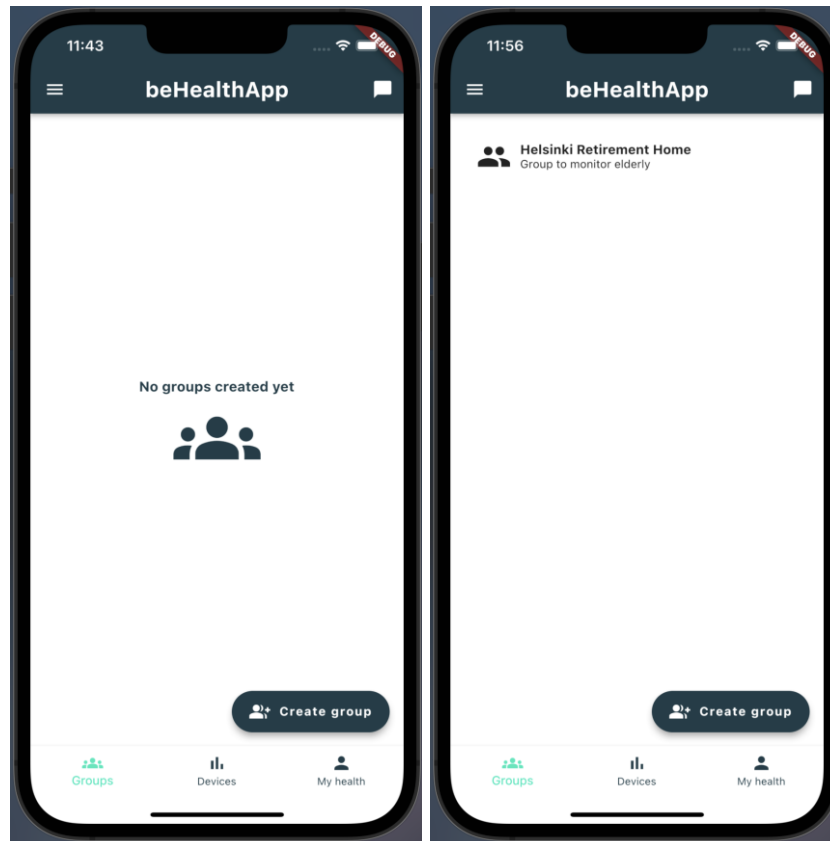
**Fig. 2.7** List of anomalies in the profile section

#### **2.2.4. Groups and Shared Data**

One of the most important features of the application that provides high degree of utility to the medical data generated by the patient is the possibility for the manager to create groups of patients. These allow the manager to have remote access to measurements carried out by any of the patients that belong to a specific group.

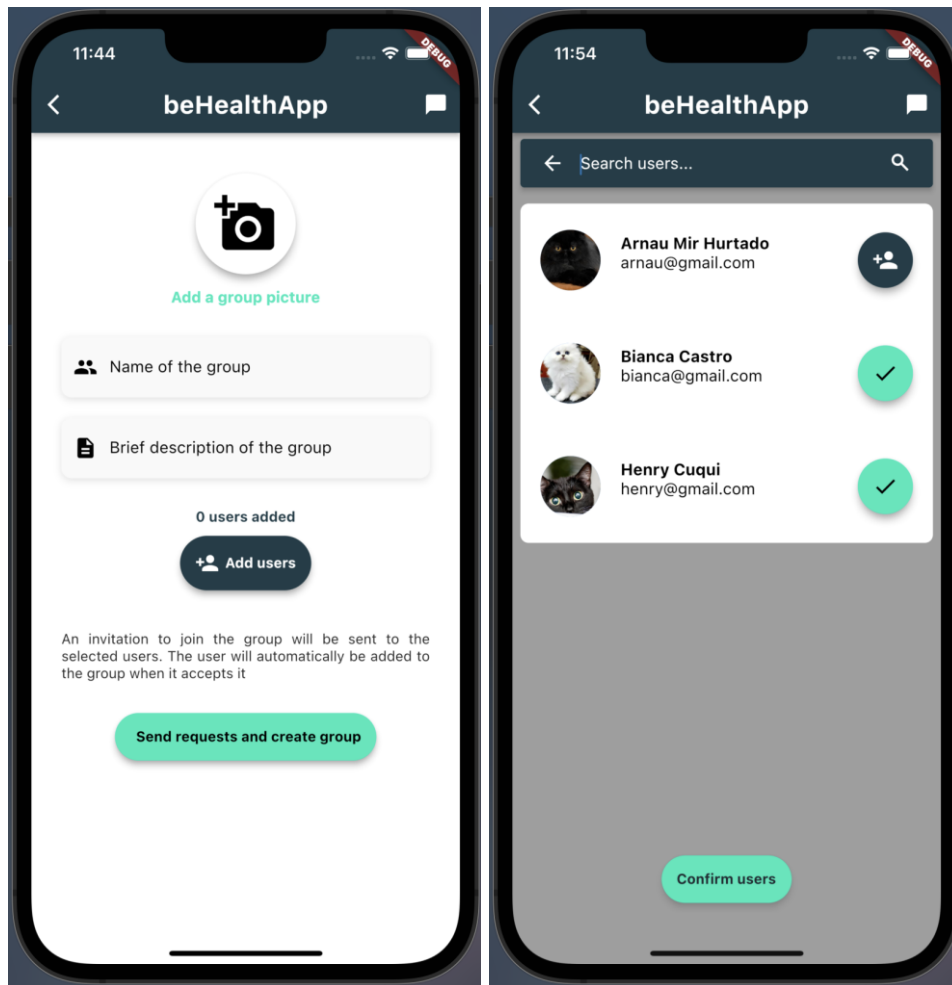
So, note that this is where the biggest differentiation between both roles within the application is more noticeable, as only users with the manager role will have the possibility to create and manage groups.

In Fig. 2.8, the manager will be able to find all created groups in the “Groups” section of the application.



**Fig. 2.8** List of groups created by the manager

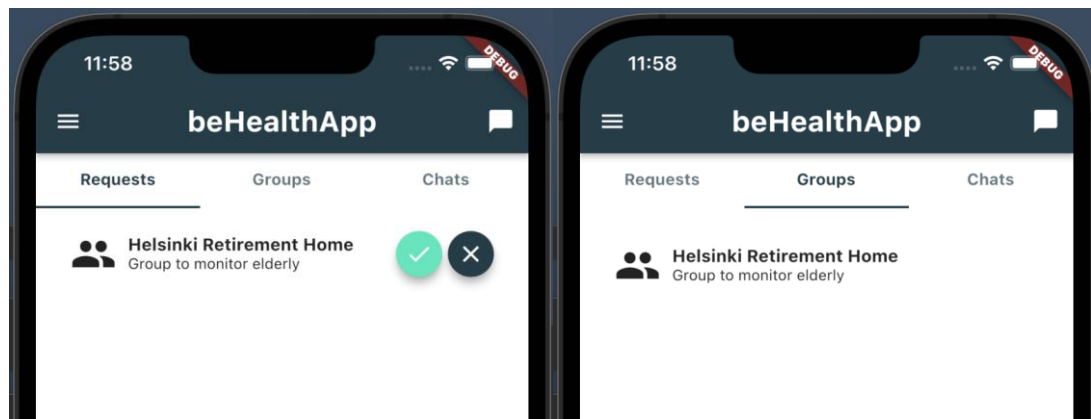
In order to create a group, the manager must provide a name, a description and a list of users that will be added. To facilitate the search of a specific user, the application offers a search bar where the manager can type the name of the wanted patient, and an automatic filtering will be carried out. This whole process can be seen in Fig. 2.9.



**Fig. 2.9** Creation process of a group by a manager

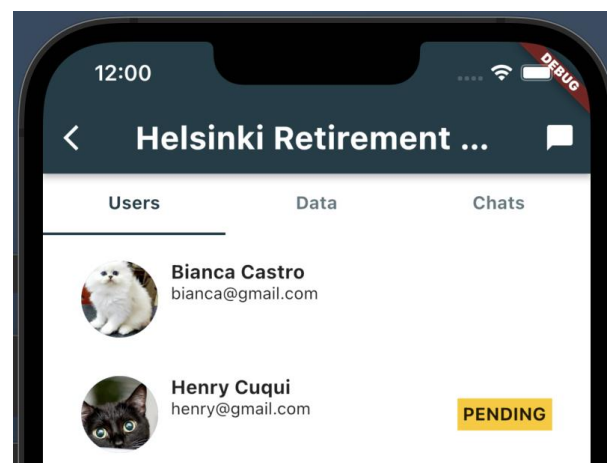
Once the group has been created, it would certainly be convenient to have some kind of intermediate process to ensure the confidentiality of the patients' data so that it can only be visible by managers that have been previously recognized and authorized by the patient. So as to do so, despite the patients have been added to a group, the manager won't have access to their data until they have authorized it by accepting the respective group request, which is a simple notification that basically offers the possibility to accept or decline the invitation.

As can be seen in Fig. 2.10, patients will find these requests in their "Groups" section. If they decline it, the patient will be definitely removed from the group, and otherwise, if the request is accepted, now the patient will be able to see the group in the groups list.



**Fig. 2.10** Group request accepted by the patient

The manager will be able to notice if a patient has accepted the invitation request when the “Pending” label next to the patient’s name within the group disappears (see Fig. 2.11).



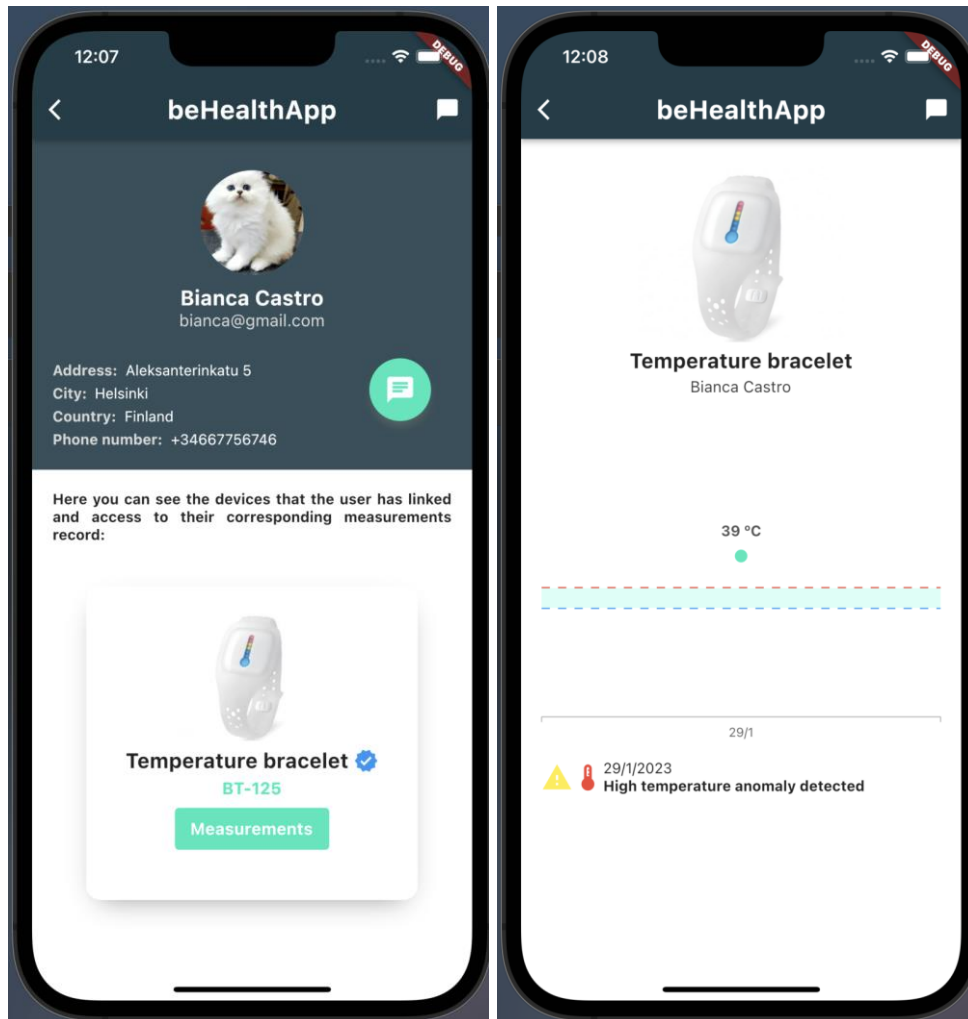
**Fig. 2.11** List of patients of a group from the manager’s perspective

The acceptance of the request by the patient also means that, the manager, henceforth, will have permissions to access the patient’s most basic personal data, along with medical data regarding measurements results generated within the context of the application.

