Diogo Jordão Filipe

REAL-TIME EVENTS DASHBOARD FOR HOSPITALS

1 2 9 0 UNIVERSIDADE Đ COIMBRA

1 2 9 0

UNIVERSIDADE Đ
COIMBRA

Diogo Jordão Filipe

REAL-TIME EVENTS DASHBOARD FOR HOSPITALS

Dissertation in the context of the Master in Informatics Engineering, specialization in Software Engineering, advised by Professor Tiago Filipe dos Santos Martins and presented to the Department of Informatics Engineering of the Faculty of Sciences and Technology of the University of Coimbra.

July 2023

1 2 9 0

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE Đ
COIMBRA

DEPARTMENT OF INFORMATICS ENGINEERING

Diogo Jordão Filipe

Real-time events dashboard for hospitals

Dissertation in the context of the Master in Informatics Engineering, specialization in Software Engineering, advised by Professor Tiago Filipe dos Santos Martins and presented to the Department of Informatics Engineering of the Faculty of Sciences and Technology of the University of Coimbra.

July 2023

Acknowledgements

I would like to start by thanking both of my dissertation advisors, Tiago Martins from DEI and Fernando Tinoco from MedicineOne, who without fail provided me with guidance and were always available to help throughout both semesters. The work described in this report would not be possible without them.

To all the people I interacted with at MedicineOne who were more than welcoming and eager to support me with my work.

I also want to express my gratitude to all of my DEI colleagues and friends, especially Bruno Gandres, Henrique Fonseca, José Reis, José Bugalho, José Gomes, Nuno Silva, and Pedro Marques, for all the enjoyable times and advice.

A special thank you to Ana Fernandes, my girlfriend, for her unwavering support and willingness to assist me during my dissertation. I appreciate you being there for me at all times.

Finally, I would like to thank my mother, Sandra Jordão, and sister, Dulce Filipe, for their loving support and encouragement.

Abstract

Due to the rise in digital data collection, data visualization tools like dashboards are becoming more and more important in a variety of industries to provide short-, medium-, and long-term decision-making support. Making these tools effective is challenging and essential for getting the most out of the vast amounts of data that are currently available, and this effectiveness is even more required in contexts of a critical nature, such as healthcare. The objective of this work is to implement a dashboard for the medical environment that can be customized for different institutions and hospital departments and is capable of consistently delivering real-time information. Key design principles and case studies were examined in published dashboard research, as well as technological alternatives, which were analyzed and compared to determine the best approach to implement the project. A web application built with the React library that communicates in real-time with the Azure SignalR Service was the chosen approach. The solution is integrated with an already existing system, from which it obtains the configuration and clinical information to display it in the designated fields with the designated appearance. This system belongs to a healthcare software company named MedicineOne, where this internship also took place. Testing was performed on the final product to ensure its reliability, as well as validation through acceptance testing, which deemed that the project met the requirements. Steps are now being taken internally within the company to conduct a pilot of the product with one of their partners and move towards its commercialization.

Keywords

Medical dashboard, Real-time information, Data visualization, Decision-making, User Interface (UI), Application Programming Interface (API).

Resumo

Devido ao crescimento da recolha de dados, ferramentas de visualização de dados como *dashboards* estão a tornar-se cada vez mais importantes numa variedade de indústrias para fornecer apoio à tomada de decisão a curto, médio, e longo prazo. Tornar estas ferramentas eficazes é desafiante e essencial para obter o máximo proveito das grandes quantidades de dados atualmente disponíveis, e esta eficácia é ainda mais requerida em contextos de natureza crítica, como o fornecimento de cuidados de saúde. O objetivo deste trabalho é implementar uma dashboard para o ambiente médico que possa ser configurada para diferentes instituições e departamentos hospitalares e que seja capaz de fornecer consistentemente informação em tempo real. Princípios de *design* chave e casos de estudo de publicações de investigação de *dashboards* foram examinados, bem como alternativas tecnológicas, que foram analisadas e comparadas para determinar a melhor abordagem possível para a implementação do projeto. Uma aplicação *web* construída com a biblioteca *React* que comunica em tempo real com o *Azure SignalR Service* foi a abordagem escolhida. A solução foi integrada com um sistema já existente por onde obtém a configuração e a informação clínica para a exibir com os campos designados com a aparência designada. Este sistema pertence a uma empresa de *software* clínico chamada *MedicineOne*, onde este estágio também decorreu. Foram realizados testes ao produto final para assegurar a sua confiabilidade, bem como validação através de testes de aceitação, que determinaram que o projeto foi ao encontro dos requisitos. Estão a ser tomados passos internamente na empresa para realizar um piloto do produto com um dos seus parceiros e mover-se em direção à sua comercialização.

Palavras-Chave

Dashboard médica, Informação em tempo real, Visualização de dados, Tomada de decisão, Interface de utilizador, Interface de programação de aplicações

Contents

Acronyms

API Application Programming Interface.

BCL Base Class Library.

CQRS Command Query Responsibility Segregation.

CSS Cascading Style Sheets.

DDD Domain-Driven Design.

DOM Document Object Model.

EHR Electronic Health Record.

FCM Firebase Cloud Messaging.

FIFO First In First Out.

GQM Goal-Question-Metric.

HTML HyperText Markup Language.

HTTP HyperText Transfer Protocol.

IDE Integrated Development Environment.

JSX JavaScript XML.

JWT JSON Web Token.

KPI Key Performance Indicator.

MAUI Multi-platform Application User Interface.

npm Node Package Manager.

ORM Object-Relational Mapping.

REST Representational State Transfer.

RGB Red-Blue-Green.

RMI Remote Method Invocation.

SDK Software Development Kit.

SLA Service Level Agreement.

SNS Simple Notification Service.

SSE Server-Sent Events.

SVG Scalable Vector Graphics.

UI User Interface.

WPF Windows Presentation Foundation.

XAML Extensible Application Markup Language.

List of Figures

List of Tables

Chapter 1

Introduction

With the aim of supporting decision-making and/or business performance, dashboards are a popular type of data visualization that is used in a variety of industries [4, 58]. The clinical and medical environment is one of these contexts where dashboard usability has a strong foundation [24, 33]. This is a result of the growing integration of Electronic Health Records (EHRs), which gives medical personnel and hospital administration access to all available medical information via a computer [35]. However, if it is not properly condensed in order to provide users with the precise information they require to make effective decisions, it is essentially useless [12, 51, 57]. Due to their ability to visually summarize complex information and enable comparison with predefined metrics and goals, dashboards are used to address this problem. Real-time monitoring, data interaction, and other features can be useful in the healthcare setting as well and are made possible by dashboards [58].

1.1 Framing

This dissertation is based on an internship at MedicineOne, Life Sciences Computing, S.A., a company from Portugal that focuses on the development of medical software.

According to feedback provided to MedicineOne by hospitals using their products, there is a need to develop a dashboard that is entirely customizable in terms of the data it presents and the form it takes. They need a tool for visual information awareness, but it must be flexible enough to meet the requirements of the various medical departments, as they have already encountered dashboards that were unreasonably rigid in the information they displayed.

1.2 Objectives

The objective of this dissertation is to implement a prototype dashboard application that can be used in different types of medical units, namely: inpatient unit, operating block, emergency room, pharmacy, treatment room, oncology day hospital, and hemodialysis day hospital. These are the types that the MedicineOne system supports and the only distinction between them is where the medical data is stored in the database.

As stated, the main objective is to give users full control by letting them decide what information will be displayed as well as the dashboard's general appearance. Additionally, by receiving real-time updates on the configured information fields, it is also intended to act as an event notification hub for physicians, nurses, and other medical professionals.

Besides the development of the application, the key factor to consider is the graphical and functional configuration of the dashboard, which will be stored in the hospital's data system and accessed through an API belonging to MedicineOne. For this dashboard application to obtain both the configuration information and the clinical data itself, endpoints must be implemented in said API. The systems manager of each organization will create the configurations in the already installed MedicineOne system that will change the dashboard's UI upon startup, setting the data accessible to medical professionals and the dashboard's visual characteristics. Real-time data updating on the dashboard is another essential feature that should be implemented. To do this, a communication channel from the MedicineOne system to the application must be set up to allow for event notification, such as admissions and discharges of patients.

Although it is not mandatory, it is intended for the dashboard to be displayed on a large screen that is located in each medical team's workspace.

1.3 Document structure

This document is divided into the following chapters: Introduction (Chapter 1), State-of-the-art (Chapter 2), Technologies analysis (Chapter 3), System specification (Chapter 4), Methodology & Planning (Chapter 5), Implementation (Chapter 6), Testing (Chapter 7), and Conclusion (Chapter 8).

In Chapter 1 a brief introduction to the subject of medical dashboards is made, followed by the framing and objectives of the project, and finally the structure of the document is described.

Chapter 2 explains the fundamental ideas behind dashboards, what factors should be taken into account when designing one, and offers examples and an analysis of dashboards used in medical settings.

Chapter 3 gives descriptions and comparisons of alternative technological solutions that can be used to develop the functionality and UI of the dashboard and put in place a system for event notification.

In Chapter 4, a more detailed overview of the intended system is provided through the definition and prioritization of requirements, the description of architecture topics, and the listing of the UI design specifications.

The work methodology used and the planning for both semesters are both covered in Chapter 5.

Chapter 6, describes what was implemented, divided by the three main components of the system: the API endpoints, the dashboard application, and the real-time communication.

In Chapter 7 the test plan for the application and the acceptance tests that were used to validate the implementation outcome are relayed.

Finally, a conclusion will be included in Chapter 8.

Chapter 2

State-of-the-art

The term "dashboard" is frequently used to refer to a variety of applications with a range of functions and aesthetics, making it challenging to define precisely. However, one aspect of a dashboard is constant: the visual representation of (useful) data [4, 58]. This fundamental feature makes it possible for a dashboard to correspond with the objectives of the intended user to improve performance and decision-making [14, 29].

2.1 Dashboard design

There are crucial elements to consider when designing a dashboard that can determine whether or not it will be useful. These aspects are the data to be conveyed and the form of visualization [14, 29]. The user's needs must be fully comprehended in order to apply accurate visual representations to useful data, so stakeholders should be involved in the design process.

When designing a dashboard, questions like the ones below can help concentrate on the important details [14]:

- What data does the user require?
- What context is crucial for communicating the chosen data?
- Which type of visualization would be more effective at conveying the desired information?

2.1.1 Choosing information

A common mistake regarding the displayed information is trying to fit as much as possible into a view, which can overwhelm the user. It is best to first determine the dashboard's function in order to more precisely define the problem's scope and, as a result, determine what data are actually required [29].

Applying a measurement model, such as a Goal-Question-Metric (GQM), will help determine which data should be displayed on the dashboard more effectively. As suggested by the name, this model entails selecting the goals, questions, and metrics, each of which corresponds to a level of the model. These models follow the structure below [29]:

- The conceptual level (goal) outlines the subject matter to be studied and explains why it is important;
- The operational level (question) establishes the study's focus, in particular its pertinent components and the characteristics of those components that define the achievement (or not) of the goal;
- Finally, the quantitative level (metric) identifies the information required to provide responses to the goal's questions.

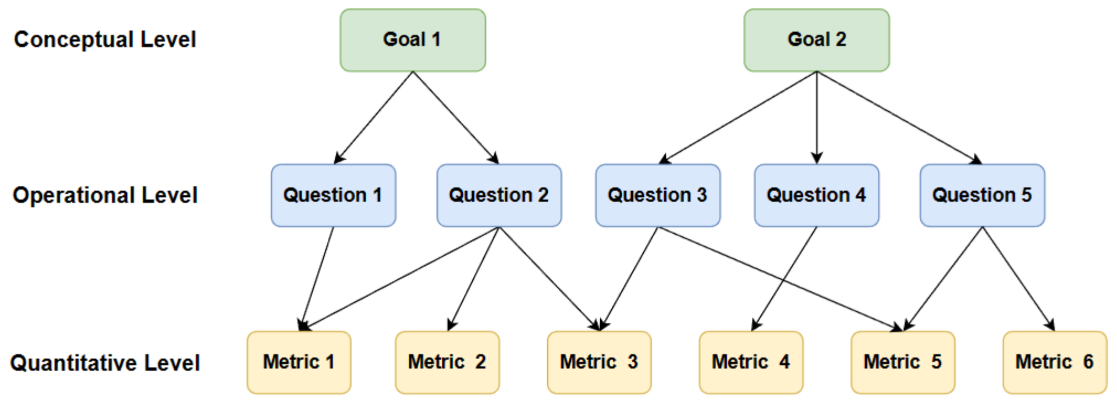


Figure 2.1: A GQM model.

As can be seen in Figure 2.1, a GQM model establishes a hierarchy with these levels that specify what will be measured and how this data should be interpreted [29]. The most important metrics in a context—those that convey performance, priority, and value in that scope—must be chosen from those that have been defined and understood. These metrics are known as Key Performance Indicators (KPIs). Any type of dashboard that intends to gauge performance and track its development should always include a definition of these metrics [58].

2.1.2 Choosing visualization

There is no right or wrong way to go about selecting visual elements for a dashboard; it all depends on the goal of the application and the setting in which it will be used [58]. As has already been mentioned, the best blueprint for creating a "good" dashboard is a list of specific user requirements. They most often bring knowledge and experience from their professional and business contexts, which is crucial [14].

A good approach to designing a dashboard, which also applies to other types of applications, is through mock-ups. They assist stakeholders in quickly conceptualizing a solution rather than creating an entire design from scratch right away, reducing the risk of delivering a suboptimal product. Since a mock-up can be a simple drawing on a white board, this idea can be effectively used in meetings with stakeholders. Any visual element can be redefined quickly and easily. By using mock-ups, the dashboard design can be developed iteratively. Since they are easier to change than actual designs, multiple iterations can be created until the best one is found [14].

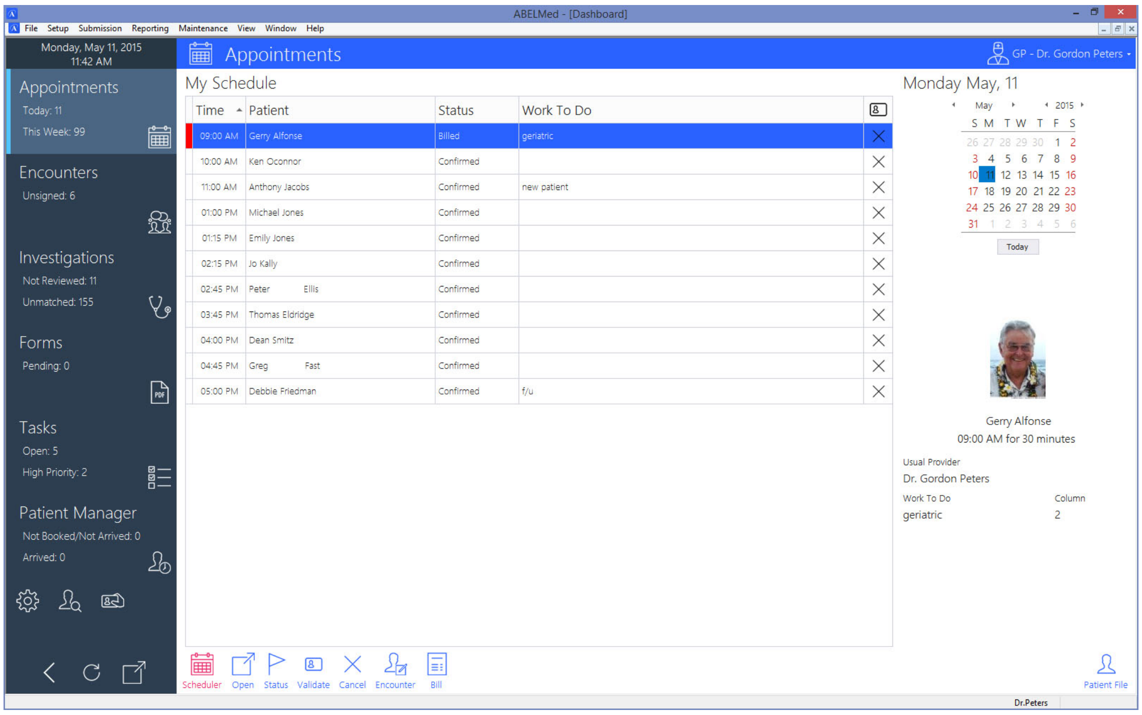
When selecting the visual elements for a dashboard, usage is another factor to consider. Should the user be "pushed" important information by the dashboard? Or is the user supposed to "pull" what they need? The dashboard in the first scenario needs to grab the user's attention and point it towards important data, whereas in the latter scenario, it needs to allow the user to freely filter and drill down on information. To "push" information effectively, it should be done in a way that makes it effortless for the user; this can be accomplished through consistent placement of information and clear visual representation to shorten the time needed to assimilate, as well as some form of highlighting or notification on the most crucial information. In order to provide efficient "pull" functionality, the dashboard must allow for the choice of detail and guide the user through the information, for example, by explaining to the user what the data represents and why it is important [29].

2.2 Case studies

Because most commercial solutions are fully integrated with an Electronic Health Records (EHRs) management tool, which clouds the dashboard component, and because there are few detailed descriptions available, gathering precise information about commercial medical dashboards has been challenging. As a result, this section will include a brief summary of some commercial solutions discovered as well as a more thorough analysis of research articles on medical dashboard implementation based on the design space described in [58], and the analysis done in [15].

2.2.1 Commercial solutions

A dashboard owned by the company ABELMed called the "Physician Dashboard" integrates with their other functions with the objective of improving the efficiency of medical professionals' activities. As can be seen in Figures 2.2, 2.3, 2.4, and 2.5, a user can access all of the other functions on the sidebar while maintaining constant access to the patient's file. For a chosen date, the first screen shows patient appointments and their status, and each appointment can be selected to view detailed information (Figure 2.2). One can view laboratory results (investigations) on the second screen, add remarks or notes to the report, and assign tasks associated with that investigation (Figure 2.3). The third screen monitors the status of each patient and provides information such as arrival time, wait time, and progression through the visit (Figure 2.4). Finally, in the last screen, medical and administrative tasks can be consulted and assigned (Figure 2.5) [1].


ABELMed - [Dashboard]
File Setup Submission Reporting Maintenance View Window Help
Appointments
GP - Dr. Gordon Peters
Scheduler Open Status Validate Cancel Encounter Bill
Patient File
Dr.Peters

Monday, May 11, 2015
11:42 AM

Appointments
Today: 11
This Week: 99

Encounters
Unsigned: 6

Investigations
Not Reviewed: 11
Unmatched: 155

Forms
Pending: 0

Tasks
Open: 5
High Priority: 2

Patient Manager
Not Booked/Not Arrived: 0
Arrived: 0

My Schedule

Time	Patient	Status	Work To Do
09:00 AM	Gerry Alfonse	Billed	geriatric
10:00 AM	Ken Oconnor	Confirmed	
11:00 AM	Anthony Jacobs	Confirmed	new patient
01:00 PM	Michael Jones	Confirmed	
01:15 PM	Emily Jones	Confirmed	
02:15 PM	Jo Kally	Confirmed	
02:45 PM	Peter Ellis	Confirmed	
03:45 PM	Thomas Eldridge	Confirmed	
04:00 PM	Dean Smitz	Confirmed	
04:45 PM	Greg Fast	Confirmed	
05:00 PM	Debbie Friedman	Confirmed	f/u

Monday May, 11

May 2015

S	M	T	W	T	F	S
26	27	28	29	30	1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30
31	1	2	3	4	5	6

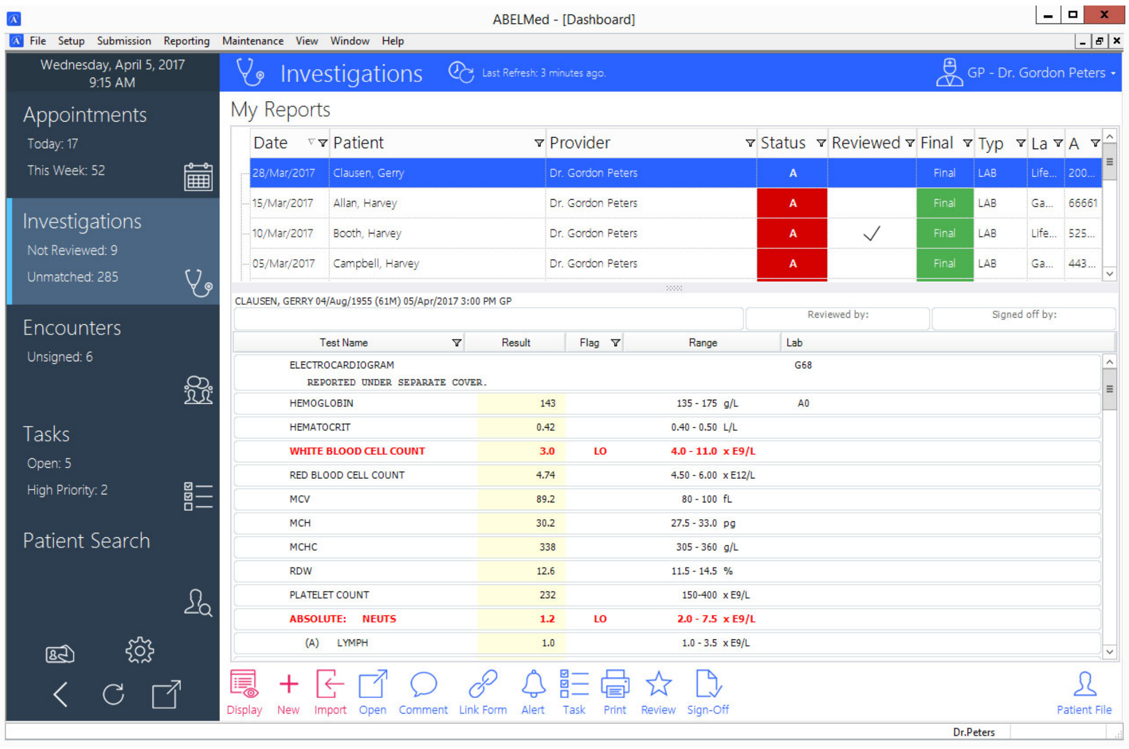
Today

Gerry Alfonse
09:00 AM for 30 minutes

Usual Provider
Dr. Gordon Peters

Work To Do: geriatric
Column: 2

Figure 2.2: ABELMed's Physician Dashboard Appointments Screen [1].


ABELMed - [Dashboard]
File Setup Submission Reporting Maintenance View Window Help
Investigations
Last Refresh: 3 minutes ago.
GP - Dr. Gordon Peters
Display New Import Open Comment Link Form Alert Task Print Review Sign-Off
Patient File
Dr.Peters

Wednesday, April 5, 2017
9:15 AM

Appointments
Today: 17
This Week: 52

Investigations
Not Reviewed: 9
Unmatched: 285

Encounters
Unsigned: 6

Tasks
Open: 5
High Priority: 2

Patient Search

My Reports

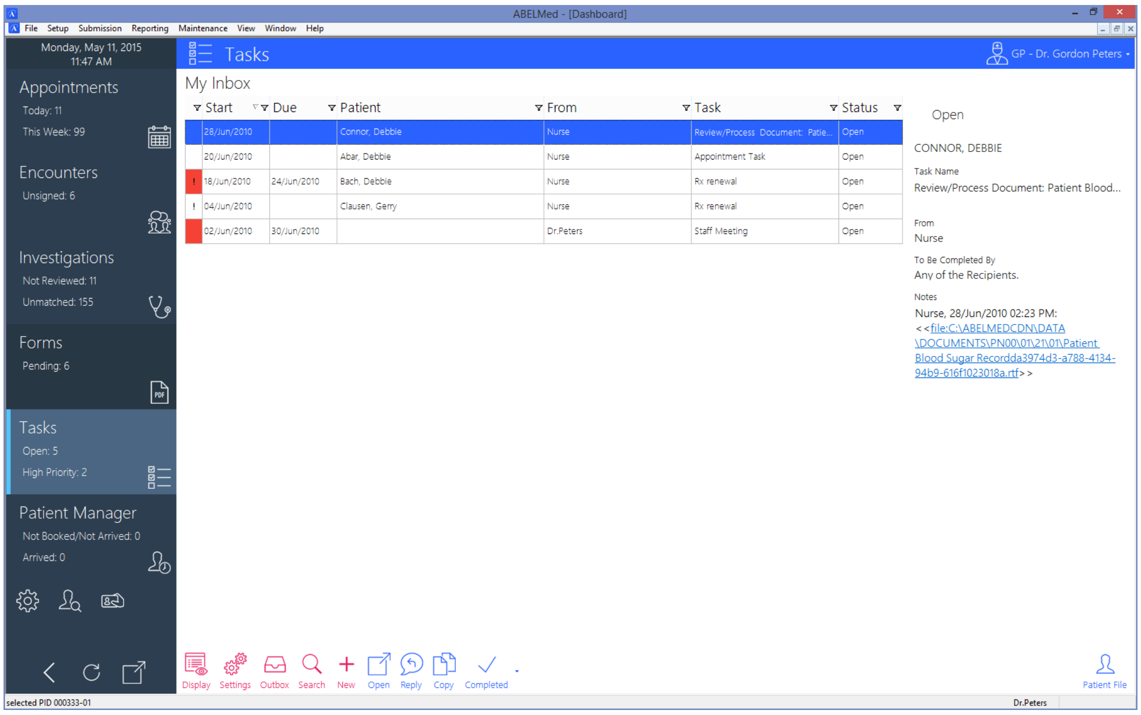
Date	Patient	Provider	Status	Reviewed	Final	Typ	La	A
28/Mar/2017	Clausen, Gerry	Dr. Gordon Peters	A		Final	LAB	Life...	200...
15/Mar/2017	Allan, Harvey	Dr. Gordon Peters	A		Final	LAB	Ga...	66661
10/Mar/2017	Booth, Harvey	Dr. Gordon Peters	A	✓	Final	LAB	Life...	525...
05/Mar/2017	Campbell, Harvey	Dr. Gordon Peters	A		Final	LAB	Ga...	443...

CLAUSEN, GERRY 04/Aug/1955 (61M) 05/Apr/2017 3:00 PM GP

Reviewed by:
Signed off by:

Test Name	Result	Flag	Range	Lab
ELECTROCARDIOGRAM REPORTED UNDER SEPARATE COVER.				G68
HEMOGLOBIN	143		135 - 175 g/L	A0
HEMATOCRIT	0.42		0.40 - 0.50 L/L	
WHITE BLOOD CELL COUNT	**3.0**	**LO**	**4.0 - 11.0 x E9/L**	
RED BLOOD CELL COUNT	4.74		4.50 - 6.00 x E12/L	
MCV	89.2		80 - 100 fL	
MCH	30.2		27.5 - 33.0 pg	
MCHC	338		305 - 360 g/L	
RDW	12.6		11.5 - 14.5 %	
PLATELET COUNT	232		150-400 x E9/L	
ABSOLUTE: NEUTS	**1.2**	**LO**	**2.0 - 7.5 x E9/L**	
(A) LYMPH	1.0		1.0 - 3.5 x E9/L	

Figure 2.3: ABELMed's Physician Dashboard Investigations Screen [1].


ABELMed - [Dashboard]
File Setup Submission Reporting Maintenance View Window Help
Monday, May 11, 2015 11:47 AM
Appointments Today: 11 This Week: 99
Encounters Unsigned: 6
Investigations Not Reviewed: 11 Unmatched: 155
Forms Pending: 6
Tasks Open: 5 High Priority: 2
Patient Manager Not Booked/Not Arrived: 0 Arrived: 0
Tasks
GP - Dr. Gordon Peters
Display Settings Outbox Search New Open Reply Copy Completed
Patient File
selected PID 000333-01
Dr.Peters

My Inbox

	Start	Due	Patient	From	Task	Status
	28/Jun/2010		Connor, Debbie	Nurse	Review/Process Document: Patie...	Open
	20/Jun/2010		Abar, Debbie	Nurse	Appointment Task	Open
!	18/Jun/2010	24/Jun/2010	Bach, Debbie	Nurse	Rx renewal	Open
!	04/Jun/2010		Clausen, Gerry	Nurse	Rx renewal	Open
🟥	02/Jun/2010	30/Jun/2010		Dr.Peters	Staff Meeting	Open

Open

CONNOR, DEBBIE

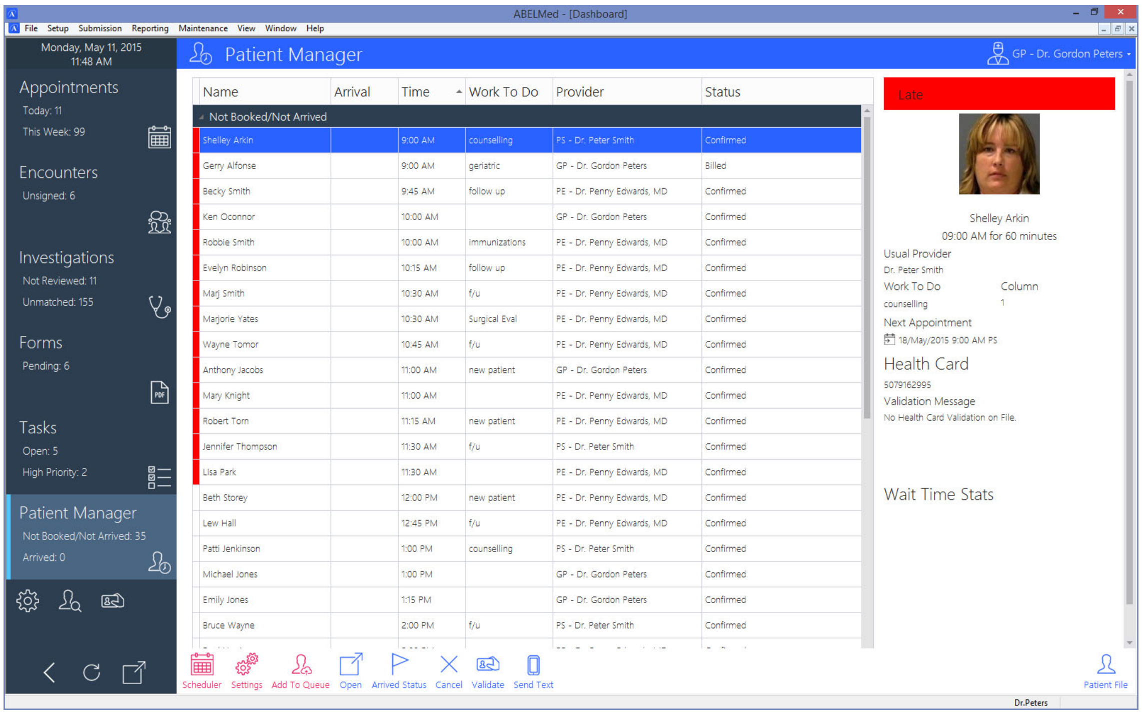
Task Name
Review/Process Document: Patient Blood...

From
Nurse

To Be Completed By
Any of the Recipients.

Notes
Nurse, 28/Jun/2010 02:23 PM:
<<file:C:\ABELMEDCDN\DATA\DOCUMENTS\PN00\01\21\01\Patient Blood Sugar Recordda3974d3-a788-4134-94b9-616f1023018a.rtf>>

Figure 2.4: ABELMed's Physician Dashboard Patient Manager Screen [1].


