



INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

**Departamento de Engenharia de Electrónica e Telecomunicações e de
Computadores**

**Software Capacitation in Hospital Professor Doutor Fernando
Fonseca Research Centre**

Valdemar Palminha Correia Antunes - 44865

Dissertação para obtenção do Grau de Mestre
em Engenharia Informática e de Computadores

Orientadores : Doutor Fernando Miguel Gamboa de Carvalho
Doutor Pedro Manuel de Almeida Carvalho Vieira

Júri:

Presidente: Doutor Nuno Miguel Soares Datia

Vogais: Doutor Fernando Miguel Gamboa De Carvalho
Doutor António Rito Silva

February, 2023



INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

**Departamento de Engenharia de Electrónica e Telecomunicações e de
Computadores**

**Software Capacitation in Hospital Professor Doutor Fernando
Fonseca Research Centre**

Valdemar Palminha Correia Antunes - 44865

Dissertação para obtenção do Grau de Mestre
em Engenharia Informática e de Computadores

Orientadores : Doutor Fernando Miguel Gamboa de Carvalho
Doutor Pedro Manuel de Almeida Carvalho Vieira

Júri:

Presidente: Doutor Nuno Miguel Soares Datia

Vogais: Doutor Fernando Miguel Gamboa De Carvalho
Doutor António Rito Silva

February, 2023

Acknowledgments

Quero expressar os meus agradecimentos aos meus orientadores, Professor Fernando Miguel Gamboa e Professor Pedro Vieira, pela orientação, pelo contínuo apoio tanto profissional como informal, pela disponibilidade no esclarecimento de dúvidas e reuniões e por terem contribuído conhecimentos que sem dúvida ajudaram a aprofundar o meu pensamento crítico e a fortalecer o meu pilar de conhecimento na minha carreira profissional como engenheiro informático.

Agradeço também à equipa do lado do Hospital que nos assistiram durante o desenvolvimento do projeto CASCIFFO no esclarecimento dos requisitos funcionais e na realização de testes de qualidade da plataforma.

Por fim, agradeço à minha família, em especial à minha mãe, e aos meus amigos por me terem sempre apoiado e incentivado no meu percurso académico.

Abstract

This document presents a thesis made on a joint-project named CASCIFFO, between the Lisbon Superior Institute of Engineering - Instituto Superior de Engenharia Lisboa (ISEL), and the Hospital Professor Doutor Fernando Fonseca (HFF). This project was made possible through the funding earned from the Clinical Investigation Agency Award and Biomedical Innovation (AICIB). This thesis aims to develop a platform and provide innovative mechanisms for interoperability with internal or external information systems, allowing, when desired, data synchronization, index search, identification data management and even access to detailed clinical data, the ability to manage and monitor clinical trials as well as their participants within the Clinical Research Center. The main contribution of this thesis will be the optimization and simplicity in managing clinical trials in order to facilitate the researchers' efforts in the management of clinical trials. We believe this platform to be an important step in the modernization of the HFF and UIC, bringing change in how the researchers view their institute and its importance. This report aims to specify the functional requisites of the platform CASCIFFO and situate the current status of development within the expected timeline.

Keywords: Management of Clinical Trials, Clinical Research, UIC, Hospital Professor Doutor Fernando Fonseca, HFF, Reactive, Non-blocking

Resumo

Com o intuito de modernizar o software utilizado pela Unidade de Investigação Clínica do Hospital Professor Doutor Fernando Fonseca, o projeto CASCIFFO foi realizado em parceria entre o Instituto Superior de Engenharia de Lisboa (ISEL) e o Hospital Professor Doutor Fernando Fonseca. O financiamento do projeto foi possibilitado pela atribuição do prémio pela Agência de Investigação Clínica e Inovação Biomédica (AICIB). O projeto tem como objetivos desenvolver uma plataforma, CASCIFFO, e fornecer mecanismos inovativos na interoperabilidade com sistemas de informação externos ou internos ao hospital. Permitir, quando desejado, sincronização de dados, índices de pesquisa, identificação de gestão de dados e acesso a detalhes de ensaios clínicos. Tem ainda como objetivo permitir a gestão e monitorização de ensaios clínicos, incluindo a gestão de participantes nos ensaios.

A contribuição principal esperada por este projeto será a otimização e simplicidade na gestão de ensaios clínicos no âmbito de facilitar o esforço feito pelos investigadores nesta gestão. Deste modo, espera-se que esta modernização venha a trazer uma mudança no pensamento dos investigadores na sua visão da instituição e a sua importância.