As can be seen in Fig. 2.12, by accessing the patient’s profile, the manager can see the list of medical devices that the patient has linked, together with their measurements. This itself already allows the application to fulfill one of its main purposes; to use the medical data generated by a patient at home for remote partial diagnosis or tracking purposes.

Note that the manager, in order to access this data, sends a request to the NodeJS API gateway, which will connect to the FHIR server, where the data is

stored. So, not only can this data be used for remote tracking and partial diagnosis within the context of the application, but other future features that use these data could be implemented in future versions.



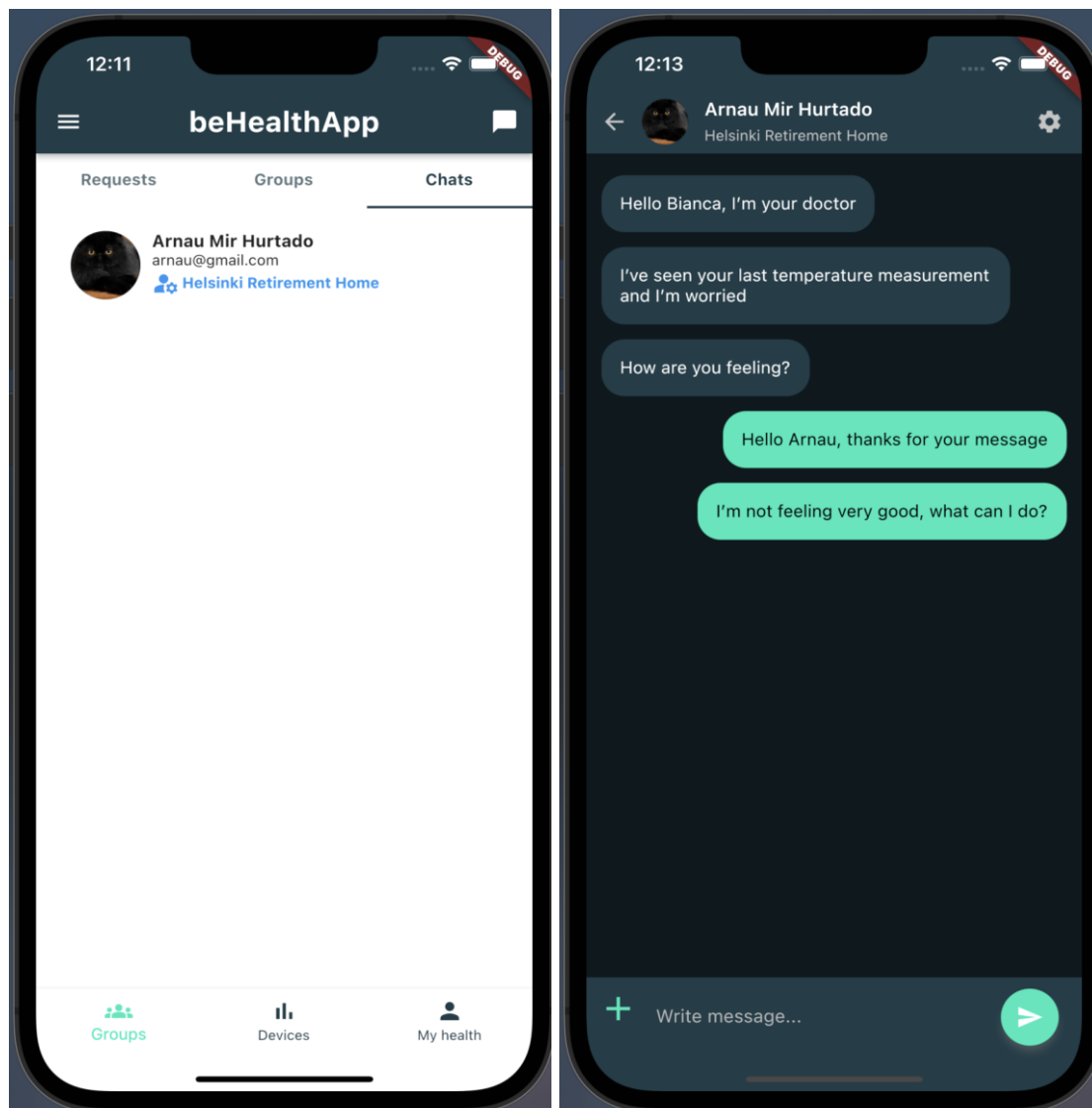
**Fig. 2.12** Patient's data from the manager's perspective

Moreover, the application also aims to be a tool that provides a unified set of features to offer a certain degree of remote healthcare, at least as far as measurements tracking is concerned, and therefore, it also offers an integrated chat service, so that the manager and the patients can exchange messages if they need to. This can be particularly useful for a similar case of Fig. 2.12, where the manager realizes that this particular patient has an abnormally high temperature in the last measurement carried out with the temperature bracelet, which is a verified and thus reliable medical device. In this situation, as can be seen in Fig. 2.13, the manager can contact the patient to ask for their health state and, if they find it necessary, arrange a traditional visit.

Persistent high temperatures, blood pressure or any other vital sign can be a reliable indicative that there's an underlying major problem, and so, its early



detection can be key to effectively treat it. In this sense, this application can be a very useful tool as well.



**Fig. 2.13** Chat service between manager and patient

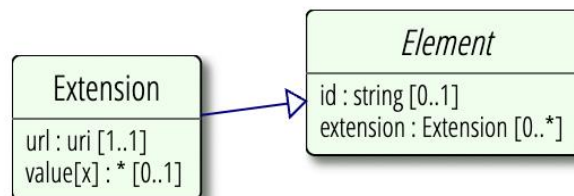
Medical data is sensitive data, no matter how simple and apparently irrelevant it is. Data concerning health means personal data related to the physical or mental health of an individual. The General Data Protection Regulation (GDPR) includes this type of data within the sensitive data category and, consequently, unauthorized disclosure may lead to various forms of discrimination and violation of fundamental rights. More emphasis in processing sensitive data legally will be put in section 2.5, but essentially, patients sharing the same manager, that is to say, they are within the same group, shouldn't be able to see other patient's medical data, being that privilege exclusive to the manager of the group.

However, throughout the implementation of the project, it was considered that it would be useful to try to exploit the fact that a patient is a group to provide them some extra value. The latter couldn't come by any means from their medical data, because of the reasons that have been just discussed, and so, the introduction of a new parameter that isn't considered as sensitive data was created; footsteps.

The application will continuously count the user's footsteps, and persist them (store the value, together with the date in the MongoDB database) at the end of the day. With this data, a lot of information can be extracted and also shared among all the users of the group.

Similarly to what happened with devices data, the number of steps of a user won't be stored in the FHIR server. HL7 FHIR standard offers the possibility to store custom information under the Extension Resource (see Fig. 2.14), which is just an extension of any other Resource. So, it exists the possibility to store the user's steps as an Extension Resource, linked to its respective Patient Resource.

However, at least in this first version of the application, it has been preferred to work under the premise to separate strictly medical standardized data from any other data generated within the context of the application. The migration of this data to a FHIR server by means of extensions is, however, an open thread that could be considered for future versions.



**Fig. 2.14** UML diagram of the Extension Resource

As can be seen in Fig. 2.15, users will be able to see their performance compared to the group average regarding daily number of steps, together with other useful information including the group's historical average per person, the registered average of the previous day with an indicator that compares the latter to the usual performance of the group, the consistency in terms of standard deviation, measuring how stable is the data, and a normal range indicator to let users know if they are within the normal range. Additionally, a ranking with the three users that registered a higher number of steps the previous day can be seen.