ABELMed - [Dashboard]
File Setup Submission Reporting Maintenance View Window Help
Monday, May 11, 2015 11:48 AM
Appointments Today: 11 This Week: 99
Encounters Unsigned: 6
Investigations Not Reviewed: 11 Unmatched: 155
Forms Pending: 6
Tasks Open: 5 High Priority: 2
Patient Manager Not Booked/Not Arrived: 35 Arrived: 0
Patient Manager
GP - Dr. Gordon Peters
Scheduler Settings Add To Queue Open Arrived Status Cancel Validate Send Text
Patient File
Dr.Peters

Name	Arrival	Time	Work To Do	Provider	Status
Not Booked/Not Arrived					
Shelley Arkin		9:00 AM	counselling	PS - Dr. Peter Smith	Confirmed
Gerry Alfonse		9:00 AM	geriatric	GP - Dr. Gordon Peters	Billed
Becky Smith		9:45 AM	follow up	PE - Dr. Penny Edwards, MD	Confirmed
Ken Oconnor		10:00 AM		GP - Dr. Gordon Peters	Confirmed
Robbie Smith		10:00 AM	immunizations	PE - Dr. Penny Edwards, MD	Confirmed
Evelyn Robinson		10:15 AM	follow up	PE - Dr. Penny Edwards, MD	Confirmed
Marj Smith		10:30 AM	f/u	PE - Dr. Penny Edwards, MD	Confirmed
Marjorie Yates		10:30 AM	Surgical Eval	PE - Dr. Penny Edwards, MD	Confirmed
Wayne Tomor		10:45 AM	f/u	PE - Dr. Penny Edwards, MD	Confirmed
Anthony Jacobs		11:00 AM	new patient	GP - Dr. Gordon Peters	Confirmed
Mary Knight		11:00 AM		PE - Dr. Penny Edwards, MD	Confirmed
Robert Torn		11:15 AM	new patient	PE - Dr. Penny Edwards, MD	Confirmed
Jennifer Thompson		11:30 AM	f/u	PS - Dr. Peter Smith	Confirmed
Lisa Park		11:30 AM		PE - Dr. Penny Edwards, MD	Confirmed
Beth Storey		12:00 PM	new patient	PE - Dr. Penny Edwards, MD	Confirmed
Lew Hall		12:45 PM	f/u	PE - Dr. Penny Edwards, MD	Confirmed
Patti Jenkinson		1:00 PM	counselling	PS - Dr. Peter Smith	Confirmed
Michael Jones		1:00 PM		GP - Dr. Gordon Peters	Confirmed
Emily Jones		1:15 PM		GP - Dr. Gordon Peters	Confirmed
Bruce Wayne		2:00 PM	f/u	PS - Dr. Peter Smith	Confirmed

Late

Shelley Arkin
09:00 AM for 60 minutes

Usual Provider
Dr. Peter Smith

Work To Do	Column
counselling	1

Next Appointment
18/May/2015 9:00 AM PS

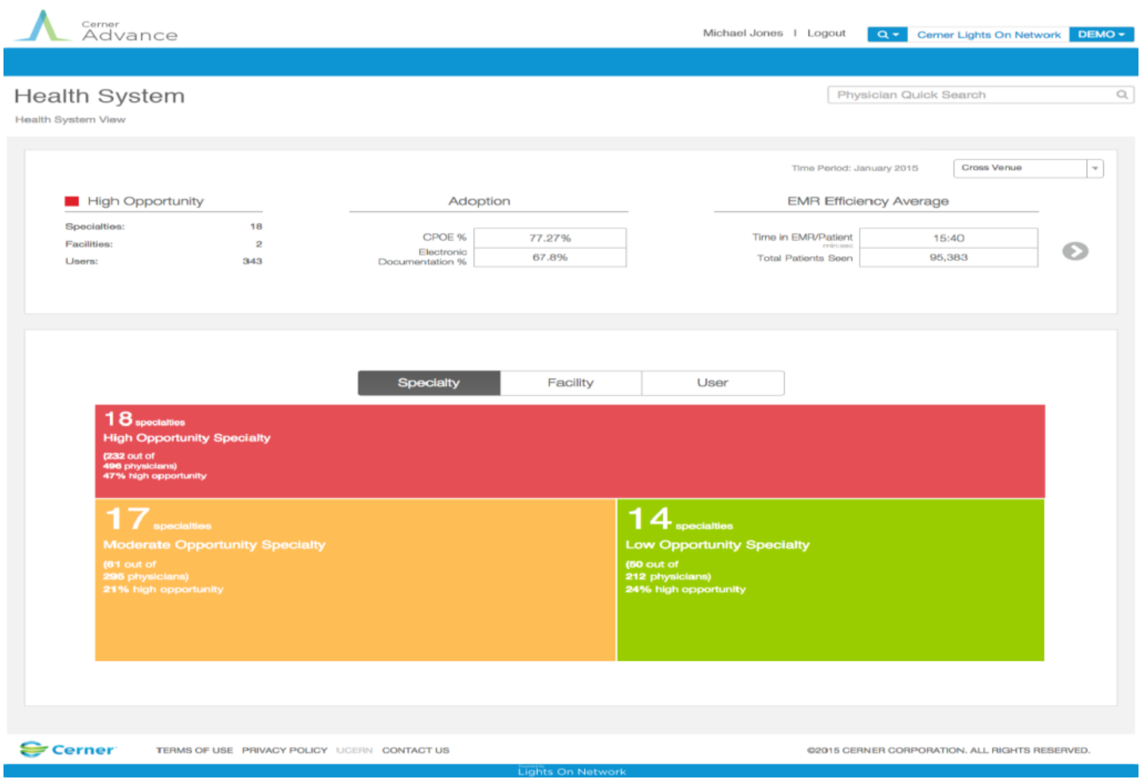
Health Card
5079162995

Validation Message
No Health Card Validation on File.

Wait Time Stats

Figure 2.5: ABELMed's Physician Dashboard Tasks Screen [1].

Cerner Advance is an aggregate of web-based tools that collect clinicians' usage data, such as time spent on Cerner's EHRs executing a certain activity (see Figure 2.6). It can present reports based on improvement opportunities it discovers and display departmental and individual information about overall productivity. Every user of a company is compared to other experts in the same field throughout a nation to determine how efficient they are. Cerner Advance maintains benchmarked KPIs to determine its return on investment. Clinical professionals can be coached on how to increase their productivity through the creation of "action plans." The overall goal of this solution is to improve efficiency and satisfaction among EHR system users [6, 17].



Health System

Health System View

Time Period: January 2015

Cross Venue

High Opportunity

Specialties:	18
Facilities:	2
Users:	343

Adoption

CPOE %	77.27%
Electronic Documentation %	67.8%

EMR Efficiency Average

Time in EMR/Patient min:sec	15:40
Total Patients Seen	95,383

18 specialties
High Opportunity Specialty
(232 out of
496 physicians)
47% high opportunity

17 specialties
Moderate Opportunity Specialty
(61 out of
295 physicians)
21% high opportunity

14 specialties
Low Opportunity Specialty
(50 out of
212 physicians)
24% high opportunity

©2015 CERNER CORPORATION. ALL RIGHTS RESERVED.

Figure 2.6: Cerner Advance dashboard [17]

Another dashboard named Cerner CareAware has a "command center" dashboard that offers real-time data to enhance awareness and decision-making for a variety of hospital professionals. It retrieves information about patient statuses and discharge options from the EHRs and allows the user to select and drill down on information to visualize detailed individual patient data. In one department, desktop monitors and large screens display the dashboard [18]. The overall solution also includes tools that relay patient flow within an unit and real-time localization of patients, staff, and equipment [16].

2.2.2 Research articles

Firstly, the medical dashboards implemented in the examined articles were classified by their **purpose**, but only on the basis of their **decision support**, which ultimately guides the direction of the visual components and data utilized in their design [58]. The aspect of communication and learning was omitted since most healthcare dashboards will focus on decision-making. Dashboards can be "operational" if they display current and near-past information that is intended to be used during day-to-day activities by low-level employees. A dashboard is referred to as "tactical" if it relays previous information with the intention of tracking performance; these dashboards are typically used by mid-level management to support decision-making and employ KPIs to quantify organizational progress. Finally, "strategic" dashboards contain a lot of data and a variety of metrics to help high-level management with organization-wide long-term planning. Dashboards can serve more than one of these functions [58]. Performance improvement, quality and safety, and management and operations are the three categories into which the purpose of the dashboards is broken down in [15]. Even though it takes a different approach, the first category mainly relates to the "tactical" kind, while the other categories are more in line with the "operational" goal.

When determining the intended **audience** for a dashboard, only two factors were considered: **visualization literacy** (low, medium, or high), which determines the difficulty of comprehending the information presented by the dashboard's visual components, and whether the user requires **advanced domain expertise**, which indicates if the dashboard only uses non-basic medical terminology [58]. The circulation factor in this category is disregarded because, in the context of medicine, the majority of dashboards will be directed to organizations, such as medical teams and departments, and rarely to the individual user [15, 58].

The following types of dashboard **features** from the supporting design space were considered: **interactivity**, meaning the user can interact with the dashboard's views and its data; **multi-page**, if the dashboard contains multiple pages to organize the various information, it displays; and **construction and composition**, as in the capability of the user changing the appearance or position of visual components [58]. Other elements in this category were absent from all of the case studies and weren't thought to be important to the research.

Finally, in relation to **data semantics**, it was determined whether the dashboards emitted **alerts and notifications** by either highlighting important events and anomalies or producing audible queues to raise awareness. Additionally, the presence of **benchmarks** was noted. These benchmarks would work in conjunction with underlying thresholds to provide users with visual cues about the status and trend of a particular monitored metric. The last consideration informed if the data displayed on the dashboard was able to be updated [58]; this was deemed insufficient as a defining factor, so it was substituted by the capability of **real-time updates** of data, which is much more pertinent in the clinical context. It was also considered the **classification of the clinical indicators** used: they were categorized as "structural" if they relayed more management-oriented information about departments and medical teams, such as bed availability and resource allocation; "process" oriented if they transmitted information about medical activities and other tasks; and "outcome" oriented for more analytical purposes, such as mortality rates and appointment statistics. Clinical indicators' specificity was not taken into account because it only indicates whether the context is a specific medical condition or a more general subject [15].

The explained categories and properties above were used in Table 2.2 to classify the analyzed case studies. These case studies are listed in Table 2.1.

As can be seen from the dashboard classification in Table 2.2, "strategic" was the least prevalent type of dashboard and was only identified once, whereas the "operational" purpose was present in all other case studies. About half of the dashboards under analysis were labeled "tactical", but only when used in conjunction with the other two. The importance of real-time awareness and quick decision-making for medical professionals engaged in routine patient care is highlighted by the fact that "operational" decision support was present in all but one case study. The frequency of "tactical" decision support is also significant because improving overall performance will ultimately lead to better hospital care. This importance is further highlighted by the fact that the majority of dashboards support real-time updates of the displayed data. Although this functionality was incredibly rare in this collection of case studies, the use of notifications and alerts is also related to the usefulness of clinical awareness because it can quickly draw the attention of medical personnel to important events.

Because they primarily used line and bar charts and tables, half of the analyzed articles were deemed to require low visual literacy, while the other half also used composed graphs (with more than one variable), which were deemed to require medium visual literacy [58]. Half of the dashboards required advanced domain knowledge from the user, about a quarter only required limited domain knowledge, and the remaining were fully understandable to the average person.

It makes sense that the two dashboards that were implemented in Microsoft Excel would be the only ones to lack any interactive features and only use a single view of data. Every other dashboard displayed different pages of information, providing at least the interactivity needed to switch between them, but most also allowed users to drill down into the data and use filters.

Over 70% of the dashboards examined used benchmarks, which is not surprising given that one of their primary objectives is to assess performance and increase awareness of established objectives [58].

Lastly, in Table 2.2, we can see a perfect overlap of "operational" dashboards and "process" oriented clinical indicators. This suggests that one property most likely won't exist in a dashboard without the other, since if we want to track data about routine medical procedures, we need to implement a dashboard that supports immediate decision-making. All other classifications of clinical indicators appeared a few times each, except the one with the "process" type.

Switching now to a more general analysis, four case studies from the group that detailed the design process chose a user-centered design approach, and one chose a co-design one. This emphasizes the importance of including the end-user when creating medical dashboards. From the perspective of development, some were said to have implemented the dashboard iteratively, while one case study used an agile approach.

Only one of these case studies lacked relevant information when it came to the technologies used. Consequently, out of the remaining population, about 64% (7/11) used web-based technologies, including JavaScript, jQuery, D3.js, ASP.NET, PHP, R Shiny, and others. From the remaining four case studies, half utilized enterprise technologies (Tableau, Epic Systems), while the other half utilized Microsoft Excel.

Table 2.1: Case studies used for the analysis. * The dashboard from case study **5** was continued and improved in case study **12**. The case study **10** contained three different dashboards (**10.1**, **10.2**, **10.3** in Table 2.2).

Case study	Article title
1	*Breaking the mould without breaking the system - the development and pilot of a clinical dashboard at The Prince Charles Hospital* [20]
2	*Dashboard visualizations - Supporting real-time throughput decision-making* [27]
3	*Developing an emergency department crowding dashboard - A design science approach* [39]
4	*Developing an Intranet-Based Lymphedema Dashboard for Breast Cancer Multidisciplinary Teams - Design Research Study* [30]
5	*Development and Evaluation of a Health Information Technology Dashboard of Quality Indicators* [38]
6	*Development and Implementation of Maternity Dashboard in Regional Hospital for Quality Improvement at Ground Level - A Pilot Study* [49]
7	*Development of dashboard for hospital logistics management* [36]
8	*Development, implementation and preliminary evaluation of clinical dashboards in a department of anesthesia* [34]
9	*Development, implementation and user experience of the Veterans Health Administration (VHA) dialysis dashboard* [26]
10	*EHDViz - clinical dashboard development using open-source technologies* [11]
11	*Improving Health Care Management in Hospitals Through a Productivity Dashboard* [50]
12*	*Usability Evaluation and Implementation of a Health Information Technology Dashboard of Evidence-Based Quality Indicators* [59]

The data sources in these case studies (when relayed) were mostly EHR systems (70%) and data warehouses, with some combining both.

Final results ranged from "good with room for improvement" to "highly usable and useful" when it came to the evaluation of the dashboards by medical professionals in these articles. This demonstrates that even a basic dashboard, such as one made in Microsoft Excel, can be helpful in a clinical setting because the gathering and organizing of information is by its very nature very beneficial.

Table 2.2: Classification of the case studies' medical dashboards, based on the design space in [58]. The "strategic", "tactical", and "operational" decision supports are denoted by **Str**, **Tac**, and **Op**, respectively. The letters **L**, **M**, and **H** stand for "low", "medium", and "high" levels of visual literacy, respectively. The letter **Y** indicates that the factor was identified or claimed to be present in the article, while the letter **N** indicates that the aspect was either explicitly absent or not acknowledged by the authors. **Limited** denotes a need for domain knowledge, but not to the extent that the average user would be completely unable to use the dashboard. **Almost** indicates that the data updates were nearly real-time, for instance with a 15-minute delay. The symbol **-** represents the impossibility of affirming whether there are or are not real-time updates in the dashboard. For the clinical indicators, **Stl**, **Pr**, and **Out** were used to represent their types: "structural", "process", and "outcome", respectively. The title and reference of each case study are registered in **Table 2.1**. * The dashboard from case study **#5** was continued and improved in case study **#12**. The case study **#10** contained three different dashboards (**10.1**, **10.2**, **10.3**).

	Purpose	**Audience**		**Features**			**Data semantics**			
Case study	**Decision support**	**Visualization literacy**	**Advanced domain expertise**	**Interactivity**	**Multipage**	**Construction and composition**	**Alerting and notification**	**Benchmarks**	**Real-time updates**	**Clinical indicators**
	Str/Tac/Op	**L/M/H**	**Y/N/Limited**	**Y/N**	**Y/N**	**Y/N**	**Y/N**	**Y/N**	**Y/N/Almost**	**Stl/Pr/Out**
1	Tac/Op	M	Limited	Y	Y	N	N	Y	Y	Stl/Pr
2	Op	L	Limited	Y	Y	N	N	Y	Almost	Stl/Pr
3	Op	M	N	Y	Y	N	N	Y	Y	Stl/Pr
4	Tac/Op	M	Y	Y	Y	N	N	N	-	Pr
5*	Op	L	Limited	N	N	N	N	Y	N	Pr
6	Tac/Op	L	Y	N	N	N	N	Y	N	Pr
7	Tac/Op	M	N	Y	Y	N	Y	Y	-	Stl/Pr
8	Tac/Op	L	Y	Y	Y	N	N	N	N	Pr/Out
9	Tac/Op	L	Y	Y	Y	N	N	Y	-	Pr/Out
10.1	Op	L	N	Y	Y	N	N	N	Almost	Pr
10.2	Op	M	Y	Y	Y	N	N	N	Y	Pr
10.3	Op	M	Y	Y	Y	N	Y	Y	Y	Pr
11	Str/Tac	M	N	Y	Y	N	N	Y	-	Out
12*	Op	L	Y	Y	Y	N	N	Y	Y	Pr

The created dashboard can be classified using the ideas covered in this chapter. First, it was determined that understanding user needs is crucial when designing a dashboard. In this project, this is guaranteed by the fact that MedicineOne has been in the market for medical software for over thirty years and has a wealth of experience working with medical professionals. Mock-ups were used in the dashboard design process, as is demonstrated in Chapter 4. The presented mock-ups demonstrate elements like consistent information placement, highlighting, and distinct visualization components. They were also mentioned as being crucial in the state-of-the-art chapter. By utilizing the three GQM model levels, we can verify the precise definition of (possible) data. For instance, the objective is to inform medical professionals of the state of affairs in their hospital division, which may lead to inquiries like "How crowded is the unit?" and "Will it be busy at the end of the day?" which can be effectively answered by values like the number of patients interned and how many have scheduled discharges. The oversaturation of screens with data, which is another important factor mentioned in the state-of-the-art, is avoided by the dashboard's design, which includes filters that change the information in the main table and open particular tabs for patient details instead of displaying all of it to the user. Finally, this dashboard prototype will now be described using the same classification criteria as the case studies. It is obvious that this dashboard is only intended for operational use because it provides up-to-date data to aid in decision-making during routine medical tasks. Because it only uses tables to display information, it only requires a minimal level of visualization literacy, which speeds up data assimilation. Anyone can understand how many patients are present and how many have scheduled surgeries, for instance, so there is also little need for advanced domain expertise. However, some patient data, like allergies and medications, requires more technical knowledge. Due to the use of filters and the ability to drill down on patient data, it offers features like interactivity and multiple pages, like the majority of case studies. Although it does not contain benchmarks, which can be thought of in the future, data semantics-wise, it is intended to provide alerts and notifications, unlike most case studies. Clinical indicators include both structural (data on unit crowding) and process-oriented (information on length of stay and scheduled medical procedures) information. Finally, as mentioned earlier, it guarantees real-time data updating.

Chapter 3

Technologies analysis

To ensure that an informed decision is made so that requirements can be met, it is crucial to first consider the alternatives before selecting the necessary technologies to create the dashboard. This analysis includes four categories: comparison between desktop and web applications; desktop alternatives; web development frameworks and libraries; and application communication services.

3.1 Desktop *vs.* Web

Aside from the technologies used for development, there are many differences between web and desktop applications that must be taken into account when choosing a development approach.

Desktop applications offer good performance and integration because they only rely on the hardware's capabilities and are created specifically to run in that environment. They are accessible even without an internet connection, and since data can be stored locally on the machine where the application is installed, it is safer. If a desktop application is offline, it usually stores data locally, but a Remote Method Invocation (RMI) server can be used to establish communication. Not utilizing an internet connection or a RMI server adds an additional layer of security but has drawbacks like: because it can only run on one machine, the application (and its data) cannot be accessed if physical access is not available (no portability); manual installation and updating are required, which may involve looking up system requirements and other compatibility issues. Most of these applications must be installed in order to be used, even though there are some workarounds. Additionally, desktop applications—whether offline or online—usually rely more on local storage, which means additional disk space is needed for installation and other features. This could be a problem, particularly for smaller devices [32, 44, 48].

Web applications, on the other hand, only need a web browser and an internet connection to function, meaning the user does not need to manually install or update anything; it is done automatically on the web. This also means that it does not take up space with installation files on the computer where the application is accessed. They are cross-platform because any device with internet access can use the web application, and because of their lightweight design, they can be accessed by machines that are less capable or sophisticated. One drawback of using a web application is that constant internet access is needed, so a reliable connection is required to avoid having the application slow down or be interrupted. These applications are also more vulnerable to security flaws, such as denial-of-service attacks, due to the fact that they are used online and, in most cases, the user's data is not stored locally on the user's devices, making them vulnerable to hacking [32, 44, 48].

In Table 3.1 a comparison of the two approaches can be seen. Desktop applications are usually more performant and secure, while web applications offer more options or are easier to benefit from in terms of portability, cross-platform, disk usage, installation, and updates. This comparison is meant to give a broader perspective, as web applications can also be performant and desktop applications can also be portable, for example.

Table 3.1: A summary of each type of application's advantages (A) and disadvantages (D) over each other.

	Performance	**Portability**	**Security**	**Cross-platform**	**Disk space**	**Installation and updates**
Desktop	A	D	A	D	D	D
Web	D	A	D	A	A	A

3.2 Desktop technologies

As the Application Programming Interface (API) used for data access utilizes .NET, only alternatives from that domain for developing the dashboard's interface are presented in this section, as it allows for easy integration if the desktop approach is chosen.

3.2.1 Windows Presentation Foundation

Windows Presentation Foundation (WPF) is an User Interface (UI) framework for building desktop applications for Windows that is part of .NET, which enables the integration with its other elements. This framework uses a vector-based rendering engine to deliver high-quality visualization components that are infinitely scalable, similarly to Scalable Vector Graphics (SVG). As a result, it offers powerful animation (2D and 3D) and styling features [68].

The typical method to develop applications with WPF, is to utilize both "markup" and "code-behind". Extensible Application Markup Language (XAML) is used as the markup language to define the visual composition and appearance of the application with components like windows, dialog boxes, pages, and others that can be filled with controls, shapes, and graphics. The term "controls" used to describe these elements in WPF has a very broad definition and refers to classes that are hosted in windows or pages, have a UI, and can perform behavior. Since it is markup-based, the composed layout is organized in a hierarchy of nested elements that is known as an "element tree". In order to implement the intended business logic and behavior for these components, such as what happens to the data when the user clicks a button, for example, these elements and their attributes are then converted at run time to instances of WPF classes that can be accessed by regular code, typically C# (code-behind). Lower development and maintenance costs and higher development efficiency are two advantages of separating the construction of UIs from the behavior implementation. With Microsoft's Integrated Development Environment (IDE), Visual Studio developers can alternatively utilize a drag-and-drop feature to populate interfaces with XAML components [68].

In WPF applications, a data binding engine that integrates with the "Binding" class at its core performs automatic tasks like copying data from objects into controls to display it to the user and keeping them synchronized after modification. There are two types of binding allowed in WPF, one-way and two-way. The second allows for complete synchronization between a control and a data object, whereas the first only allows for the UI to be updated with underlying data updates, the opposite, as in the underlying data being altered through the control, and a "one-time" mode, where the source only updates the other on initialization. This binding offers other features like data validation, sorting, filtering, and grouping and can be applied directly to the XAML layout. Additionally, data binding supports the use of data templates to design UIs for bound data [68].

3.2.2 Multi-platform Application User Interface

.NET Multi-platform Application User Interface (MAUI) is a cross-platform framework that enables the creation of desktop and native mobile applications, utilizing C# and XAML. This framework enables the development of applications that can be deployed on a variety of platforms, including Android, macOS, Windows, and others, from a single shared code base. It is an open-source evolution and replacement for Xamarin.Forms, Microsoft's now-deprecated mobile development framework. Leveraging some of its functionality while rebuilding others for performance and usability improvements and extending it to the desktop environment. .NET MAUI's has as one of its defining goals allowing developers to implement most of their applications' logic and UI in a single code base [45].

By utilizing the native .NET APIs for platforms like Windows and iOS, which all have access to a common Base Class Library (BCL), .NET MAUI unifies various platforms under a single code base. This library is able to offer native execution environments for applications that permit the sharing of business logic, while .NET MAUI provides a framework for the sharing of UIs. The application code, as seen in Figure 3.1, primarily interacts with the .NET MAUI API, which in turn consumes other .NET native APIs. However, in some circumstances, it can also access these native APIs directly [45].

This framework supports data-binding properties (one-way and two-way), has a

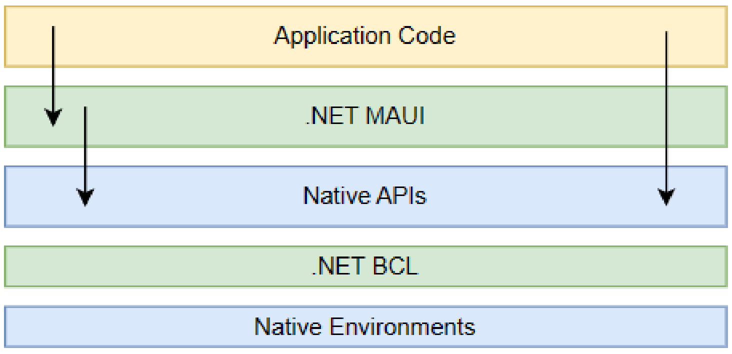


Figure 3.1: A high-level visual representation of the .NET MAUI framework architecture.

powerful layout engine and various page types to create rich UIs, the ability to customize control handlers, and "hot reload" functionality, which allows developers to change either the XAML markup or the code-behind while the application is running and the changes are propagated without restarting the application [45].

3.2.3 Desktop technologies comparison

The two technologies addressed above serve different functions: WPF is used to create Windows desktop applications, while .NET MAUI is used to create cross-platform applications that can run on Windows, macOS, and mobile devices. The first offers benefits like straightforward multimedia integration, resolution independence, hardware acceleration, declarative programming using XAML, and extensive control customization. Because it is not WPF's intended use, it lacks platform compatibility and cannot be used with Windows 2000 or prior versions. The benefits of .NET MAUI are primarily attributable to its cross-platform capabilities, including platform reach, code reuse and sharing, and lower cost of bug fixing than an application implemented for various platforms separately. Coming from another background may allow for a lower learning curve because it uses common .NET technologies and languages. On the other hand, it is a very recent framework that was only released in 2022, does not support Linux, and does not support web on its own [45, 68].

3.3 Web technologies

There are many methods one can use to build web applications. Although it is possible to only use pure JavaScript, there are some advantages to using libraries and frameworks, including faster development and fewer coding errors [19].

For the main application implementation, web development frameworks and a more complete library are examined first and then compared, as they are part of the main decision to be made. Subsequently, another, smaller library was examined that is specially intended for graphical visualization.

3.3.1 React

React has become one of the most popular technology choices in recent years and is currently the most popular choice for front-end web development [63], so it makes sense to take it into consideration for this project. Although React is technically a library for creating UIs, it is frequently referred to as a JavaScript framework due to its widespread use and popularity in numerous types of applications. It is kept up by Facebook as well as other businesses and developers [13].

React has many features that enable it to compete with JavaScript frameworks in the market for web development. Its main characteristic is that it is based on components, which is the name given to the parts that make up the application and are housed inside distinct modules. Developers can easily pass complex data through the application without confusing the Document Object Model (DOM) with state by using these components because the logic for them is not written in templates but in JavaScript [13, 54]. The code for React components is written using JavaScript XML (JSX), a JavaScript extension that enables code to be written similarly to HyperText Markup Language (HTML), making it more readable and easier to learn the library [13, 54].

React by default only supports one-way data binding, while further steps are required to implement two-way data binding [13, 54]. This binding is from the components to the views, so by default, only the logic of the components can change the views and not the other way around. This makes the application more modular and faster [54].

The last and one of the most important features of React is its DOM management (see Figure 3.2). For every browser DOM object already in existence, React creates a virtual copy of it that is stored in memory. It becomes no longer necessary to repeatedly re-render entire pages because of the existence of this replica, which allows React to only update the actual DOM when a change is detected in the virtual one. These changes are detected in the state of components, which developers can programmatically request to occur, and in doing so, a re-render is applied to the component; this follows the life cycle of a React component (mount, update, unmount). Because no data is drawn on the page when manipulating the virtual DOM, it is faster than doing so with the original. This significantly improves the performance of web applications built with React in comparison with the usual DOM manipulation techniques [13, 54].

3.3.2 Angular

Angular is one of the most widely used front-end frameworks for web development [63]; it was created and is maintained by Google, which gives it a very solid and reliable reputation. It was developed using the AngularJS framework with the intention of converting from JavaScript to TypeScript, which is one of its key features. With TypeScript, the Angular framework has access to strong features like object-oriented programming, static types, iterators, lambda functions, and

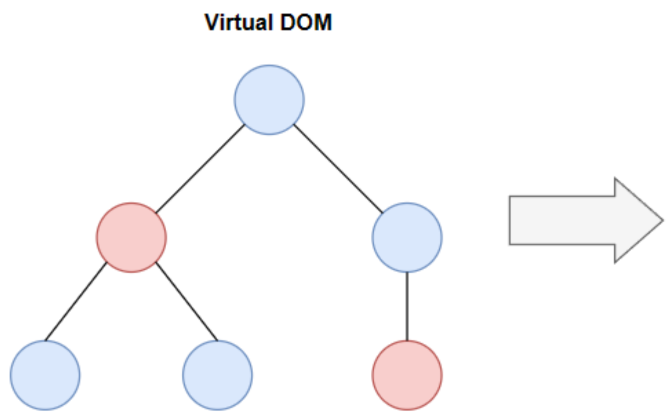


Figure 3.2: A visual representation of React's DOM manipulation. The red circles on the virtual DOM represent altered components that were discovered after comparing the two DOMs. The orange circles on the browser's DOM stand in for components that had to be re-rendered as a result of changes to the virtual one.

others [7, 13, 65]. The fact that Angular as a framework integrates other libraries to offer solutions for routing, form management, client-server communication, and other issues makes it simpler to create enterprise-level web applications [7].

Similar to React, Angular emphasizes componentizing development to improve scalability. These components provide a TypeScript class to implement functionality, a HTML template to indicate how the component should be rendered, and the possibility to include style sheets to enhance the appearance of the templates. Encapsulation is enabled as a result, which facilitates testing and strengthens the application's structure [7].

Data binding is supported by Angular, but unlike React, two-way data binding is enabled by default. This binding occurs between components' templates and data models in the class files, and it can be used to change the template or the data, as well as both at once (see Figure 3.3). Angular's change detection algorithm listens for these data changes, and when one is detected, the defined updates to the view and component are applied [7].

Dependency injection is arguably the most significant feature of the Angular framework. It permits the declaration of TypeScript classes' dependencies while removing the user from the process of instantiating them because Angular takes care of that task on its own. When a dependency is requested, an abstraction known as an "injector" determines whether a memory instance of that dependency already exists; if not, one is created and saved. Once every dependency is fulfilled, Angular calls on the components' constructors. This design pattern also adds to the flexibility and testability of the developed web applications [7].

In addition, Angular offers a command-line interface (Angular CLI) that can be used to launch, create scaffolds for, deploy, and maintain applications [7].

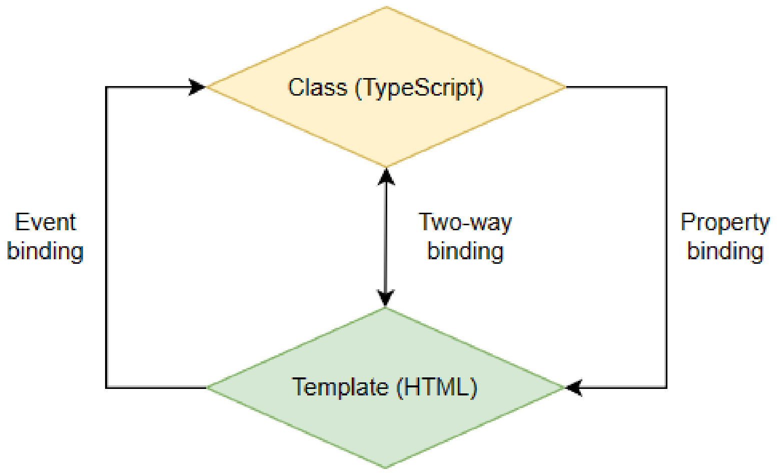


Figure 3.3: A visual representation of data binding in Angular. A button click on the template that activates a function on the class is an example of event binding. A value in the template that is defined by a value in the class is an example of property binding. Two-way binding is the combination of both types.

3.3.3 Vue.js

Vue.js is a JavaScript framework that has grown in popularity in the front-end web development environment unlike any other in recent years, and is now almost as popular as Angular [13, 63]. This framework was created with adaptability as its guiding principle; as a result, it can easily integrate with other libraries and projects due to its main library's focus on the "view" layer. For instance, if we have a server-side application made with another JavaScript framework, Vue.js can be integrated to provide interface development and management. Vue.js can be utilized independently for simpler applications, such as single-page ones [13, 67].

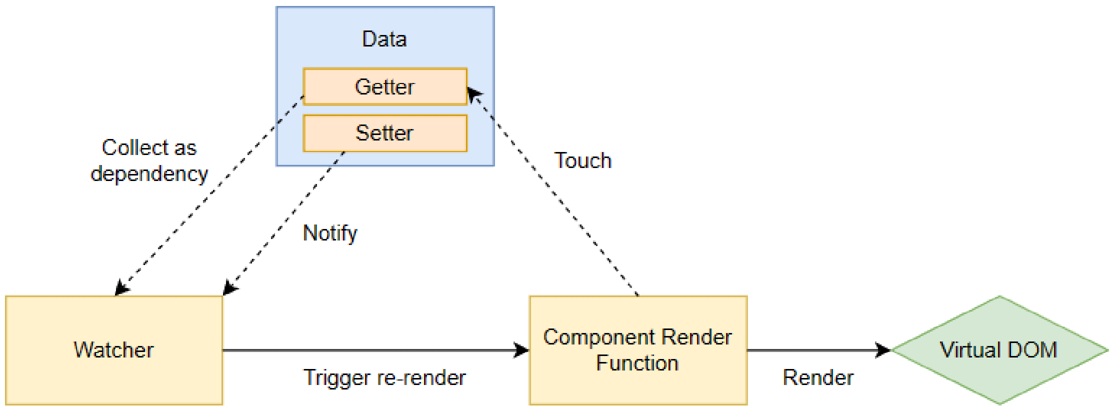


Figure 3.4: A visual representation of reactivity in Vue.js. There is a watcher for each component, which registers when properties are "touched" during the render of the component by gathering them as dependencies. The watcher is alerted if a property setter is later used, which causes the component to re-render.