Palavras-chave: Gestão de ensaios clínicos, Investigação Clínica, Unidade de Investigação Clínica, Hospital Professor Doutor Fernando Fonseca, HFF, Reativo, Não-bloqueante

Contents

List of Figures	xv
List of Tables	xvii
List of Listings	xix
Acronyms	xxi
1 Introduction	1
1.1 Context and Motivation	1
1.2 Hospital Professor Doutor Fernando Fonseca	1
1.2.1 HFF's mission	2
1.2.2 Clinicial Research Unit	2
1.2.3 UIC's mission	2
1.2.4 Clinical Research	2
1.2.4.1 Types of Clinical Research	3
1.3 Project overview and main goals	3
1.3.1 Challenges	4
1.3.2 Main Goals	4
1.4 Document Structure	4

2	State of the art	7
2.1	Related work	7
2.2	Infrastructure	8
2.3	Access control	9
2.4	Processes	9
2.4.1	Clinical Investigation Proposals	9
2.4.2	Clinical Trials	11
2.4.3	Addenda to the contract	11
3	Functional and Non-Functional Requirements	13
3.1	General Features	13
3.1.1	Visualization and Management of Clinical Trials as a Process . .	14
3.1.2	Ability to edit and validate data (edit checks)	15
3.1.3	Access control based on different user profiles	16
3.1.4	Access by computer, tablet or smartphone	17
3.1.5	Ability to export information in numerical or graphical mode . .	17
3.1.6	Ability to customize the form of visualization	18
3.2	Clinical Component Features	18
3.2.1	View detailed Characteristics and evolution of clinical Trials in- cluding the tested medicine or technique in question	18
3.2.2	Monitoring the set of patients included in clinical Trials and their characteristics	25
3.2.3	Insertion of patient data in face-to-face or tele-consultation	27
3.2.4	Characteristics of the treatment associated with the clinical trial .	28
3.2.5	Monitoring of the patient's behavior under trial and its attendance	28
3.2.6	Monitoring of physical and financial assets	29
3.2.7	Monitoring of visits & recording of adverse events	31
3.3	Non-Functional Requirements	32

4	Architecture	35
4.1	Global architecture	35
4.1.1	Back-End module	36
4.1.2	Front-End module	37
4.2	Data model	38
4.2.1	Users, Roles and States	39
4.2.2	Proposal Entity	40
4.2.3	Clinical Research	41
5	Implementation	43
5.1	Back-end module	43
5.1.1	Framework Stack	44
5.1.2	PostgreSQL database	45
5.1.3	Initial Development	45
5.1.4	Spring Configuration	46
5.1.5	Repositories	48
5.1.6	Services	53
5.1.6.1	User service	53
5.1.6.2	Proposal service	55
5.1.7	Controllers	59
5.1.7.1	Mono/Flux syntax vs Suspend function syntax	61
5.1.8	Mappers	63
5.1.9	Authentication	64
5.1.10	Exception handling	66
5.2	Front-end module	68
5.2.1	Framework Stack	69
5.2.2	Configuration & Required Modules	69
5.2.3	Model Layer	73
5.2.4	Service Layer	73

5.2.5	View Layer	76
5.2.5.1	Security	79
5.2.6	Meeting the Functional Requirements	79
5.2.6.1	Financial contract tab	80
5.2.6.2	User management page	81
5.2.6.3	Data management page	81
5.2.6.4	Dashboard	82
5.3	Installation & Deployments	83
5.3.1	Installation	84
5.3.2	Heroku Deployment	84
5.3.3	On-Premises Deployment	86
5.3.4	Deploying the Application	87
5.3.5	Apache Hosting	88
6	Module Evaluation	91
6.1	Back-End Evaluation	91
6.2	Front-End Evaluation	95
7	Conclusions and Future Work	97
	References	101

List of Figures

3.1	Mock overview of clinical investigations proposals.	14
3.2	Mock overview of clinical trials.	15
3.3	Mock creation of a clinical investigation proposal.	16
3.4	Mock screen selecting clinical proposals to export into excel.	17
3.5	Mock overview of a clinical investigation proposal.	19
3.6	Mock overview of the contacts tab.	20
3.7	Mock overview of the observations tab.	21
3.8	Mock overview of the partnerships tab.	21
3.9	Mock overview of the protocol tab.	22
3.10	Mock overview of the chronology tab.	23
3.11	Mock overview of the details of a clinical trial.	24
3.12	Mock overview of scientific activities made within the scope of the investigation.	25
3.13	Mock overview of all participants included in the study.	26
3.14	Mock screen of adding a new patient as a participant in the study.	27
3.15	Mock overview of a visit's details.	28
3.16	Mock overview of patient details.	29
3.17	Mock detailed view of the income flow made in the investigation.	30
3.18	Mock detailed view of income flow divided by the investigator team.	31
3.19	Mock overview of visits scheduled in the clinical trial.	32