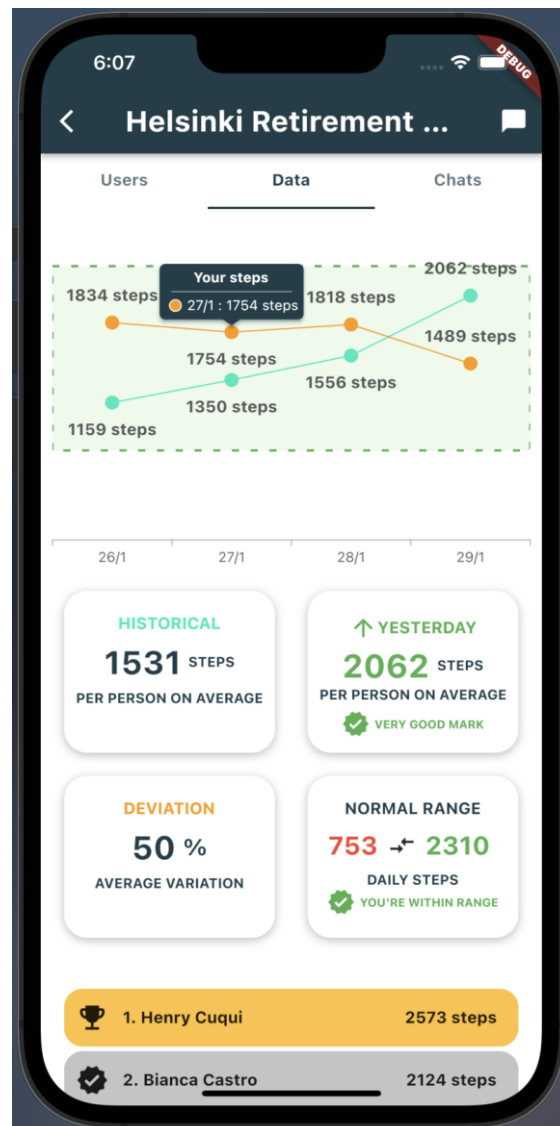
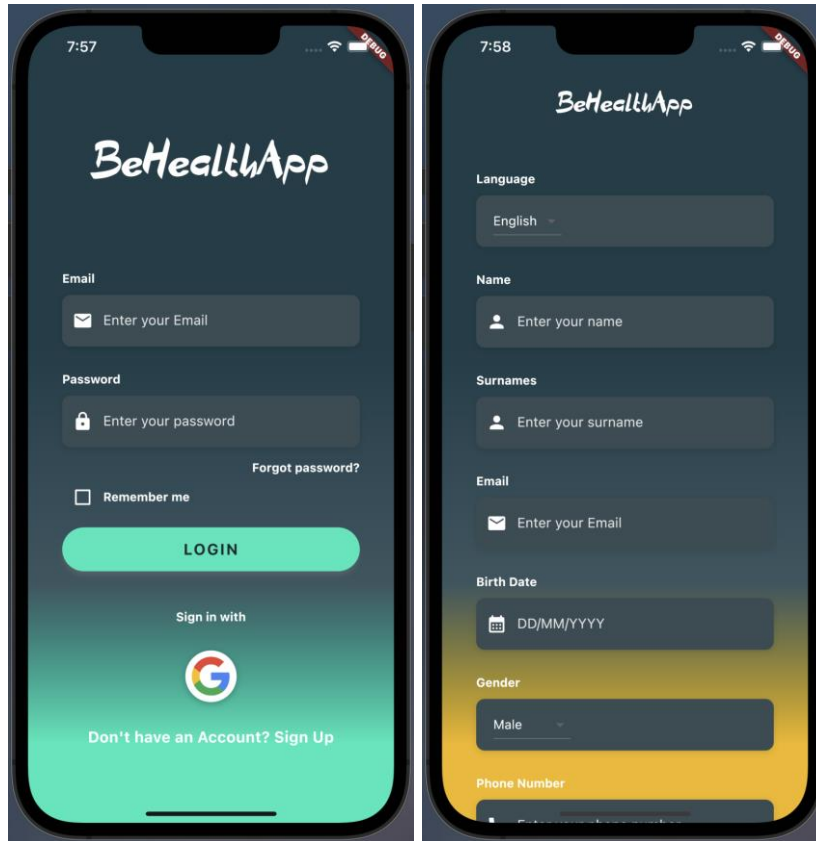


Fig. 2.15 Group's performance and statistics page

### 2.2.5. User credentials and profile management

In order to access the contents of the application, the user will need to authenticate by means of a login request to the NodeJS server. To do so, they will need to provide their email and password. In this first version of the application, the authentication by means of the user's Google account is still not supported, but should be in future versions.

If the user doesn't have credentials yet, first a sign in process is required. Throughout the latter, the user will be asked to fill a form with some personal data, which will be used to generate a FHIR Patient Resource. As was mentioned in previous sections, the role of the user is also assigned during the registration process. Both login and register pages can be seen in Fig. 2.16.



**Fig. 2.16** Login (left) and register (right) pages

For security reasons, the user's password is not stored in clear in the MongoDB database, but a hashed version of the password (see Fig. 2.17). A hash is a mathematical function that converts an input of arbitrary length into an encrypted output of a fixed length. The key feature of hashes is that they cannot be used to "reverse-engineer" the input from the hashed output. That is to say that hashes are one-way functions that always generate the same output for a given input, and so, the only way for the server to verify if the user's sent password is correct is to hash it, and compare if both hashes are identical.

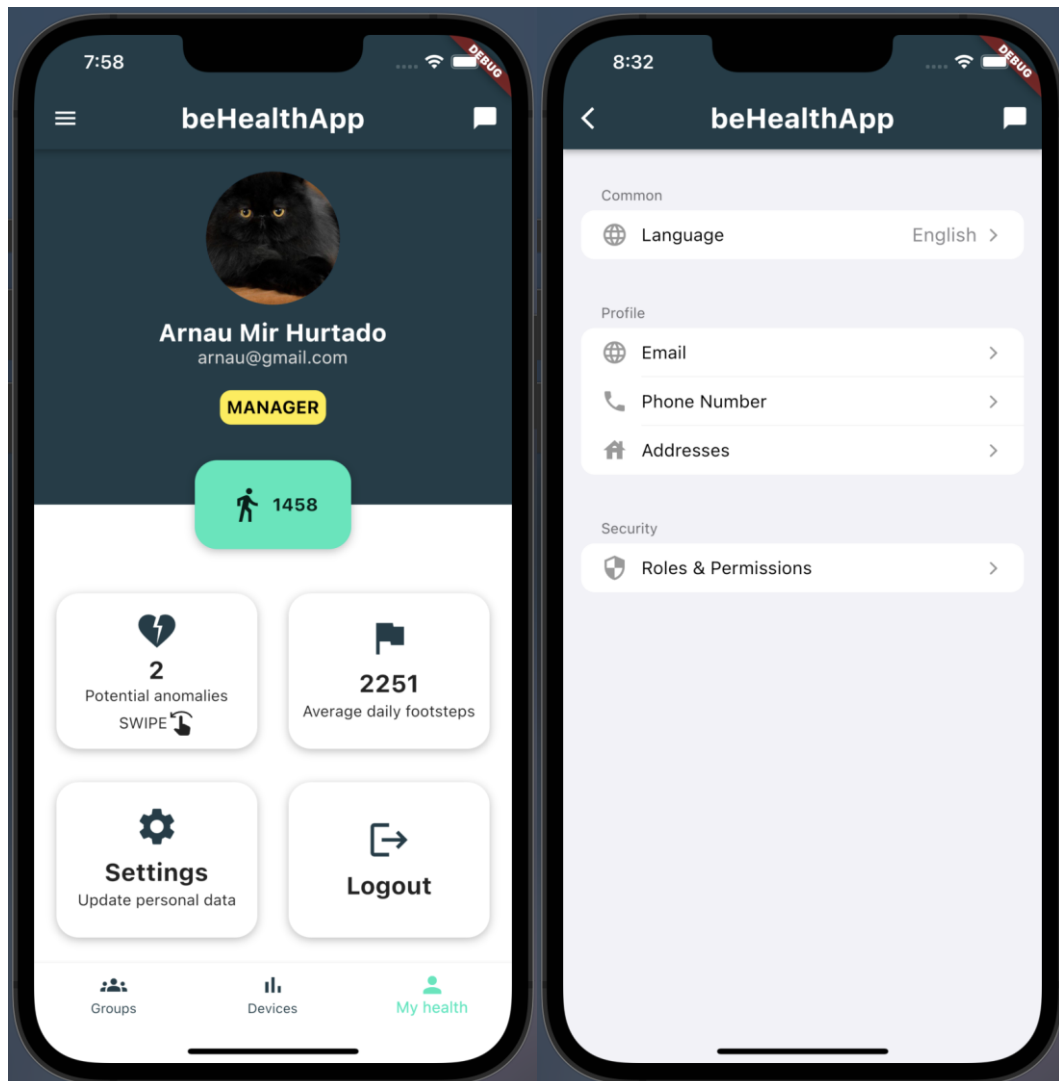
```

_id: ObjectId('63d64bd31ed0d7336efca507')
full_name: "Bianca Castro"
email: "bianca@gmail.com"
phone_number: "+34667756746"
password: "$2a$10$.WHhCkMPQB0LZc5b81nSuecJcyQ1w8JzMRgX2NJx4.5apEN7durZy" ←
language: "en"
  ▶ role: Array
  ▶ groups: Array
  ▶ footsteps: Array
  today_footsteps: 2124
  image_url: "https://media.donedeal.ie/eyJidWNrZXQiOiJkb25lZGVhbC5pZS1waG90b3MiLCJl..."
  creationDate: 2023-01-29T10:34:59.969+00:00
  ▶ devices: Array
  __v: 0
  fhir_id: "556"

```

**Fig. 2.17** Hashed password stored in the MongoDB database

And finally, once the user has valid credentials, they can modify some of their data when desired, by means of the “Settings” section in the profile page of the application, as can be seen in Fig. 2.18.



**Fig. 2.18** User's profile (left) and settings (right) pages

### 2.3. Backend elements

The project's ecosystem can be separated in two different parts; the backend and the frontend. The former refers to the user interface, the application itself in this case, and the latter encompasses all the logic that is being carried out behind the user interface, including the servers and databases that are involved to deliver information to the user.

While section 2.4 will address the frontend part, this section aims to describe the different elements of the project's ecosystem that build the backend network

that communicates with the application to deliver information and to handle the user's petitions.

There are three elements involved in the project's backend; the HAPI FHIR API, where the medical data is stored, the NodeJS API Gateway, which is a proprietary API developed throughout this project that acts as an intermediate entity to communicate the application with the HAPI FHIR API and the MongoDB database (see Fig. 2.1), and the MongoDB database itself, where other data is stored.

In this chapter, a close description to these elements has been carried out.

### **2.3.1. HAPI FHIR API**

Over this document, a generic server where the HL7 FHIR Resources, such as the Patient and Observation Resources, are being stored has been mentioned several times, but little detail has been put into how this element has been set up.

To create from scratch a functional implementation of a server supporting the HL7 FHIR standard would be extremely challenging itself, and certainly inefficient, given the fact that there's an open-source implementation that have already been developed and that has been around for eighteen years; HAPI FHIR.

HAPI FHIR is a complete implementation of the HL7 FHIR standard for healthcare interoperability developed in Java by an open community developing software licensed under the business-friendly Apache Software License 2.0. The FHIR standard is still relatively new, and so, the HAPI FHIR library is still under constant modifications and updates that add more features and flexibility to the standard.

In this project, a public Docker image implementing the HAPI FHIR library for the server side has been used, and so, the HAPI FHIR server has been locally used as a Docker container. There's already a public Docker image implementing the HAPI FHIR library in Docker Hub, and so, its incorporation to the project was immediate.

Please, note that the HAPI FHIR library provides a full mechanism for connecting to FHIR REST servers and to easily handle Resources with built-in methods to set the different attributes of a resource. This initial version of the project handles very simple data without any complexity, and therefore, a custom implementation of the Patient and Observation resources has been created. Moreover, there's extra custom data that is being used in the project that is obviously not considered in the HAPI FHIR implementation.

So, the HAPI FHIR implementation has only been used in the server's side to deploy a FHIR RESTful API, and not in the client's. However, the migration to a client's side server built in Java using the HAPI FHIR library that would replace

the currently NodeJS API Gateway should be considered as data becomes more complex.

### 2.3.2. NodeJS API Gateway

One of the proposed objectives when carrying out the project was to isolate the mobile application from all the logic behind data handling, so that the application never communicates with the HAPI FHIR server nor the MongoDB database, getting itself rid from all that workload. Instead, it only communicates with an intermediary server using an HTTP connection.