The use of components is one of the features that this framework has in common with the other two. The components in Vue.js are referred to as "Single-File Components" and combine the logic, template, and style of the component in a syntax resembling HTML. These components can be written based on two different APIs: "Options API" and "Composition API". The first one can be used by creating the following objects inside the component: "data", which holds the information to be used as the component's updatable state; "methods", which change the state and cause updates (they can be set as listeners in templates); and "mounted", which denotes the stage of the component's lifecycle where it is prepared for use. The second relies on importing API functions and declaring reactive state variables within the scope of a function. Understanding the reactivity property of Vue.js, which is briefly described next, is necessary to use this method. Another feature that Vue.js and React have in common is the virtual DOM, which improves performance when working with the browser's DOM [67].

The reactivity system in Vue.js can be characterized as unobtrusive because components' states are implemented as JavaScript objects that, when modified, cause the view to update. Vue.js accomplishes this by keeping track of the "getters" and "setters" for these objects, which allows it to determine when and what needs to be updated (see Figure 3.4). This approach to state management can be regarded as being simpler and more developer-friendly [67].

Finally, two-way data binding is a feature of Vue.js that, like Angular but unlike React, enables the template and the data to make changes to each other [67].

3.3.4 Web technologies comparison

As already mentioned, React, Angular, and Vue are some of the most well-liked web technologies for building UIs. The first is by far the most used, and the last is a newcomer [63]. This new adherence to Vue may be related to having the lowest learning curve out of the three, since it is based on plain JavaScript, HTML and Cascading Style Sheets (CSS), while Angular utilizes TypeScript and React utilizes JSX, which might prove more cumbersome to developers learning the technology. Even with the need to learn JSX, React is said to be easier to learn than Angular, mostly because the latter contains more built-in functionalities [19, 42, 61].

React and Vue are very similar to one another in terms of memory usage and performance because they are both lightweight and use the virtual DOM technique. Due to its included packages, Angular is typically slower, uses more memory, and has a larger project size in regular applications. However, when working with large-scale enterprise web applications, this size disparity is diminished because React and Vue need to import external libraries and packages in order to implement such complex systems, while Angular is already prepared for this scenario. Compared to the other two technologies, Angular is less flexible in terms of how developers can structure their applications and which packages to use. The other two, however, offer complete freedom in terms of how to design the application architecture and integrate it with other libraries [19, 42, 61].

A summary of the comparison made can be seen in Table 3.2, which characterizes the mentioned characteristics with qualitative values.

Table 3.2: Comparison of web technologies summarized.

	React	**Angular**	**Vue**
Popularity	High	Medium	Medium
Learning curve	Medium	High	Low
Flexibility	High	Medium/Low	High
Data binding	One-way	Two-way	Two-way
Size	Low	Medium	Very low
Performance	High	Medium/High	Very High

3.3.5 D3.js

Data-Driven Documents, or D3.js, is a JavaScript library for data-based document manipulation that can enhance data visualization by utilizing CSS, HTML, and SVG. It addresses the core of the "problem", which is effective manipulation of data-based documents, rather than trying to give developers access to every feature related to web-based information representation. Focusing on web standards gives developers complete access to the features of modern browsers, which in turn offer a ton of flexibility. Due to its low overhead on manipulations, D3.js is also extremely quick, supporting the use of large datasets, intricate visual animations, and dynamic data interaction [22].

In order for D3.js to create visual representations, it first binds information to a DOM and then applies data-driven transformations to that document. An example could be creating a table in HTML, or an interactive pie chart in SVG, from a list of values. HTML tags, classes, and identifiers can be used to select nodes in D3.js. These nodes can then be modified by setting attributes and styles, registering event listeners, or even being deleted and sorted. Accessing the underlying DOM is also a possibility. When it comes to visual transitions, D3.js only modifies the necessary properties in order to be more efficient. With a wide range of official and community-developed modules, D3.js's functional style enables the reutilization of code [22]. A wide variety of examples can be seen at [21].

3.4 Application communication

As was previously mentioned, in order to be able to give medical professionals real-time information, a channel of communication between the main system and the dashboard must be established. A few of the methods for doing this are discussed in this section.

3.4.1 Azure SignalR Service

An application can receive real-time functionality over HyperText Transfer Protocol (HTTP) from the Azure SignalR Service, which enables a source to push updates to linked client systems. This is made possible by the SignalR hubs, which can be viewed as topics or channels that broadcast the messages or send them to a particular connected client. As a result, clients are no longer required to periodically poll the server or submit update requests. A wide range of system types, including gaming, polling, and voting applications, real-time dashboards and monitoring systems, message chats, push notifications in social networks and emails, and many more, use this service to acquire real-time functionality [10].

Different transport methodologies can be used with Azure SignalR Service to create real-time web applications. It can determine which approach is more appropriate based on the capabilities of both the server and the client. Although this service prefers WebSockets in most cases, it can also use Long Polling or Server-Sent Events (SSE) if that is not an option. Thus, WebSockets will be used whenever possible to enable two-way communication. Using this method, ASP.NET Core and ASP.NET can both be used with native programming on the server side. It can work with a variety of client platforms, including browsers, IoT devices, game consoles, and mobile, web, and desktop applications. Additionally, serverless support is offered, for instance, through Azure Functions and Representational State Transfer (REST) APIs [10].

3.4.2 Firebase Cloud Messaging

Firebase Cloud Messaging (FCM) is a reliable cross-platform solution for sending and receiving messages in mobile and web applications. It can be used to alert clients to data-related server events, and programmers can specify what should happen next in the application based on that notification. Due to the two-way nature of the communication, either the server or the clients can send messages to an individual client, a group of clients, or all clients who have subscribed to a topic [25].

To use this service, message requests must first be developed in a reliable environment that supports either the Firebase Admin Software Development Kit (SDK) or FCM server protocols. Instead of being a Cloud Functions for Firebase environment, this environment may be a custom application server, but it must be able to send message requests that are properly formatted, handle request emission retries, and securely store client tokens and authorization credentials. No matter the method, message requests are sent to the FCM backend, which primarily receives them and generates an identifier for each one while also performing other operations on them. This backend uses the corresponding platform service (Android, iOS, and web) to route the messages through a platform-level transport layer to their final destination, where the FCM SDK on the device will handle and operate on the messages [25].

3.4.3 Amazon Simple Notification Service

Amazon Simple Notification Service (SNS) is a service that provides message delivery based on the publisher-subscriber pattern. A publisher can interact asynchronously with subscribers using these kinds of communication systems by posting messages in a topic that serves as the communication channel. Once subscribed, users can start receiving messages from a SNS topic via a supported endpoint, such as other Amazon components, HTTP, email, or mobile notifications and messages [5].

This service provides a variety of features, which are the following [5]:

- Application-to-application messaging that permits communication between applications via HTTP or other Amazon services;
- Application-to-person notifications for emitting notification to end-users devices and email addresses;
- First In First Out (FIFO) topics to ensure message ordering and non-duplications. These topics can only be subscribed to by the Amazon Simple Queue Service FIFO;
- By storing messages in various locations, employing a delivery retry policy, and keeping undeliverable messages in a designated queue, Amazon SNS ensures message durability;
- Message filtering so that subscribers only receive the messages they need;
- To ensure security, messages are encrypted on the server side.

3.4.4 Application communication technologies comparison

The three options mentioned, will now be contrasted, to help decide which service to use to enable real-time communication between the medical system and the dashboard. First off, in terms of multi-purpose cloud providers, Amazon Web Services, Microsoft Azure, and Firebase are, respectively, the first, second, and fourth most popular platforms at this time [63]. This popularity brings some sort of reliability and certification factor into play when choosing a cloud service provider.

All of these services are accompanied by decent documentation overall, including tutorials, API references, and system architecture, among other things. Additionally, each corresponding development console is fairly simple to use, with Amazon's SNS being the least so of the three, and they all offer solid support in terms of analysis and reports, particularly FCM. The APIs are all well organized, with FCM having the best organization while the Azure SignalR Service's is more difficult to use [2, 3].

All three services offer SDKs for the intended languages and environments, which are JavaScript for web development and .NET for desktop and backend development, respectively. The supported communication patterns and protocols for each of these services are listed on Table 3.3 [2, 3, 5, 10, 25].

Table 3.3: The three services' capacities for supporting various communication protocols and patterns. The letters Y and N indicate whether or not the service can apply the method, whereas Limited indicates that either the full method cannot be applied or that it can be applied but only when integrated with another service.

	Patterns			**Protocols**		
Services	**Publisher/Subscriber**	**Message Queue**	**WebHooks**	**WebSockets**	**HTTP**	**SSE**
Azure SignalR Service	Limited	N	Limited	Y	Y	Y
FCM	Y	N	Y	N	Y	N
Amazon SNS	Y	Limited	Y	N	Y	N

Only Amazon SNS can guarantee message order and the deliver of a message only once, but only when sending messages to FIFO queues of Amazon Simple Queue Service. Only the Azure SignalR Service cannot make use of native push notifications, which allow messages to be sent to clients who are not connected to the internet. All of the services can make calls to serverless functions in their respective cloud environments. The security of the three services is comparable; they all provide API key authentication, token authentication, programmable rules and permissions, and encryption of the data while it is in storage and transit, but only Amazon SNS guarantees encryption of the message payload. In Table 3.4 limits such as throughput, number of connections and Service Level Agreement (SLA) can be seen [2, 3, 5, 10, 25].

3.5 Technological decisions

The decisions made in relation to the dashboard's implementation will now be discussed.

To begin with, **web technologies** were chosen as the method for creating the dashboard interface. This decision is supported by a few factors. The first is the state-of-the-art, where it was discovered that, from the article sample, more than half of the case studies made use of web-based technologies for the creation of the corresponding dashboards. This might be attributed to the application's simple

Table 3.4: The limits of the three services. The symbol "-" states that no clear information was found to fit the limit. In the Azure SignalR Service a unit represents a sub-instance that processes messages, and the value 99.95% for the SLA is for the premium subscription.

Services	**Throughput (messages/second)**	**Message size (KBytes)**	**Number of channels/topics**	**Number of connections/subscriptions**	**SLA**
Azure SignalR Service	-	Unlimited, messages are divided into chunks of 2	100 units/instance (standard tier)	1000 connections/unit (standard tier)	**99.9%-99.95%**
FCM	4 (1 to a single device)	4	Unlimited, an application can only subscribe to 2.000 topics	2.500 connections/project	**99.95%**
Amazon SNS	20	256	100.000 standard topics/ account (1.000 for FIFO topics)	12.500.000 subscriptions/ standard topic (100 for FIFO topics)	**99.9%**

dissemination and accessibility among medical professionals, as well as its integration with tools for producing eye-catching data visualizations like D3.js and R Shiny and lower development costs. Another factor is the configuration data, which can be addressed by building a dashboard out of adaptable components, which fits well with the component-based design of the web UI building tools under consideration. Finally, in terms of drawbacks, performance is not thought to be at risk because the intended application only has a limited set of features beyond information display and the ability for users to drill down on particular units and patients. It has more security risks than desktop alternatives, but in this case, the web application will be used in the hospital's private networks, which are more secure than regular public ones. Benefits like not having to manually install and update the application and the assurance that the dashboard will display correctly on any computer in any hospital are valued.

React was the web technology of choice. It should be noted that any of the component-based frameworks and libraries examined could be used to implement the dashboard, but React was preferred because there was no need for two-way data binding because the dashboard is only intended to be read-only, thus it is favorable to not add redundancy by including a blatantly undesirable functionality present in Angular and Vue. In addition, React offers a number of benefits, including a not-so-steep learning curve, a large and active community, and the fact that it is lightweight and performant. Also, Angular has the downside of not being as flexible, which is key in the intended application. Due to the simplicity of the necessary visual representations, D3.js was not deemed necessary; however, it is an option and recommendation for the implementation of such graphic components in the future.

The **Azure SignalR Service** was chosen as the cloud option to deliver real-time communication of events to the dashboard. First of all, in terms of supported patterns and protocols, FCM and Amazon SNS are the most comparable of the three, however, SNS offers better throughput and message size limits, as well as being already used by MedicineOne, so FCM was immediately disqualified. The Azure SignalR Service, which is also currently in use, namely in the MedicineOne API to communicate with the server, provides unlimited message size and is able to use the SSE transport protocol, which corresponds well with the intended use; a one-way communication channel for the server to inform the dashboard about specific data altering events.

Chapter 4

System specification

The specification of the requirements, architecture, and UI design serves as a comprehensive guide to establish a solid foundation for fully understanding the scope of the project and its development.

4.1 Requirements and use cases

The following requirements were listed and described based on an informal description of what the system should enable users to do and what it should offer. Then, according to their impact on the project's successful implementation, they were prioritized using the MoSCoW method, which divides requirements into four categories: "Must have", those that are essential to the project; "Should have", those that are significant but not entirely necessary; "Could have", those that are desirable but not essential; and "Won't have", those that are the least important [43].

The requirements and their MoSCoW priority can be seen in Table 4.1.

The "Must have" requirements mostly concern the user inputs necessary for the application to function, such as the server address from which the configuration and medical data is obtained, the organization where the system is deployed and a dashboard configuration present in the organization, and lastly, the desired patient list to be displayed. These also include the correct application of the dashboard configuration to the several UI components and other crucial functionalities.

The "Should have" and "Could have" requirements are primarily focused on quality of life features, with the first set being more crucial.

The final requirement is a "Will not have" requirement because it is outside the current project scope, but it may be addressed in the future.

From these requirements, use cases were created for the present user-centered actions, as shown in Tables 4.2, 4.3, 4.4, 4.5 and 4.6.

Table 4.1: System requirements priotized with the MoSCoW method. The letters M, S, C, and W stand for "Must have", "Should have", "Could have", "Will not have", respectively.

Code	Requirement	Priority
R1	Users must be able to navigate through the application pages through buttons present in the header.	M
R2	Users must be able to insert the API address for the application to communicate with.	M
R3	Users must be able to choose their organization and an associated configuration for the application to use.	M
R4	The application must support displaying patients from at least inpatient units, operating blocks, and emergency rooms.	M
R5	Users must be able to choose a patient list for the application to display.	M
R6	The application should save the user's settings, these being: the server address, the organization, the configuration, and the patient list.	S
R7	Users should be able to expand and collapse the patient lists of a parent node.	S
R8	The application must apply the configured patient panel attributes.	M
R9	The application must display the configured indicators.	M
R10	Users should be able to select one of the indicators to apply a filter to the patient list.	S
R11	The application must display the configured patient list.	M
R12	The application could display a message if the patient list is empty.	C
R13	Users must be able to select a patient from the list to open the patient detail.	M
R14	The application must display the configured patient detail components.	M
R15	Users must be able to close the patient detail by selecting the same patient from the list.	M
R16	The application should receive notifications when the data is updated in real time.	S
R17	The application must at least update the information with a set interval.	M
R18	The application could stay on the same page when it updates information.	C
R19	The application could maintain the selected indicator filter and update the filtered list when it updates information.	S
R20	The application could maintain the selected patient detail open when it updates information, unless the patient is no longer present in the list.	C
R21	The application could emit sound notifications.	C
R22	The application could emit graphical notifications.	W

Table 4.2: Description of use case #1 - Set the application settings.

UC-1 - Set the application settings		
Source	User	
Goal	The user wants to set the server address, organization, and configuration to be used in the application.	
Preconditions	The user is on the settings page. At least a dashboard configuration has been created for an available organization.	
Flow	1. The user inserts the server address;	
		2. The application enables the "connect" button;
	3. The user clicks the "connect" button;	
		4. The application enables the first dropdown menu with the organizations that contain dashboard configurations;
	5. The user selects the desired organization;	
		6. The application enables the second dropdown menu with the selected organization's configurations;
	7. The user selects the desired configuration;	
		8. The application enables the "continue" button;
	9. The user clicks the "continue" button;	
		10. The application loads the patient list selection page.
Postconditions	The user is on the patient list selection page.	

Table 4.3: Description of use case #2 - Select a patient list.

UC-2 - Select a patient list		
Source	User	
Goal	The user wants to select the patient list to be displayed.	
Preconditions	The user has already set the settings in the corresponding page. The user is on the patient list selection page. At least a patient list exists within the organization.	
Flow		1. The application loads the available patient lists grouped by their "parent" node;
	2. The user selects the desired patient list by: i. clicking on a "child" or "parent" node; ii. double-clicking on a "child" node.	
		3ii. The application loads the patient panel page with the selected patient list;
	3i. The user manually navigates to the patient panel page.	
Postconditions	The user is on the patient panel page.	

Table 4.4: Description of use case #3 - Apply a filter to the patient list.

UC-3 - Apply a filter to the patient list		
Source	User	
Goal	The user wants to filter the patient list by one of the available indicators.	
Preconditions	The user has already set the settings in the corresponding page. The user is on the patient panel page. There is at least one indicator apart from the main one.	
Flow	1. The user clicks one of indicators: i. the indicator was not already selected and corresponds to at least one patient; ii. the indicator was already selected or had a value of zero.	
		2i. The application filters the patients that do not correspond to the selected filter; 2ii. The application takes no action.
Postconditions	The user is presented with the filtered patient list.	

Table 4.5: Description of use case #4 - Open a patient's detail.

UC-4 - Open a patient's detail		
Source	User	
Goal	The user wants to open the detailed view of a patient from the patient list.	
Preconditions	The user has already set the settings in the corresponding page. The user is on the patient panel page. There is at least one patient in the list.	
Flow	1. The user clicks one of the patients;	2. The application opens the selected patient detail.
Postconditions	The user is presented with the detailed view of the selected patient, which he can close by clicking on the same patient.	

Table 4.6: Description of use case #5 - Navigate the application pages with header buttons.

UC-5 - Navigate the application pages with header buttons		
Source	User	
Goal	The user wants to navigate the application pages (settings page, patient list selection page, and patient panel page) through the corresponding header buttons.	
Preconditions	A server address, an organization, and a configuration have been set, to navigate to the patient list selection page. Additionaly, a patient list has been chosen, to navigate to the patient panel page.	
Flow	1. The user clicks one of the header buttons;	2. The application loads the corresponding page.
Postconditions	The user is on the selected page.	

4.2 Architecture

In this section aspects related to the architecture of the system are relayed.

Firstly, in Figure 4.1 the different components and interactions of the system can be seen. A web application that houses the dashboard and its features is the system's main component. The application uses the MedicineOne API to retrieve the clinical data for the dashboard and the configuration it needs to apply. The API communicates directly with the hospital database where the mentioned data is present. Medical professionals use the MedicineOne system daily, and it is here that dashboard configurations can be made and then saved in the same database. In order to meet the requirement of providing real-time data updates in the application, it is planned for the MedicineOne system to call the API whenever specific events take place. The API will then send messages pertaining to these events to the Azure SignalR Service instance. After that, the dashboard will have access to these messages and can update as necessary.

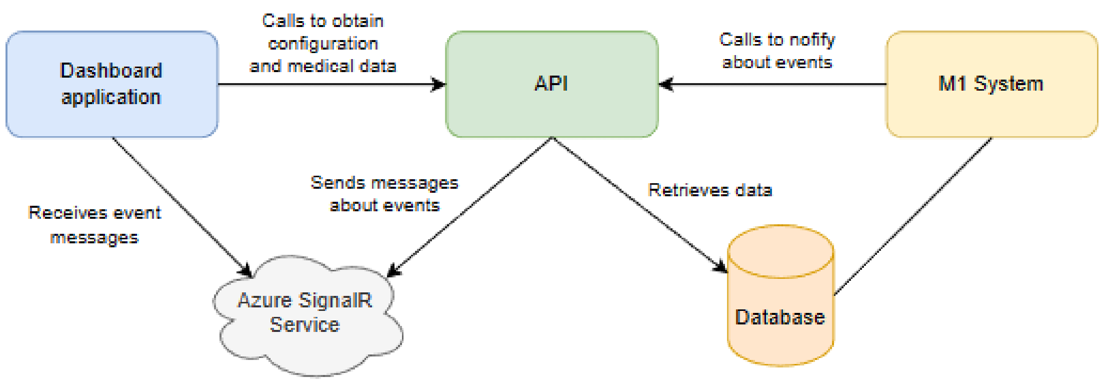

Dashboard application
Calls to obtain configuration and medical data
API
Calls to nofify about events
M1 System
Receives event messages
Sends messages about events
Retrieves data
Azure SignalR Service
Database

Figure 4.1: A visual representation of the system architecture.

To fulfill the requirement of saving the user's settings the browser's local storage was used, as these settings are not considered sensitive data, and this storage method is persistent, allowing the web application to maintain them even after a restart. The MedicineOne API uses JSON Web Tokens (JWTs) as its method of authentication, so any tokens that are generated are also saved in the local storage. Given the constraints and the fact that the application will be used exclusively within the hospital's private networks, which can increase security, it is thought to be a reasonable approach in this case.

4.2.1 Configuration schema and dashboard data

The database schema for the configuration data that the application obtains through the API, is shown in Figure 4.5. It provides insight into how such information is organized and how it structures the dashboard. In the center is the list of patients ("PatientListPanel") which represents the main table on the dashboard interface. This table can have a single type ("PatientListType") and multiple indicators ("PatientListPanelMetric"), the types being the hospital units such as internment and the indicators the different filters that can be seen in the top of Figure 4.9. Being a table, it contains multiple columns ("PatientListPanelColumn") each having a field type ("ListFieldType"). The table has multiple detailed patient views ("PatientDetailComponent") which can be accessed by clicking a patient (see Figure 4.10). This detailed view contains various columns ("PatientDetailComponentColumn") which also have a type.

The "PatientListType" database table contains a list of the supported types, which includes inpatient unit, operating block, emergency room, pharmacy, treatment room, oncology day hospital, and hemodialysis day hospital. These types only limit the hospital areas from which patient lists can be chosen. For example, if a configuration is of the type operating block, it makes sense to only access patients who have scheduled surgeries; this does not impose any restrictions on the choice of the displayed fields, indicators, or dashboard's appearance. Additionally, all types, with the exception of the inpatient unit and pharmacy types, permit the selection of one or more lists at once by selecting the corresponding parent node of the lists.

```
{
    "entries": [
        {"patientId": 1234, "fieldName": "abc"},
        ...
    ]
}
```

Figure 4.2: Indicator data schema.

Regarding the dashboard data, a few allowed types are already defined in the "ListFieldType" database table. These are:

- Text, the primary type used to display anything from names, dates, numbers, etc.;
- Photo, which mainly serves for the patients' picture, although it can be used freely. A dashboard field of this type expects to receive binary data to render the photo;
- Image 16/24/32/48/64, from a collection of local icons in various sizes (16x16, 24x24, 32x32, 48x48 and 64x64 pixels). This kind of field anticipates receiving a string of numbers that correspond to images as well as text enclosed in quotation marks that serve as image labels.

The information itself can be anything that is stored in the organization's systems; as such, the data is fetched from the database through stored procedures. This way, queries can be tailored to each configuration's requirements without the need to alter the codebase. MedicineOne will supply this parameterization so that their clients can choose every field of information that is shown on the dashboard.

Because the application needs to receive the data in a consistent manner in order to use it, data models were developed to enforce some rules on how stored procedures will return the data.

The schema for the indicator values is shown in Figure 4.2. It is made up of a single list called "entries" that includes patient entries for a particular indicator. Each entry has a "patientId" which is an integer unique identifier, and a "fieldName" which is a string that corresponds to a configured indicator. This structure allows the application to calculate each indicator value by counting the corresponding entries. Additionally, because the patient list, whose schema is described shortly, also includes the patient identifier, it enables patient filtering when an indicator filter is applied.

The schema shown in Figure 4.3 is used for the patient list. A list of patients is included, and for the previously mentioned reason, each one must include their unique patient identifier. Other than that, every other field is configurable, with its name defined in the configuration and its value following.

Finally, the schema shown in Figure 4.4 was developed for the patient detail data.

```
{
    "patients": [
        {
            "patientId": 1234,
            *Field Name*: "value",
            ...
        },
        ...
    ]
}
```

Figure 4.3: Patient list data schema.

```
{
    "patientDetail": {
        "photo": "value",
        "name": "value",
        "birthdate": "DD-MM-YYYY - XX years",
        "processNumber": "value",
        "gender": "value",
        "nationality": "value",
        "doctor": "value",
        "contacts": "value",
        *Field Name*: "value",
        ...
    }
}
```

Figure 4.4: Patient detail data schema.

It includes a few default fields created for an identification column in the detail section, which is explained in Section 4.3. A separate stored procedure that can be used for any configuration returns these fields. The configured field names and values make up the remaining data. For each detail component, a stored procedure is to be used; as such, all the stored procedure results should be merged before returning.

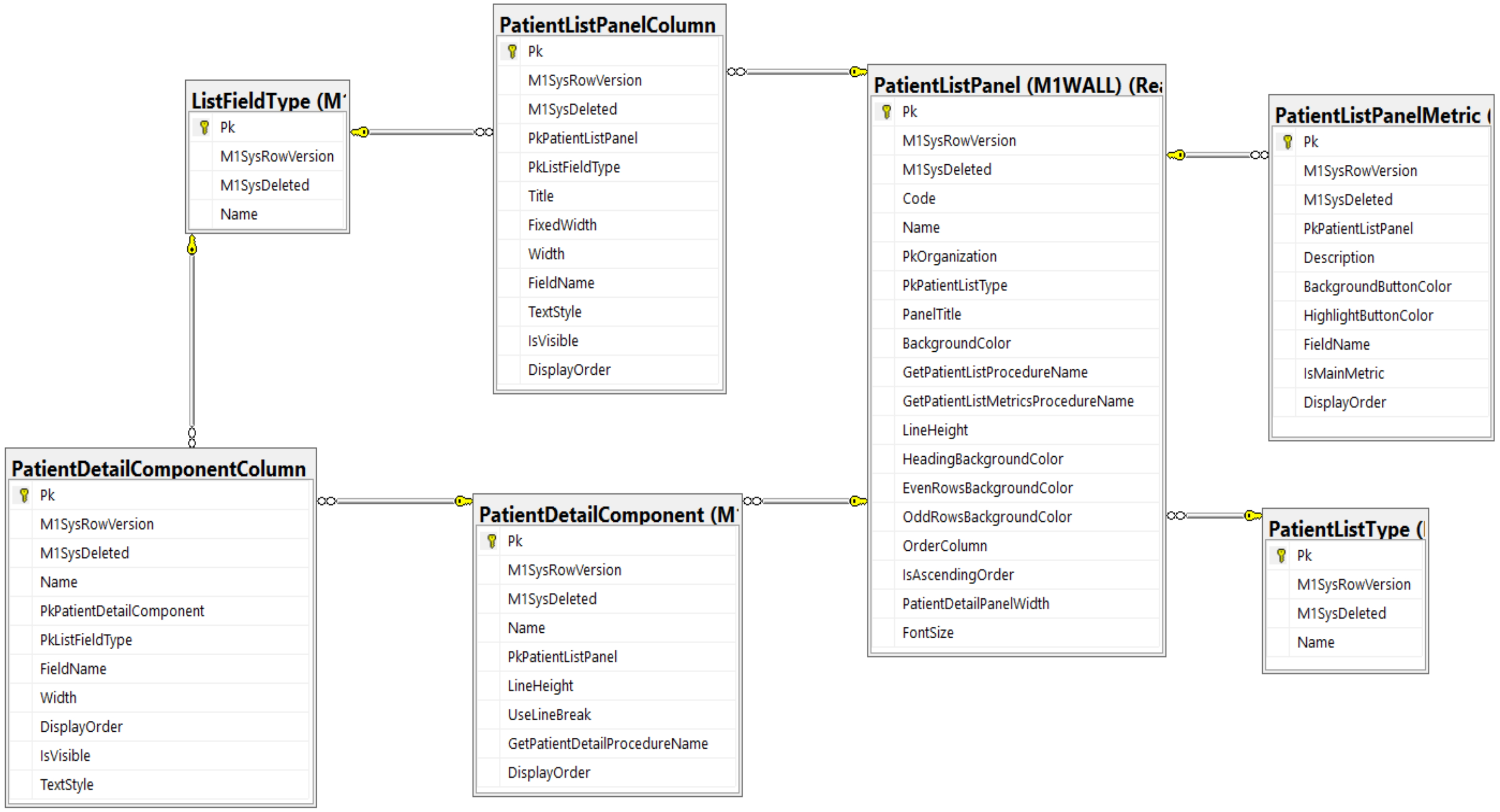


Figure 4.5: The database schema for the dashboard configuration data.

4.2.2 API description

As previously mentioned, MedicineOne's REST API is used to access medical data gathered and generated by their system in order to populate the dashboard application with the necessary information. Structurally, this API is composed of several components based on the Onion Architecture and the Domain-Driven Design (DDD) pattern. The Onion Architecture (see Figure 4.6) is an architectural pattern that focuses on layering systems so that layers at the center cannot depend on the ones at the edges but only on more internal layers. The domain model, which represents the business and behavior objects, is always the core layer. The repository interfaces, which offer saving and fetching functionalities, are typically found in the first layer after the domain model. On the edges reside the UI, infrastructure, and test layers. This pattern's ability to decouple the rest of the system from the data by having the application database outside of it is one of its most distinctive features. Since there is more decoupling in this architecture than in the typical system, applications that use it benefit from lower maintenance costs, especially if they are complex and enterprise-focused. When creating the internal domain model, the use of the design pattern DDD complements the architecture. The fundamental idea behind DDD is the use of a standard nomenclature by developers and subject matter experts in both the code and business domains; terms like "patient", "doctor", and "user" should all mean the same thing in those contexts, which enables better communication between stakeholders [31, 37].

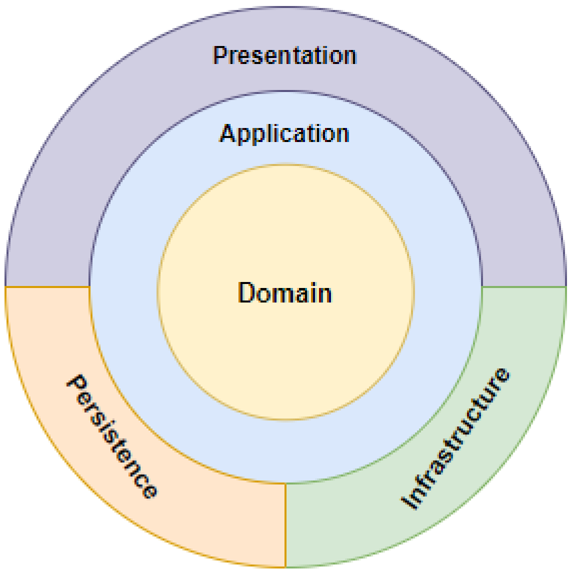


Figure 4.6: A visual representation of the Onion Architecture in the API. The presentation layer is where the API to be consumed is implemented; the persistence layer is where database operations are implemented; the infrastructure layer is where external code to the application is written; the application layer is where the business logic resides; and finally in the center is the modeled domain.

The API implementation (main component) can be consumed by different types of applications, and for each there is a sub-directory in the "Controllers" location.

These categories are:

- M1 client and web applications;
- Mobile applications;
- Third-party applications (public API).

Utilizing ASP.NET Identity [8] and a customized implementation of OpenIddict [47], authentication is used to access the API endpoints. These sub-directories also contain information related to the API supported versions. Every time the API undergoes a significant change, a new version is released with backwards compatibility for earlier releases.

The MediatR package implements the mediator pattern, which aids in minimizing object dependency management issues [41]. This pattern requires that objects communicate through a mediator object, which reduces the "web" of dependencies by requiring that objects rely solely on the mediator [56]. The API endpoints and their contracts are documented using Swagger and based on the OpenAPI specification [46]. In order to maintain the separation between the various API versions, these contracts are housed in a separate project and, most importantly, are not reused.

A project for application services is maintained and harbors code related to business logic. The pattern Command Query Responsibility Segregation (CQRS) which has as its core principle the implementation of two different models—one to read information and another to write it—is applied in this component. This pattern proves useful with complex domains that also benefit from DDD [40]. This service component is divided into domains and sub-domains, and they can contain:

- Commands: There is one for each operation that writes data, and they communicate with it using the database layer;
- Queries: There is one for each read-only operation, and Dapper is used as its Object-Relational Mapping (ORM) [23] to interact with the database directly;
- Commons: classes used to share business logic between queries and commands.