4.1	Layer Architecture	36
4.2	Back-end module global architecture.	37
4.3	Front-end module global architecture.	37
4.4	Entity relationship diagram of users, roles and states.	40
4.5	Entity relationship diagram centered on the Proposal Entity.	41
4.6	Entity relationship diagram centered on the Clinical Research Entity. . .	42
5.1	Initial project structure using <i>Spring Initializr</i>	45
5.2	Example of three existing services in CASCIFFO.	53
5.3	Financial contract tab UI.	80
5.4	User Management UI.	81
5.5	Data management UI.	82
5.6	Dashboard UI 1/2.	83
5.7	Dashboard UI 2/2.	83
5.8	Command to list currently listening processes.	88
6.1	H2 configurations.	93
6.2	Authentication request made in Postman.	95

List of Tables

3.1	Non-Functional Requirements Specification.	33
-----	--	----

List of Listings

5.1	Gradle configurations for production build.	47
5.2	Repository used for the Proposal entity.	49
5.3	A proposal aggregate repository function that fetches data from multiple sources.	51
5.4	Data class representing the entity Proposal.	51
5.5	User service interface.	54
5.6	User service implementation declaration.	54
5.7	Login method implementation.	55
5.8	Proposal service interface.	56
5.9	Proposal service implementation declaration.	56
5.10	Signature and implementation of the transitionState method.	57
5.11	Implementation of the handleStateTransition method 1 of 2.	58
5.12	Implementation of the handleStateTransition method 2 of 2.	59
5.13	Loading nested entities in an object using Mono/Flux syntax	62
5.14	Loading nested entities in an object using suspend function syntax	62
5.15	Mapper interface.	63
5.16	Mapper for state entity.	63
5.17	Configuration of spring security filter chain	65
5.18	Example of an exception annotated with @ResponseStatus	66
5.19	Example response using <i>Spring's</i> automatic response converter.	66

5.20	Sample controller with exception handler	67
5.21	Implementation of a global exception handler.	68
5.22	TypeScript compileer configuration file.	70
5.23	Package.json configuration file.	71
5.24	User Model Interface.	73
5.25	Utility fetch methods.	74
5.26	Utility fetch wrapper method implementation.	75
5.27	User Service implementation snippet.	75
5.28	Routes snippet for FE module.	78
5.29	Configuration snippet of Spring static content routing.	78
5.30	Gradle task added for <i>Heroku</i> deployment.	85
5.31	Apache configuration file.	90
6.1	Spring properties configurations with H2 database.	92
6.2	Spring test dependencies snippet.	92
6.3	Test method for the Proposal Repository.	93
6.4	Unit integration test for adding a patient to an on-going research.	94

Acronyms

BE	Back-End. 4, 36, 37, 44, 72, 73, 74, 78, 84, 85, 86, 97, 98, 100
DTO	Data Transfer Object. 60
FE	Front-End. 4, 36, 37, 46, 68, 69, 72, 74, 77, 78, 84, 85, 86, 95, 97, 98, 99, 100
HFF	Hospital Professor Doutor Fernando Fonseca. 1, 2, 3, 4, 9, 33, 84, 97, 99
HOC	Higher Order Component. 79
IoC	Inversion of Control. 64
JWT	Json Web Token. 64, 65, 79
PWA	Progressive Web Application. 69, 79, 86, 98, 99, 100
SPA	Single Page Application. 69
UI	User Interface. 13, 18, 35, 36, 37, 76, 77, 97, 99
UIC	Unidade de Investigação Clínica. 2, 3, 4, 8, 9, 10, 11, 12, 15, 16, 99

1

Introduction

This chapter reviews the introduction of the Hospital Professor Doutor Fernando Fonseca, followed by its Clinical Research Unit and their missions. It also introduces the platform being developed in the scope of this thesis, the motive behind it and its main goals to be achieved. The end of chapter describes the structure of the document.

1.1 Context and Motivation

This thesis is a part of the joint-project CASCIFFO and it's made within the scope of the curricular unit Tese Final de Mestrado, in the course Mestrado em Engenharia Informática e de Computadores. A strong motivator behind this project is the impact it'll have within the Clinical Research Unit, by facilitating the management and monitoring of clinical trials, CASCIFFO aims to alleviate this burden off the workload of researchers.

1.2 Hospital Professor Doutor Fernando Fonseca

The [Hospital Professor Doutor Fernando Fonseca \(HFF\)](#), first opened in 1995, is a first of the line hospital combining the professionals excelency with the most modern medical practices, searching to answer to a population of over 600.000 inhabitants of the municipalities of Amadora and Sintra [5]. The Institution develops assistance and research activities as well as providing education, pre- and post-graduation training.

1.2.1 HFF's mission

HFF's mission is to provide humanized and differentiated health care throughout a person's life cycle, in collaboration with primary and continuing health care, as well as other hospitals in the National Health Service's ('Serviço Nacional de Saúde') network [6].