The entity created from scratch to serve as the mentioned intermediary is a server built in Typescript using NodeJS and Express. In Fig. 2.19, the reader will be able to find the server's file where the connection with the MongoDB database and the different API endpoints base URL can be found, together with some middlewares that have also been used to improve the server's capabilities, offering a certain degree of security in front of some cracking attacks, the possibility to send data in a compressed format, and automatically handling CORS issues, among others. Regarding the endpoints, this file sets the base URL for each of the logic endpoints handled by the server. There are four categories: patients, managers, groups, and conversations.

```
class Server {
  public app: express.Application;

  constructor(){
    this.app = express();
    this.config();
    this.routes();
  }

  config() {
    //MongoDB settings
    let DB_URL = 'mongodb://localhost:27017/beHealthApp';
    DB_URL = DB_URL.replace("user", process.env.DB_USER!);
    DB_URL = DB_URL.replace("password", process.env.DB_PASSWORD!);

    mongoose.connect(DB_URL).then(db => console.log("DB is connected"));

    //Settings
    this.app.set('port', process.env.PORT || 3000);

    //Middlewares
    this.app.use(morgan('dev')); //Allows to see by console the petitions that eventually arrive.
    this.app.use(express.urlencoded({extended:false}));
    this.app.use(helmet()); //Offers automatically security in front of some cracking attacks.
    this.app.use(compression()); //Allows to send the data back in a compressed format.
    this.app.use(cors()); //It automatically configures and leads with CORS issues and configurations.

    this.app.use(express.json({limit: '25mb'}));
    this.app.use(express.urlencoded({limit: '25mb',extended: true,}));
  }

  routes() {
    this.app.use('/api/patients', patientRoutes);
    this.app.use('/api/managers', managerRoutes);
    this.app.use('/api/groups', groupRoutes);
    this.app.use('/api/conversations', conversationRoutes);
  }
}
```

Fig. 2.19 Server's settings and setup file

In Fig. 2.20, the structure of the Group entity can be seen. Again, groups have only been created within the context of the application to provide some users under the manager role the possibility to access their patient's medical data, and so, all data regarding groups and group requests will remain within the local BeHealthApp ecosystem, the left block of the general architecture shown in Fig. 2.1.

In this first version, groups have a simple structure, containing a name and a description that will be set by the manager that creates the group (see Fig. 2.9), the manager id, the list of patient's id that belong to that group, a list of requests id (group requests are another entity themselves, as can be seen in Fig. 2.21), a list of daily step averages (used in the group information page seen in Fig. 2.15) and a creation date.

```
const GroupSchema = new Schema({
  name: {type: String, required: true},
  description: {type: String},
  manager: { type: mongoose.Schema.Types.ObjectId, ref: 'User' },
  patients: [{ type: mongoose.Schema.Types.ObjectId, ref: 'User' }],
  requests: [{ type: mongoose.Schema.Types.ObjectId, ref: 'GroupRequest' }],
  dailyStepsAverages: [
    {
      date: {type: String},
      value: {type: Number}
    }
  ],
  creationDate: {type: Date, default:Date.now},
})

export default model('Group', GroupSchema);
```

**Fig. 2.20** Group Schema in the NodeJS server

```
const GroupRequestSchema = new Schema({
  group: { type: mongoose.Schema.Types.ObjectId, ref: Group },
  patient: { type: mongoose.Schema.Types.ObjectId, ref: Patient },
  creationDate: {type: Date, default:Date.now},
})

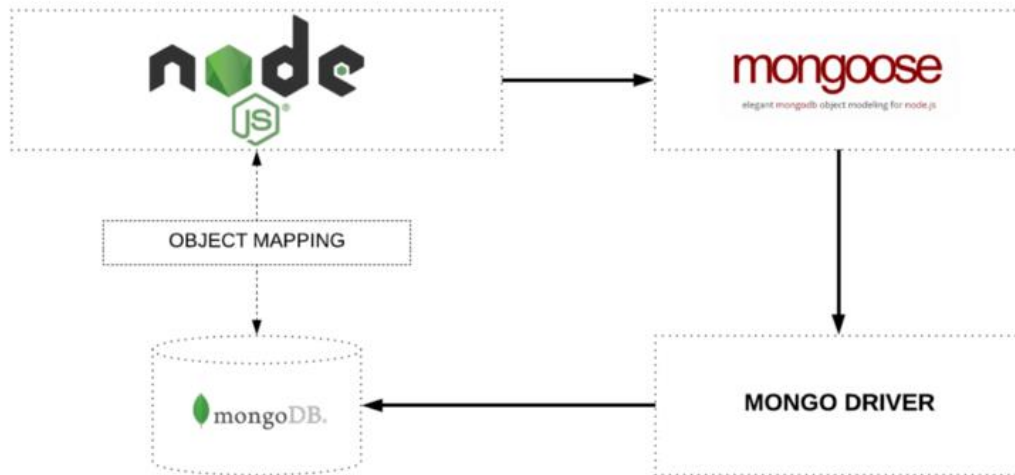
export default model('GroupRequest', GroupRequestSchema);
```

**Fig. 2.21** Group Request Schema in the NodeJS server

As will be seen in the next section, this data will be stored in the MongoDB database as a document. In order to establish the connection between the database and the NodeJS server is by means of the Mongoose library, which is an Object Data Modeling (ODM) library for MongoDB and NodeJS. It manages relationships between data, provides schema validation, and is used to translate between objects in code and the representation of those objects in MongoDB (see Fig. 2.22).



To create these objects, Mongoose allows to create their structure in NodeJS by means of Schemas, which can be seen both in Fig. 2.20 and Fig. 2.21. A Mongoose schema is a document data structure that is enforced via the application layer.



**Fig. 2.22** Mongoose Mapping between NodeJS and MongoDB

Every instance of a particular Schema that is created (the creation of different groups, for example) will be stored under the same “collection”. Collections are the MongoDB equivalence to tables in relational databases such as an SQL table.

The requirements of this project have led to the creation of four collections, as can be seen in Fig. 2.23.

Collection Name	Storage size	Documents	Avg. document size	Indexes	Total index size
conversations	20.48 kB	1	672.00 B	1	20.48 kB
grouprequests	20.48 kB	1	93.00 B	1	36.86 kB
groups	20.48 kB	1	507.00 B	1	20.48 kB
users	20.48 kB	3	729.00 B	1	36.86 kB

**Fig. 2.23** The four beHealthApp collections seen from MongoDB Compass

There are a lot of methods defined in the NodeJS server that are used as endpoints by the application, but the example of figures Fig. 2.24 and Fig. 2.25 really illustrates how the project’s ecosystem works, regarding the internal

communication between the different blocs. Fig. 2.24 shows the code defined within the application itself, where the user needs to download the data of their FHIR Patient Resource stored in the HAPI FHIR API, but instead of directly fetching the latter, the application fetches the NodeJS API endpoint, so that the external communication with the HAPI FHIR API is completely transparent for the application. In the application, this request is done after the user is logged in, and so, the user already knows their internal id, which is the unique identification of that particular User document, who in turn contains the FHIR Patient Resource id (referred as “fhir\_id” in the User schema) stored in the MongoDB database. This id is passed as a path parameter in the GET HTTP request to the NodeJS API. Then, the latter has to perform a search in the MongoDB database to find a user instance with the received id. Once found, the “fhir\_id” attribute of the instance is used by the NodeJS server to finally fetch the HAPI FHIR server, where the corresponding Patient Resource will be sent back with status code 200, if there’s actually an existing instance holding that id. Finally, in the last stage of the communication, the NodeJS server will send the received Patient Resource JSON to the application, which will be accordingly parsed. This whole exchange of data is carried out by means of an asynchronous communication.

```
Future<User?> getPatientFromFhir(String idApi) async {
  var res = await http.get(Uri.parse('$baseUrlApi/fhir/$idApi'), headers: {
    'accept': 'application/fhir+json',
    'authorization': LocalStorage('key').getItem('token'),
  });
  if (res.statusCode == 200) {
    User patient = User.fromJSON(jsonDecode(res.body));
    return patient;
  }
  return null;
}
```

**Fig. 2.24** FHIR Patient request data to the NodeJS server

```
public async getPatientFromFhirByApiId(req: Request, res: Response) : Promise<void> {
  const patientFound = await Patient.findById(req.params._id);

  if(patientFound == null) {
    res.status(404).send("Patient not found.");
  }
  else{
    var urlFhir = "http://localhost:8080/fhir/Patient/" + patientFound.fhir_id;
    const response = await fetch(urlFhir, {
      method: 'get',
      headers: {'Accept': 'application/fhir+json'} });

    const data = await response.json();
    res.status(200).send(data);
  }
}
```

**Fig. 2.25** User verification before request

Finally, in Fig. 2.26, the reader will be able to find an example showing some of the NodeJS API endpoints regarding a patient, used by the application to get some information or to carry out certain actions.

```
routes() {
  this.router.get('/', [authJwt.VerifyToken], this.getAllPatients);
  this.router.get('/filter/:_text', [authJwt.VerifyToken], this.filterUsers);
  this.router.get('/:_id', [authJwt.VerifyToken], this.getPatientById);
  this.router.get('/fhir/:_id', [authJwt.VerifyToken], this.getPatientFromFhirByApiId);
  this.router.get('/devices_patient/:_id', [authJwt.VerifyToken], this.getPatientDevicesById);
  this.router.get('/name/:fullName', [authJwt.VerifyToken], this.getPatientByFullName);
  this.router.post('/', [authJwt.VerifyToken], this.addPatient);
  this.router.post('/observation', [authJwt.VerifyToken], this.addObservation);
  this.router.post('/add_device/:_id', [authJwt.VerifyTokenPatient], this.addDeviceToPatient);
  this.router.post('/remove_device/:_id', [authJwt.VerifyTokenPatient], this.removeDeviceFromPatient);
  this.router.post('/login', [authJwt.VerifyToken], this.login);
  this.router.put('/:_id', [authJwt.VerifyTokenPatient], this.updatePatient);
  this.router.delete('/:_id', [authJwt.VerifyTokenPatient], this.deletePatient);
  this.router.get('/anomalies/:_id', [authJwt.VerifyToken], this.getAnomaliesPatient);
}
```

Fig. 2.26 Patient's endpoints in the NodeJS server.

### 2.3.3. MongoDB

As the reader already knows, MongoDB has been used as the internal database of the project. In this section, a more detailed analysis of this block of the project's ecosystem will be carried out.

MongoDB is a document-oriented non-relational (NoSQL) database used for high volume data storage. NoSQL refers to the wide variety of technologies that have been developed as a response to modern applications needs, which include the generation of large volumes of data in constant evolution, among others. Relational databases weren't designed to face the scalability and agility that modern applications require, and neither to take advantage of the processing power that exists nowadays.

NoSQL databases are more scalable and offer an increased performance compared to the SQL databases. Moreover, their data models address several issues that the relational model ignores, such as:

- The equal consideration between huge volumes of structured, semi-structured and not structured under constant variation data.
- In line with OOP
- Horizontal Scaling instead of Vertical Scaling (MongoDB collections are self-contained and not couple relationally, leading to the possibility to share the load to different nodes, one for each collection, for example, instead of increasing the processing power of a single server, as would happen with relational databases).

Fig. 2.27 and Fig. 2.28 are an example to illustrate the difference between relational and non-relational databases.

In the case of Fig. 2.27, if a user needs to have a list of hobbies, two different tables would likely be created, and a relation is established by means of the id of the user.