In the database project, Entity Framework Core is utilized to interact with the database, and it does so with the following structure:

- Contexts: One context maps the whole MedicineOne database;
- Entities: Contains a variety of database schemas, each of which contains entities that correspond to database tables;
- Configurations: Each database schema has configuration for matching entities with database tables;

- Repositories: For each domain, there are classes that can fetch, create, update, and remove domain models.

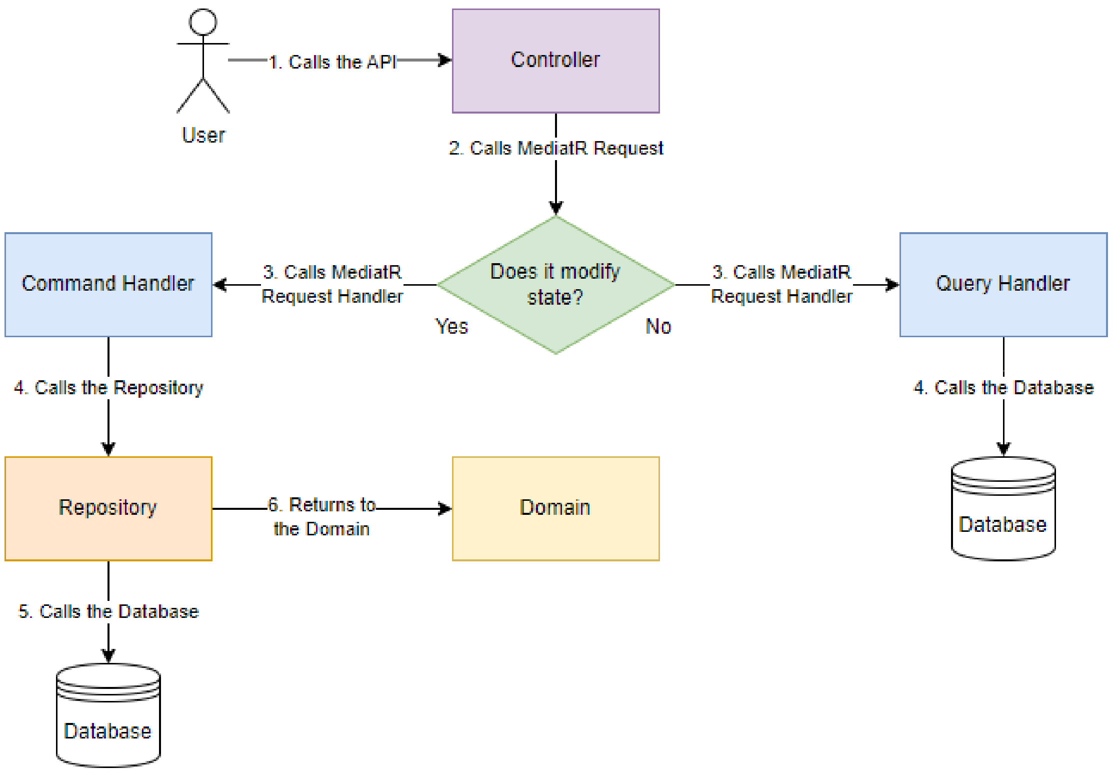


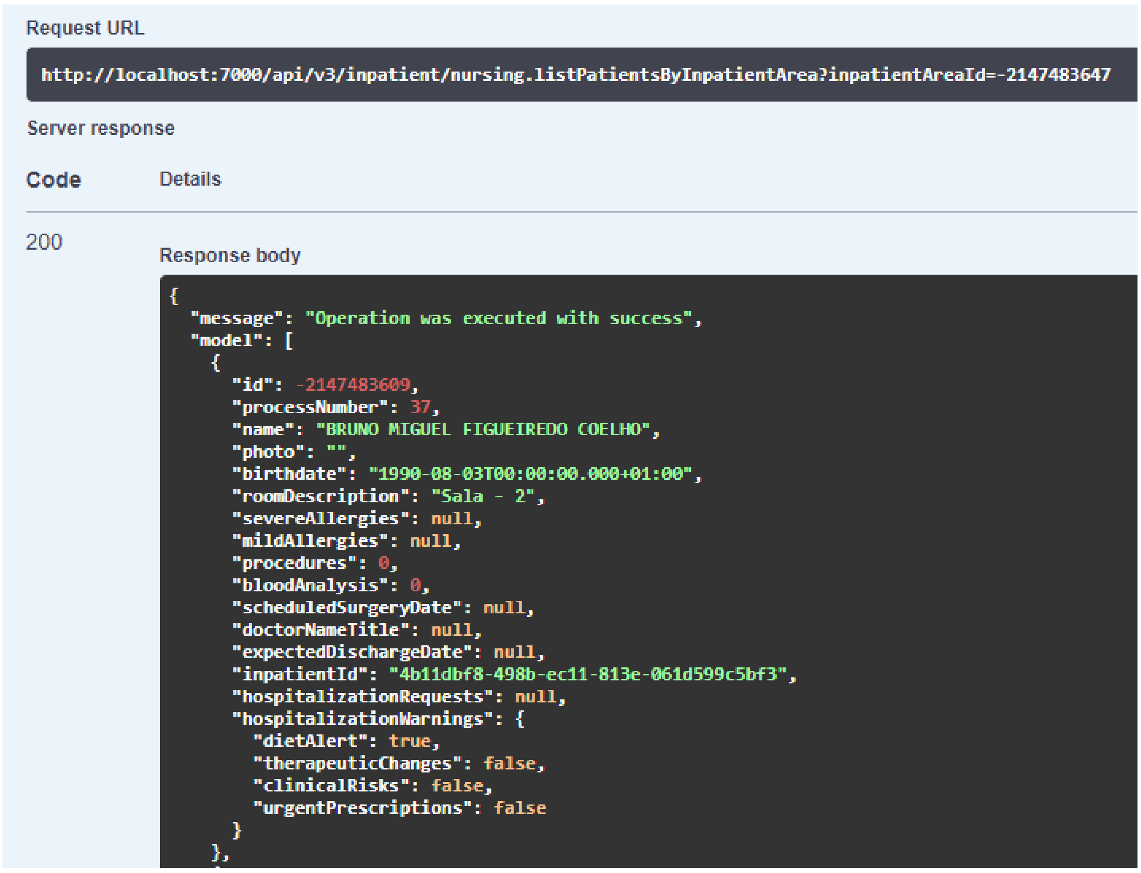
Figure 4.7: A visual representation of the application flow in the MedicineOne API. After a controller is invoked, it is determined whether the request modifies the state of the applications. If it does, a repository accesses the database for the Command; after that, the repository propagates the changes to the domain. If the state won't change, a Query is used to call the database directly.

The infrastructure layer is where the code for utilizing other technologies and external services is located. These include Hangfire [28], which allows the API to run tasks in the background; the MedicineOne legacy services, which serve as a bridge between the API and the server, allowing the reuse of code already on the server-side; and Microsoft's real-time framework SignalR [62], which is used to communicate with MedicineOne's mobile applications.

Additionally, there are projects for writing test code, writing code that is used by each layer that makes up the API, implementing a SDK that can be installed, via NuGet Packet Manager, in other applications to allow access to the API, and updating the primary database and the ASP.NET Identity database.

The MedicineOne API application flow can be see in Figure 4.7.

As a final example, a request and response made with Swagger UI to an endpoint from the API are shown on Figure 4.8. This endpoint accepts the identifier of an inpatient area and returns the patients who are present in that area, along with a variety of other data about them, including personal information, the room in which they are located, allergies, medical precautions and warnings, and a host of other things. This is some of the information that can be displayed on the dashboard prototype.



Request URL

```
http://localhost:7000/api/v3/inpatient/nursing.listPatientsByInpatientArea?inpatientAreaId=-2147483647
```

Server response

Code Details

200 Response body

```
{
  "message": "Operation was executed with success",
  "model": [
    {
      "id": -2147483609,
      "processNumber": 37,
      "name": "BRUNO MIGUEL FIGUEIREDO COELHO",
      "photo": "",
      "birthdate": "1990-08-03T00:00:00.000+01:00",
      "roomDescription": "Sala - 2",
      "severeAllergies": null,
      "mildAllergies": null,
      "procedures": 0,
      "bloodAnalysis": 0,
      "scheduledSurgeryDate": null,
      "doctorNameTitle": null,
      "expectedDischargeDate": null,
      "inpatientId": "4b11dbf8-498b-ec11-813e-061d599c5bf3",
      "hospitalizationRequests": null,
      "hospitalizationWarnings": {
        "dietAlert": true,
        "therapeuticChanges": false,
        "clinicalRisks": false,
        "urgentPrescriptions": false
      }
    },
```

Figure 4.8: An example of a request and response from a MedicineOne API endpoint.

This study of the MedicineOne API sets the knowledge base for the development of the endpoints needed to access the information required to build and organize the dashboard prototype.

4.3 UI design specification

Next, a description of the defined UI design properties that were followed during development, are given, organized by components.

A header that serves as a navigation bar is present and has the following characteristics and elements:

- White background color;
- The current time in the left side with the format “HH:MM”;
- A MedicineOne logo in the middle;

- Buttons for navigation on the right side. The first button has a patients icon, the second a hospital icon, and the third a gear icon. These buttons all have three states: enabled, selected, and disabled, each with a distinct appearance;
- Each element of the header must be vertically and horizontally aligned.

The settings page must contain the following elements:

- An input field for entering the server address;
- A "Connect" button;
- A drop-down menu with an alphabetical list of the organization names. If there is only one organization, it must be selected by default;
- A second drop-down menu with the list of configuration codes of the selected organization, also in alphabetical order. If there is only one configuration, it must be selected by default;
- A “Continue” button.

The patient list selection page must contain the following elements:

- A title portraying the type of the configuration selected;
- A table that lists all the available patient lists, grouped by the "parent" node. These groups can be expanded and collapsed by clicking on the "parent" node. By default, they ought to be expanded.

The patient panel page must have in the upper section a list of indicators with the following characteristics:

- A title with the name of the selected patient list;
- Each indicator should comprised of a 45x35 pixels rectangle with the value of the indicator and another rectangle with the same height of the previous, but with variable width, containing the indicator's description;
- If there are less than 5 indicators, the width of all second rectangles should 300 pixels;
- If there are between 5 and 7 indicators, inclusively, the width of all second rectangles should the total width split equally;
- If there are more than 7 indicators, the width should the same as the previous rule, and the height will reduce to 15 pixels;
- Between each indicator there should be a 10 pixel horizontal spacing;

- The indicators' content should be vertically aligned;
- The content of the first rectangles should be in the center;
- The content of the second rectangle should align on the left side but leaving a 10 pixel spacing;
- When an indicator is selected, there should be a horizontal line below it matching the colors of the rectangles.

Also on the patient panel page there must be a list of patients with the following characteristics:

- A table header with the field names;
- Each table row represents a patient from the patient list;
- Columns of type "text" should have their content aligned to the left side;
- Columns with type "photo", should render them in a round shape, in the center, and with 3 pixels of upper and lower margin;
- For columns of type "image", the size should be according to the number in the column type (16/24/32/48/64), and if there are multiple images in the same cell, a space of 15 pixels should separate them.
- If there isn't any patient on the table, there should be an indicative message with a 15 point size, regular style, and the same color as the content labels.

When a patient is selected, a section of the page is filled with the patient's detailed information, while the patient list shrinks to occupy the remaining width, assuming this component is configured. This section must have the following elements and characteristics:

- A 20 pixels spacing between the detail section and the patient list;
- An universal column for patient identification, present in every configuration on the left side;
- The configured patient detail components on the right side.

The identification column must have the following elements and characteristics:

- A width of 280 pixels;
- An upper square with upper and lower padding of 20 pixels;
- This top square should include the patient's image, which should be round and 130 pixels in diameter; the patient's first and last names, which should be capitalized, bold, and 15 points in size; the color "1; 174; 212" (light blue) in Red-Blue-Green (RGB) values; and the patient's birthdate, which should be written in the following format: "DD-MM-YYYY - XX years";

- These three elements should be horizontally centered and have between them 25 pixels of spacing;
- A rectangular area 20 pixels below the upper square;
- This area contains various pieces of information such as: process number, patient gender, nationality, the attending doctor, and emergency contacts;
- The labels for these patient data should have a point size of 11, be in bold, and have a color corresponding to the RGB values "102; 102; 102" (dark gray);
- The corresponding values should appear directly below the labels;
- A vertical space of 30 pixels should be used to separate each pair of label and value;
- This rectangle's content should have a 10 pixel left margin and be aligned to the left.

The patient detail components must appear on the right side of the identification column with the following characteristics:

- There should be a 10 pixel space below lines that are designated as "using line break" in the configuration before the subsequent column;
- Lines with height set to 0 should expand to occupy the area not used by the other lines;
- Each cell should have a padding of 10 pixels;
- According to the type, the value of each cell should be displayed as in the patient list.

A mock-up of the intended UI design created by MedicineOne is shown in Figure 4.9, and it provides some visual aid to some of the design specifications detailed above, as well as an example of what information can be displayed. In this illustration, we can see details about the internment unit, including the number of patients housed there (selected tab), as well as other filters like those that show patients with upcoming surgeries and those who will be discharged that day. Each patient's row contains details about their room and bed, any allergies they may have, their doctor, and other things. It also provides alerts regarding medication and meals.

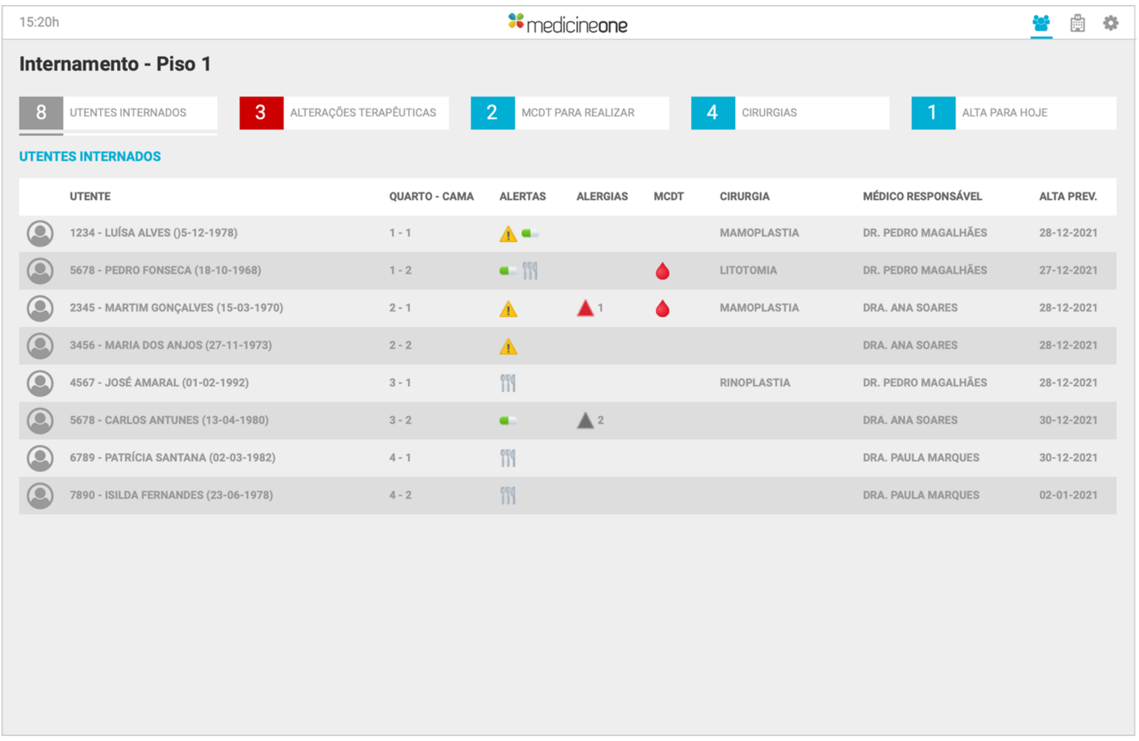
Figure 4.10 depicts a different mock-up that corresponds to the choice of a specific patient. A new tab is opened, and the patient's personal details, a list of their diagnoses, the date of their admission, and other details are displayed. It clearly shows the division of this section with the identification column on the left, and the configured components on the right.

Table 4.7: List of configurable properties of each dashboard component.

Indicators	Description; Left and right background colors; Display order.
Patient list panel	Title; Page background color; Line height; Table header, odd rows, and even rows background colors; Font size; Column value used to order patients; Ascending or descending order of patients; Width occipied by the patient detail section.
Patient list column	Title; Width; If width is fixed or occupies remaining area; Regular, bold, or italic text style; Display order.
Patient detail component	Line height; If uses line break or not; Display order.
Patient detail component column	Width; Regular, bold, or italic text style; Display order.

As was already stated, several appearance properties can be set in the configuration. These properties are listed on Table 4.7 and correspond to fields from the database schema in Figure 4.5.

The window on the main system that creates the configuration for the dashboard application is shown in Figure 4.11. Along with the appearance properties already listed, the data for the patient board and the indicators can be chosen.


15:20h

medicineone

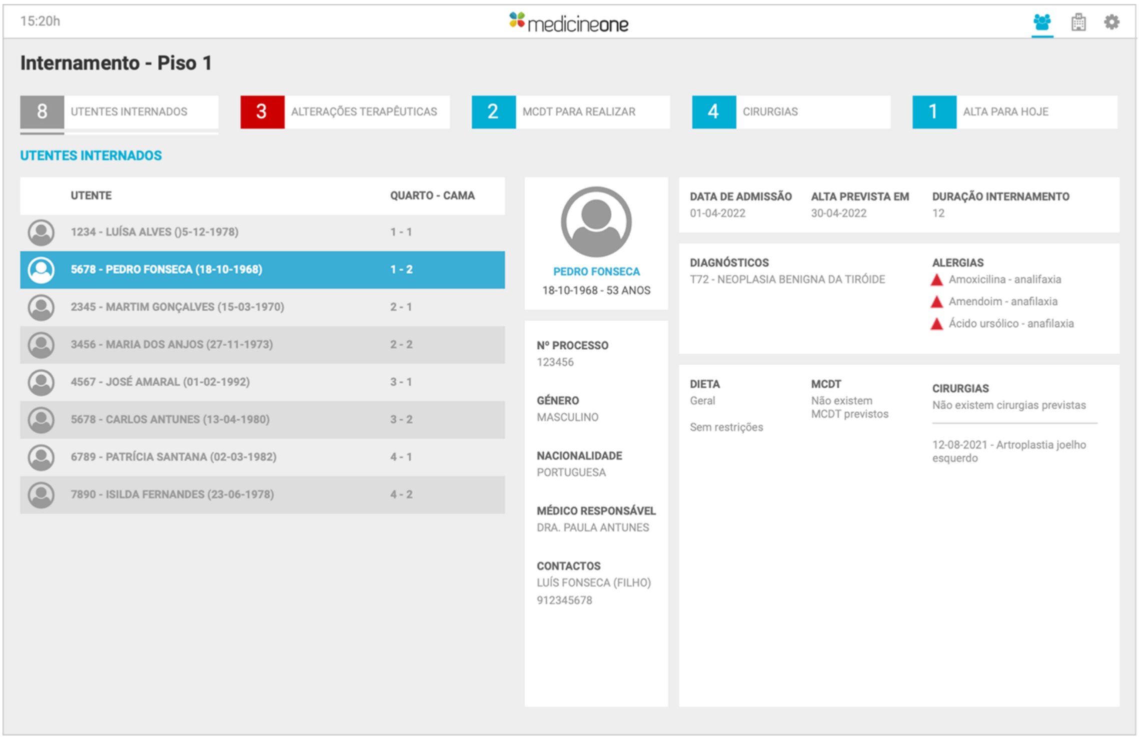
Internamento - Piso 1

8 UTENTES INTERNADOS | 3 ALTERAÇÕES TERAPÊUTICAS | 2 MCDT PARA REALIZAR | 4 CIRURGIAS | 1 ALTA PARA HOJE

UTENTES INTERNADOS

UTENTE	QUARTO - CAMA	ALERTAS	ALERGIAS	MCDT	CIRURGIA	MÉDICO RESPONSÁVEL	ALTA PREV.
1234 - LUÍSA ALVES ()5-12-1978)	1 - 1	⚠			MAMOPLASTIA	DR. PEDRO MAGALHÃES	28-12-2021
5678 - PEDRO FONSECA (18-10-1968)	1 - 2			●	LITOTOMIA	DR. PEDRO MAGALHÃES	27-12-2021
2345 - MARTIM GONÇALVES (15-03-1970)	2 - 1	⚠	▲ 1	●	MAMOPLASTIA	DRA. ANA SOARES	28-12-2021
3456 - MARIA DOS ANJOS (27-11-1973)	2 - 2	⚠				DRA. ANA SOARES	28-12-2021
4567 - JOSÉ AMARAL (01-02-1992)	3 - 1				RINOPLASTIA	DR. PEDRO MAGALHÃES	28-12-2021
5678 - CARLOS ANTUNES (13-04-1980)	3 - 2		▲ 2			DRA. ANA SOARES	30-12-2021
6789 - PATRÍCIA SANTANA (02-03-1982)	4 - 1					DRA. PAULA MARQUES	30-12-2021
7890 - ISILDA FERNANDES (23-06-1978)	4 - 2					DRA. PAULA MARQUES	02-01-2021

Figure 4.9: A mock-up of the dashboard main screen designed by MedicineOne.



15:20h

medicineone

Internamento - Piso 1

8	UTENTES INTERNADOS
3	ALTERAÇÕES TERAPÊUTICAS
2	MCDT PARA REALIZAR
4	CIRURGIAS
1	ALTA PARA HOJE

UTENTES INTERNADOS

UTENTE	QUARTO - CAMA
1234 - LUÍSA ALVES ()5-12-1978)	1 - 1
5678 - PEDRO FONSECA (18-10-1968)	1 - 2
2345 - MARTIM GONÇALVES (15-03-1970)	2 - 1
3456 - MARIA DOS ANJOS (27-11-1973)	2 - 2
4567 - JOSÉ AMARAL (01-02-1992)	3 - 1
5678 - CARLOS ANTUNES (13-04-1980)	3 - 2
6789 - PATRÍCIA SANTANA (02-03-1982)	4 - 1
7890 - ISILDA FERNANDES (23-06-1978)	4 - 2

PEDRO FONSECA
18-10-1968 - 53 ANOS

Nº PROCESSO
123456

GÉNERO
MASCULINO

NACIONALIDADE
PORTUGUESA

MÉDICO RESPONSÁVEL
DRA. PAULA ANTUNES

CONTACTOS
LUÍS FONSECA (FILHO)
912345678

DATA DE ADMISSÃO	ALTA PREVISTA EM	DURAÇÃO INTERNAMENTO
01-04-2022	30-04-2022	12

DIAGNÓSTICOS
T72 - NEOPLASIA BENIGNA DA TIRÓIDE

ALERGIAS
- Amoxicilina - analifaxia
- Amendoim - anafilaxia
- Ácido ursólico - anafilaxia

DIETA
Geral

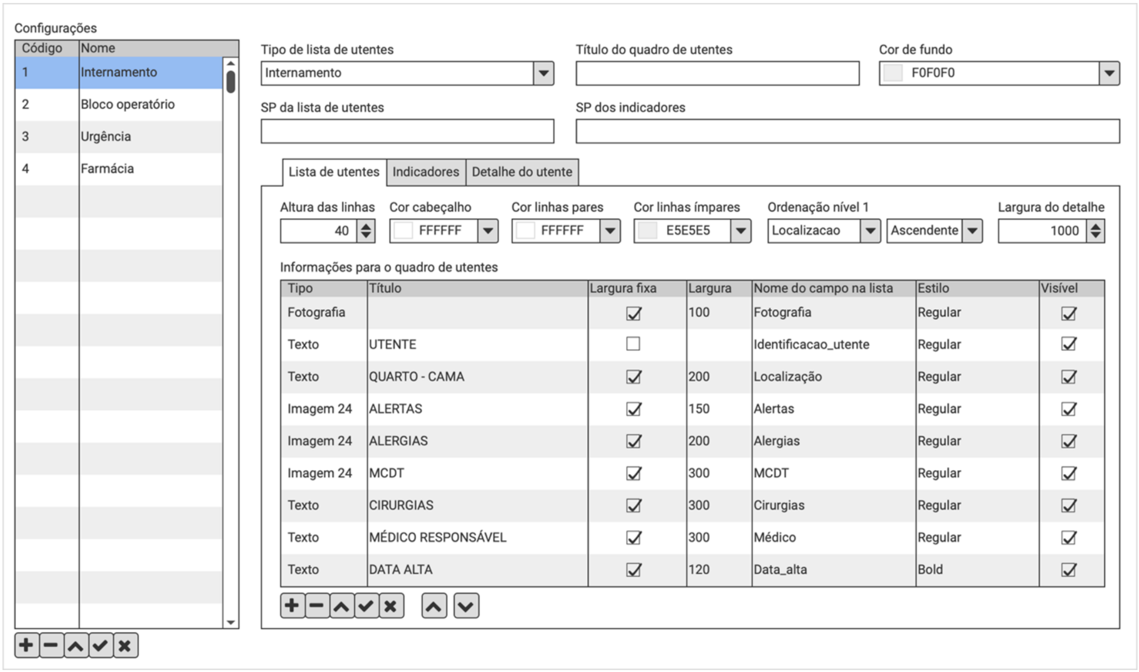
Sem restrições

MCDT
Não existem
MCDT previstos

CIRURGIAS
Não existem cirurgias previstas

12-08-2021 - Artroplastia joelho esquerdo

Figure 4.10: A mock-up of the dashboard patient details screen designed by MedicineOne.



Configurações

Código	Nome
1	Internamento
2	Bloco operatório
3	Urgência
4	Farmácia

Tipo de lista de utentes: Internamento
Título do quadro de utentes:
Cor de fundo: F0F0F0
SP da lista de utentes:
SP dos indicadores:

Lista de utentes | Indicadores | Detalhe do utente

Altura das linhas: 40
Cor cabeçalho: FFFFFF
Cor linhas pares: FFFFFF
Cor linhas ímpares: E5E5E5
Ordenação nível 1: Localizacao | Ascendente
Largura do detalhe: 1000

Informações para o quadro de utentes

Tipo	Título	Largura fixa	Largura	Nome do campo na lista	Estilo	Visível
Fotografia		☑	100	Fotografia	Regular	☑
Texto	UTENTE	☐		Identificacao_utente	Regular	☑
Texto	QUARTO - CAMA	☑	200	Localização	Regular	☑
Imagem 24	ALERTAS	☑	150	Alertas	Regular	☑
Imagem 24	ALERGIAS	☑	200	Alergias	Regular	☑
Imagem 24	MCDT	☑	300	MCDT	Regular	☑
Texto	CIRURGIAS	☑	300	Cirurgias	Regular	☑
Texto	MÉDICO RESPONSÁVEL	☑	300	Médico	Regular	☑
Texto	DATA ALTA	☑	120	Data_alta	Bold	☑

Figure 4.11: The window on the MedicineOne system that generates the configuration for the dashboard.

Chapter 5

Methodology & Planning

The work methodology for this project, as well as its planning, are described in this chapter.

5.1 Methodology

This project was developed using the Scrum methodology, with some deviations occurring during the second semester, which are explained at the end of the methodology portion. Being an agile project management framework, Scrum places a strong emphasis on moving forward in manageable increments of work and uses a continuous process of feedback and inspection to guide the project toward its intended completion. The three main categories of attributes in this framework are Accountabilities, Events, and Artifacts [60]. A visual representation of this framework can be seen on Figure 5.1.

Scrum Accountabilities are the people and roles present in the Scrum Team, which can be [60]:

- Scrum Master: The person who applies Scrum knowledge to maximize the effectiveness of the team and organization. They do this by coaching, teaching, facilitating, and mentoring. In this project the Scrum Master was the dissertation advisor from MedicineOne Fernando Tinoco;
- Product Owner: The member of the Scrum Team who oversees the production of the most valuable product possible. The Product Owner was Bruno Doutor from MedicineOne;
- Developers: The members of the Scrum Team who collaborate to create the product. In this context the intern integrated an engineering team in MedicineOne that provided support specially when working with the described API, but was solely responsible for the tasks related to this project.

The Scrum Events, which are developed to compel regularity and lighten the workload of other meetings, can take the following forms [60]:

- Sprint: The work is completed in sprints, which are brief cycles of one month or less and include all other Scrum Events. Immediately following the conclusion of the prior Sprint, a new one begins. Sprint duration for this project is three weeks each;
- Sprint Planning: activity for organizing the work that will be done during the Sprint;
- Daily Scrum: Every day, this daily event is held where the developers assess their progress toward the sprint goal, identify any obstacles, and make necessary adjustments;
- Sprint Review: An event that takes place at the conclusion of the Sprint, where the Scrum Team and important stakeholders discuss what was accomplished in the Sprint and what changed in their environment;
- Sprint Retrospective: During this meeting, the Scrum Team discusses how the previous Sprint went and determines the most beneficial changes to increase their effectiveness.

Lastly, Scrum Artifacts are the work and plans that are transparent, inspectable, and allow for future adaptation. Each artifact has a Commitment that aids the team in determining whether they are progressing. These artifacts can be [60]:

- Product Backlog: an ever-evolving, structured list of what must be done to enhance the product; it is the Scrum Team's sole source of work. Its Commitment is the Product Goal which is the goal that the group plans to achieve;
- Sprint Backlog: a visible list of tasks that represents the developer's sprint plan and may change as they gain knowledge. Its Commitment is the Sprint Goal, which represents the sole goal of the Sprint;
- Increments: Small pieces of work that act as tangible steps toward the Product Goal. There is no requirement to only release once during a Sprint; deliveries may be as frequent as necessary. This artifact's Commitment is the Definition of Done, meaning the requirements that must be met for an Increment to be deemed complete.

For this project, a Sprint Planning meeting was held on a Tuesday, marking the beginning of a new Sprint, with a duration of about an hour and a half, where priorities for the Sprint were set. It is intended to hold Daily Scrum meetings for at least fifteen minutes to talk about issues and progress. There was a Sprint Refinement meeting every Thursday for about an hour and a half, during which tasks were estimated. The Sprint Review and Retrospective meetings took place on the last Tuesday of the Sprint to discuss what was accomplished during the sprint and how it can be improved for the following, with a duration of about an hour and a half and half an hour, respectively. The Scrum Master was present at all meetings, and only at Daily Scrum meetings was the Product Owner not present.

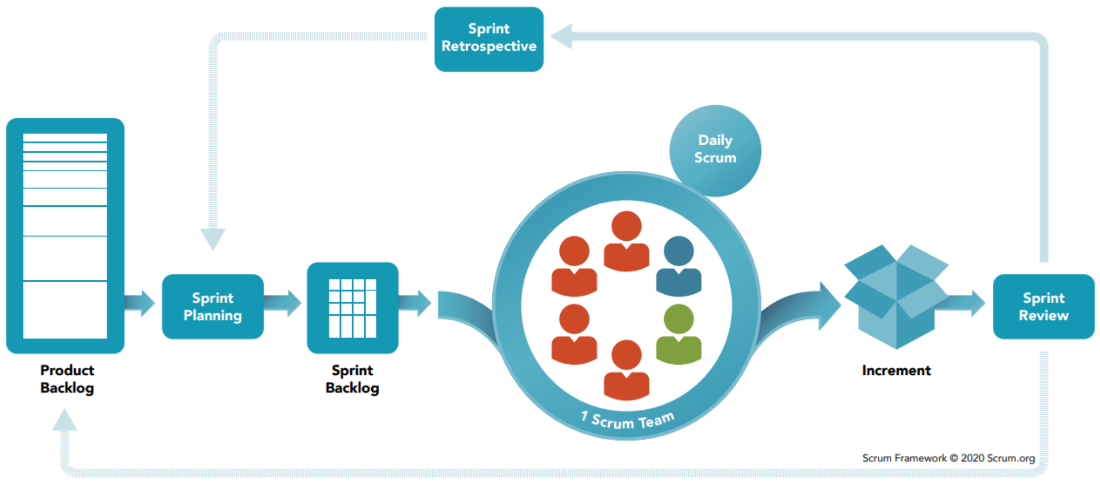


Figure 5.1: A visual representation of the Scrum Framework [60].