1.2.2 Clinicial Research Unit

The Unidade de Investigação Clínica (UIC), created in March 2018, is an internal department within the Hospital that incorporates fundamental concepts of activities in line with the strategic objectives of the institution. The UIC is responsible for managing clinical trials and is characterized by a multidisciplinary team responsible for ensuring accuracy in the scientific planning of the submitted studies, for the fulfillment of clinical best practices by researchers and for the negotiation of contracts for projects financed by external promoters.

1.2.3 UIC's mission

UIC's mission is to promote quality clinical trials in an organized and sustainable manner. It achieves this by following guidelines that value systematic knowledge through the management of interfaces associated with Research, Development, and Innovation. In addition, it also complies with applicable ethical and legal provisions, for the benefit of the Hospital, the Community, the Patients, and the Families/Caregivers. The goal is to become a reference in the promotion of best practices in hospital clinical research and to consolidate a transversal scientific culture within the institution [6].

1.2.4 Clinical Research

Clinical research is a very important sector to the world of medicine and health care. It's through it that new medicine and new ways of treatment are discovered and tested through strict adherence to scientific accuracy measurements and best practices before being administrated to the general public.

1.2.4.1 Types of Clinical Research

There are two main types of clinical investigations, without intervention, which includes observational trials ("Estudos Observacionais"), and with intervention, which includes clinical trials ("Ensaio Clínicos"). The distinction between these trials comes down to the type of intervention between the study and the participants. Active intervention occurs when the researchers in a trial introduce any variable, such as a new medicine, that provokes any sort of change within the participant's behavior, health care or mindset. No intervention means that the team of researchers will not intervene in any way with the participants besides only monitoring them. Having stated these differences, we can clearly define observational trials and clinical trials. Observational Trials have no active intervention, they instead contemplate purely the aspect of observation, for example in the evaluation of a potential risk factor. On the other hand, Clinical Trials engage in active intervention, as such they can be characterized as a Clinical Research which involves any intervention that foresees any change, influence or programming of health care, behavior or knowledge of the participating patients or caretakers, with the end goal of discovering the effects it had on the participant's health. Clinical trials consist of a scientific controlled investigation, done on humans (healthy or ill), with the end-goal of establishing or confirming the safety and efficiency of the experimental medicine [21].

Clinical trials have shown to be a vital tool in the development and testing of vaccinations and treatments for the safety of the entire world population, especially now during the present *Novel Coronavirus (SARS-CoV-2)* pandemic. For this reason, the efficiency and simplicity in managing and monitoring the evolution of a clinical trial is essential.

1.3 Project overview and main goals

CASCIFFO is a joint project between [HFF](#) and [ISEL](#) and was developed through funding by the Clinical Investigation Agency Award and Biomedical Innovation (AICIB). CASCIFFO is a web-app that aligns with the UIC's goals by promoting efficiency and quality in the management of clinical research. CASCIFFO strives to make the visualization, monitoring, and management of clinical Trials as simple and straightforward as possible. It will allow the UIC/HFF to be modernized, bringing a shift in how patients, researchers, and promoters view and value their institution.

1.3.1 Challenges

The current procedure of a clinical investigation relies on e-mail exchanges between the external and internal parties involved, which adds a considerable amount of effort in the management and monitoring of clinical trials. Another concern is with the scheduling and monitoring of Clinical Trials patients, as there is currently no systematic way of distinguishing the types of appointments made for each patient. In this context, the application CASCIFFO, aims to provide a solution in order to enhance the efficiency in the management and monitoring of clinical research.

1.3.2 Main Goals

The application aims to develop and provide innovative mechanisms for interoperability with internal and external information systems, allowing, when desired, data synchronization, index search, identification data management and even access to detailed clinical data. CASCIFFO consists of two core modules, the Front-End (FE) module and the Back-End (BE) module. The FE supports interaction with users while the BE connects to an internal database system to the HFF/UIC and which aggregates the total information of this ecosystem. The interaction with users will depend on their role within the platform, displaying the appropriate information to each one.

1.4 Document Structure

This document is divided into the following 6 chapters:

- Chapter 1, this one, consists of the introduction of the HFF and HFF, their missions and goals, followed by explaining what a clinical research is and their types and concluding with an overview of the project and the goals to achieve.
- Chapter 2 describes the state of art of CASCIFFO and other related work.
- Chapter 3 consists of the business logic rules of CASCIFFO and the functional requirements.
- Chapter 4 presents the architecture and the infrastructure of the platform, as well as the justifications for the choices made.
- Chapter 5 describes the implementation details of both modules that make up the platform.

- Chapter 6 describes the method of evaluating and testing of each module.
- Lastly chapter 7 consists of the conclusions made and future work in the scope of the platform.