Users				
ID	first_name	last_name	cell	city
1	Leslie	Yepp	8125552344	Pawnee

Hobbies		
ID	user_id	hobby
10	1	scrapbooking
11	1	eating waffles
12	1	working

**Fig. 2.27** Relational database (SQL) example

However, if the same example had to be considered using a non-relational model (see Fig. 2.28), all the information could be included in a single JSON file, so that no joints are required, resulting in faster queries.

```
{
  "_id": 1,
  "first_name": "Leslie",
  "last_name": "Yepp",
  "cell": "8125552344",
  "city": "Pawnee",
  "hobbies": ["scrapbooking", "eating waffles", "working"]
}
```

**Fig. 2.28** Non-relational database (NoSQL) example

Moreover, the reader might correctly have realized that, since the HAPI FHIR server continuously handles FHIR resources data using JSON, it must also use a NoSQL database. Certainly, NoSQL databases are used in nearly every industry, including the most highly critical use cases, where one could find healthcare records storage.

To conclude the section, Fig. 2.29 shows the MongoDB document for the group shown in Fig. 2.11, and sets a perfect example of a continuously changing structure, as requests and patients have to be dynamically added or removed, and every day a new entry to the daily steps average list have to be added. To define an equivalent structure in a relational context would be highly complex,

especially given the dynamism that a group document has within the context of the application.

```
_id: ObjectId('63d650d61ed0d7336efca533')
name: "Helsinki Retirement Home"
description: "Group to monitor elderly"
manager: ObjectId('63ca536f40f3d3de539e9ee4')
▼ patients: Array
  0: ObjectId('63d64bd31ed0d7336efca507')
  1: ObjectId('63d64d941ed0d7336efca522')
▼ requests: Array
  0: ObjectId('63d650d71ed0d7336efca543')
▼ dailyStepsAverages: Array
  ▼ 0: Object
    date: "2023-01-26T00:00:00.000+00:00"
    value: "2318"
  ▶ 1: Object
  ▶ 2: Object
  ▶ 3: Object
creationDate: 2023-01-29T10:56:22.953+00:00
__v: 0
```

**Fig. 2.29** Real example from the project of a Group document in MongoDB

## 2.4. User Interface Development

At this point, the reader is already familiar with all blocks of the ecosystem of the project that provide the “background logic”. As has been seen, each of them has its role and, together, they carry out all the necessary operations and data exchanges so that the application can be provided by the necessary data that will be shown to the user by means of a User Interface (UI) which is the application itself.

The features that the application provides have already been described throughout section 2.2, and so, the aim of this section is to address the technical details regarding the development of the application as for the technology that have been used and how the internal management of the data has been approached.

### 2.4.1. Technology choice: Flutter

In the last few years, the app development market has grown a lot, and is expected to grow exponentially in the coming decade. So, much development and research has been done to deliver the best performance and to make the app development process faster and much simpler.

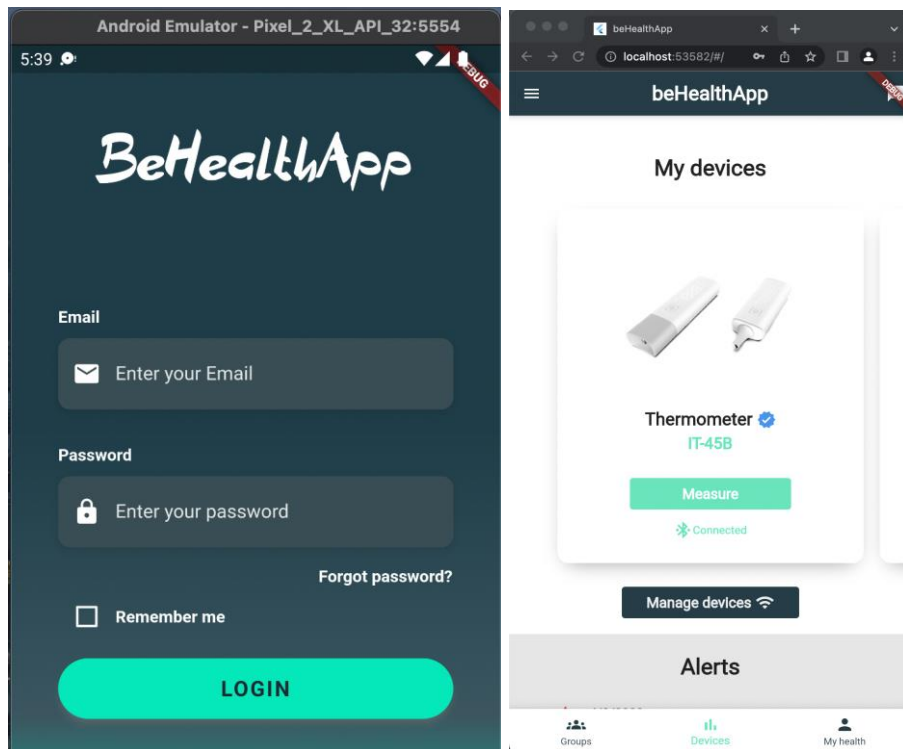
Applications can be broadly categorized as:

- **iOS Apps:** These apps are made for Apple devices and are made using the Swift language.
- **Android Apps:** These apps are made for Android devices and are created using Java and Kotlin.

Thus, this differentiation leads to a large list of problems, among which one can find the fact that iOS and Android applications work very differently internally and, consequently, there's no Cross-Platform Dependency that could carry out a "translation", leaving no other option than the parallel development of two versions of the application, one for each Operative System (OS). This obviously also means more employees and the subsequent increased development cost.

However, most of these problems were solved with the launch of Flutter in May 2017. Flutter is a mobile app Software Development Kit (SDK) created by the Google that allows developers to create web, desktop, and cross-platform apps that run both on Android and iOS devices, as well as in web. Among other reasons that will be discussed in a bit, this was the main pillar that led to the decision of using Flutter to develop the application, as it sets a good solution for a scenario where a single person is carrying out all the code.

Along this document, it can be seen that all figures showing features from the application have been running on iOS (specifically using an iPhone 13 Pro Max simulator) but, as can be seen in Fig. 2.30, thanks to having developed the application using Flutter, it can also run in Android and web with no issues.



**Fig. 2.30** Cross-platform demonstration with Android (left) and web (right)

The Flutter framework uses the Dart programming language and facilitates the task to build user interfaces that are beautiful, fast and responsive. Flutter works with “widgets”, which are the basic building blocks of the app. Some straight examples of Flutter widgets are buttons and text, which are a very illustrating example of the concept of widgets hierarchy, as a button widget can have a text widget as a so called “child”. In fact, there are widgets (such as the container, column or row widgets) which are only intended to serve as an external “skeleton” to organize inner widgets in a particular way.

There are two types of widgets:

- **Stateless widgets:** They don’t have an inner state, and thus, they are not expected to dynamically undergo changes that could affect their visualization. Some examples would be buttons or static image widgets.
- **Stateful widgets:** They have an internal state which can change over time. This can be reflected in how the widget looks and behaves. Some examples could be input fields, or a whole application page which, in essence, is a huge widget containing a lot of children widgets (which can be both stateless and stateful).

In Fig. 2.31, the reader will be able to find the Dart code where the login button shown in the left part of Fig. 2.27 can be seen. The button is created under as a “RaisedButton” widget, which has a list of attributes for customization reasons, together with the “onPressed” attribute, which contains the code that will be executed when it is clicked (in this case, a trigger to initiate the login process). Moreover, it has a “child” attribute, which contains the “Text” button with its own properties.

```
return RaisedButton(
  elevation: 5.0,
  onPressed: () {
    if (emailController.text.isNotEmpty && passwordController.text.isNotEmpty) {
      BlocProvider.of<AuthorizationBloc>(context).add(LoginEvent(emailController.text, passwordController.text));
    }
  },
  padding: EdgeInsets.all(15.0),
  shape: RoundedRectangleBorder(
    borderRadius: BorderRadius.circular(30.0),
  ), // RoundedRectangleBorder
  color: Color.fromARGB(255, 5, 232, 185),
  child: Text(
    translate('pages.login_page.login'),
    style: TextStyle(
      color: Color.fromARGB(255, 33, 40, 48),
      letterSpacing: 1.5,
      fontSize: 18.0,
      fontWeight: FontWeight.bold,
      fontFamily: 'OpenSans',
    ), // TextStyle
  ), // Text
); // RaisedButton
```

**Fig. 2.31** Flutter widget’s structure

After seeing Fig. 2.31, the reader might have realized that the text of the button is actually the result of a translate method provided by the Flutter Translate library. This is because this application also provides is the support for both English and Spanish languages. The user will be able to select their language

from the very beginning in the registration process, and will be able to change it from the profile configuration section. This has been achieved by statically defining all the text within the application twice in two JSON files (see Fig. 2.32), and then referencing the shared identifier of that particular word or sentence.

```

assets > i18n > {} es.json > ...
34 "pages":
35 {
36   "login_page":
37   {
38     "enter_email": "Escribe correo electrónico",
39     "enter_password": "Escribe tu contraseña",
40     "email": "Correo electrónico",
41     "password": "Contraseña",
42     "forgot_password": "Has olvidado tu contraseña?",
43     "remember": "Recuérdame",
44     "login": "ACCEDER",
45     "sign_in_with": "Acceder con",
46     "no_account": "No tienes una Cuenta?",
47     "sign_up": "Regístrate"
48   },
49   "register_page":
50   {
51     "name": "Nombre",
52     "surnames": "Apellidos",
53     "enter_surnames": "Escribe tus apellidos",
54     "enter_name": "Escribe tu nombre",
55     "birth_date": "Fecha de Nacimiento",

```

```

assets > i18n > {} en.json > ...
34 "pages":
35 {
36   "login_page":
37   {
38     "enter_email": "Enter your Email",
39     "enter_password": "Enter your password",
40     "email": "Email",
41     "password": "Password",
42     "forgot_password": "Forgot password?",
43     "remember": "Remember me",
44     "login": "LOGIN",
45     "sign_in_with": "Sign in with",
46     "no_account": "Don't have an Account? ",
47     "sign_up": "Sign Up"
48   },
49   "register_page":
50   {
51     "name": "Name",
52     "surnames": "Surnames",
53     "enter_surnames": "Enter your surname",
54     "enter_name": "Enter your name",
55     "birth_date": "Birth Date",

```

**Fig. 2.32** Spanish (left) and English (right) JSON files

So, one can easily create both Stateless and Stateful widgets using the Dart programming language, and one can also use various other development tools such as the Dart Analyzer and the Flutter Inspector.

So, as a summary, there are several key benefits that led to the decision to use Flutter as the technology to develop the “beHealthApp” mobile application, including the following ones:

- Flutter is fast, as Dart is compiled into native code, meaning that there is no need for a JavaScript bridge, leading to fast and responsive results.
- Flutter gives the possibility to create cross-platform applications, which means that the same code can be used to run the project in iOS, Android and web from a single codebase, eliminating the need to carry out a parallel independent development for each OS.
- Flutter has a rich set of widgets and a very well developed and instructive documentation with a lot of examples.
- Flutter is open source, meaning that anyone can contribute to its development by creating libraries with new widgets.
- Flutter is free.
- It's created by Google, which gives a certain degree of confidence as Flutter will continue growing and getting improved.
- Different screen adaptability, as can be seen in Fig. 2.27. The screen sizes and ratios dynamically adapt the device where the application is running.