5.1.1 Deviancies

In the beginning of the second semester, the intern joined a team that worked on the MedicineOne API, where the specified Scrum methodology was carried out as intended. After about a month, MedicineOne underwent an internal reorganization of teams and departments that resulted in the relocation of the team's members, which led to its dissolution. Consequently, the project methodology was altered. Daily meetings with the Scrum Master and dissertation advisor from MedicineOne continued to be held, but other Scrum Events were replaced with another meeting that took place three times. Along with the Scrum Master, this meeting included the Chief Vision Officer of MedicineOne and the majority of the Project Development team, specifically its Manager, Product Specialist, Designer, and two Product Responsibles. During this meeting, the intern presented the project's progress, provided a demonstration, and asked questions about the project's development. The other attendees to the meeting then offered feedback and clarification.

5.2 Planning

The work plan for this project is divided into two semesters.

In the first semester, tasks that required research and analysis predominated, as well as a brief onboarding training period. These tasks included learning about general dashboard development, then learning about medical dashboards to develop a state-of-the-art. After that, desktop and web-based options for creating a dashboard prototype and real-time communication technologies were examined. The MedicineOne API, which is used to supply data to the dashboard, was also studied. Throughout the semester, the report was intermittently worked on, with an emphasis on the later stages. This process can be seen on Figure 5.2.

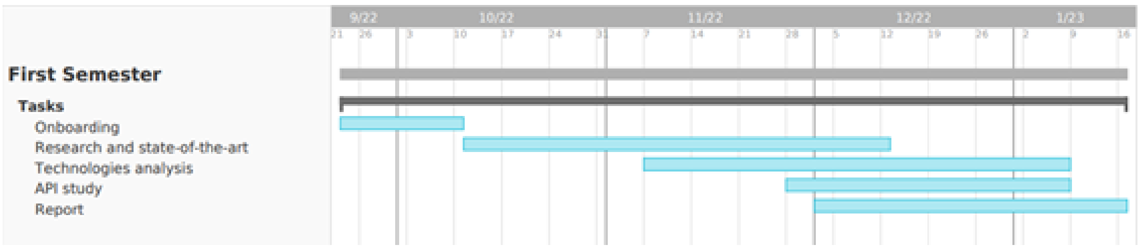


Figure 5.2: The work plan for the first semester.

The work planned for the second semester can be seen in Figure 5.3, and was divided into the following areas: establishing endpoints in the MedicineOne API to retrieve the configuration and the clinical data specified in said configuration; creating the dashboard application that uses the endpoints created to accomplish its goal; establishing the client side of the real-time communication and processing the potential events to be received; testing and correction of bugs found; and the elaboration of the final report.

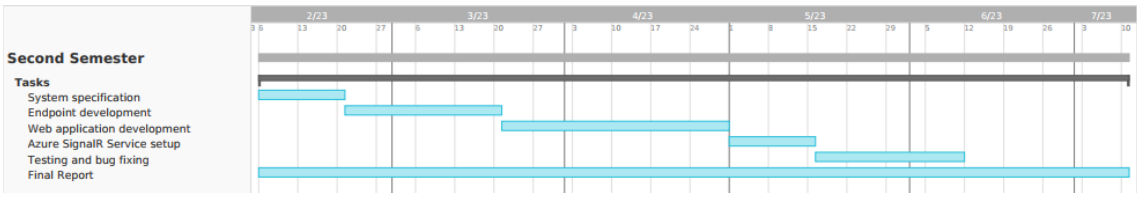


Figure 5.3: The work plan for the second semester.

5.2.1 Deviancies

The second semester's actual work differed slightly from the planning that had been done. It was primarily caused by the patient list selection component of the system not being optimally specified at the outset. This made it necessary to review the specification, create additional endpoints, and write web application code. In general, it had no negative effects on how the project was being developed. This can be seen in the diagram in Figure 5.4.

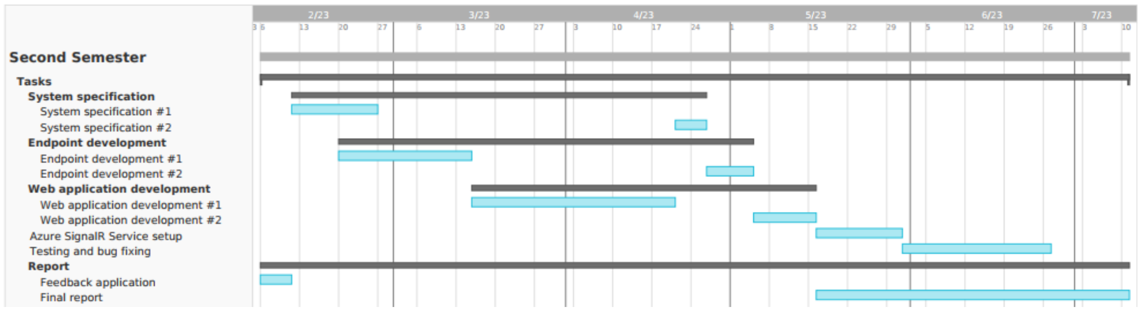


Figure 5.4: The work done in the second semester.

Chapter 6

Implementation

This chapter provides an overview of the implementation process, covering the endpoint creation in the MedicineOne API, the development of the web application's UI and features, and the setup of an Azure SignalR Service instance and the necessary code to ensure real-time communication between the medical system and the dashboard.

6.1 API

This development followed the API structure detailed in Chapter 4. In summary, controllers were created in the "Controllers" directory inside the main project component of the MedicineOne API. Inside the "Contracts" project, the API routes for the endpoints were added, as well as the success and error responses for each endpoint and the definition of the data object compositions to be used in the controllers. The query definitions, their potential error codes, and the handlers that carry out the requests were added to the "Services" project.

In order to support the web application, the API must have endpoints that can retrieve the configuration of the dashboard from the database and run the defined stored procedures to get the data needed to populate the dashboard. As a result, two primary API controllers were developed, one with endpoints to return configuration data and the other with endpoints to execute and get the output of the stored procedures for medical data. Since only a one-way information flow from the server to the client is required, all of the endpoints created for these controllers used the "GET" HTTP method, which is enough to implement the needed functionality. Using the MediatR package and the CQRS pattern, these requests are mapped to classes called queries, which are then passed on to a handler that will execute them and return the result. A third API controller was later created to support the Azure SignalR Service usage, which are described later in Section 6.3.

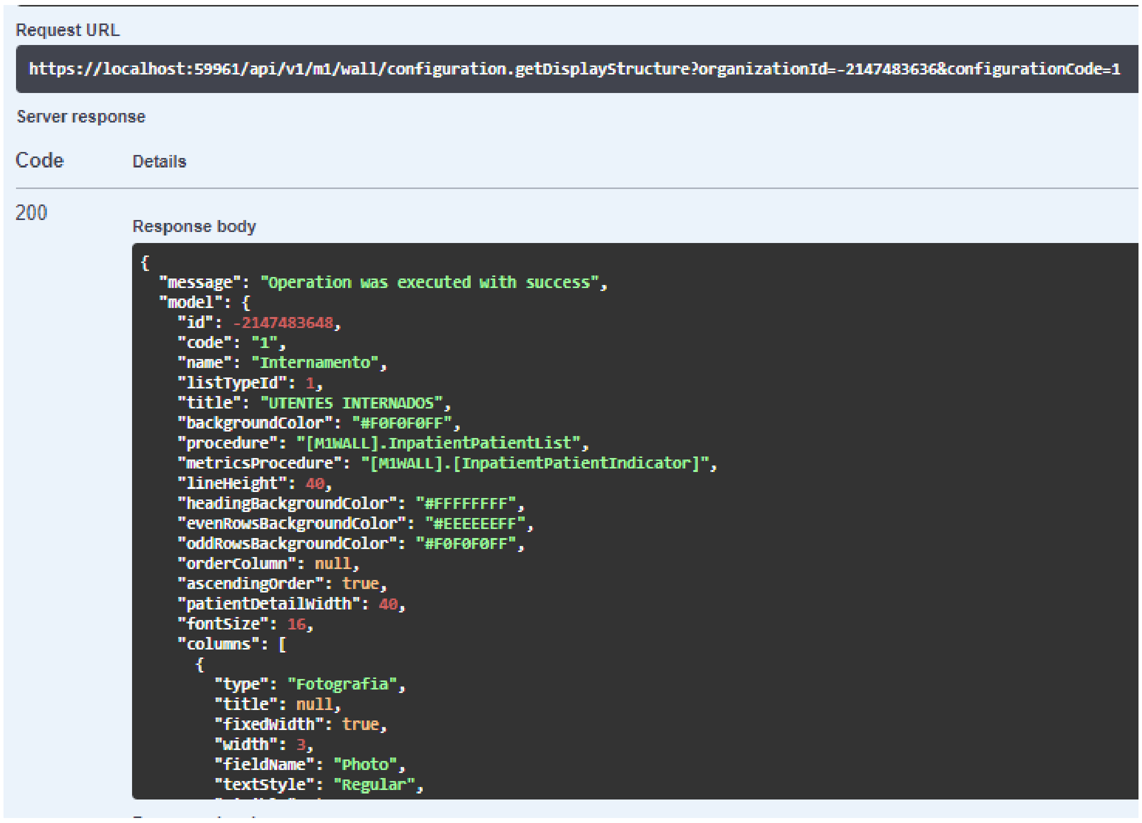
The controllers themselves are not particularly complex because, as stated, they pass on the requests to handlers in the "Services" project. They contain an asynchronous method for each endpoint, on which the HTTP method and the types of responses are defined with annotations. They also state in the parameters the expected type, requirement (mandatory or not), and location of the input values. Each method creates a corresponding query object for a request and sends it through the mediator object; the result is then returned as a success response or an error response, depending on whether an error occurred.

In the following sections, the created API endpoints are described in detail.

6.1.1 Dashboard configuration endpoints

The "GetDisplayStructure" endpoint was created mainly to return all the configuration data for the dashboard in a single API call. It requires two parameters from the caller: an integer representing the organization identifier and a string corresponding to a configuration code. Following that, validations are made on the query handler to see if the organization is present in the system, if the authenticated user has access to it, and if the configuration code matches any configurations that are accessible to that organization. Following these checks, a database query using the Dapper package is run to retrieve data from the "PatientListPanel" and "PatientListType" tables (shown in Figure 4.5) corresponding to the provided configuration code and organization identifier. The configuration information for the indicators, patient panel columns, and patient detail components is retrieved from the other tables using the patient panel identifier if a result is obtained. Finally, all the results are concatenated, but before returning, the fields containing color values are converted to hexadecimal because the format in which they are created in the main MedicineOne system is not easily usable in the web application. An example of a call to this endpoint and its output is shown in Figure 6.1.

Later, two more endpoints were added to the configuration controller to implement the patient list selection. One of the endpoints is designed to return the list of organizations present in the system and their corresponding configuration codes, and the other returns the various patient lists accessible to the previously selected configuration. The first one, called "ListOrganizationsConfigurations" takes no arguments and simply returns all the organization names with the nested configuration codes that the user has access (see Figure 6.2). The second is named "ListSectors" because the patient lists that it returns actually represent the various medical divisions and subdivisions (sectors and sections were chosen as a general nomenclature). It requires both the chosen organization's identifier and the chosen configuration's list type identifier as parameters. Similar to the "GetDisplayStructure" endpoint, the corresponding handler verifies that the organization exists and that the user is authorized to access it. Following that, a switch case is used with the provided type identifier to determine which type is being dealt with, in order to execute the proper query because different patient list types correspond to different medical departments, which are saved in the system differently. Figure 6.3 shows an example call made to this endpoint and its result.



Request URL

```
https://localhost:59961/api/v1/m1/wall/configuration.getDisplayStructure?organizationId=-2147483636&configurationCode=1
```

Server response

Code Details

200 Response body

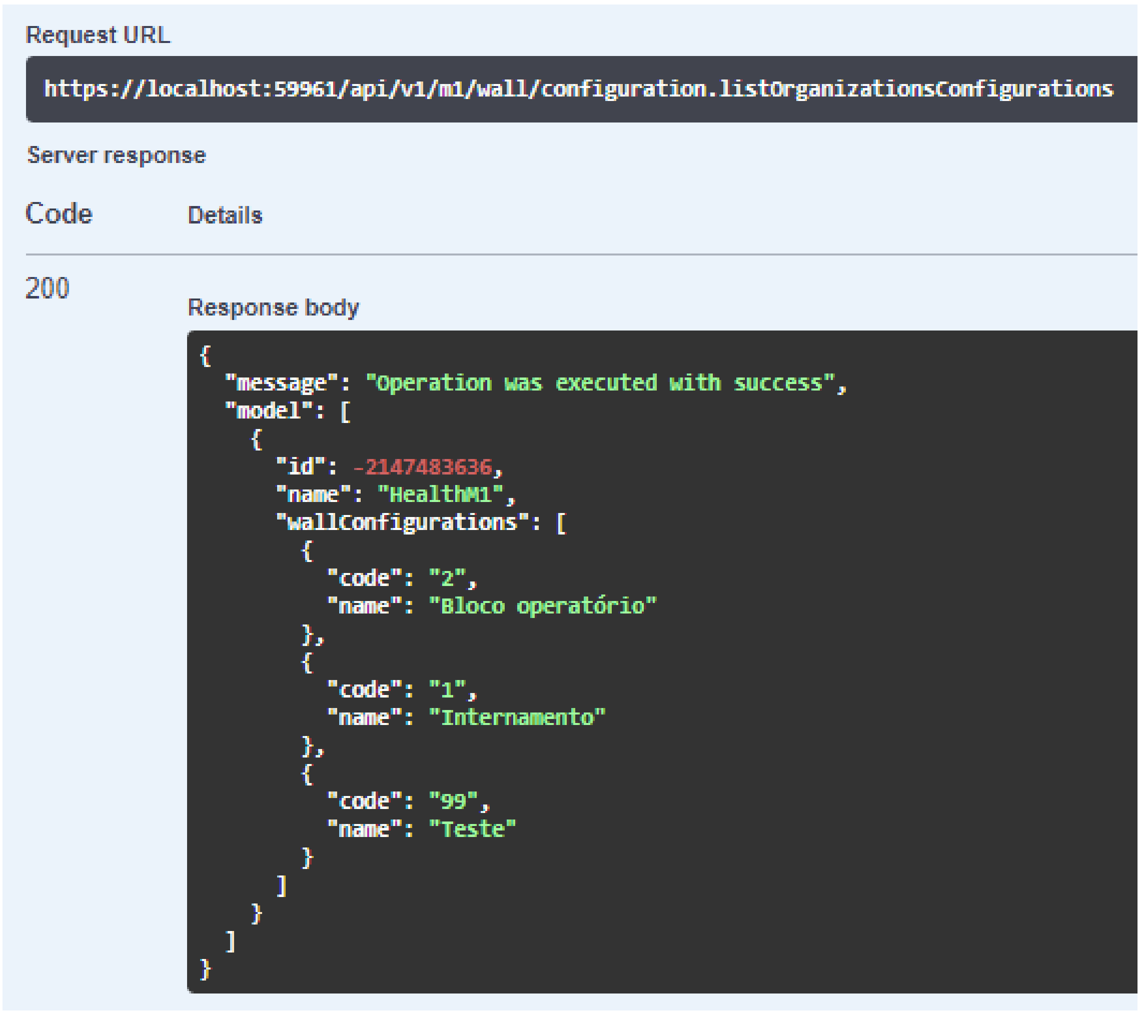
```
{
  "message": "Operation was executed with success",
  "model": {
    "id": -2147483648,
    "code": "1",
    "name": "Internamento",
    "listTypeId": 1,
    "title": "UTENTES INTERNADOS",
    "backgroundColor": "#F0F0F0FF",
    "procedure": "[M1WALL].InpatientPatientList",
    "metricsProcedure": "[M1WALL].[InpatientPatientIndicator]",
    "lineHeight": 40,
    "headingBackgroundColor": "#FFFFFFFF",
    "evenRowsBackgroundColor": "#EEEEEEFF",
    "oddRowsBackgroundColor": "#F0F0F0FF",
    "orderColumn": null,
    "ascendingOrder": true,
    "patientDetailWidth": 40,
    "fontSize": 16,
    "columns": [
      {
        "type": "Fotografia",
        "title": null,
        "fixedWidth": true,
        "width": 3,
        "fieldName": "Photo",
        "textStyle": "Regular",
```

Figure 6.1: An example request and response of the "GetDisplayStructure" endpoint in the Swagger UI.

6.1.2 Dashboard data endpoints

As stated previously, in order to access the indicator, patient, and patient detail data from the database, stored procedures must be executed from the endpoints. Three endpoints were created from these categories, one for each. “ListMetrics” and “ListPatients” are the endpoints that return the indicator data and the patients from the selected list, respectively, and both of them are structured similarly. They both begin by receiving the same arguments: the organization identifier, the name of the configured stored procedure, and a list of section identifiers. This last parameter is a list because, depending on the configuration type, some patient list selections may include more than one medical division, necessitating a fetch from multiple sections. After completing the organization validations inside of each handler, the stored procedures are run, and their results are returned. Examples of calls to these endpoints and their outputs are visible in Figures 6.4 and 6.5.

For obtaining the patient detail data, the endpoint "GetPatientDetail" was created. The arguments it receives are the organization identifier, the selected patient’s identifier, and a list of stored procedure names because, as stated, each patient detail component has a corresponding stored procedure. The query handler first confirms that the given organization exists and that the authenticated user is authorized to access it. Next, it executes the global patient identification stored procedure that was mentioned in Chapter 4 before running the configured



```
https://localhost:59961/api/v1/m1/wall/configuration.listOrganizationsConfigurations
```

```
{
  "message": "Operation was executed with success",
  "model": [
    {
      "id": -2147483636,
      "name": "HealthM1",
      "wallConfigurations": [
        {
          "code": "2",
          "name": "Bloco operatório"
        },
        {
          "code": "1",
          "name": "Internamento"
        },
        {
          "code": "99",
          "name": "Teste"
        }
      ]
    }
  ]
}
```

Figure 6.2: An example request and response of the "ListOrganizationsConfigurations" endpoint in the Swagger UI.

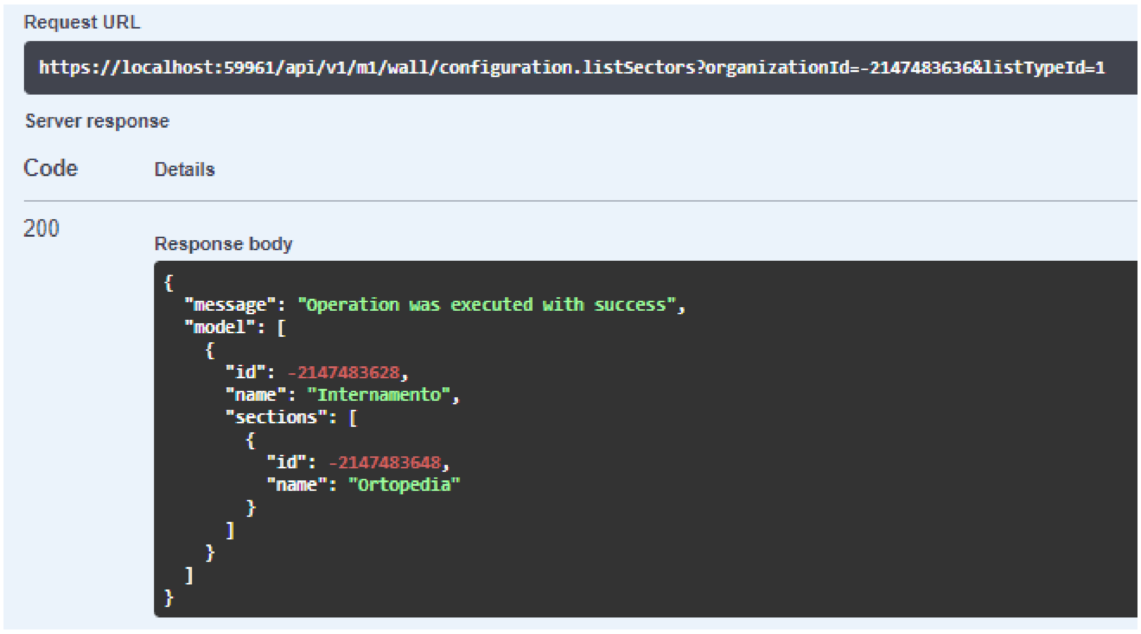
stored procedures. Finally, every output is concatenated and returned. In Figure 6.6 an example call and response to this endpoint can be seen.

6.2 Web application

In this section the web application development is described, starting with the project setup, and then covering relevant development information.

6.2.1 Project setup

To start the web application development, the project was created with Vite, a build tool and development server for modern web applications, using the Node Package Manager (npm) command "**npm create vite@latest**" which fetches the latest version of Vite and create with it a project template [66]. When this command is run, a prompt asks for the preferred framework and variant, from which


Request URL

```
https://localhost:59961/api/v1/m1/wall/configuration.listSectors?organizationId=-2147483636&listTypeId=1
```

Server response

Code Details

200

Response body

```
{
  "message": "Operation was executed with success",
  "model": [
    {
      "id": -2147483628,
      "name": "Internamento",
      "sections": [
        {
          "id": -2147483648,
          "name": "Ortopedia"
        }
      ]
    }
  ]
}
```

Figure 6.3: An example request and response of the "ListSectors" endpoint in the Swagger UI.

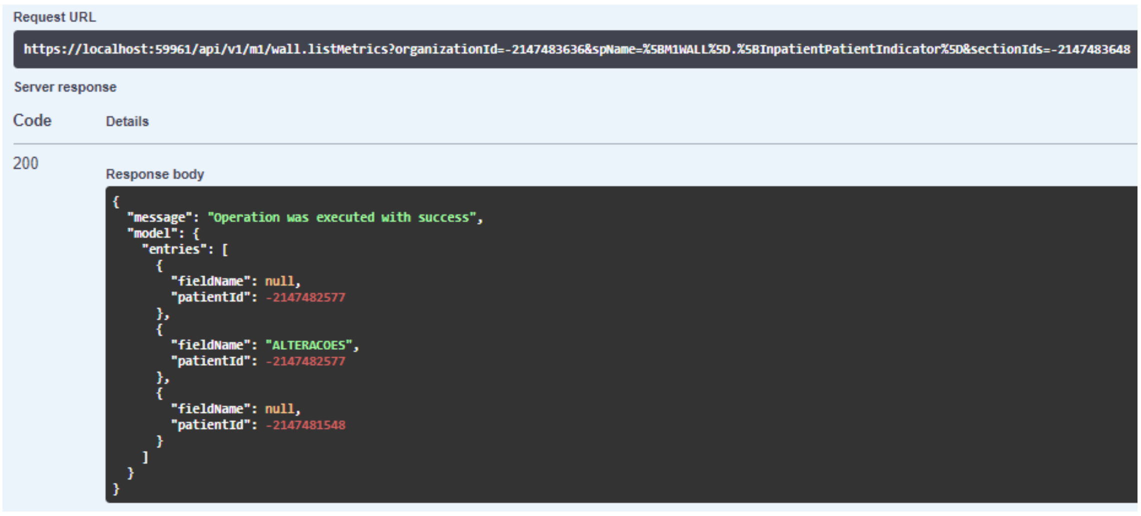
React and JavaScript were selected, respectively. The resulting project structure can be seen in Figure 6.7.

Tailwind CSS and Axios were added right after the project's creation, the first one to aid in the development of the application UI and the second to make HTTP calls [9, 64]. To install Axios, the npm command "**npm install axios**" sufficed. On the other hand, Tailwind was installed alongside PostCSS and Autoprefixer with the command "**npm install -D tailwindcss postcss autoprefixer**", as development dependencies due to the CSS classes being converted to plain CSS when building for production. PostCSS and Autoprefixer were added to allow the use of configurable Tailwind classes and improve browser compatibility, respectively [64]. The command "**npx tailwind init**" was run to setup the Tailwind configuration files, and finally, Tailwind and Autoprefixer were added to the plugin in the PostCSS configuration file.

6.2.2 Development

In order to take advantage of the component capabilities of React, it was examined how to divide the dashboard UI into components after the project was set up. This resulted in the creation of the following files:

- Header.jsx, for implementing the application header;
- Body.jsx, to contain the main content of the application;
- PatientPanel.jsx, to contain the dashboard content;
- MetricList.jsx, to display the configured indicators;



```
https://localhost:59961/api/v1/m1/wall.listMetrics?organizationId=-2147483636&spName=%5BM1WALL%5D.%5BInpatientPatientIndicator%5D&sectionIds=-2147483648
```

```
{
  "message": "Operation was executed with success",
  "model": {
    "entries": [
      {
        "fieldName": null,
        "patientId": -2147482577
      },
      {
        "fieldName": "ALTERACOES",
        "patientId": -2147482577
      },
      {
        "fieldName": null,
        "patientId": -2147481548
      }
    ]
  }
}
```

Figure 6.4: An example request and response of the "ListMetrics" endpoint in the Swagger UI.

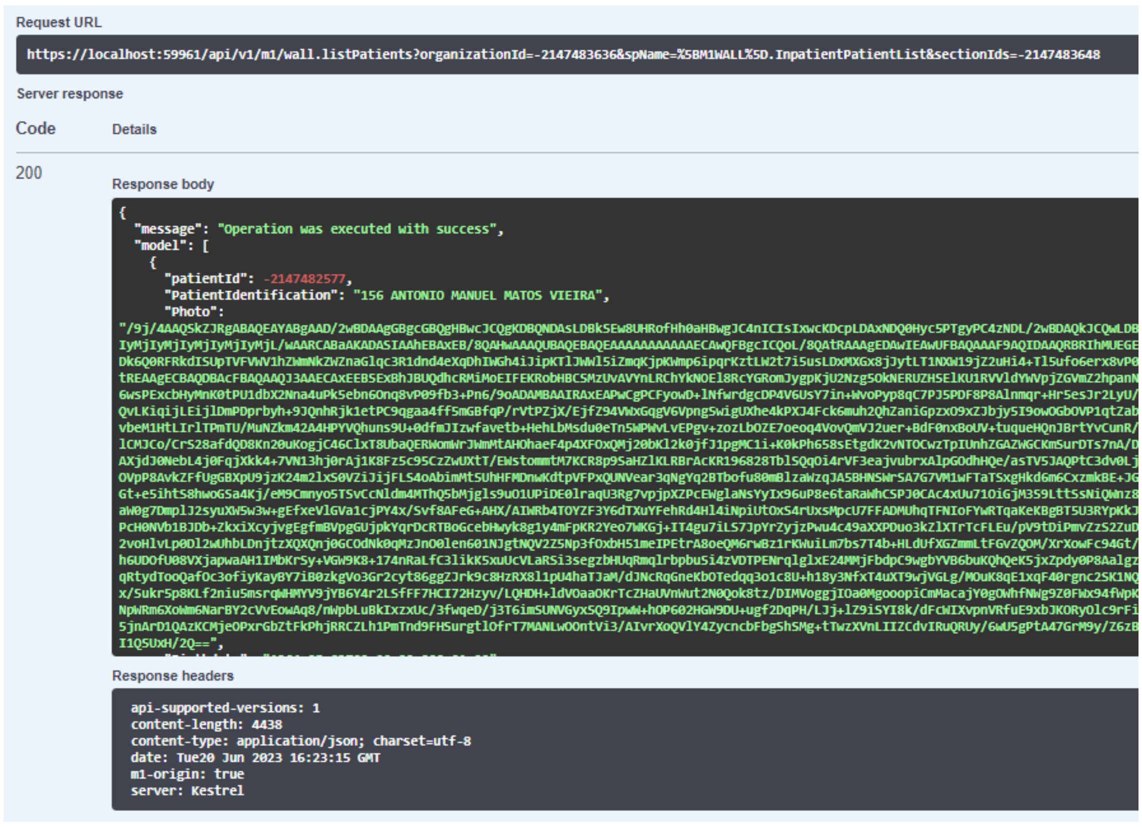
- Metric.jsx, to implement an indicator;
- PatientList.jsx, to display the configured patient list;
- PatientListHeader.jsx, to implement the patient list header;
- Patient.jsx, to implement a patient;
- PatientDetail.jsx, to display the configured patient detail;
- PatientDetailIdentification.jsx, to implement the patient detail identification column;
- PatientDetailComponent.jsx, to implement a patient detail component.

Other components were created for other functionalities, such as the settings page (Settings.jsx), the patient selection page (SectorList.jsx and Sector.jsx), and a loading spinner component (Spinner.jsx). A screenshot of the full components directory can be seen in Figure 6.8.

In order to isolate the component code from newly created reusable functions, the latter were placed inside service files. These include API request functions, browser local storage management, and other utilities.

The use of the browser's local storage was centralized and made simpler by the creation of a service with a named export that contained logic to set, get, and remove items from the local storage.

The logic for the application's authentication to the MedicineOne API was stored in a file called "AuthenticationService.jsx." In it, the authentication server address and credentials are imported from environment variables, and they are used in an Axios call to authenticate the application, obtaining a token. Inside the service, a function called "createAxiosInstance" was developed, which, as the name states, creates an Axios instance. In this instance, the following interceptors are added:



Request URL

```
https://localhost:59961/api/v1/m1/wall.listPatients?organizationId=-2147483636&spName=%5BM1WALL%5D.InpatientPatientList&sectionIds=-2147483648
```

Server response

Code Details

200

Response body

```
{
  "message": "Operation was executed with success",
  "model": [
    {
      "patientId": -2147482577,
      "PatientIdentification": "156 ANTONIO MANUEL MATOS VIEIRA",
      "Photo":
"/9j/4AAQSkZJRgABAQEAYABgAAD/2wBDAAgGBgcGBQgHBwcJCQgKDBQNDAsLDBkSEw8UHRofHh0aHBwgJC4nICIsIxwcKDcpLDAxNDQ0Hyc5PTgyPC4zNDL/2wBDAQkJCQwLDB
IyMjIyMjIyMjIyMjIyMjL/wAARCABaAKADASIAAhEBAxEB/8QAHwAAAQUBAQEBAQEAAAAAAAAAAAECAwQFBgcICQoL/8QAtRAAAgEDAwIEAwUFBAQAAAF9AQIDAAQRBRIhMUEGE
Dk6Q0RFRkdISUpTVFVWV1hZWmNkZWZnaGlqc3R1dnd4eXqDhIWGh4iJipKTlJWWl5iZmqKjpKWmp6ipqrKztLW2t7i5usLDxMXGx8jJytLT1NXW19jZ2uHi4+Tl5ufo6erx8vP0
tREAAgECBAQDBAcFBAQAAQJ3AAECAxEEBSExBhJBUQdhcRMiMoEIFEKRobHBCSMzUvAVYnLRChYkNOEl8RcYGRomJygpKjU2Nzg5OkNERUZHSElKU1RVVldYWVpjZGVmZ2hpanN
6wsPExcbHyMnK0tPU1dbX2Nna4uPk5ebn6Onq8vP09fb3+Pn6/9oADAMBAAIRAxEAPwCgPCFyowD+lNfwrdgcDP4V6UsY7in+WvoPyp8qC7PJ5PDF8P8Alnmqr+Hr5esJr2LyU/
QvLKiqijLEijlDmPDprbyh+9JQnhRjk1etPC9qgaa4ff5mGBfqP/rVtPZjX/EjfZ94VWxGqgV6VpngSwigUXhe4kPXJ4Fck6muh2QhZaniGpzxO9xZJbjy5I9owOGbOVP1qtZab
vbeM1HtLIrlTPmTU/MuNZkm42A4HPYVQhuns9U+0dfmJIzwfavetb+HehLbMsdu0eTn5WPWvLvEPgv+zozLbOZE7oeoq4VovQmVJ2uer+BdF0nxBoUV+tuqueHQnJBrtYvCunR/
lCMJCo/CrS28afdQD8Kn20uKogjC46ClxT8UbaQERWomWrJWmMtAHOhaeF4p4XFOxQMj20bKl2k0jfJ1pgMC1i+K0kPh658sEtgdK2vNTOCwzTpIUnhZGAZWGCKmSurDTs7nA/D
AXjdJ0NebL4j0FqjXkk4+7VN13hj0rAj1K8Fz5c95CzZwUXtT/EWstommtM7KCR8p9SaHZlKLRBrAcKR196828TblSQqOi4rVF3eajvubrxAlpGOdhHQe/asTV5JAQPtC3dv0Lj
OVpP8AvkZFfUgGBXpU9jzK24m2lxS0VZiJijFLS4oAbimMtSUhHFMDnwKdtpVFPxQUNVear3qNgYq2BTbofu80mBlzaWzqJA5BHNSWrSA7G7VM1wFTaTSxgHkd6m6CxzmkBE+JG
Gt+e5ihtS8hwoGSa4Kj/eM9Cmnyo5TSvCcNldm4MThQ5bMjgls9uO1UPiDE0lraqU3Rg7vpjpXZPcEWglaNsYyIx96uP8e6taRaWhCSPJ0CAc4xUu71OiGjM3S9LttSsNiQWnz8
aW0g7DmplJ2syuXW5w3w+gEfxeVlGVa1cjPY4x/Svf8AFeG+AHX/AIWRb4TOYZF3Y6dTXuYFehRd4Hl4iNpiUtOxS4rUxsMpcU7FFADMUhqTFNIoFYwRTqaKeKBgBT5U3RYpKkJ
PcH0NVb1BJDb+ZkxiXcyjvgEgfmBVpgGUjpkYqrDcRTBoGcebHwyk8g1y4mFpKR2Yeo7WKGj+IT4gu7iLS7JpYrZyjzPwu4c49aXXPDuo3kZlXTrTcFLEu/pV9tDiPmvZzS2ZuD
2voHlvLp0Dl2wUhbLDnjtzXQXQnj0GCOdNk0qMzJnO0len601NJgtNQV2Z5Np3fOxbH51meIPEtrA8oeQM6rwBz1rKWuiLm7bs7T4b+HLdUfXGZmmLtFGvZQOM/XrXowFc94Gt/
h6UDOfU08VXjapwaAH1IMbKrSy+VGW9K8+174nRaLfC3likK5xuUcVLaRSi3segzbHUqRmqlrbpbuSi4zVDTPENrqlglxE24MMjFbdpC9wgbYVB6buKQhQeK5jxZpdy0P8AalgZ
qRtydTooQafOc3ofiyKayBY7iB0zkgVo3Gr2cyt86ggZJrk9c8HzRX8l1pU4haTJaM/dJNcRqGneKbOTedqq3o1c8U+h18y3NfxT4uXT9wjVGLg/MOuK8qE1xqF40rgnc2SK1NQ
x/Sukr5p8KLf2niu5msrqWHMYV9jYB6Y4r2LSfFF7HCI72Hzyv/LQHDH+ldVOaaOKrTcZHaUVnWut2N0Qok8tz/DIMVoggjIOa0MgooopiCmMacajY0gOWhfNWg9Z0FWx94fWpK
NpWRm6XoWm6NarBY2cVvEowAq8/mWpbLuBkIxzxUc/3fwqeD/j3T6imSUNVGyxSQ9IpwW+hOP602HGW9DU+ugf2DqPH/LJj+lZ9iSYI8k/dFcWIXvpnVRfuE9xbJKORyOlc9rFi
5jnArD1QAzKCMjeOPxrGbZtFkPhjRRCZLh1PmTnd9FHSurgtlOfrT7MANLwOOntVi3/AIvrXoQVlY4ZycncbFbgShSMg+tTwzXVnLIIZCdvIRuQRUy/6wU5gPtA47GrM9y/Z6zB
I1QSUxH/2Q==",
```

Response headers

```
api-supported-versions: 1
content-length: 4438
content-type: application/json; charset=utf-8
date: Tue20 Jun 2023 16:23:15 GMT
m1-origin: true
server: Kestrel
```

Figure 6.5: An example request and response of the "ListPatients" endpoint in the Swagger UI.