State of the art

This chapter consists of a detailed view over the concept and functional requirements of the CASCIFFO platform. It is structured with the following sections:

- Related work: Analysis of platforms with similar functionalities.
- Infrastructure: Description of technologies and framework used for the development of CASCIFFO.
- Access control: Identification and categorization of actors and their roles.
- Processes: Identification and detailing of the process flow.

2.1 Related work

Clinical trial management is an important factor to consider for any health care organization, to be able to track and manage any data concerning clinical trials and studies, from the moment of the proposal, to the clinical trial itself, the patient recruitment and monitoring management. To this effect, after an brief search and analysis of platforms providing these features, two platforms stood out: the Clinical Conductor Clinical Trial Management System (CTMS) [3] by Advarra [1] and RealTime-CTMS. Clinical Conductor CTMS is a premium service scalable to optimize finances, regulatory compliance, and overall clinical research operations such as financial, patient and

visit management including patient recruitment. The platform RealTime-CTMS is a leader in cloud-based software solutions for the clinical research industry and is dedicated to solving problems and providing systems that make the research process more efficient and more profitable [17]. CASCIFFO takes a tailored approach in building a straightforward platform that fits the needs of the Hospital Professor Doutor Fernando Fonseca. It features, in similarity to the aforementioned platforms, clinical trial management, from the moment its a proposal to the end of the clinical trial, allowing the entire progress to be tracked and analyzed; time line management in the aspect of tracking progress and reporting the completion date as well as possible overdue target dates; patient monitoring and visit management; financial management of clinical trials and each individual member of the investigation team and a role-based system for users of different internal departments. Furthermore the way CASCIFFO is planned to be developed, will allow for offline use. It is not as robust as a leading cloud-based software solution like RealTime-CTMS, however, it accomplishes its purpose of bringing innovation and simplicity to the management of clinical trials of the Hospital Professor Doutor Fernando Fonseca's Clinical Research Unit (Unidade de Investigação Clínica).

2.2 Infrastructure

The infrastructure of CASCIFFO, as mentioned previously, consists of two core modules, the front-end and the back-end. The front-end runs on a *Node.js* environment, using [React](#), a JavaScript library for building user interfaces [16], with Typescript to build all front-end functionalities. The dependencies are managed and installed using [npm](#), a software package manager, installer and the worlds largest software library [12]. [npm](#) was chosen over [yarn](#) [23] due to its larger community and support. The back-end executes on a Java virtual environment, utilizing *Spring Boot* and *Spring WebFlux* as the basis for building the server. *Spring Boot* facilitates the building and deployment of web applications by removing much of the boilerplate code and configuration associated with web development [18]. *Spring-WebFlux*, despite being a new technology, was chosen for its non-blocking web stack framework [20]. This technology allows for the use of *Spring Data R2DBC* [15], a driver that overcomes the inherent blocking limitation of drivers such as *JDBC* when accessing data in a database. The database connections via the *R2DBC* driver are non-blocking, returning streams of data in the form of *Flux* and *Mono* upon access. CASCIFFO has its own database, utilizing the framework [PostgreSQL](#), a powerful open-source object-relation database system. [PostgreSQL](#) was chosen for its earned reputation in its proven architecture, reliability, data integrity and

community support [13]. While CASCIFFO has its own database, the patient and medical staff data is planned to be imported from an internal database, "Admission", within the HFF/UIC. There are restrictions on the amount of queries made on the medical staff information, limiting this procedure to once per day.

2.3 Access control

Within the app CASCIFFO, in order for the management of clinical investigations to progress, it needs to be reviewed by many entities, such as the Administrative Council ("Concelho administrativo", CA), the Finance and Juridical department. Given this nature of CASCIFFO, there needs to be a well-defined structure of access control, so that each entity can contribute to the management of clinical investigations within the scope of their responsibilities. Each involved entity must have a role and a set of permissions. The roles identified are as follows: the UIC role, given to the investigators who can create and edit clinical investigation proposals; the Team Member role, given to investigators belonging to the team conducting the clinical investigation; the Management role, able to approve and reject clinical investigation proposals; the Finance and Juridical roles, given to collaborators whose function belongs within the Finance and Juridical departments, respectfully; and finally the Superuser role, who has complete access to every feature CASCIFFO has to offer. Each user of CASCIFFO have multiple roles.

2.4 Processes

This section details the types of processes occurring within the scope of the project. There are three identified processes consisting of the life-cycle of a clinical investigation proposal, the Clinical Trials and the contract addenda.