## 2.4.2. States Management: BLoC Pattern Architecture

The discussion of how to structure an app is among the most heavily debated topics that arise in the development stage. Android and iOS developers use the Model-View-Controller (MVC) pattern as a default choice when building apps. However, Flutter brings a new reactive style that is not entirely compatible with MVC. As a consequence, a variation of this classical pattern emerged from the Flutter community; the BLoC pattern, which is the architectural pattern that have been used to manage the internal states of the application.

BLoC stands for Business Logic Components, and its basic premise is that everything within the app should be represented as a stream of events that cause a change of the app's state. In simple words, BLoC is a component that mediates between what the user sees and the logic behind it, and thus, separating the presentation layer from the business logic.

Essentially, the application is constantly linked to a particular state (or multiple states, as will be seen later), which affects what the user sees in the app. A simple example would be when the user authenticates by carrying out the login. Before their credentials are validated, the application's state is "UnauthorizedState", but once the system has recognized the user, the state switches to "AuthorizedState", which obviously leads to a change in the UI, indicating the user that the login has been successful. To trigger this change of state, an event ("LoginEvent", for example) is sent from the UI to the BLoC element when the user presses the "login" button after having introduced the respective credentials. The BLoC knows that after receiving such event, it must send a request to the NodeJS server, within the context of the project, and depending on the response, emit a particular state ("AuthorizationFailedState or "AuthorizedState, for instance) that, again, will affect the UI. This whole process is illustrated by Fig. 2.33. Within the project, in fact, there are multiple BLoC components, each one to manage state and events related with a particular service. The previous example regarding authorization (which will be analyzed at the code level later in this section) would be handled by an "AuthorizationBloc".

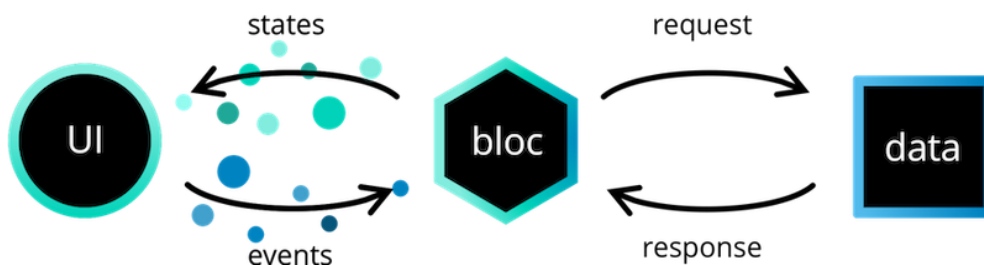
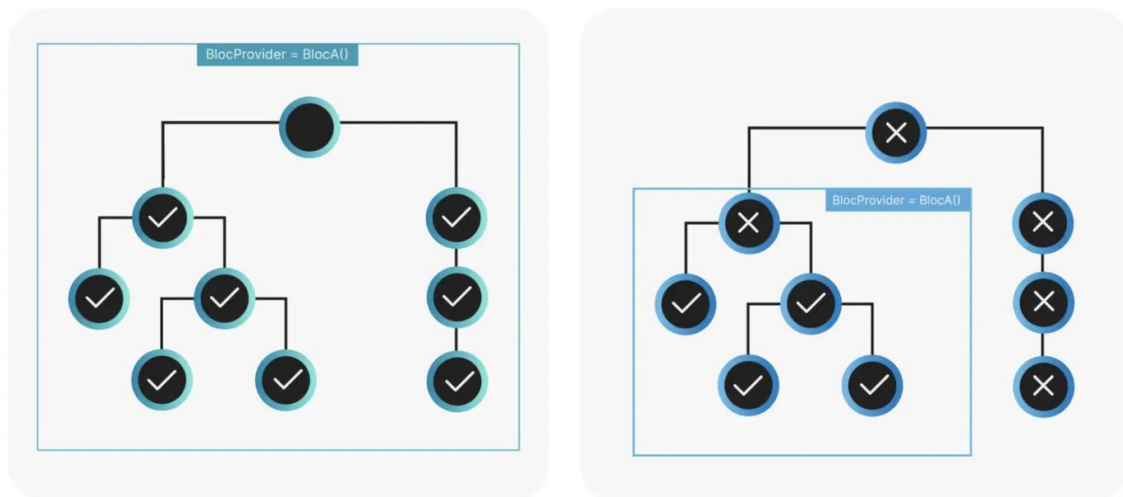


Fig. 2.33 BLoC general structure

So, now that the reader knows that in Flutter the whole UI is made of a hierarchy of Widgets and that the BLoC element is constantly listening to a

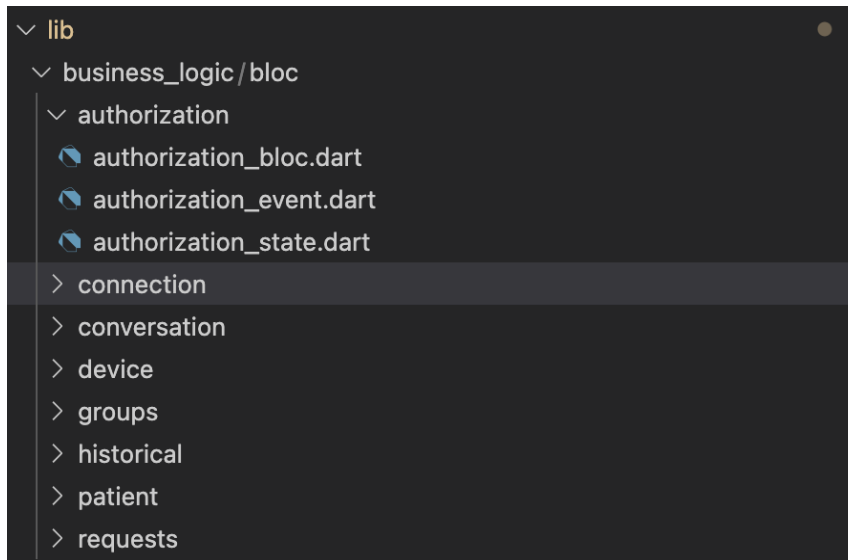
stream of events, the next upcoming question would be where does this element listen within the UI? A button itself can clearly trigger a login event, so does the block only need to provide service to that specific small widget? The answer depends on the type of service. If a widget is contained by a BLoC instance, the latter will only be accessible (to generate events or to access the current state) within the widget itself and from the child ones (see Fig. 2.34). States and events carry information (the “LoginEvent” will need to contain the credentials provided by the user, and the “AuthorizedState” will contain the profile’s information of the user that has been sent by the NodeJS server after the authentication has been successfully carried out. So, for this specific use case, it is quite obvious that the user’s information should be accessible from everywhere within the information, as other requests that require some user’s data (the id, in most endpoints, as has been seen in section 2.3.2). That’s why the “AuthorizationBloc” is actually declared in the root widget of the application.



**Fig. 2.34** BLoC Hierarchy

However, there are other services that only affect very specific parts of the application (groups, conversations, or historical measurements data, for example) and, in this case, the respective BLoC is declared way lower as far as the widgets hierarchy is concerned.

In Fig. 2.35, the different BLoC entities that have been used in the project can be found within the business logic folder. As can be seen, each BLoC contains three files; one where the different states are defined, another one where the list of events are defined, and a third one where the actions carried out by the bloc and the states that are emitted are defined.



**Fig. 2.35** BLoC entities used in the project

In Fig. 2.36, the reader will be able to take a closer look to the files where the Authorization BLoC states and events have been defined. There are some states which don't need to hold any information, and are only used to determine what the UI must show to the user, while other states and events do hold information, such as the "AuthorizedState" and the "LoginEvent".

```

part of 'authorization_bloc.dart';

abstract class AuthorizationState extends Equatable {
  const AuthorizationState();
}

class UnauthorizedState extends AuthorizationState {
  @override
  List<Object?> get props => [];
}

class RegisteringState extends AuthorizationState {
  @override
  List<Object?> get props => [];
}

class AuthorizingState extends AuthorizationState {
  @override
  List<Object?> get props => [];
}

class AuthorizedState extends AuthorizationState {
  final User user;
  const AuthorizedState(this.user);
  @override
  List<Object?> get props => [user];
}

part of 'authorization_bloc.dart';

abstract class AuthorizationEvent extends Equatable {
  const AuthorizationEvent();
}

class RegisterEvent extends AuthorizationEvent {
  final User patient;
  final bool isManager;
  const RegisterEvent(this.patient, this.isManager);
  @override
  List<Object?> get props => [patient, isManager];
}

class LoginEvent extends AuthorizationEvent {
  final String email;
  final String password;
  const LoginEvent(this.email, this.password);
  @override
  List<Object?> get props => [email, password];
}

class LogoutEvent extends AuthorizationEvent {
  @override
  List<Object?> get props => [];
}

```

**Fig. 2.36** Authorization BLoC states (left) and events (right) definition

Based on the previous definitions of states and events, now the inner logic and mapping within the BLoC element itself can be carried out, as shown in Fig. 2.37. As can be seen, each event received by the BLoC leads to the emission of new states. For instance, when a "LoginEvent" is received, the first thing that the BLoC does is emit a state to let the user know that the system is processing

the request (“AuthorizingState”), and then, depending on the server’s response, the corresponding authorized or not authorized state will be emitted.

```
class AuthorizationBloc extends Bloc<AuthorizationEvent, AuthorizationState> {
  final UserService _patientService;
  AuthorizationBloc(this._patientService) : super(UnauthorizedState()) {
    on<RegisterEvent>((event, emit) async {
      emit(RegisteringState());
      final response = await _patientService.addUser(event.patient, event.isManager);
      if (response != null) {
        emit(UnauthorizedState());
      }
    });
    on<LoginEvent>((event, emit) async {
      emit(AuthorizingState());
      final userApi = await _patientService.login(event.email, event.password);
      if (userApi != null) {
        emit(AuthorizedState(userApi));
      } else {
        emit(UnauthorizedState());
      }
    });
    on<LogoutEvent>((event, emit) async {
      emit(UnauthorizedState());
    });
  }
}
```

**Fig. 2.37** Authorization BLoC logic definition

As far as the presentation layer is concerned, different widgets will be displayed to the user depending on the current state of the application. This mapping between the state and the widget that have to be displayed can be done by means of an intermediary function that returns a Widget. An example for the BLoC entity managing devices’ states can be seen in Fig. 2.38.

```
Widget getWidget(DeviceState state, BuildContext c) {
  if (state is DeviceMeasureDoneState) { ...
  }
  if (state is DeviceSelectedState) { ...
  }
  if (state is DeviceMeasuringState) { ...
  }
}
```

**Fig. 2.38** Mapping function between states and widgets

## 2.5. User Authorization: JWT

As was covered in section 2.2.1, there are two different roles within the application context; managers and patients. Managers are also patients, but they have extra privileges to carry out some more actions, such as creating and managing groups, or accessing their patients' medical data.