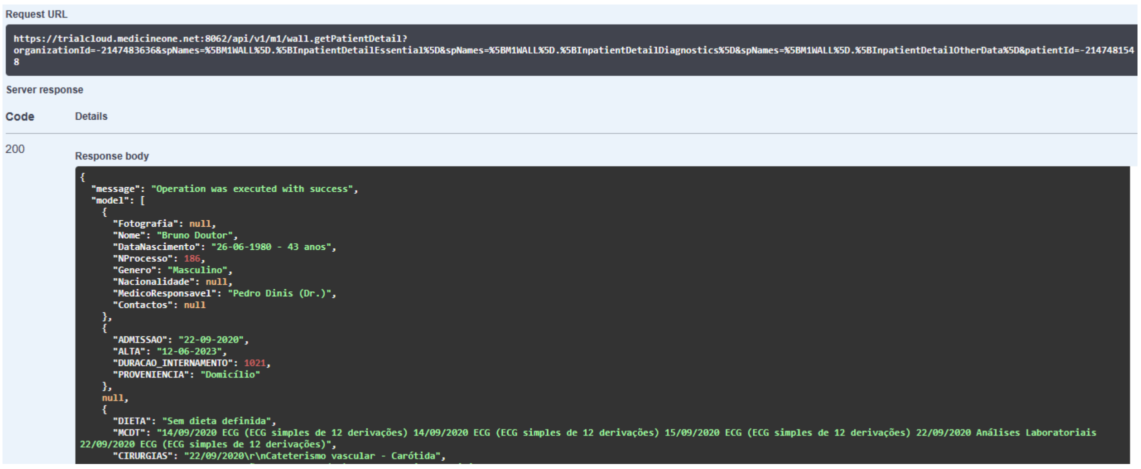
- An interceptor for the requests that gets the token from the local storage and adds it to the request's authorization header, if the token exists;

- An interceptor that calls the previous authentication function in error responses with status code "401 Unauthorized". In the event that authentication is successful, the interceptor saves the token in the local storage of the browser, adds it to the failed request header in a manner similar to the first interceptor, and reroutes the call. Authentication errors are retried once more, and if they happen again, they are simply returned.

The address is set on application startup if it is present on the browser's local storage or when one is provided in the settings page. This instance is then imported and used throughout the application whenever a need to request data from the API arises.

Two services were made for the API calls: “WallConfigurationService.jsx” for the configuration controller and “WallService.jsx” for the dashboard data controller. Both services import the Axios instance built in the prior service to implement a function for each endpoint that returns a JavaScript promise of the request, which can then be resolved by the application components to acquire the data. Error handling was enforced on the usage of these services, in case the fetching of information fails.

Additionally, a function was developed to parse fields of type image, which, as



Request URL

```
https://trialcloud.medicineone.net:8062/api/v1/m1/wall.getPatientDetail?
organizationId=-2147483636&spNames=%5BM1WALL%5D.%5BInpatientDetailEssential%5D&spNames=%5BM1WALL%5D.%5BInpatientDetailDiagnostics%5D&spNames=%5BM1WALL%5D.%5BInpatientDetailOtherData%5D&patientId=-214748154
8
```

Server response

Code Details

200 Response body

```
{
  "message": "Operation was executed with success",
  "model": [
    {
      "Fotografia": null,
      "Nome": "Bruno Doutor",
      "DataNascimento": "26-06-1980 - 43 anos",
      "NProcesso": 186,
      "Genero": "Masculino",
      "Nacionalidade": null,
      "MedicoResponsavel": "Pedro Dinis (Dr.)",
      "Contactos": null
    },
    {
      "ADMISSAO": "22-09-2020",
      "ALTA": "12-06-2023",
      "DURACAO_INTERNAMENTO": 1021,
      "PROVENIENCIA": "Domicílio"
    },
    null,
    {
      "DIETA": "Sem dieta definida",
      "MCDT": "14/09/2020 ECG (ECG simples de 12 derivações) 14/09/2020 ECG (ECG simples de 12 derivações) 15/09/2020 ECG (ECG simples de 12 derivações) 22/09/2020 Análises Laboratoriais
22/09/2020 ECG (ECG simples de 12 derivações)",
      "CIRURGIAS": "22/09/2020\r\nCateterismo vascular - Carótida",
```

Figure 6.6: An example request and response of the "GetPatientDetail" endpoint in the Swagger UI.

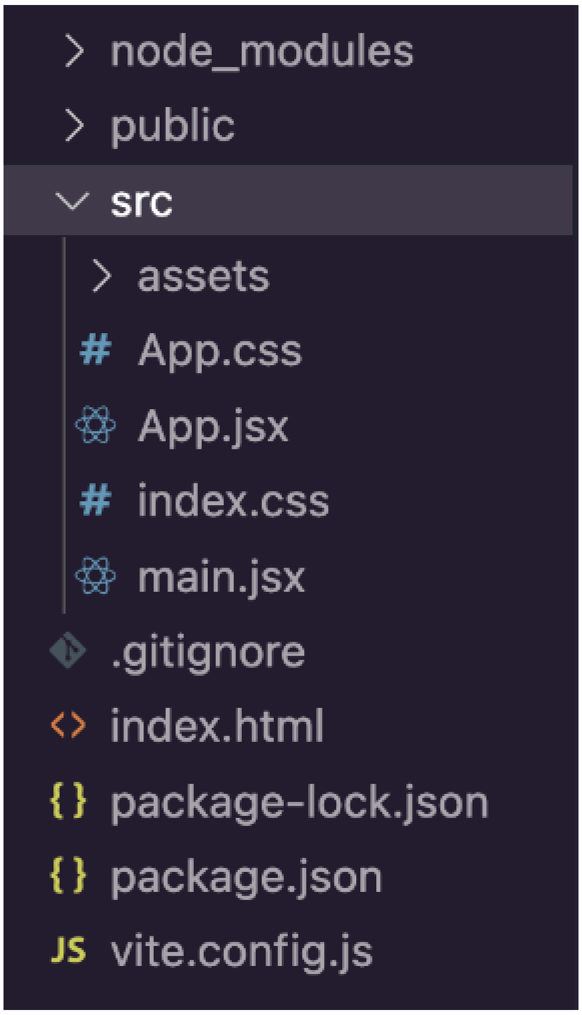
stated in Chapter 4, contain a string that can represent multiple images and text labels in a particular format. As this was needed for more than one application component, the patient list and the patient detail components, it was extracted to a service.

The application's routing, which was carried out using the React Router package, is a relevant detail to address. The package used supports nested routing, client-side page routing, and other features [55]. On the first routing level, the settings and wildcard routes are specified. The first maps the path "/settings" to the corresponding page, while the second one is used to ensure the settings are present before accessing other pages. In order to accomplish this, the element is configured to render the return of a function that, if all settings are defined, returns the component with the second level of routing; otherwise, it redirects the user to the settings page. The second level of routing operates similarly to the first by mapping the path "/selectList" to the appropriate page and using the wildcard route to make sure that a list is selected. This validation employs the same method. The element that needs to be rendered is the output of a function that, if a patient list is chosen, returns the patient panel component; otherwise, it redirects the user to the page where they can choose a patient list. In Figure 6.9 a diagram illustrates the routing described.

As was stated, the appearance of the application must adjust to the configuration information obtained through the API. Therefore, to apply the incoming properties, a couple methods were used, such as: the HTML "style" attribute, which was mainly used to set the width, height, and background color of elements, with some cases logic expressions being used; string interpolation in the Tailwind CSS "classNames" attribute, which was used primarily to set classes like visibility and text style, among others.

During development, a number of React functionalities that are considered best practices were used. The following practices can enhance the maintainability, reusability, and overall quality of the codebase:

- As was demonstrated, the application was divided into various compo-



```
> node_modules
> public
v src
  > assets
  # App.css
    App.jsx
  # index.css
    main.jsx
  .gitignore
<> index.html
{} package-lock.json
{} package.json
JS vite.config.js
```

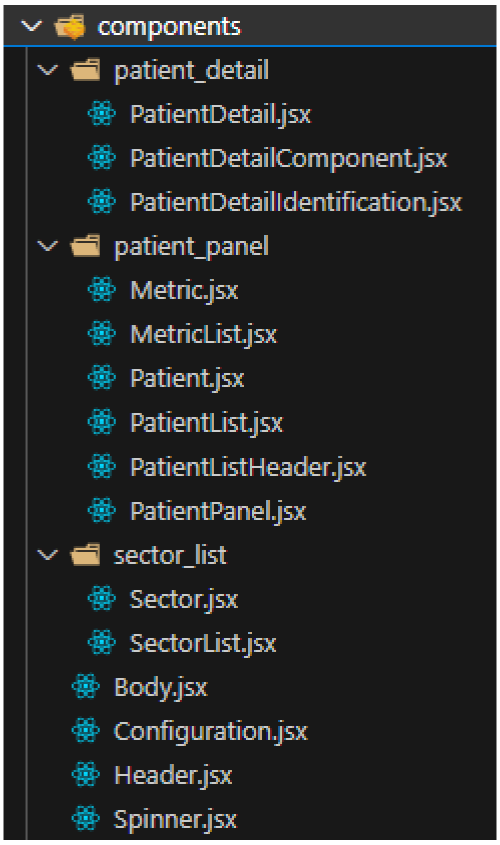
Figure 6.7: The initial Vite project structure.

nents, each of which was responsible for rendering a different section of the "UI;

- React's state management was utilized to manage and distribute data among components, specifically to share configuration and dashboard data with the in need components;
- The created components are all functional, meaning they encourage the use of React hooks for state management and lifecycle methods, for instance;
- To ensure predictable behavior, the state and props of the component were largely kept immutable by creating new instances if the values were to be updated.

6.3 Azure SignalR Service

To utilize the Azure SignalR Service in the system, several steps had to be taken. First, an instance of this service was created on Microsoft Azure. This instance



```
components
  patient_detail
    PatientDetail.jsx
    PatientDetailComponent.jsx
    PatientDetailIdentification.jsx
  patient_panel
    Metric.jsx
    MetricList.jsx
    Patient.jsx
    PatientList.jsx
    PatientListHeader.jsx
    PatientPanel.jsx
  sector_list
    Sector.jsx
    SectorList.jsx
  Body.jsx
  Configuration.jsx
  Header.jsx
  Spinner.jsx
```

Figure 6.8: The components directory structure.

was created in serverless mode, which, as the name suggests, does not need or allow for server connections. As it is advised when using a serverless instance, the package "Microsoft.Azure.SignalR.Management" was installed in the API project to set up the message transmission [10]. This strategy was chosen because it was confirmed that, in contrast to other approaches, it did not obstruct MedicineOne's API use of the non-cloud based SignalR.

First, a service was built and named “SignalRService” in the corresponding project folder of the API. Its function is to establish and maintain the hub context, a management package object that represents the connection to the Azure SignalR Service hub and enables interaction with it. There were three methods created in it: one to start the service and create the hub context instance, one to dispose of the instance, and one to stop the service. A service manager object is created and configured with the connection string for the Azure SignalR Service instance when the service is started, and it is then used to create the hub context. The hub context is then saved on an interface that was developed to act as a store for injection when necessary. Finally, since only one instance of each is required, the service and the hub context interface are registered as singletons. The first is also registered as a hosted service that is started when the API starts and as the hub context interface's implementation.

Following the initial configuration, a controller was implemented similarly to the others but with an injection of the hub context store. This controller contains endpoints to send messages and for the negotiation process, which is a necessary

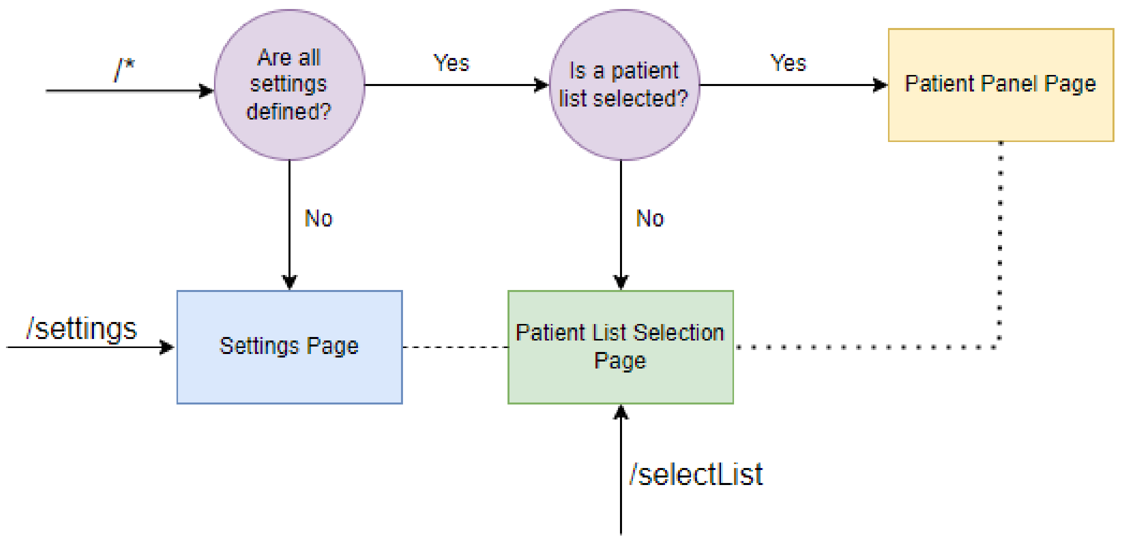
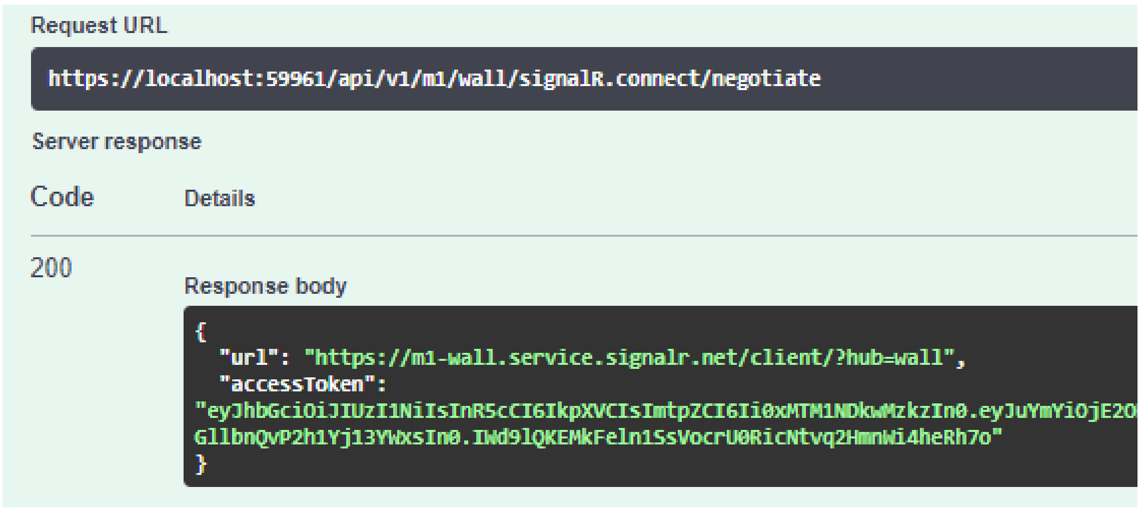

/*
Are all settings defined?
Yes
Is a patient list selected?
Yes
Patient Panel Page
No
No
/settings
Settings Page
Patient List Selection Page
/selectList

Figure 6.9: A visual representation of the application routing.

process to begin communication with a new client. No query, command, or handler classes were developed for these endpoints because they do not access the database. The negotiation process is done by sending a HTTP POST request to obtain the connection identifier and available transport protocols that the client needs to create the connection object. A request may also include the preferred transport protocol, and if the host of the negotiation endpoint requests one, an access token may also be needed. Authentication is required in this instance because this endpoint was incorporated into the MedicineOne API.

The negotiation endpoint was named "Negotiate", and it acts as a middleman in the negotiation process. After receiving the client's request, it invokes the "NegotitateAsync()" method from the SignalR management package, which negotiates on the client's behalf and redirects it to the Azure SignalR instance by returning the address of that instance and an access token (see Figure 6.10). The client then sends a second request, this time to the specified address, along with the token to prove it is authenticated. It receives the already mentioned connection identifier and the available communication protocols from the Azure SignalR Service, concluding this process. Figure 6.11 illustrates this process.

Four endpoints were made specifically for message sending that will be called from the MedicineOne main system to notify the dashboard clients of events. They all function similarly, taking parameters from the request body and sending them to all Azure SignalR clients using the corresponding method (topic) through the hub context. One employs the method "ConfigurationUpdate," which is invoked whenever changes are made to the configuration data. The message includes the updated configuration's code so that dashboards using it can update their visuals without having to be restarted. The others use the methods "PatientAdmission," "PatientDischarge," and "PatientUpdate" to alert clients to changes in the dashboard data itself. The first receives and sends to the Azure SignalR instance the identifier of the patient's location in the hospital, while both the latter receive and send the patient's identifier.


Request URL

```
https://localhost:59961/api/v1/m1/wall/signalR.connect/negotiate
```

Server response

Code	Details
200	Response body

```
{
  "url": "https://m1-wall.service.signalr.net/client/?hub=wall",
  "accessToken":
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCIsImtpZCI6Ii0xMTM1NDkwMzkzIn0.eyJuYmYiOjE2O
GllbnQvP2h1Yj13YWxsIn0.IWd9lQKEMkFeln1SsVocrU0RicNtvq2HmnWi4heRh7o"
}
```

Figure 6.10: An example request and response of the "Negotiate" endpoint in the Swagger UI.

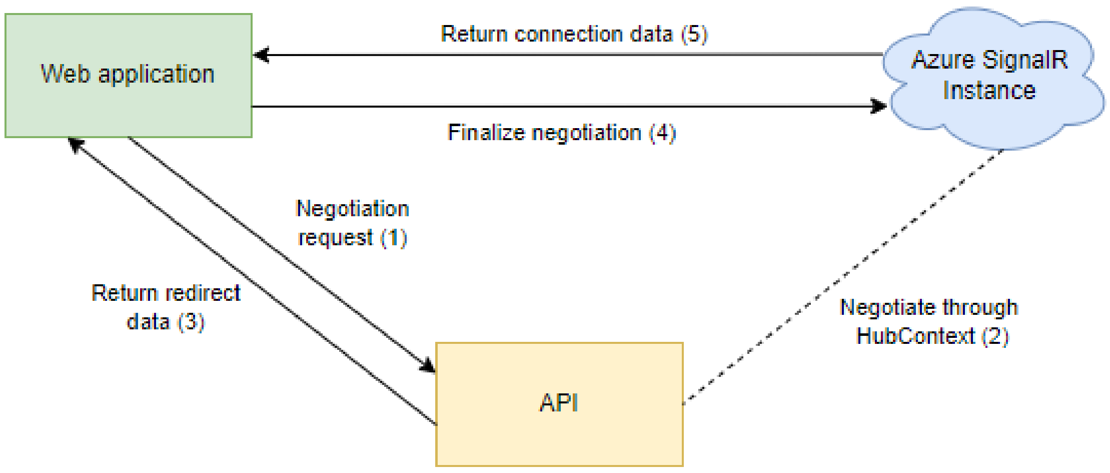


Figure 6.11: A visual representation of the client negotiation process with the Azure SignalR instance.

Moving on to the web application, first the package “@microsoft/signalr” was installed. A new method called "connectSignalR" was created for the authentication service, and it is in charge of connecting to the instance using the hub connection builder from the previous package. The address for the newly created negotiation endpoint and the preferred transport protocol, SSE, were both specified in this builder. As was mentioned in Chapter 3 this protocol fits in well with the preferred communication approach, which is a one-way channel that only allows clients to receive event notifications. There is also a parameter in the builder called "accessTokenFactory" that supplies the token kept in local storage for the API authentication, and the builder method for automatic reconnection in case of network or server error was also added. Finally, this function returns the newly created hub connection object. It is called in the component code with an interval retry method, and when the connection is obtained, it is started with the corresponding method.

Next, handlers for the methods used to send the notifications must be set up in order to receive messages from the Azure SignalR instance. The "on" method of the hub connection object, with the parameters being the method and the handler function, is used to carry out this action. This is applied to two separate components: the body component, which sets a handler for the "ConfigurationUpdate" method, and the patient panel component, which sets handlers for the dashboard data methods. In the first handler, a simple check is made to see if the configuration code in the message matches the configuration that was chosen for this instance of the application, and if it does, a new request for its data is made using the API. In a similar manner, the "PatientAdmission" method handler requests the indicator and patient list data if the location identifier of the new patient matches the patient list that has been selected for the dashboard. The "PatientUpdate" method handler makes the same requests to the API, but only if the patient identifier in the message matches one of the patients who are listed. Furthermore, if the updated patient is currently selected and has their detailed information displayed, a request is also made for that data. Last but not least, the "Patient-Discharge" method handler checks to see if it contains the patient matching the provided identifier, but since it relates to a discharge from the hospital, it just removes the patient record from the list and its indicator entries, and if the patient was selected, it also closes the detail information component. If any of these handlers perform an action that is not intended for the patient who is currently being selected (if there is one), the patient detail component will not be closed.

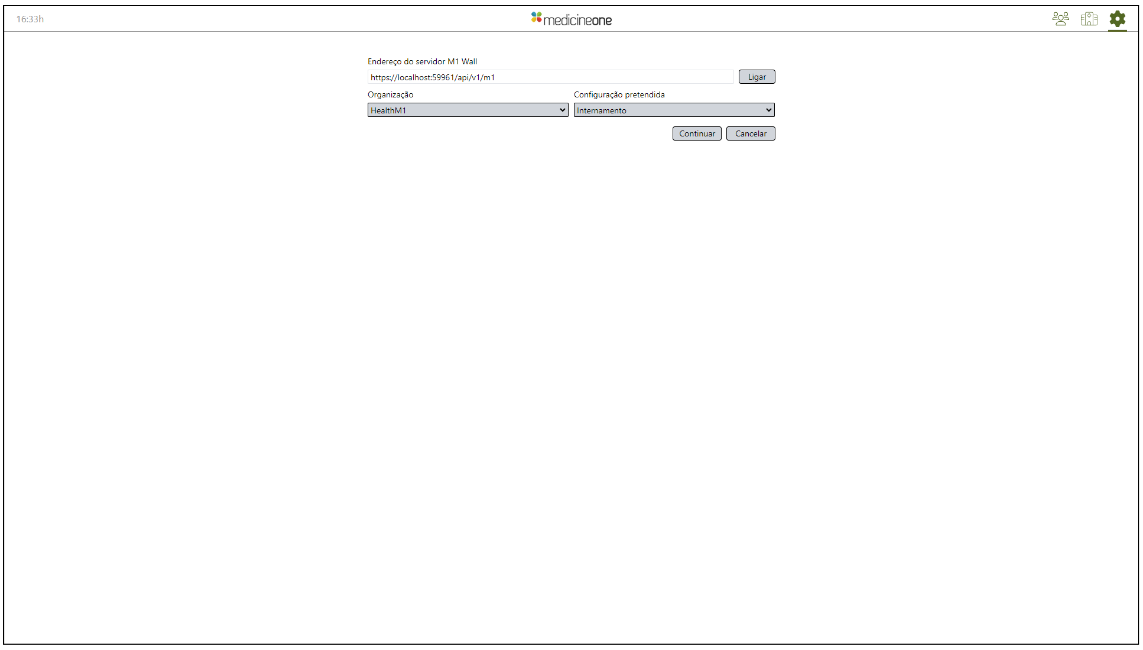
Because each dashboard may be unique, contain a variety of medical data, and name each field differently, there are limitations on how messages can be sent. For instance, even if a dashboard lists this patient and contains the exact fields of information, there is no guarantee that the fields will have the same names if we send an updated patient record. As a result, the strategy of sending the patient identifier was selected since it is required that this field exist with a consistent name in every configuration, even if it is not displayed. The same applies to the patient's location identifier, which, at least for the selected patient list, will be present in every dashboard.

The implementation that is required on the MedicineOne server to send notifications when specific events occur is not included in the dissertation's scope, so only the configuration update was fully implemented by MedicineOne to assert that it communicated properly with the Azure SignalR instance. As a result, timers were set up on each endpoint call for dashboard data so they could periodically refetch it as a temporary measure.

6.4 Result

In this section, screenshots of the web application are shown for three different dashboard configurations, which show how the UI and its data can change by simply changing configurations.

The settings page is the first page shown when launching the application for the first time. As previously mentioned, a server address, an organization, and a configuration are prompted before continuing. This page can be seen in Figure 6.12.



Endereço do servidor M1 Wall: https://localhost:59961/api/v1/m1

Organização: HealthM1

Configuração pretendida: Internamento

Figure 6.12: A screenshot of the settings page.

The patient list selection page is displayed following completion of the settings page's required inputs. The background color and list attributes are already taken into account on this page thanks to the previously chosen configuration. It displays the available lists for each configuration obtained through the API, which vary depending on the configuration type selected.

In Figures 6.13, 6.14, and 6.15 this page can be seen for an inpatient unit, operating block, and emergency room configurations, respectively.

Every example allows for the collapsing and expanding of lists by clicking on the parent node, but only the final two allow for the selection of the parent node, which selects all of the lists it contains, this is due to the configuration type, as was stated in Chapter 4.

After a selection is made, the dashboard page is displayed, with the patients and indicators listed according to the list that was chosen and the data from the selected configuration applied to the UI. All the information is obtained by calling on the developed endpoints that fetch it from the database, using the defined stored procedures for the patient and indicator data. This information is then updated on receiving notification messages from the Azure SignalR Service.

The different configured dashboards can be seen in Figures 6.16, 6.18, and 6.19.

Additionally, in Figure 6.17 an indicator was selected and consequently applied a filter to the patient list of the first dashboard.

Finally, by selecting a patient from the list, the patient detail component is opened and populated with data. Figures 6.20, 6.21, and 6.22, show the different appearances and information fields on the three dashboards.

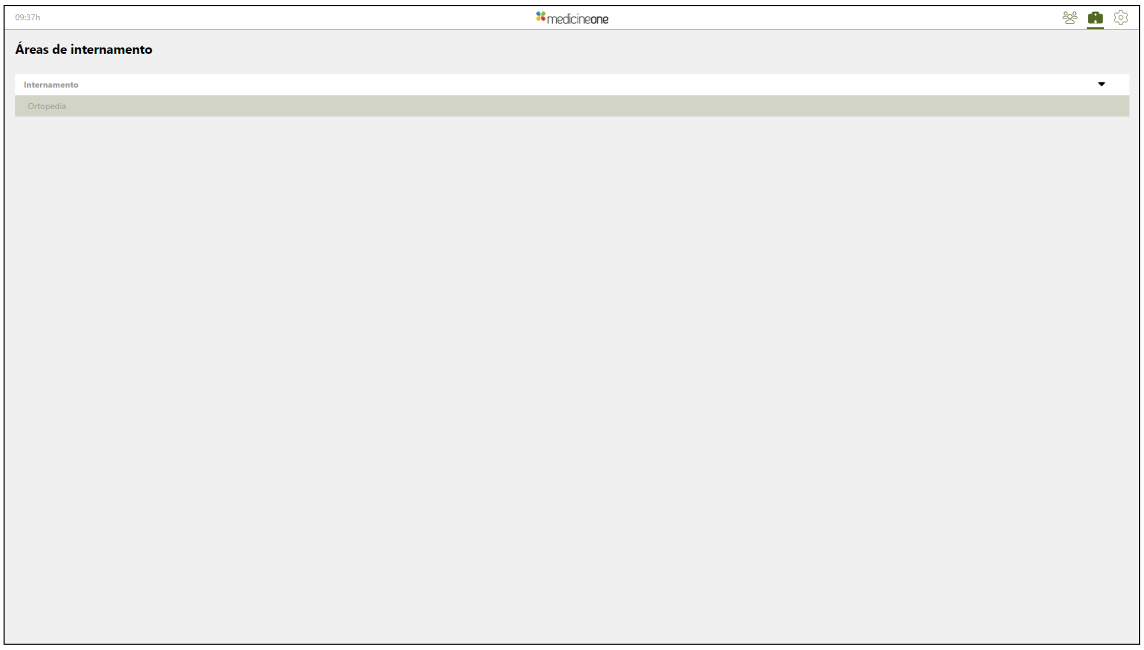


Figure 6.13: A screenshot of the patient list selection page for an inpatient unit configuration.