2.4.1 Clinical Investigation Proposals

From the instant a clinical investigation is kicked-off, it follows through a series of states and protocols that must be adhered to, in order to be completely validated. There are two types of clinical investigations: Clinical Trials and Observational Trials. Each state, except the terminal one, has an entity responsible, 'owner', for advancing the state. The flow of states is as follows:

1. Submitted ("Submetido"), 'owner=UIC';
2. Financial Contract Validation ("Validação do CF"), 'owner=Finance, Juridical';
3. External validation ("Validação externa"), 'owner=UIC';
4. Submission to the CA ("Submissão ao CA"), 'owner=UIC';
5. Internal validation ("Validação interna"), 'owner=CA';
6. Validated ("Validado").

The enumerated set of states corresponds to the life-cycle of a clinical trial Proposal. An Observational Trial Proposal consists of the enumerated states 1, 4, 5 and 6; it lacks a financial component and a promoter.

Taking the example of the submission of a clinical trial Proposal, an investigator starts by creating and submitting a proposal. Once it's submitted, the CA will be notified, via app and email. This state is described as *Submitted*. When the negotiation of the financial contract begins, the principal Investigator, who belongs to the UIC role, will advance the state to its next step in the proposal's evolution, *Validation of financial contract*, where users with roles of either 'Finance' and/or 'Juridical', which represents the Financial and Juridical internal departments, respectfully, will be notified that a proposal is ready to be validated. The validation will consist of a simple 'Accept' or 'Refuse' with added justification for the choice. In the case of either user with 'Finance' or 'Juridical' role reject the financial contract, both the validations will be reset and the principal investigator will be notified of the occurrence. Once it's accepted by both roles, the proposal will automatically advance into the next state, *External validation*. In this state the UIC will be notified of the change and asked to verify all the documents, including the final version of the financial contract. The UIC to the external promoter, requesting their signatures. When the reply is received via email, the UIC adds the received signatures and possible additional documents to the proposal in the CASCIFFO platform, advancing it to the next stage *Submission to CA*. In the state *Submission to CA*, the principal investigator will be notified both via the platform and email that a proposal requires their signature. Once the principal investigator submits their signature into the platform, he can advance the proposal's state into *Internal Validation*. The progression to the mentioned state will notify users with the 'CA' role stating that a proposal is ready for its final evaluation. Once a user with the role of 'CA' checks the proposal he can either validate it or not. In case it's not validated, the proposal will become 'canceled' with its life-cycle ending there, however, if it is validated, the proposal can become fully validated once the termination of another process is ends

successfully. This process, which can be considered a sub-process, is called the validation protocol. It starts in parallel when the proposal is first submitted. The purpose of this protocol is to validate the clinical investigation's ethical and safety values. It consists in the validation of the proposal by an internal agency, the [clinical investigations Ethics Comity](#) ("Comissão de Ética para Investigação Clínica", CEIC [2]). The protocol ends when the mentioned agency approves or rejects the proposal. Once it has successfully passed through the described validation protocol, the proposal becomes *Validated* and a Clinical or Observational Trial is automatically created, importing the core information from the proposal. If either process declares the proposal invalid, its state becomes 'canceled', notifying the UIC and showing the root cause of cancellation.

Each proposal is distinguished by six main properties, the principal investigator, the type of investigation, the type of therapeutic service it's integrated into (*i.e.* Oncology), the 'Sigla' which represents the name of the therapeutic or medicine, the partnerships involved in the investigation and the medical team participating in the investigation. Proposals with a financial component must also include the promoter of the investigation, in addition to the properties listed.

2.4.2 Clinical Trials

The life-cycle of a clinical trial is divided into three states: active, completed, and canceled. Starting with the active state, a clinical trial will become available for viewing and editing once its proposal has been accepted. Clinical trials, as a process, consist on the experimentation of new medicine or treatment on a set of participants. These participants can be added to the clinical trial directly from the application. The experimentation requires constant monitoring, through visits, on each participant. These visits can be scheduled either when a participant is added or created afterwards. In addition to monitoring participants, several studies can be made in the scope of the clinical trial, such as scientific articles, presentations, reports, etc.

2.4.3 Addenda to the contract

Throughout the life of a clinical trial, there can be made changes to the study's contract, be it changing the investigator team or other factors that impact the standard run of the study. These changes pass through two entities before being applied; the UIC and the CA. The addenda can only be made once a clinical trial is active, which means its proposal has already been approved. The addenda has five different states: submitted 'Submetido', internal validation by UIC 'Validação interna', internal validation by

CA 'Validação interna', the terminal state validated 'Validado' and finally the terminal canceled state 'Indeferido'. The first four mentioned states are sequential, with the last one being an exception state. The sequential flow of states have an entity responsible for advancing their state, the 'owner'. Listed below, in similar fashion to the states presented in the proposal process, is the aforementioned sequence:

1. Submitted ("Submetido"), 'owner=UIC';
2. Internal validation ("Validação interna"), 'owner=UIC';
3. Internal validation ("Validação interna"), 'owner=CA';
4. Validated ("Validado").