The NodeJS server has a vast list of endpoints which, if no further protection is implemented, can be accessed publicly. So, it's obvious that some kind of authorization method has to be implemented, so that only registered users and depending on their role can access the server's endpoints. In this project, JSON Web Token (JWT) has been used to achieve so.

JWT is an open industry standard used to share information between two entities (in this case the application and the NodeJS server) that contains JSON objects which have the information that needs to be shared. Moreover, each JWT is signed by means of a hash, to ensure the JSON's content integrity, so that it cannot be altered by the client or a malicious party.

So, when the user sends a "login" request, after verifying that the user exists in the MongoDB database, the server will generate a token that will be sent to the application. This token is the result of signing (by means of a server's private key) a JSON containing some information regarding the user's identity. From then on, the user will need to attach this token as an authorization header, so that the server can verify the integrity of the request.

As can be seen in Fig. 2.39, the structure of a JWT contains three parts:

- **Header:** Indicates the signing algorithm that is being used so that the server can verify the signature, and the type of token, which in this case it's JWT.
- **Payload:** Contains the JSON object itself with the data that the server will use to verify the user.
- **Signature:** A string that is generated via a cryptographic algorithm that is used to verify the integrity of the JSON payload.



Fig. 2.39 Structure of a JSON Web Token (JWT)

Now that the reader is already familiar with JWT basics, let's take a closer look to the specific use case of the project. As can be seen in Fig. 2.40, the JWT payload only contains the MongoDB document id of the user together with their full name and role. This code is defined within the "login" function defined in the NodeJS server, and the "patientFound" variable is the user's data sent by the MongoDB database after verifying that the user exists. This JSON is then signed using a secret key, and sent to the application. The validity of this token has been set to one hour, and so, after this time, the session expires and the user needs to authenticate again in order to receive a new token. This adds an extra security layer in case the user's token is somehow intercepted by a malicious party.

```
const SECRET = process.env.JWT_SECRET;
const token = jwt.sign({
  {
    id: patientFound._id,
    full_name: patientFound.full_name,
    role: patientFound.role
  },
  SECRET!,
  {
    expiresIn: 3600
  }
});

res.status(200).send({ 'token': token, 'user': patientFound});
```

**Fig. 2.40** Signature process of the JWT payload in the NodeJS server

When the application receives the token, it stores it (see Fig. 2.41), and attaches it to every single request as an authorization header (see Fig. 2.42).

```
if (res.statusCode == 200) {
  var token = JWTtoken.fromJson(await jsonDecode(res.body));
  storage.setItem('token', token.toString());
  User user = User.fromJSON(jsonDecode(res.body)['user']);
  return user;
}
```

**Fig. 2.41** The application stores the received token from the NodeJS server

```
Future<List<AnomalyReport>?> getAnomaliesPatient(String patientId) async {
  var res = await http.get(Uri.parse('$baseUrlApi/anomalies/$patientId'), headers: {
    'authorization': LocalStorage('key').getItem('token'), ←
  });

  if (res.statusCode == 200) {
    var data = jsonDecode(res.body);
    List<AnomalyReport> listAnomalies = [];
    for (int i = 0; i < data.length; i++) {
      listAnomalies.add(AnomalyReport(codingDisplay: data[i]['codingDisplay'], value: data[i]['value'],
    )
    )
    return listAnomalies;
  }
  return null;
}
```

**Fig. 2.42** The application attaches the token as an authorization header

When the NodeJS server receives a request, the first thing it does is to check the JWT. First, it fetches the header part, where it is able to discover the algorithm that has been used to sign the payload. By definition, a hash is not reversible, and so, the only way for the server to verify that the signature is valid is to generate it again with the header and the body of the JWT using its private key, and then checking if they coincide. Note that if the incoming JWT's body is different, this step will generate a different signature and the request will be discarded. The request will also be ignored if JWT is expired.

As was seen before, the JWT payload contains the roles of the patients, and thus, after the JWT has been validated (valid signature and not expired), the server can proceed to check the content of the payload and, depending on the user's permissions, the request will be accepted or declined with a 403 forbidden code, indicating that the user doesn't have permissions to access that endpoint. This process can be seen in Fig. 2.43.

```
try {
  decoded = jwt.verify(token!, SECRET!);
} catch (e) {
  res.status(403).send({ message: "Invalid token" });
  return;
}

const role: Array<String> = decoded.role;

if (!role.includes(Roles.MANAGER)) {
  res.status(403).send({ message: "Not authorized. User needs elevation" });
  return;
}
```

**Fig. 2.43** Server verification of the user's permissions

## CONCLUSIONS AND CONTINUITY

The digitalization of data within the healthcare system has proven to be an important trend that is continuously transforming the way patients and professionals communicate and treat medical conditions. One of the most important developments in digital healthcare is the widespread use of telehealth, and the last few years have witnessed so with the SARS-CoV-2 pandemics.

Thanks to digitalization, healthcare providers can access patients' data remotely, regardless their location, which can be particularly useful in cases where the patient is unable to be treated in a healthcare center environment, due to physical or mental pathologies, as well as other external reasons that make it impossible for the doctor to treat their patients conventionally.

Moreover, a more holistic approach can be reached by means of the utilization of digitalized medical data in a vast list of realms within the healthcare industry, such as individual diagnosis, relying on the patient's health record, large scale studies or remote health state tracking.

There's, however, some potentially relevant medical data that is constantly generated, but never used, due to the lack of a mechanism to persist it in a database following a worldwide known standardized format. This is the case for rutinary medical devices commonly owned at home, such as thermometers, pressure bracelets, oximeters, etc. Note that this is only possible if the medical device allows a Bluetooth connection, and given that their price is generally higher, so far this application could be used in more clustered environments and situations, such as nursing homes, where a unique device could be used to monitor all residents' vital signs and be remotely checked by a doctor, or a case where the hospital itself provides the device to the user.

This project has presented a solution to this use case, by means of the creation of a mobile application that allows to connect medical devices and carry out measurements, which will be stored following the current HL7 FHIR standard format, making this data compatible for real healthcare system practical utilization and allowing interconnectivity with other medical services that follow the same standard. Moreover, the application offers an integrated service for healthcare professionals to manage groups of patients and remote tracking, as well as a certain degree of automatic anomaly detection of measurements. To do so, a whole ecosystem together with the application has been implemented, including an internal MongoDB database connected to a NodeJS server that acts as a gateway to allow the communication between the application and the HAPI FHIR server.

However, this first version of the software still lacks a list of features to be implemented so that it could really be released as a commercial product, such as the implementation of a mechanism to physically establish the connection between the devices and the application, and some security implementations to



protect the user's data, such as the incorporation of Transport Layer Security (TLS) in the connection between the different elements that make up the project's architecture given that, currently, only user authorization is implemented in the NodeJS server by means of JWT.

Moreover, so far, the UI considers that a device is only able to carry out measurements for a unique vital sign, but certainly there are devices that can provide measurements for multiple ones, and so, this use case should also be addresses in a future version.

As far as roles administration is concerned, a future version of the application should also include a mechanism to ensure that a user can only be registered with a "manager" role if they can really prove that they are real healthcare professionals with a license of some sort.

Due the simplicity of the generated data within the application and for other reasons regarding efficiency, the server gateway was developed in TypeScript using NodeJS and Express. However, the migration to a Java server that fully implements the HAPI FHIR library on the client's side should be considered, as a lot of agility regarding data modification and a more organized code structure could be achieved.

And as a last potential improvement for a future version of the application, once more data regarding the device is known, after the implementation of the system to establish the connection between the latter and the application, the possibility to store the device's data under a FHIR Device Resource in the HAPI FHIR server instead of in the internal MongoDB database should be considered.

Healthcare data management implies a certain Information and Communication Technology (ICT) resources that will lead to certain energy costs from the server's side, and so, the final solution should provide some kind of proposal to minimize these effects, such as unifying both HAPI FHIR and MongoDB databases under a unique entity that manages both blocks internally, for example. However, the collection of digitalized clinical data can lead medical devices to have more powerful and valuable information, which implicitly means that decision making procedures in front of emergencies will be considerably expedited, and the possibility to gather and access clinical data remotely avoids unnecessary travel, contributing to the reduction of emissions.

## ANNEX A. HL7v3 XML message analysis

In Fig. A.1, the XML structure of the Transmission Wrapper can be seen, and it's the perfect example to justify the previously mentioned feature of the new messaging protocol, the one that contemplated expanding the protocol beyond the application layer of the OSI Model, as it serves as the lowest layer, identifying the message, its type, the trigger event and the receiver's responsibilities. The receiving application is described in the receiver/device tags and the sending application is described in the sender/device and the asLocatedEntity subsection identifies the facility. In this case, the receiver of the message is GHH LAB, located in ELAB-3, and the receiver is GHH OE, located in BLDG24. The Transmission Wrapper, acting as the root element, wraps the payload, which is another wrapper; the Control Act Wrapper.

```
<POLB_IN224200 ITSVersion="XML_1.0" xmlns="urn:hl7-org:v3"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<id root="2.16.840.1.113883.19.1122.7" extension="CNTRL-3456"/>
<creationTime value="200202150930-0400"/>
<!-- The version of the datatypes/RIM/vocabulary used is that of May 2006 -->
<versionCode code="2006-05"/>
<!-- interaction id= Observation Event Complete, w/o Receiver Responsibilities -->
<interactionId root="2.16.840.1.113883.1.6" extension="POLB_IN224200"/>
<processingCode code="P"/>
<processingModeCode nullFlavor="OTH"/>
<acceptAckCode code="ER"/>
<receiver typeCode="RCV">
  <device classCode="DEV" determinerCode="INSTANCE">
    <id extension="GHH LAB" root="2.16.840.1.113883.19.1122.1"/>
    <asLocatedEntity classCode="LOCE">
      <location classCode="PLC" determinerCode="INSTANCE">
        <id root="2.16.840.1.113883.19.1122.2" extension="ELAB-3"/>
      </location>
    </asLocatedEntity>
  </device>
</receiver>
<sender typeCode="SND">
  <device classCode="DEV" determinerCode="INSTANCE">
    <id root="2.16.840.1.113883.19.1122.1" extension="GHH OE"/>
    <asLocatedEntity classCode="LOCE">
      <location classCode="PLC" determinerCode="INSTANCE">
        <id root="2.16.840.1.113883.19.1122.2" extension="BLDG24"/>
      </location>
    </asLocatedEntity>
  </device>
</sender>
<!-- Trigger Event Control Act & Domain Content -- >
</POLB_IN224200>
```

**Fig. A.1** HL7v3 Transmission Wrapper

The Trigger Event Control Act Wrapper is another wrapper around the actual message. Its structure (visible in Fig. A.2), includes the data relative to the trigger event, in this case the event POLB\_TE224200 and information about the date and time when the trigger event occurred. Although the responsible parties for the trigger event are not present in this example, they could also be

conveyed as part of the wrapper. For a clearer visualization, the domain content part of this wrapper, where all the explicit information relative to the observation itself is contained, has been approached separately.

```
<controlActProcess classCode="CACT" moodCode="EVN">
  <code code="POLB_TE224200" codeSystem="2.16.840.1.113883.1.18"/>
  <subject typeCode="SUBJ" contextConductionInd="false">

    <!-- domain content has been removed
         see next section of this whitepaper-->

  </subject>
</controlActProcess>
```

**Fig. A.2** HL7v3 Trigger Event Control Act Wrapper

The domain context (labeled as Observation Event) contains de directly-related data about the measurement, including the observation itself, the author who has carried out the measurement, the patient and, as in HL7v2, the field containing the data about the original observation order.