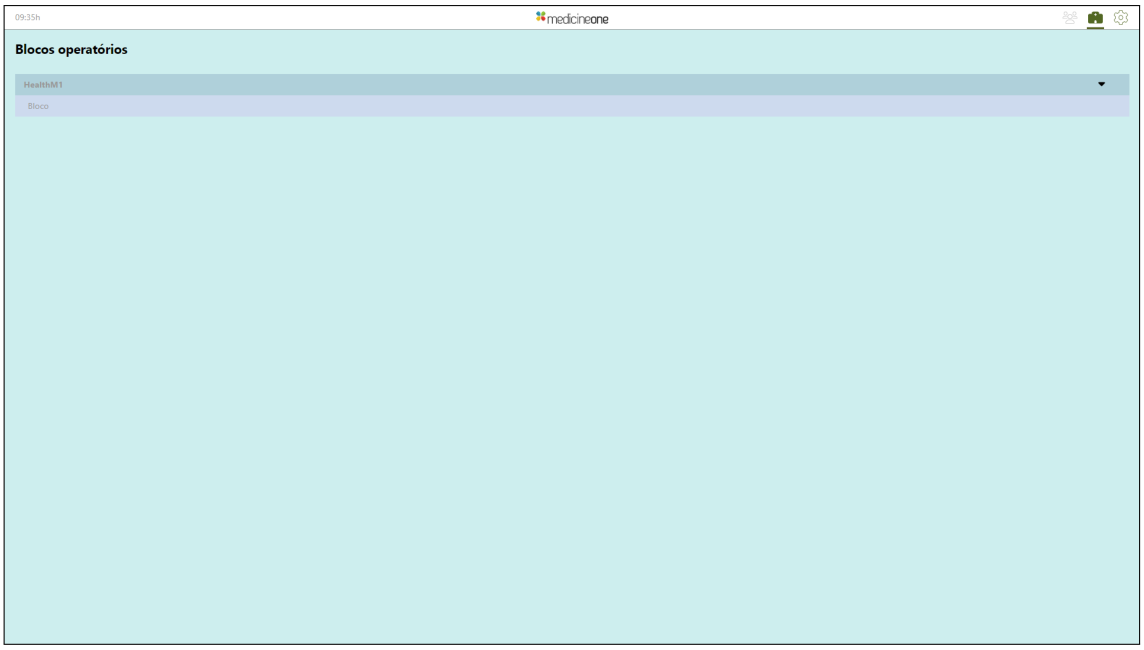
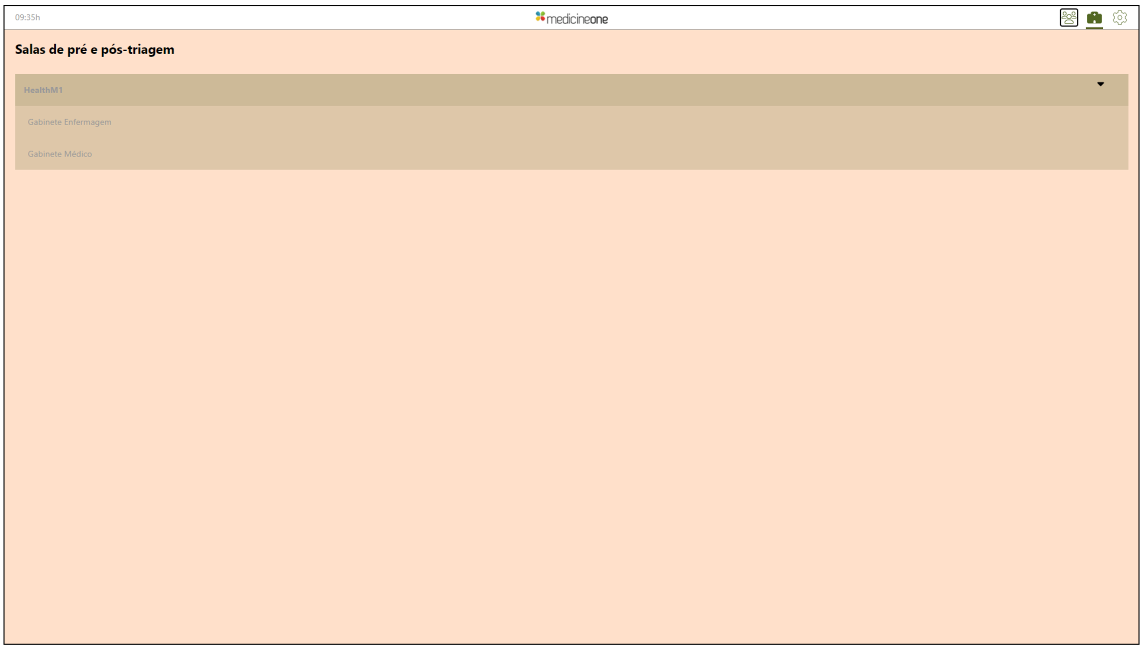


Figure 6.14: A screenshot of the patient list selection page for an operating block configuration.



09:35h

medicineone

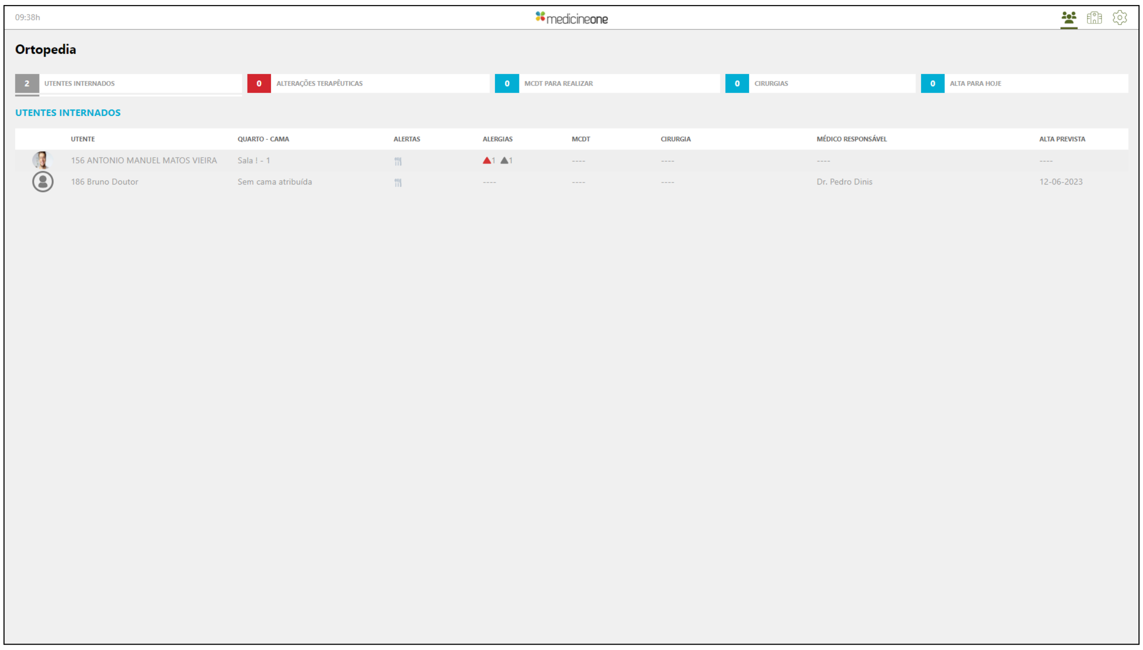
Salas de pré e pós-triagem

HealthM1

Gabinete Enfermagem

Gabinete Médico

Figure 6.15: A screenshot of the patient list selection page for an emergency room configuration.



09:38h

medicineone

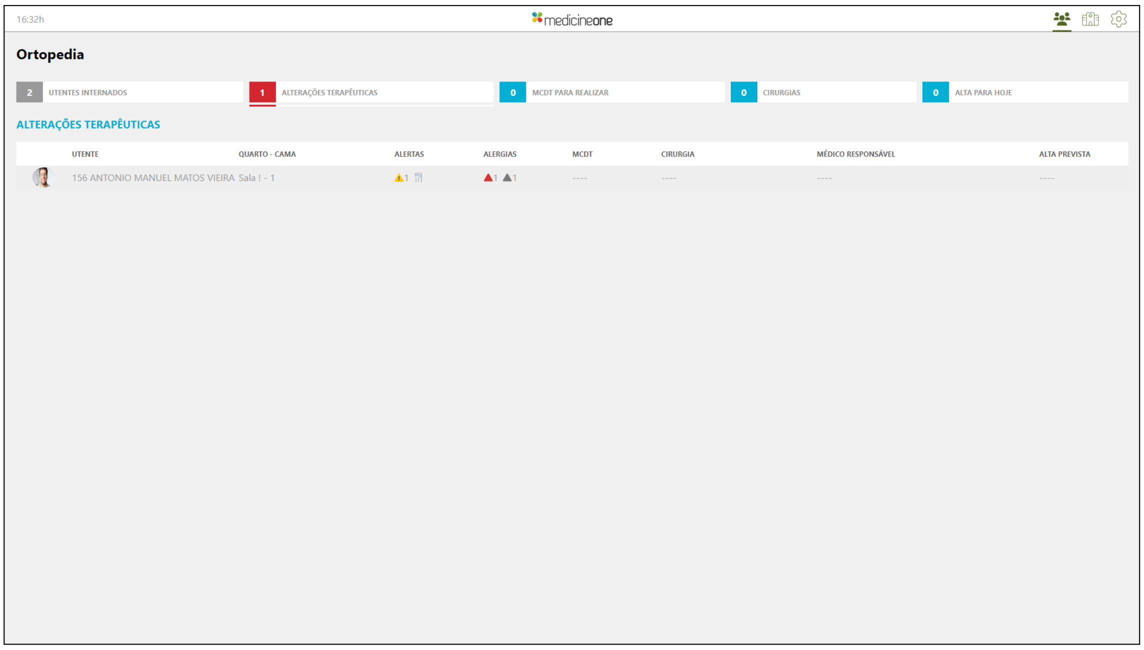
Ortopedia

2 UTENTES INTERNADOS | 0 ALTERAÇÕES TERAPÊUTICAS | 0 MCDT PARA REALIZAR | 0 CIRURGIAS | 0 ALTA PARA HOJE

UTENTES INTERNADOS

UTENTE	QUARTO - CAMA	ALERTAS	ALERGIAS	MCDT	CIRURGIA	MÉDICO RESPONSÁVEL	ALTA PREVISTA
156 ANTONIO MANUEL MATOS VIEIRA	Sala ! - 1		▲1 ▲1	----	----	----	----
186 Bruno Doutor	Sem cama atribuída		----	----	----	Dr. Pedro Dinis	12-06-2023

Figure 6.16: A screenshot of the patient list panel page for an inpatient unit configuration.



16:32h

medicineone

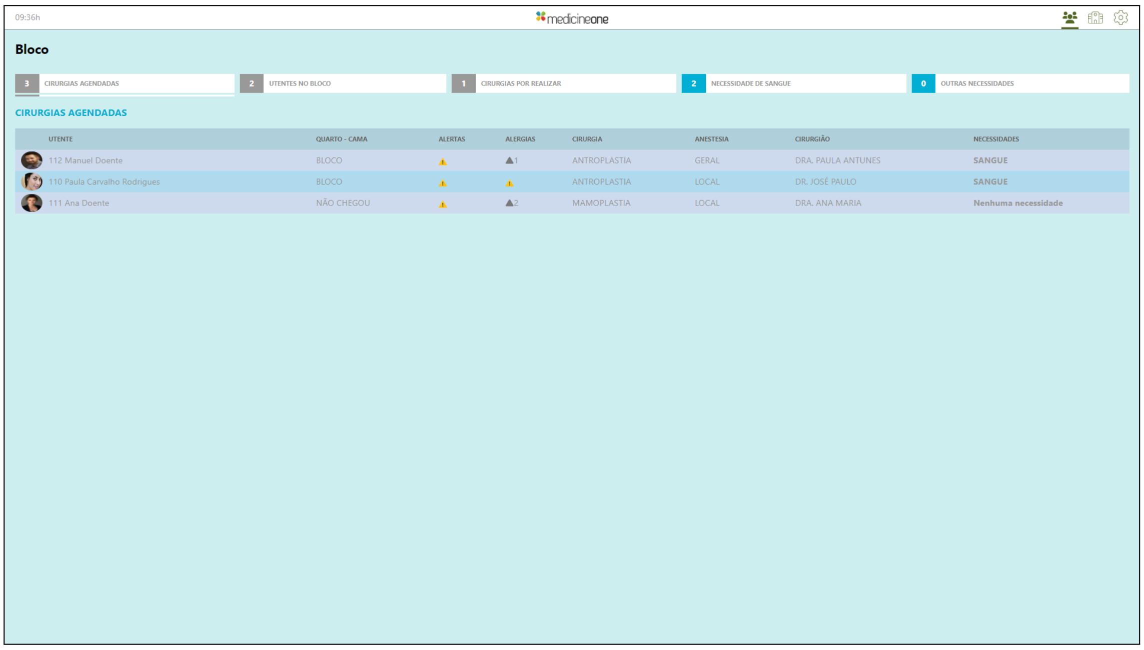
Ortopedia

2	UTENTES INTERNADOS	1	ALTERAÇÕES TERAPÊUTICAS	0	MCDT PARA REALIZAR	0	CIRURGIAS	0	ALTA PARA HOJE

ALTERAÇÕES TERAPÊUTICAS

UTENTE	QUARTO - CAMA	ALERTAS	ALERGIAS	MCDT	CIRURGIA	MÉDICO RESPONSÁVEL	ALTA PREVISTA
156 ANTONIO MANUEL MATOS VIEIRA	Sala ! - 1	⚠1	▲1 ▲1	----	----	----	----

Figure 6.17: A screenshot of the patient list panel page for an inpatient unit configuration with a selected indicator filter.



09:36h

medicineone

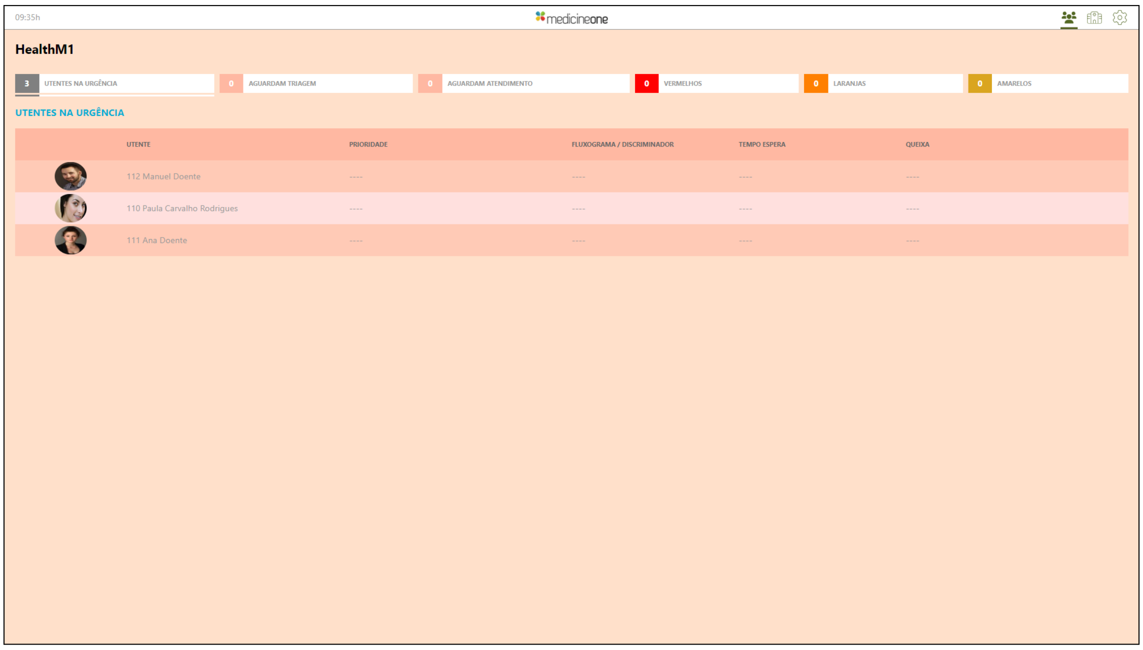
Bloco

3	CIRURGIAS AGENDADAS	2	UTENTES NO BLOCO	1	CIRURGIAS POR REALIZAR	2	NECESSIDADE DE SANGUE	0	OUTRAS NECESSIDADES

CIRURGIAS AGENDADAS

UTENTE	QUARTO - CAMA	ALERTAS	ALERGIAS	CIRURGIA	ANESTESIA	CIRURGIÃO	NECESSIDADES
112 Manuel Doente	BLOCO	⚠	▲1	ANTROPLASTIA	GERAL	DRA. PAULA ANTUNES	**SANGUE**
110 Paula Carvalho Rodrigues	BLOCO	⚠	⚠	ANTROPLASTIA	LOCAL	DR. JOSÉ PAULO	**SANGUE**
111 Ana Doente	NÃO CHEGOU	⚠	▲2	MAMOPLASTIA	LOCAL	DRA. ANA MARIA	**Nenhuma necessidade**

Figure 6.18: A screenshot of the patient list panel page for an operating block configuration.



09:35h

medicineone

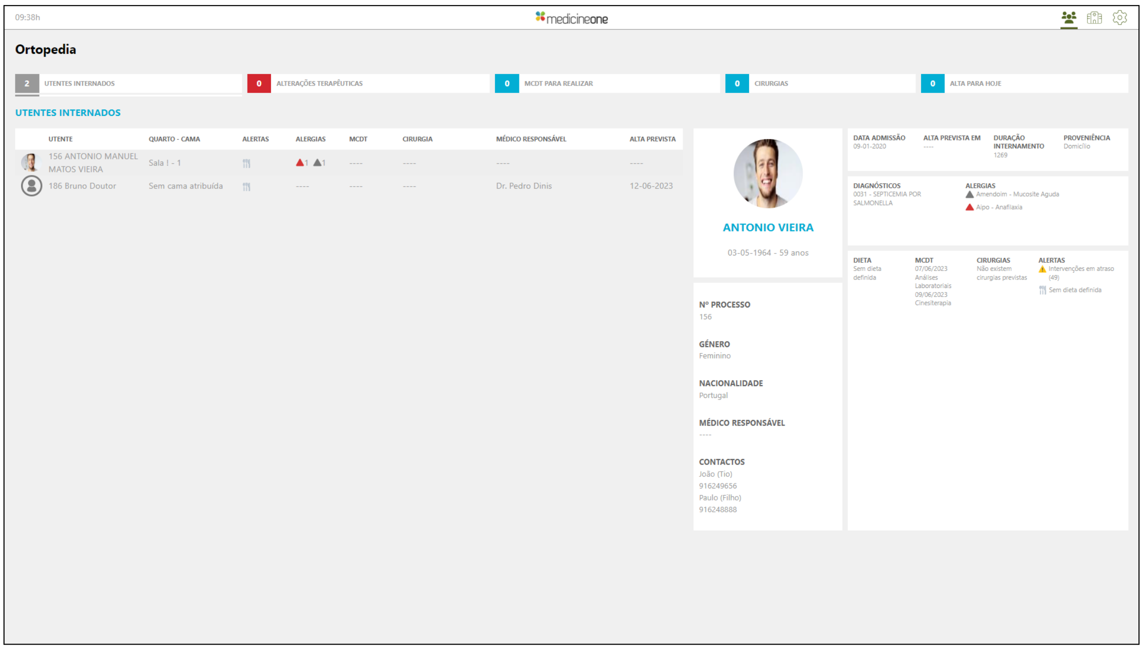
HealthM1

3	UTENTES NA URGÊNCIA
0	AGUARDAM TRIAGEM
0	AGUARDAM ATENDIMENTO
0	VERMELHOS
0	LARANJAS
0	AMARELOS

UTENTES NA URGÊNCIA

UTENTE	PRIORIDADE	FLUXOGRAMA / DISCRIMINADOR	TEMPO ESPERA	QUEIXA
112 Manuel Doente	----	----	----	----
110 Paula Carvalho Rodrigues	----	----	----	----
111 Ana Doente	----	----	----	----

Figure 6.19: A screenshot of the patient list panel page for an emergency room configuration.



09:38h

medicineone

Ortopedia

2	UTENTES INTERNADOS
0	ALTERAÇÕES TERAPÊUTICAS
0	MCDT PARA REALIZAR
0	CIRURGIAS
0	ALTA PARA HOJE

UTENTES INTERNADOS

UTENTE	QUARTO - CAMA	ALERTAS	ALERGIAS	MCDT	CIRURGIA	MÉDICO RESPONSÁVEL	ALTA PREVISTA
156 ANTONIO MANUEL MATOS VIEIRA	Sala ! - 1	🍴	▲1 ▲1	----	----	----	----
186 Bruno Doutor	Sem cama atribuída	🍴	----	----	----	Dr. Pedro Dinis	12-06-2023

ANTONIO VIEIRA

03-05-1964 - 59 anos

Nº PROCESSO
156

GÉNERO
Feminino

NACIONALIDADE
Portugal

MÉDICO RESPONSÁVEL

CONTACTOS
João (Tio)
916249656
Paulo (Filho)
916248888

DATA ADMISSÃO	ALTA PREVISTA EM	DURAÇÃO INTERNAMENTO	PROVENIÊNCIA
09-01-2020	----	1269	Domicílio

DIAGNÓSTICOS
0031 - SEPTICEMIA POR SALMONELLA

ALERGIAS
▲ Amendoim - Mucosite Aguda
▲ Aipo - Anafilaxia

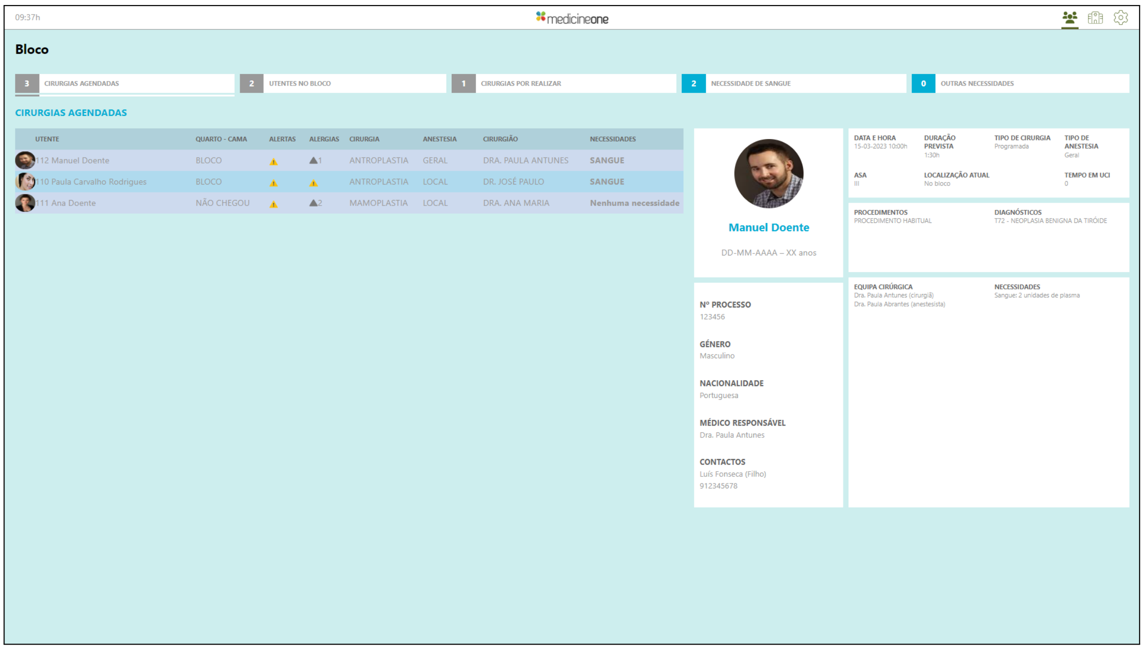
DIETA
Sem dieta definida

MCDT
07/06/2023 Análises Laboratoriais
09/06/2023 Cinesiterapia

CIRURGIAS
Não existem cirurgias previstas

ALERTAS
⚠ Intervenções em atraso (49)
🍴 Sem dieta definida

Figure 6.20: A screenshot of the patient list panel page for an inpatient unit configuration with a selected patient.


09:37h

Bloco

3 CIRURGIAS AGENDADAS | 2 UTENTES NO BLOCO | 1 CIRURGIAS POR REALIZAR | 2 NECESSIDADE DE SANGUE | 0 OUTRAS NECESSIDADES

CIRURGIAS AGENDADAS

UTENTE	QUARTO - CAMA	ALERTAS	ALERGIAS	CIRURGIA	ANESTESIA	CIRURGIÃO	NECESSIDADES
112 Manuel Doente	BLOCO	⚠	▲1	ANTROPLASTIA	GERAL	DRA. PAULA ANTUNES	SANGUE
110 Paula Carvalho Rodrigues	BLOCO	⚠	⚠	ANTROPLASTIA	LOCAL	DR. JOSÉ PAULO	SANGUE
111 Ana Doente	NÃO CHEGOU	⚠	▲2	MAMOPLASTIA	LOCAL	DRA. ANA MARIA	Nenhuma necessidade

Manuel Doente

DD-MM-AAAA – XX anos

Nº PROCESSO
123456

GÉNERO
Masculino

NACIONALIDADE
Portuguesa

MÉDICO RESPONSÁVEL
Dra. Paula Antunes

CONTACTOS
Luís Fonseca (Filho)
912345678

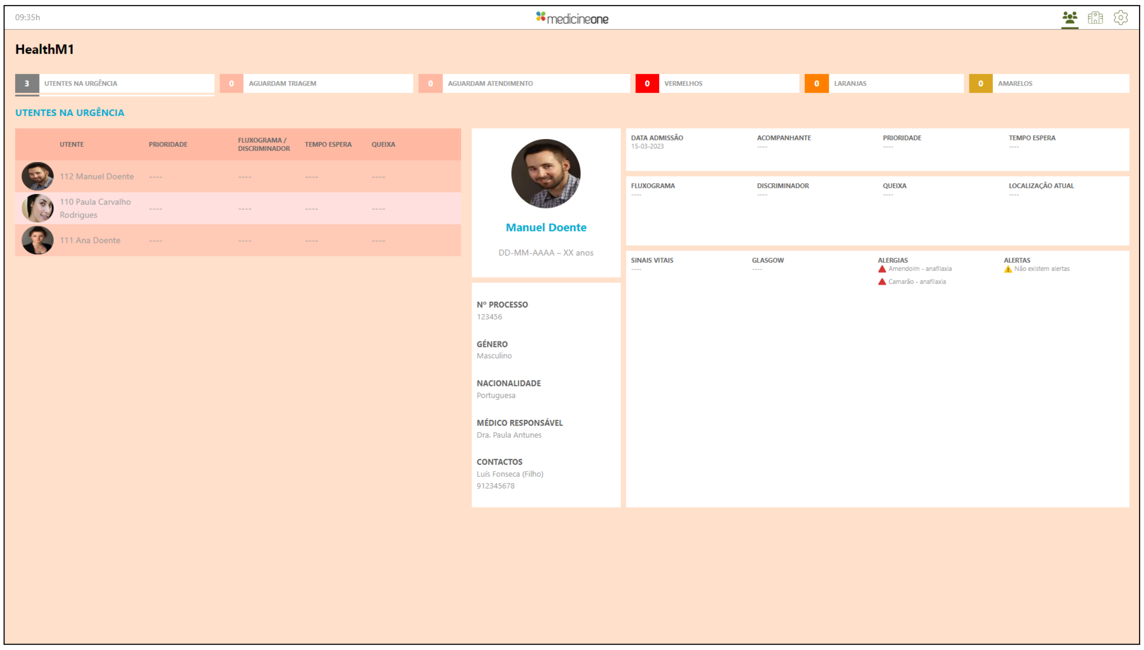
DATA E HORA: 15-03-2023 10:00h
DURAÇÃO PREVISTA: 1:30h
TIPO DE CIRURGIA: Programada
TIPO DE ANESTESIA: Geral
ASA: III
LOCALIZAÇÃO ATUAL: No bloco
TEMPO EM UCI: 0

PROCEDIMENTOS: PROCEDIMENTO HABITUAL
DIAGNÓSTICOS: T72 - NEOPLASIA BENIGNA DA TIRÓIDE

EQUIPA CIRÚRGICA
Dra. Paula Antunes (cirurgiã)
Dra. Paula Abrantes (anestesista)

NECESSIDADES
Sangue: 2 unidades de plasma

Figure 6.21: A screenshot of the patient list panel page for an operating block configuration with a selected patient.


09:35h

HealthM1

3 UTENTES NA URGÊNCIA | 0 AGUARDAM TRIAGEM | 0 AGUARDAM ATENDIMENTO | 0 VERMELHOS | 0 LARANJAS | 0 AMARELOS

UTENTES NA URGÊNCIA

UTENTE	PRIORIDADE	FLUXOGRAMA / DISCRIMINADOR	TEMPO ESPERA	QUEIXA
112 Manuel Doente	----	----	----	----
110 Paula Carvalho Rodrigues	----	----	----	----
111 Ana Doente	----	----	----	----

Manuel Doente

DD-MM-AAAA – XX anos

Nº PROCESSO
123456

GÉNERO
Masculino

NACIONALIDADE
Portuguesa

MÉDICO RESPONSÁVEL
Dra. Paula Antunes

CONTACTOS
Luís Fonseca (Filho)
912345678

DATA ADMISSÃO: 15-03-2023
ACOMPANHANTE: ----
PRIORIDADE: ----
TEMPO ESPERA: ----

FLUXOGRAMA: ----
DISCRIMINADOR: ----
QUEIXA: ----
LOCALIZAÇÃO ATUAL: ----

SINAIS VITAIS: ----
GLASGOW: ----
ALERGIAS
▲ Amendoim - anafilaxia
▲ Camarão - anafilaxia
ALERTAS
⚠ Não existem alertas

Figure 6.22: A screenshot of the patient list panel page for an emergency room configuration with a selected patient.

Chapter 7

Testing

As software testing is crucial to ensuring an application's dependability and overall quality, several tests were created and executed, with a main focus on the key components of the developed system. For each component tested, a brief description, an example of a test case, and the overall results are provided. Finally, acceptance tests are also discussed.

7.1 API testing

Due to the developed endpoints' importance to the application, a set of black-box tests was created for them. The testing tool used was Postman, which has a number of features to aid in the design, development, and testing of APIs [53]. It was used to quickly make test calls throughout the endpoint development process, and later a more structured test plan was enforced. These tests were written inside Postman, and they were run alongside a call to an endpoint.

The tests mainly consisted of validating the responses received from the API in relation to the input parameters. For instance, sending a request with valid parameters should result in a 200 (OK) response status code and an object; sending a request with valid parameters but no matches in the database should result in a 422 (Unprocessable Entity) response status code; and sending a request with invalid or missing parameters should result in a 400 (Bad Request) response status code. The description of one of the tests done on an endpoint can be seen in Table 7.1.

Eight out of the forty-eight tests that were conducted failed, yielding an initial test pass rate of 83% (40/48). All of the tests that failed were the result of incorrect error handling and validation. One such instance occurs in the "listSecotrs" endpoint, which accepts an integer representing a list type identifier. If a value was provided that was lower than 1 or higher than 3, it simply returned an empty list, despite the fact that the intended behavior was to return an error message informing the user that no types in the system match those identifiers. Another instance was on the dashboard data endpoints, where an exception was returned rather than an error response when a valid string was provided as a stored procedure name that did not exist in the system. These problems were promptly fixed, and tests were then repeated until all ultimately passed. The complete list and results of these test cases can be seen in Appendix A.

Table 7.1: Test #2, where the endpoint that returns the configuration data is tested with an inexistent configuration.

Code	Endpoint	Input parameters	Expected response	Result

TC2	configuration.getDisplayStructure	organizationId = -2147483636
		configurationCode = "99"
			200 Ok
			(Returns the correct configuration)
				Pass

7.2 Dashboard configuration testing

The configurable dashboard appearance is also one of the application's defining features. As such, test cases were planned to ensure that the correct styling was rendered on the dashboard UI. Playwright, an end-to-end testing framework for web applications, was used to create and carry out these tests. It enables tests to be run on various browsers to ensure cross-browser compatibility [52]. With this framework, it is possible to assert that the application rendering will be consistent across various browsers, in addition to testing the proper usage of the dashboard configuration data. Each test is run by Playwright, which launches the application in a browser and enables interaction with the rendered elements of the page. All tests were run with the following browsers: Chromium, Firefox, and WebKit. After choosing a configuration in the settings page, the method "getComputedStyle()", which returns all of the element's CSS properties after all stylesheets and computations are made, is used to obtain the attributes intended for testing. Finally, the obtained properties are compared with the selected configuration data using assertions.

These end-to-end tests used three configurations as inputs, each with different types and fields. They were used to test a wide range of configuration values, including different colors, line heights, indicator counts, and all of the different column fields. In Table 7.2 a description of one of these tests can be seen.

Ninety-nine tests were run, and a pass rate of 100% was achieved (99/99), though some circumstances called for some leniency. For instance, a line height of 40 pixels would be extended to fit an image with a 64x64 pixel resolution if a field's type was "Image 64", as such this was not regarded as a test failure. The complete list and results of these test cases can be seen in Appendix B.

Table 7.2: Test case #103, where the application is tested for the correct application of the patient list background colors and line height in the Chromium browser.

Code	Input configuration	Browser	Expectation	Result
TC103	#1	Chromium	Should have the configured list background colors and line height.	Pass

7.3 Real-time notifications testing

Additionally, end-to-end tests were conducted for the Azure SignalR Service, which sends real-time updates to the dashboard. As it was mentioned that only the update of configurations is set up to emit events, aside from its test, which can be performed by changing a configuration on the MedicineOne system, all other tests involved calling the endpoints with different message inputs with Postman. Their purpose was to ensure that messages were properly received and that web application instances correctly decided whether to act on messages or whether they were only meant for other instances. It was also checked to see if all instances of the application that were currently running were receiving the messages. Tables 7.3 and 7.4 show two of the tests executed for testing the application's interaction with the Azure SignalR Service, the first one being run directly with the MedicineOne system and the second with Postman.