3

Functional and Non-Functional Requirements

This chapter details the functional requirements and presents a mock User Interface (UI) that will satisfy the requirement.

The main features of CASCIFFO can be separated into two groups: general functionalities and clinical component. These two groups contain the functional requirements of CASCIFFO.

This chapter's structure is as follows:

- General features - details each of the general features of the platform;
- Clinical component features - details each of the clinical component features of the platform;
- Non-functional requirements - describes the non-functional requirements.

3.1 General Features

The general features of CASCIFFO consist in the following:

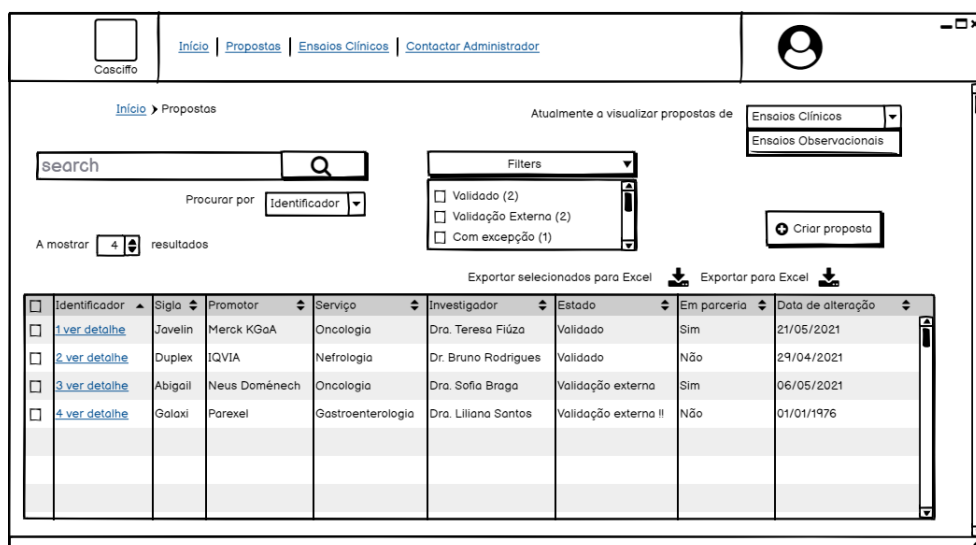
- Visualization and management of Clinical Trials as a process;

- Ability to edit and validate data (edit checks);
- Access control based on different user profiles;
- Access by computer, tablet or smartphone;
- Ability to export information in numerical or graphical mode;
- Ability to customize the form of visualization.

Each of these features will be detailed in this section.

3.1.1 Visualization and Management of Clinical Trials as a Process

To view and manage a clinical trial as a process, a user needs only to view the general overview of Clinical Trials or clinical investigations Proposals. As in figure 3.1 and figure 3.2, the user has an overview of the all submitted proposals and clinical trials with their main characteristics, such as, the identification, current state, last alteration date, the principal investigator and whether it has partnerships or not.



The image shows a web application interface for managing clinical trial proposals. At the top, there is a navigation bar with links: 'Início', 'Propostas', 'Ensaios Clínicos', and 'Contactar Administrador'. A user profile icon is on the right. Below the navigation bar, the main content area has a search bar with the text 'search' and a magnifying glass icon. To the right of the search bar is a dropdown menu labeled 'Atualmente a visualizar propostas de' with options 'Ensaios Clínicos' and 'Ensaios Observacionais'. Below the search bar is a section labeled 'Procurar por' with a dropdown menu showing 'Identificador'. To the right of this is a 'Filters' section with checkboxes for 'Validado (2)', 'Validação Externa (2)', and 'Com exceção (1)'. Below the filters is a 'Criar proposta' button. At the bottom, there is a table with columns: 'Identificador', 'Sigla', 'Promotor', 'Serviço', 'Investigador', 'Estado', 'Em parceria', and 'Data de alteração'. The table contains four rows of data, each with a link to view details (e.g., '1.ver detalhe').

Identificador	Sigla	Promotor	Serviço	Investigador	Estado	Em parceria	Data de alteração
1.ver detalhe	Javelin	Merck KGaA	Oncologia	Dra. Teresa Fiúza	Validado	Sim	21/05/2021
2.ver detalhe	Duplex	IQVIA	Nefrologia	Dr. Bruno Rodrigues	Validado	Não	29/04/2021
3.ver detalhe	Abigail	Neus Domènech	Oncologia	Dra. Sofia Braga	Validação externa	Sim	06/05/2021
4.ver detalhe	Galaxi	Parexel	Gastroenterologia	Dra. Liliana Santos	Validação externa II	Não	01/01/1976

Figure 3.1: Mock overview of clinical investigations proposals.