The Observation Event section contains the data about the measurement (see Fig. A.3), including main data as the date and time when it was carried out and the result itself, but also other complementary fields such as a status code, indicating the current state of the observation (in this case, it's already completed), and a reference range, which indicates the low and high values between which the result of the measurement should be placed to consider the patient's health state within a normalcy scenario. As it will be seen in the next section, these latter have a great similarity with the FHIR implementation, leading to a very similar structure to the one that this project has used to store the generated observation results coming from the user's medical devices.

```
<observationEvent>
  <id root="2.16.840.1.113883.19.1122.4" extension="1045813"
    assigningAuthorityName="GHH LAB Filler Orders"/>
  <code code="1554-5" codeSystemName="LN"
    codeSystem="2.16.840.1.113883.6.1"
    displayName="GLUCOSE^POST 12H CFST:MCNC:PT:SER/PLAS:QN"/>
  <statusCode code="completed"/>
  <effectiveTime value="200202150730"/>
  <priorityCode code="R"/>
  <confidentialityCode code="N"
    codeSystem="2.16.840.1.113883.5.25"/>
  <value xsi:type="PQ" value="182" unit="mg/dL"/>
  <interpretationCode code="H"/>
  <referenceRange>
    <interpretationRange>
      <value xsi:type="IVL_PQ">
        <low value="70" unit="mg/dL"/>
        <high value="105" unit="mg/dL"/>
      </value>
      <interpretationCode code="N"/>
    </interpretationRange>
  </referenceRange>
```

### Fig. A.3 Observation Event (I). Measurement results and parameters

In the Author section (see Fig. A.4), shows two levels of information regarding the performing provider (Mr. Harold H Hippocrates). The first level is the practitioner level, where his ID is present, and the personal level, where it's full name and qualifier are shown.

This structural bisection is also carried out on the patient's (referenced as record target) side (see Fig. A.5). Note that this is just a section of an observation report of a measurement, and thus, the information as for the patient is very limited, just enough to provide some form of error-checking. Other specific and detailed information about the patient could be found in another type of HL7v3 message (a demographic message for updating a patient's personal information, for instance).

```

<author>
  <time value="200202150730"/>
  <modeCode code="WRITTEN"/>
  <signatureCode code="S"/>
  <assignedEntity>
    <id root="2.16.840.1.113883.19.1122.3" extension="444-444-4444"/>
    <assignedPerson>
      <name>
        <given>Harold</given>
        <given>H</given>
        <family>Hippocrates</family>
        <suffix qualifier="AC">MD</suffix>
      </name>
    </assignedPerson>
  </assignedEntity>
</author>

```

### Fig. A.4 Observation Event (II). Author section

```

<recordTarget>
  <patientClinical>
    <id root="2.16.840.1.113883.19.1122.5" extension="444-22-2222"
      assigningAuthorityName="GHH Lab Patient IDs"/>
    <statusCode code="active"/>
    <patientPerson>
      <name use="L">
        <given>Eve</given>
        <given>E</given>
        <family>Everywoman</family>
      </name>
      <asOtherIDs>
        <id extension="AC555444444" assigningAuthorityName="SSN"
          root="2.16.840.1.113883.4.1"/>
      </asOtherIDs>
    </patientPerson>
  </patientClinical>
</recordTarget>

```

### Fig. A.5 Observation Event (III). Record target section

The last section of the Observation Result section contains the reference to the original observation order, identifies by a placer number, which is used by the receiver to match the results to the order (see Fig. A.6).

```
<inFulfillmentOf>
  <placerOrder>
    <id root="2.16.840.1.113883.19.1122.14" extension="845439"
      assigningAuthorityName="GHH OE Placer orders"/>
  </placerOrder>
</inFulfillmentOf>
</observationEvent>
```

**Fig. A.6** Observation Event (IV). Original order information.

## ANNEX B. CODE

The code that has been developed throughout this project can be found in the following GitHub repositories.

The following link corresponds to the frontend's code, the application itself:  
<https://github.com/eetac/medicaldevices>

And, as for the backend's, the NodeJS server, the code can be found in:  
<https://github.com/eetac/medicalDevices-backend>

And regarding the Docker Image of the HAPI FHIR server, it is public in Docker Hub, accessible through the following link:  
<https://hub.docker.com/r/hapiproject/hapi>

## BIBLIOGRAPHY

- [1] 'HL7 Standards - Section 2: Clinical and Administrative Domains', *HL7.org*. [Online]. Available: [https://www.hl7.org/implement/standards/product\\_section.cfm?section=20&ref=nav](https://www.hl7.org/implement/standards/product_section.cfm?section=20&ref=nav). [Accessed: 08-Feb-2023].
- [2] 'HL7 Version 2 Product Suite', *HL7.org*. [Online]. Available: [https://www.hl7.org/implement/standards/product\\_brief.cfm?product\\_id=185](https://www.hl7.org/implement/standards/product_brief.cfm?product_id=185). [Accessed: 08-Feb-2023].
- [3] *HL7spain.org*. [Online]. Available: [http://www.hl7spain.org/wp-content/uploads/2012/08/semHL7\\_presentacionV3.pdf](http://www.hl7spain.org/wp-content/uploads/2012/08/semHL7_presentacionV3.pdf). [Accessed: 08-Feb-2023].
- [4] E. C. M. Mario, 'HL7 en Español', *Blogspot.com*. [Online]. Available: <http://hl7es.blogspot.com/2013/12/el-ocaso-de-hl7-v3.html>. [Accessed: 08-Feb-2023].
- [5] 'Documentation - FHIR v4.3.0', *HL7.org*. [Online]. Available: <https://www.hl7.org/FHIR/documentation.html>. [Accessed: 08-Feb-2023].
- [6] Amanda, 'Intercambio de datos clínicos con HL7 FHIR e integración con Red Hat Fuse (Apache Camel)', *Chakray*, 01-Sep-2021. [Online]. Available: <https://www.chakray.com/es/utilizar-hl7-fhir-intercambio-informacion-clinica-integracion-apache-camel/>. [Accessed: 08-Feb-2023].
- [7] 'IBM Documentation', *ibm.com*, 13-Apr-2021. [Online]. Available: <https://www.ibm.com/docs/es/app-connect/11.0.0?topic=healthcare-hl7-fhir-pattern>. [Accessed: 08-Feb-2023].
- [8] 'HAPI FHIR - the open source FHIR API for java', *Hapifhir.io*. [Online]. Available: <https://hapifhir.io/>. [Accessed: 08-Feb-2023].
- [9] 'Introduction - HAPI FHIR documentation', *Hapifhir.io*. [Online]. Available: [https://hapifhir.io/hapi-fhir/docs/getting\\_started/introduction.html](https://hapifhir.io/hapi-fhir/docs/getting_started/introduction.html). [Accessed: 08-Feb-2023].
- [10] 'Home -', *LOINC*. [Online]. Available: <https://loinc.org/>. [Accessed: 08-Feb-2023].
- [11] '¿Qué es SNOMED CT?', *Gob.es*. [Online]. Available: <https://www.sanidad.gob.es/profesionales/hcdsns/areaRecursosSem/snomed-ct/quees.htm>. [Accessed: 08-Feb-2023].
- [12] 'Patient - FHIR v4.3.0', *HL7.org*. [Online]. Available: <https://hl7.org/fhir/patient.html>. [Accessed: 08-Feb-2023].

- [13] 'Observation - FHIR v4.3.0', *HL7.org*. [Online]. Available: <https://hl7.org/fhir/observation.html>. [Accessed: 08-Feb-2023].
- [14] 'Device - FHIR v4.3.0', *HL7.org*. [Online]. Available: <https://hl7.org/fhir/device.html>. [Accessed: 08-Feb-2023].
- [15] S. Mangione, 'Vital Signs', in *Physical Diagnosis Secrets*, Elsevier, 2008, pp. 34–62.
- [16] A. Sapra, A. Malik, and P. Bhandari, *Vital Sign Assessment*. StatPearls Publishing, 2022.
- [17] J. Ratliff, 'Docker: Accelerated, containerized application development', *Docker*, 10-May-2022. [Online]. Available: <https://www.docker.com/>. [Accessed: 08-Feb-2023].
- [18] D. Moliner, 'LifeVit devices', *LifeVit*. [Online]. Available: <https://lifevit.es/en/dispositivos/>. [Accessed: 08-Feb-2023].
- [19] 'Node.js', *Node.js*. [Online]. Available: <https://nodejs.org/en/>. [Accessed: 08-Feb-2023].
- [20] 'Express - Node.js web application framework', *Expressjs.com*. [Online]. Available: <https://expressjs.com/>. [Accessed: 08-Feb-2023].
- [21] 'Mongoose', *Mongoosejs.com*. [Online]. Available: <https://mongoosejs.com/>. [Accessed: 08-Feb-2023].
- [22] 'MongoDB: The developer data platform', *MongoDB*. [Online]. Available: <https://www.mongodb.com/>. [Accessed: 08-Feb-2023].
- [23] 'Relational vs. Non-relational databases', *MongoDB*. [Online]. Available: <https://www.mongodb.com/compare/relational-vs-non-relational-databases>. [Accessed: 08-Feb-2023].
- [24] 'Build apps for any screen', *Flutter.dev*. [Online]. Available: <https://flutter.dev/>. [Accessed: 08-Feb-2023].
- [25] J. Schmitt, 'Native vs cross-platform mobile app development', *CircleCI*, 24-Aug-2022. [Online]. Available: <https://circleci.com/blog/native-vs-cross-platform-mobile-dev/>. [Accessed: 08-Feb-2023].
- [26] R. Barbosa, 'Flutter Internationalization the Easy Way — using Provider and JSON', *Medium*, 15-Jul-2020. [Online]. Available: <https://medium.com/@rafavinncce/flutter-internationalization-the-easy-way-using-provider-and-json-986eb5d76822>. [Accessed: 08-Feb-2023].



- [27] 'Bloc state management library', *Bloclibrary.dev*. [Online]. Available: <https://bloclibrary.dev/#/>. [Accessed: 08-Feb-2023].
- [28] 'JSON web tokens - jwt.io', *Jwt.io*. [Online]. Available: <https://jwt.io/>. [Accessed: 08-Feb-2023].
- [29] Institute of Medicine (US) Committee on Regional Health Data Networks, M. S. Donaldson, and K. N. Lohr, *Confidentiality and privacy of personal data*. Washington, D.C., DC, USA: National Academies Press, 1994.
- [30] J. E. Peña, M. Letrado, J. Servicio, and J. D. Icom, 'LA LEY ORGANICA DE PROTECCION DE DATOS Y LAS CONSULTAS MEDICAS', *Fesemi.org*. [Online]. Available: <https://www.fesemi.org/sites/default/files/documentos/ponencias/xxxii-congreso-semi/23-%20Pena%20Martin.pdf>. [Accessed: 08-Feb-2023].