Thirty test cases were elaborated and run in total, and a 100% pass rate was achieved (30/30). The complete list and results of these test cases can be seen in Appendix C.

Table 7.3: Test case #151, where the application is tested for the correct course of action when receiving a configuration update notification for the configuration it is currently using.

Code	Notification	Action	Message (contains the configuration code)	Recipient (configuration code)	Expectation (besides receiving the message)	Result
TC151	signalr. sendConfigurationUpdate	Add a field to the patient list	98	98	Refetches configuration	Pass

Table 7.4: Test case #158, where the application is tested for the correct course of action when receiving a patient admission notification for its displayed list.

Code	Notification endpoint	Message (contains the location identifier)	Precondition	Expectation (besides receiving the message)	Result
TC158	signalr. sendPatientAdmission	-2147483647	The dashboard displays the location	Refetches indicators and patient panel	Pass

7.4 Acceptance testing

To verify that the implemented requirements were met and validate the work done, acceptance tests were performed. As a result of the configuration end-to-end tests, some requirements had already been indirectly tested. For instance, the correct application of the configuration to the patient panel (R8) was tested in numerous test cases in Appendix B. The remainder were then put to the test as well. The application emitting graphical notifications (R22), which had the lowest MoSCoW priority, and the emission of sound notifications (R21), which had the second lowest priority, were the only two requirements that were not implemented. The first had already been deemed out of the scope of the project during its specification, while the second was disregarded during development. The full list of these tests and their results can be seen in Appendix D.

Besides the intern, MedicineOne advisor Fernando Tinoco also performed informal tests by experimenting with the application while also supplying feedback and reporting bugs found throughout development.

During the meetings with most of the Project Development team and MedicineOne's Chief Vision Officer, it was transmitted that the work done was in line with the requirements and their expectations. Additionally, a desire to perform a test pilot with one of MedicineOne's partners was also demonstrated, and it is being internally planned for the end of the year.

Chapter 8

Conclusion

The need for dashboards to provide clear and concise data visualization to support decision-making, particularly in the medical environment, was described in this report. To gain a better understanding of how a medical dashboard implementation should be approached, a study on dashboard design principles and published implementations for clinical purposes was conducted. It was emphasized the significance of selecting the data to display on the dashboard as well as the means of visualization, and several factors were relayed to describe the purpose, functionality, and data properties of medical dashboards. Based on the written state-of-the-art, the defined dashboard properties were validated, and the intended solution was characterized to compare with the analyzed case studies.

An analysis of UI development technologies for web and desktop environments, as well as services that offer real-time application communication, was done. The decisions were revealed and justified, and they included the use of web technologies, specifically the UI development library React, and to create a real-time communication channel between the dashboard and the main MedicineOne system, the Azure SignalR Service was chosen.

In order to lay the groundwork for development and to provide a means of project validation at the end, the overall system was specified, starting with the listing and prioritization of requirements and the creation of user stories. Additionally, architectural details were communicated, including the system's overall structure and internal interactions, the description of data schemas, and UI design requirements. A structural and functional explanation of the MedicineOne API, which will supply the medical data displayed on the dashboard and serve as a proxy for real-time communication, was also provided.

An overview of the development process was written, which addressed the implementation of the system's essential parts: the endpoints built into MediciNeOne's API, the customizable web application, and the setup of an Azure SignalR instance for real-time notifications. The development's outcome was demonstrated by screenshots from the application, which showed various configured dashboards showcasing the system's potential.

It was explained how the applied testing process worked, which had an emphasis on the important system components, establishing a solid framework for the system's robustness. Regarding acceptance testing, it was also discussed how the testing of the requirements and the involvement of MedicineOne validated the work completed.

Overall, there were no issues with the project's development, and even though it deviated from the methodology and planning as stated and justified, it ultimately had no detrimental effects on the work. The outcome was validated through the fulfillment of requirements and input from MedicineOne, and it validated the expectations of being an effective and customizable medical dashboard with real-time event notification in place.

There is work that can be done in the future to complement and enhance the product, such as finishing the support for the configuration types that are currently offered, putting the server-side changes into place so that more events are emitted to the dashboard, and carrying out a more thorough testing plan with hospitals and medical professionals' participation. Concerning this final element, MedicineOne is already working to implement a pilot with one of their partners by the end of the year to test the system in a real-world environment and obtain feedback from their future users.

References

[1] ABELMed. Physician Dashboard - ABELSoft Inc. - ABELMed. `https://www.abelmed.com/Physician-Dashboard`. last accessed: 22-12-2022.

[2] Amazon SNS vs Azure SignalR Service. `https://ably.com/compare/amazon-sns-vs-azure-signalr-service`, 2020. last accessed: 10-01-2023.

[3] Azure SignalR Service vs Firebase. `https://ably.com/compare/azure-signalr-service-vs-firebase`, 2020. last accessed: 10-01-2023.

[4] Mohammed Alhamadi. Challenges, strategies and adaptations on interactive dashboards. In *Proceedings of the 28th ACM Conference on User Modeling, Adaptation and Personalization*, page 368–371. Association for Computing Machinery, 2020.

[5] What is Amazon SNS? - Amazon Simple Notification Service. `https://docs.aws.amazon.com/sns/latest/dg/welcome.html`. last accessed: 10-01-2023.

[6] Jacob Anderson, Jason Leubner, and Steven R. Brown. EHR overtime: An analysis of time spent after hours by family physicians. *Family Medicine*, 52(2):135–137, 2020.

[7] Angular. `https://angular.io/`. last accessed: 26-12-2022.

[8] Introduction to ASP.NET Identity - ASP.NET 4.x | Microsoft Learn. `https://learn.microsoft.com/en-us/aspnet/identity/overview/getting-started/introduction-to-aspnet-identity`. last accessed: 16-01-2023.

[9] Getting Started | Axios Docs. `https://axios-http.com/docs/intro`. last accessed: 23-06-2023.

[10] What is Azure SignalR Service? - Microsoft Learn. `https://learn.microsoft.com/en-us/azure/azure-signalr/signalr-overview`. last accessed: 27-06-2023.

[11] Marcus Badgeley, Shameer Khader, Benjamin Glicksberg, Max Tomlinson, Matthew Levin, Patrick McCormick, Andrew Kasarskis, David Reich, and Joel Dudley. EHDViz: Clinical dashboard development using open-source technologies. *BMJ Open*, 6:e010579, 2016.

[12] John W. Beasley, Tosha B. Wetterneck, Jon Temte, Jamie A. Lapin, Paul Smith, A. Joy Rivera-Rodriguez, and Ben-Tzion Karsh. Information chaos in primary care: Implications for physician performance and patient safety. *The Journal of the American Board of Family Medicine*, 24(6):745–751, 2011.

[13] Sufyan Bin Uzayr, Nicholas Cloud, and Tim Ambler. *JavaScript Frameworks for Modern Web Development*. Springer, 2019. (Chapters 7, 13 and 14).

[14] Richard Brath and Michael Peters. Dashboard design: Why design is important. *DM Direct*, 85:1011285–1, 2004.

[15] Bernard Bucalon, Tim Shaw, Kerri Brown, and Judy Kay. State-of-the-art dashboards on clinical indicator data to support reflection on practice: Scoping review. *JMIR Med Inform*, 10(2):e32695, 2022.

[16] Cerner. CareAware capacity management solutions - Oracle Cerner. `https://www.cerner.com/solutions/capacity-management`. last accessed: 22-12-2022.

[17] Cerner. Cerner Advance. `https://advance.cerner.com/`. last accessed: 22-12-2022.

[18] Cerner. Achieving situational awareness with real-time data. `https://healthcareexecutive.org/archives/september-october-2021/achieving-situational-awareness-with-real-time-data`, 2021. last accessed: 22-12-2022.

[19] Jelica Cincović and Marija Punt. Comparison: Angular vs. React vs. Vue. which framework is the best choice? *Belgrade, Universidad de Belgrade*, 2020.

[20] Kevin W Clark, Elizabeth Whiting, Jeffrey Rowland, Leah E Thompson, Ian Missenden, and Gerhard Schellein. Breaking the mould without breaking the system: The development and pilot of a clinical dashboard at the Prince Charles Hospital. *Australian Health Review*, 37(3):304–308, 2013.

[21] D3 gallery - observable. `https://observablehq.com/@d3/gallery`. last accessed: 30-12-2022.

[22] D3.js - data-driven documents. `https://d3js.org/`. last accessed: 30-12-2022.

[23] Dapper - a simple object mapper for .Net - GitHub. `https://github.com/DapperLib/Dapper`. last accessed: 14-01-2023.

[24] Dawn Dowding, Rebecca Randell, Peter Gardner, Geraldine Fitzpatrick, Patricia Dykes, Jesus Favela, Susan Hamer, Zac Whitewood-Moores, Nicholas Hardiker, Elizabeth Borycki, and Leanne Currie. Dashboards for improving patient care: Review of the literature. *International Journal of Medical Informatics*, 84(2):87–100, 2015.

[25] Firebase Cloud Messaging. `https://firebase.google.com/docs/cloud-messaging`. last accessed: 08-01-2023.

[26] Michael Fischer, Wissam Kourany, Karen Sovern, Kurt Forrester, Cassandra Griffin, Nancy Lightner, Shawn Loftus, Katherine Murphy, Greg Roth, Paul Palevsky, and Susan Crowley. Development, implementation and user experience of the Veterans Health Administration (VHA) dialysis dashboard. *BMC Nephrology*, 21, 2020.

[27] Amy Franklin, Swaroop Gantela, Salsawit Shifarraw, Todd R. Johnson, David J. Robinson, Brent R. King, Amit M. Mehta, Charles L. Maddow, Nathan R. Hoot, Vickie Nguyen, Adriana Rubio, Jiajie Zhang, and Nnaemeka G. Okafor. Dashboard visualizations: Supporting real-time throughput decision-making. *Journal of Biomedical Informatics*, 71:211–221, 2017.

[28] Hangfire – background jobs and workers for .NET and .NET Core. `https://www.hangfire.io/`. last accessed: 14-01-2023.

[29] Andrea Janes, Alberto Sillitti, and Giancarlo Succi. Effective dashboard design. *Cutter IT Journal*, 26:17–24, 01 2013.

[30] Anna Janssen, Candice Donnelly, Judy Kay, Peter Thiem, Aldo Saavedra, Nirmala Pathmanathan, Elisabeth Elder, Phuong Dinh, Masrura Kabir, Kirsten Jackson, Paul Harnett, and Tim Shaw. Developing an intranet-based lymphedema dashboard for breast cancer multidisciplinary teams: Design research study. *J Med Internet Res*, 22(4), 2020.

[31] Jeffrey Palermo. The Onion Architecture : part 1 | Programming with Palermo. `https://jeffreypalermo.com/2008/07/the-onion-architecture-part-1/`, 2008. last accessed: 09-01-2023.

[32] Jemin Desai. Web application vs desktop application: Pros and cons. `https://positiwise.com/blog/web-application-vs-desktop-application-pros-and-cons/`. last accessed: 11-01-2023.

[33] Saif Sherif Khairat, Aniesha Dukkipati, Heather Alico Lauria, Thomas Bice, Debbie Travers, and Shannon S Carson. The impact of visualization dashboards on quality of care and clinician satisfaction: Integrative literature review. *JMIR Hum Factors*, 5(2):e22, 2018.

[34] Géry Laurent, Mouhamed Moussa, Cedric Cirenei, Benoit Tavernier, Romaric Marcilly, and Antoine Lamer. Development, implementation and preliminary evaluation of clinical dashboards in a department of anesthesia. *Journal of Clinical Monitoring and Computing*, 35, 2021.

[35] Keehyuck Lee, Se Young Jung, Hee Hwang, Sooyoung Yoo, Hyun Young Baek, Rong-Min Baek, and Seok Kim. A novel concept for integrating and delivering health information using a comprehensive digital dashboard: An analysis of healthcare professionals' intention to adopt a new system and the trend of its real usage. *International Journal of Medical Informatics*, 97:98–108, 2017.

[36] ER Mahendrawathi, Danu Pranantha, and Johansyah Dwi Utomo. Development of dashboard for hospital logistics management. In *2010 IEEE Conference on Open Systems (ICOS 2010)*, pages 86–90, 2010.

[37] Marco Schaefer. Onion Architecture explained — building maintainable software. `https://marcoatschaefer.medium.com/onion-architecture-explained-building-maintainable-software-54996ff8e464`, 2020. last accessed: 09-01-2023.

[38] Jr. Mark C. Schall, Howard Chen, Priyadarshini R. Pennathur, and Laura Cullen. Development and evaluation of a health information technology dashboard of quality indicators. *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*, 59(1):461–465, 2015.

[39] Niels Martin, Jochen Bergs, Dorien Eerdekens, Benoît Depaire, and Sandra Verelst. Developing an emergency department crowding dashboard: A design science approach. *International Emergency Nursing*, 39:68–76, 2018.

[40] Martin Fowler. CQRS - Martin Fowler. `https://martinfowler.com/bliki/CQRS.html`, 2011. last accessed: 14-01-2023.

[41] jbogard/MediatR: Simple, unambitious mediator implementation in .NET - GitHub. `https://github.com/jbogard/MediatR`. last accessed: 14-01-2023.

[42] Mohit Joshi. Angular vs React vs Vue: Core differences | BrowserStack. `https://www.browserstack.com/guide/angular-vs-react-vs-vue`, 2022. last accessed: 11-01-2023.

[43] What is the MoSCoW Method? - TechTarget. `https://www.techtarget.com/searchsoftwarequality/definition/MoSCoW-method`. last accessed: 01-06-2023.

[44] Nadezhda Mal. Web app vs. desktop app - Qulix Systems. `https://www.qulix.com/about/web-app-vs-desktop-app/`, 2022. last accessed: 11-01-2023.

[45] What is .NET MAUI? - .NET MAUI - Microsoft Learn. `https://learn.microsoft.com/en-us/dotnet/maui/what-is-maui?view=net-maui-7.0`. last accessed: 05-01-2023.

[46] OpenAPI Specification - version 3.0.3 - Swagger. `https://swagger.io/specification/`. last accessed: 14-01-2023.

[47] What's OpenIddict? `https://documentation.openiddict.com/guides/index.html`. last accessed: 16-01-2023.

[48] Web vs desktop apps: a weigh-up - Parker Software. `https://www.parkersoftware.com/blog/web-vs-desktop-apps-a-weigh-up/`. last accessed: 11-01-2023.

[49] Malini Patel, Bhawna Rathi, and Mansoor Yarubi. Development and implementation of maternity dashboard in regional hospital for quality improvement at ground level: A pilot study. *Oman Medical Journal*, 34:194–199, 2019.

[50] Miguel Pestana, Ruben Pereira, and Sérgio Moro. Improving health care management in hospitals through a productivity dashboard. *Journal of medical systems*, 44(4):87, 2020.

[51] Rimma Pivovarov and Noémie Elhadad. Automated methods for the summarization of electronic health records. *Journal of the American Medical Informatics Association*, 22(5):938–947, 2015.

[52] Playwright enables reliable end-to-end testing for modern web apps. `https://playwright.dev/`. last accessed: 02-07-2023.

[53] What is Postman? Postman API Platform. `https://www.postman.com/product/what-is-postman/`. last accessed: 03-07-2023.

[54] React – a JavaScript library for building user interfaces. `https://reactjs.org/`. last accessed: 26-12-2022.

[55] React Router: Home v6.13.0. `https://reactrouter.com/en/main`. last accessed: 23-06-2023.

[56] Mediator - Refactoring.Guru. `https://refactoring.guru/design-patterns/mediator`. last accessed: 09-01-2023.

[57] Saeed Rouhani and Shooka Zamenian. An architectural framework for healthcare dashboards design. *Journal of Healthcare Engineering*, 2021:1–12, 2021.

[58] Alper Sarikaya, Michael Correll, Lyn Bartram, Melanie Tory, and Danyel Fisher. What do we talk about when we talk about dashboards? *IEEE Transactions on Visualization and Computer Graphics*, 25(1):682–692, 2019.

[59] Mark Schall, Laura Cullen, Priyadarshini Pennathur, Howard Chen, Keith Burrell, and Grace Matthews. Usability evaluation and implementation of a health information technology dashboard of evidence-based quality indicators. *Computers, informatics, nursing : CIN*, 35, 2016.

[60] What is Scrum? `https://www.scrum.org/resources/what-is-scrum`. last accessed: 12-01-2023.

[61] Shaumik Daityari. Angular vs React vs Vue: Which framework to choose in 2023. `https://www.codeinwp.com/blog/angular-vs-vue-vs-react/`, 2023. last accessed: 11-01-2023.

[62] Real-time ASP.NET with SignalR. `https://dotnet.microsoft.com/en-us/apps/aspnet/signalr`. last accessed: 14-01-2023.

[63] Stack Overflow Developer Survey 2022. `https://survey.stackoverflow.co/2022/`. last accessed: 10-01-2023.

[64] Tailwind CSS - Rapidly build modern websites without ever leaving your HTML. `https://tailwindcss.com/`. last accessed: 23-06-2023.

[65] TypeScript: JavaScript with syntax for types. `https://www.typescriptlang.org/`. last accessed: 26-12-2022.

[66] Vite | Next Generation Frontend Tooling. https://vitejs.dev/. last accessed: 23-06-2023.

[67] Vue.js - the progressive JavaScript framework | vue.js. https://vuejs.org/. last accessed: 29-12-2022.

[68] What is Windows Presentation Foundation - WPF .NET. https://learn.microsoft.com/en-us/dotnet/desktop/wpf/?view=netdesktop-6.0. last accessed: 05-01-2023.

Appendices

Appendix A

Endpoint tests

Table A.1: List and results of the test cases run for the configuration endpoints.

Code	Endpoint	Input parameters	Expected response	Result

TC1	configuration.getDisplayStructure	organizationId = -2147483636
		configurationCode = "1"
			200 OK
			(Returns the correct configuration)
				Pass

TC2		organizationId = -2147483636
		configurationCode = "99"
				Pass

TC3		organizationId = -2147483646
		configurationCode = "2"
				Pass

TC4		organizationId = -2147483636
		configurationCode = "2"
			422 Unprocessable Entity
			(Configuration doesn't exist)
				Pass

TC5		organizationId = -2147483646
		configurationCode = "1"
				Pass

TC6		organizationId = -2147483647
		configurationCode = "1"
			422 Unprocessable Entity
			(Organization doesn't exist)
				Pass

TC7		organizationId = -1
		configurationCode = "1"
				Pass

TC8		organizationId = "a"
		configurationCode = "1"
			400 Bad Request
			(Invalid organization identifier)
				Pass

TC9		organizationId = -2147483646 configurationCode =	400 Bad Request (Configuration code missing)	Pass
TC10		organizationId = configurationCode = "1"	400 Bad Request (Organization identifier missing)	Pass
TC11	configuration. listOrganizationsConfigurations		200 OK (Returns both organizations with configurations)	Pass
TC12			200 OK (Returns the only organization with configurations)	Pass

Table A.1: List and results of the test cases run for the configuration endpoints.

Code	Endpoint	Input parameters	Expected response	Result

TC13	configuration.listSectors	organizationId = -2147483636
		listTypeId = 1
			200 OK
			(Returns organic units with inpatient areas)
				Pass

TC14		organizationId = -2147483646
		listTypeId = 1
				Pass

TC15		organizationId = -2147483636
		listTypeId = 2
			200 OK
			(Returns operating blocks with operating rooms)
				Pass

TC16		organizationId = -2147483646
		listTypeId = 2
				Pass

TC17		organizationId = -2147483636
		listTypeId = 3
			200 OK
			(Returns emergency units with pre and post triage rooms)
				Pass

TC18		organizationId = -2147483646
		listTypeId = 3
				Pass

TC19		organizationId = -2147483636
		listTypeId = 0
			422 Unprocessable Entity
			(List type doesn't exist)
				Fail

TC20		organizationId = -2147483646
		listTypeId = 4
				Fail

TC21		organizationId = -2147483636
		listTypeId = "a"
			400 Bad Request
			(Invalid list type identifier)
				Pass

TC22		organizationId = -2147483647
		listTypeId = 1
			422 Unprocessable Entity
			(Organization doesn't exist)
				Pass

TC23		organizationId = listTypeId = 1	400 Bad Request (Organization identifier missing)	Pass
TC24		organizationId = -2147483636 listTypeId =	400 Bad Request (List type identifier missing)	Pass

Table A.2: List and results of the test cases run for the indicator data endpoint.

Code	Endpoint	Input parameters	Expected response	Result

TC25	wall. listMetrics	organizationId = -2147483636
		spName = "[M1WALL].[InpatientPatientIndicator]"
		sectionIds = -2147483648
			200 OK
			(Returns the section's indicator entries)
				Pass

TC26		organizationId = -2147483636
		spName = "[M1WALL].[InpatientPatientIndicator]"
		sectionIds = -2147483647
				Pass

TC27		organizationId = -2147483636
		spName = "[M1WALL].[InpatientPatientIndicator]"
		sectionIds = -2147483647
		sectionIds = -2147483648
			200 OK
			(Returns the sections' indicator entries)
				Pass

TC28		organizationId = -2147483636
		spName = "a"
		sectionIds = -2147483647
			422 Unprocessable Entity
			(Stored procedure doesn't exist)
				Fail

TC29		organizationId =
		spName = "[M1WALL].[InpatientPatientIndicator]"
		sectionIds = -2147483647
			400 Bad Request
			(Organization identifier missing)
				Pass

TC30		organizationId = -2147483636
		spName =
		sectionIds = -2147483647
			400 Bad Request
			(Stored procedure name missing)
				Pass

TC31		organizationId = -2147483636
		spName ="[M1WALL].[InpatientPatientIndicator]"
		sectionIds =
			400 Bad Request
			(Section(s) identifier(s) missing)
				Fail

Table A.3: List and results of the test cases run for the patient data endpoint.

Code	Endpoint	Input parameters	Expected response	Result

TC32	wall.listPatients	organizationId = -2147483636
		spName = "[M1WALL].InpatientPatientList"
		sectionIds = -2147483648
			200 OK
			(Returns the section's patients)
				Pass

TC33		organizationId = -2147483636
		spName = "[M1WALL].InpatientPatientList"
		sectionIds = -2147483647
				Pass

TC34		organizationId = -2147483636
		spName = "[M1WALL].InpatientPatientList"
		sectionIds = -2147483647
		sectionIds = -2147483648
			200 OK
			(Returns the sections' patients)
				Pass

TC35		organizationId = -2147483636
		spName = "a"
		sectionIds = -2147483647
			422 Unprocessable Entity
			(Stored procedure doesn't exist)
				Fail

TC36		organizationId =
		spName = "[M1WALL].InpatientPatientList"
		sectionIds = -2147483647
			400 Bad Request
			(Organization identifier missing)
				Pass

TC37		organizationId = -2147483636
		spName =
		sectionIds = -2147483647
			400 Bad Request
			(Stored procedure name missing)
				Pass

TC38		organizationId = -2147483636
		spName ="[M1WALL].InpatientPatientList"
		sectionIds =
			400 Bad Request
			(Section(s) identifier(s) missing)
				Fail

Table A.4: List and results of the test cases run for the patient detail data endpoint.

Code	Endpoint	Input parameters	Expected response	Result

TC39	wall.getPatientDetail	organizationId = -2147483636
		spNames = "[M1WALL].[InpatientDetailEssential]"
		spNames = "[M1WALL].[InpatientDetailDiagnostics]"
		spNames = "[M1WALL].[InpatientDetailOtherData]"
		patientId = -2147482577
			200 OK
			(Returns the patient's details and identification)
				Pass

TC40		organizationId = -2147483636
		spNames = "[M1WALL].[InpatientDetailEssential]"
		spNames = "[M1WALL].[InpatientDetailDiagnostics]"
		spNames = "[M1WALL].[InpatientDetailOtherData]"
		patientId = -2147481548
				Pass

TC41		organizationId = -2147483636
		spNames = "[M1WALL].[InpatientDetailEssential]"
		spNames = "[M1WALL].[InpatientDetailDiagnostics]"
		patientId = -2147482577
				Pass

TC42		organizationId = -2147483636
		spNames = "[M1WALL].[InpatientDetailEssential]"
		spNames = "[M1WALL].[InpatientDetailDiagnostics]"
		patientId = -2147481548
				Pass

TC43		organizationId = -2147483636
		spNames = "[M1WALL].[InpatientDetailEssential]"
		patientId = -2147482577
				Pass

TC44		organizationId = -2147483636
		spNames = "[M1WALL].[InpatientDetailEssential]"
		patientId = -2147481548
				Pass

TC45		organizationId = -2147483636
		spNames = "a"
		patientId = -2147482577
			422 Unprocessable Entity
			(Stored procedure doesn't exist)
				Fail

Table A.4 continued from previous page

TC46		organizationId = -2147483636
		spNames = "[M1WALL].[InpatientDetailEssential]"
		patientId = -2147482577
			400 Bad Request
			(Organization identifier missing)
				Pass

TC47		organizationId = -2147483636
		spNames =
		patientId = -2147482577
			400 Bad Request
			(Stored procedure name(s) missing)
				Fail

TC48		organizationId = -2147483636
		spNames = "[M1WALL].[InpatientDetailEssential]"
		patientId =
			400 Bad Request
			(Patient identifier missing)
				Pass

Appendix B

Configuration tests

For these tests, configuration #1 had the following properties: 5 indicators; a 50 pixels line height; patient list column types of photo, text (regular, bold, and italic), and 16x16/24x24/32x32 pixels images; patient detail column types of photo, text (regular, bold, and italic), and 16x16/24x24 pixels images; and patient detail components with line break.

Configuration #2 had the following different properties: 1 indicator; a 80 pixels line height; patient list column types of 48x48/64x64 pixels images; and patient detail column types of 32x32/48x48/64x64 pixels images.

Configuration #3 had the following different properties: 10 indicators and a 25 pixels line height.

Table B.1: List and results of the test cases run for the dashboard configuration.

Code	Input configuration	Browser	Expectation	Result
TC49	#1	Chromium	Should have the configured background color.	Pass
TC50	#2			Pass
TC51	#3			Pass
TC52	#1	Firefox		Pass
TC53	#2			Pass
TC54	#3			Pass
TC55	#1	WebKit		Pass
TC56	#2			Pass
TC57	#3			Pass
TC58	#1	Chromium	Should display the selected list name.	Pass
TC59	#2			Pass
TC60	#3			Pass
TC61	#1	Firefox		Pass
TC62	#2			Pass
TC63	#3			Pass
TC64	#1	WebKit		Pass
TC65	#2			Pass
TC66	#3			Pass
TC67	#1	Chromium	Should have the configured metric descriptions.	Pass
TC68	#2			Pass
TC69	#3			Pass
TC70	#1	Firefox		Pass
TC71	#2			Pass
TC72	#3			Pass
TC73	#1	WebKit		Pass

Table B.1 continued from previous page

TC74	#2			Pass
TC75	#3			Pass
TC76	#1	Chromium	Should have the configured metric background colors.	Pass
TC77	#2			Pass
TC78	#3			Pass
TC79	#1	Firefox		Pass
TC80	#2			Pass
TC81	#3			Pass
TC82	#1	WebKit		Pass
TC83	#2			Pass
TC84	#3			Pass
TC85	#1	Chromium	Should have the configured list header color and height.	Pass
TC86	#2			Pass
TC87	#3			Pass
TC88	#1	Firefox		Pass
TC89	#2			Pass
TC90	#3			Pass
TC91	#1	WebKit		Pass
TC92	#2			Pass
TC93	#3			Pass
TC94	#1	Chromium	Should have the configured list column width and title.	Pass
TC95	#2			Pass
TC96	#3			Pass
TC97	#1	Firefox		Pass
TC98	#2			Pass
TC99	#3			Pass
TC100	#1	WebKit		Pass
TC101	#2			Pass

Table B.1 continued from previous page

TC102	#3			Pass
TC103	#1	Chromium	Should have the configured list background colors and line height.	Pass
TC104	#2			Pass
TC105	#3			Pass
TC106	#1	Firefox		Pass
TC107	#2			Pass
TC108	#3			Pass
TC109	#1	WebKit		Pass
TC110	#2			Pass
TC111	#3			Pass
TC112	#1	Chromium	Should have the configured list column width and type properties.	Pass
TC113	#2			Pass
TC114	#3			Pass
TC115	#1	Firefox		Pass
TC116	#2			Pass
TC117	#3			Pass
TC118	#1	WebKit		Pass
TC119	#2			Pass
TC120	#3			Pass
TC121	#1	Chromium	Should have the configured patient detail width.	Pass
TC122	#2			Pass
TC123	#3			Pass
TC124	#1	Firefox		Pass
TC125	#2			Pass
TC126	#3			Pass
TC127	#1	WebKit		Pass
TC128	#2			Pass
TC129	#3			Pass

Table B.1 continued from previous page

TC130	#1	Chromium	Should have the configured patient detail components bottom margins and heights.	Pass
TC131	#2			Pass
TC132	#3			Pass
TC133	#1	Firefox		Pass
TC134	#2			Pass
TC135	#3			Pass
TC136	#1	WebKit		Pass
TC137	#2			Pass
TC138	#3			Pass
TC139	#1	Chromium	Should have the configured patient detail components columns.	Pass
TC140	#2			Pass
TC141	#3			Pass
TC142	#1	Firefox		Pass
TC143	#2			Pass
TC144	#3			Pass
TC145	#1	WebKit		Pass
TC146	#2			Pass
TC147	#3			Pass

Appendix C

Real-time notifications tests

Table C.1: List and results of the test cases run for the configuration update notifications.

Code	Notification	Action	Message (contains the configuration code)	Recipients (configuration codes)	Expectation (besides receiving message)	Result
TC148	signalR. sendConfigurationUpdate	Change configuration name	1	98	No action is taken	Pass
TC149				1	Refetches configuration	Pass
TC150		Add a field to the patient list	98	99	No action is taken	Pass
TC151				98	Refetches configuration	Pass
TC152		Change an indicator's color	99	100	No action is taken	Pass
TC153				99	Refetches configuration	Pass
TC154		Remove a patient detail component	100	101	No action is taken	Pass
TC155				100	Refetches configuration	Pass

Table C.2: List and results of the test cases run for the patient admission notifications.

Code	Notification endpoint	Message (contains the location identifier)	Precondition	Expectation (besides receiving message)	Result
TC156	signalR. sendPatientAdmission	-2147483648	The dashboard displays the location	Refetches indicators and patient panel	Pass
TC157			The dashboard doesn't display the location	No action is taken	Pass
TC158		-2147483647	The dashboard displays the location	Refetches indicators and patient panel	Pass
TC159			The dashboard doesn't display the location	No action is taken	Pass
TC160		"a"		400 Bad Request and message not sent	Pass
TC161				400 Bad Request and message not sent	Pass

Table C.3: List and results of the test cases run for the patient discharge and patient update notifications.

Code	Notification endpoint	Message (contains the patient identifier)	Precondition	Expectation (besides receiving message)	Result
TC162	signalR. sendPatientDischarge	-2147482577	The dashboard displays the patient	Removes the patient from the list and its indicator entries	Pass
TC163		-2147481548			Pass
TC164		-2147482577	The dashboard doesn't display the patient	No action is taken	Pass
TC165		-2147481548			Pass
TC166		-2147482577	The dashboard displays the patient and has it selected	Removes the patient from the list, its indicator entries, and closes its details	Pass
TC167		-2147481548			Pass
TC168		"a"		400 Bad Request and message not sent	Pass
TC169				400 Bad Request and message not sent	Pass
TC170	signalR. sendPatientUpdate	-2147482577	The dashboard displays the patient	Refetches indicators and patient panel	Pass
TC171		-2147481548			Pass
TC172		-2147482577	The dashboard doesn't display the patient	No action is taken	Pass
TC173		-2147481548			Pass
TC174		-2147482577	The dashboard displays the patient and has it selected	Refetches indicators, patient panel and its details	Pass
TC175		-2147481548			Pass
TC176		"a"		400 Bad Request and message not sent	Pass
TC177				400 Bad Request and message not sent	Pass

Appendix D

Acceptance tests

Table D.1: List and results of the acceptance tests.

Code	**Priority**	**Implemented**	**Result**
R1	Must	Yes	Pass
R2			Pass
R3			Pass
R4			Pass
R5			Pass
R8			Pass
R9			Pass
R11			Pass
R13			Pass
R14			Pass
R15			Pass
R17			Pass
R6	Should		Pass
R7			Pass
R10			Pass
R16			Pass
R19			Pass
R12	Could		Pass
R18			Pass
R20			Pass
R21		No	-
R22	Won’t		-