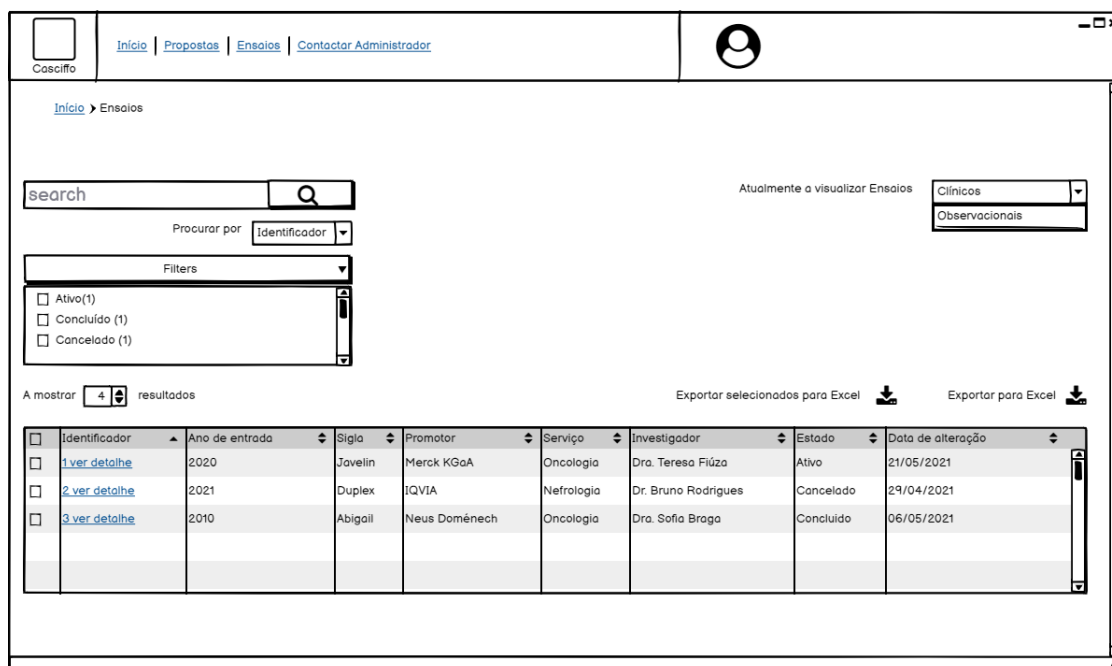


Figure 3.2: Mock overview of clinical trials.

When a user clicks the link view details ("ver detalhe"), he will be redirected to a screen displaying the details of the target clinical investigation.

3.1.2 Ability to edit and validate data (edit checks)

Within the CASCIFFO platform, the UIC and internal departments can view the details of a certain clinical investigation proposal and edit or validate according to their roles. Users with 'UIC' role will be able to create and edit their own Investigations, whereas the internal departments, with roles of 'CA', 'Finance' and 'Juridical' will only be able to validate the proposals. The creation of a proposal starts in the overview of proposals screen, from there a user can click on Create Proposal ("Criar proposta") and will be redirected to the screen illustrated in figure 3.3. Here an investigator can choose what type of investigation this proposal corresponds to, either an observational or clinical trial. In the case a clinical trial is selected, more options will be shown since clinical trials have a corresponding financial component. The data fields corresponding to the therapeutic area, the service and the pathology of an investigation are restricted to a set of possible inputs. In addition, the members of a medical team also belong to an already defined database, being validated at the time of creation of the medical team for the investigation. Certain fields cannot be changed once a proposal has been submitted, such as the promoter, the partnerships and the type of investigation can only be defined during the creation of a proposal.

The mockup shows a web interface for creating a clinical investigation proposal. The top navigation bar includes links for 'Início', 'Propostas', 'Ensaio Clínico', and 'Contactar Administrador'. The main content area is titled 'Proposta' and contains several sections:

- Parcerias:** A toggle switch is turned on. Below it are fields for 'Sigla' (with an asterisk) and 'Serviço' (with a dropdown menu labeled 'Escolher Serviço').
- Área terapêutica:** A dropdown menu labeled 'Escolher área'.
- Patologias:** A dropdown menu labeled 'Escolher Patologia'.
- Investigador:** A text input field.
- Tipo de investigação:** Two radio buttons: 'Ensaio Clínico' (selected) and 'Ensaio Observacional'.
- Promotor:** A text input field.
- Tipo de Promotor:** Two radio buttons: 'Comercial (e.g. Farmaceutico)' (selected) and 'Não comercial (e.g. Académico)'.
- Adicionar Contrato Financeiro:** A button with a document icon and an asterisk.
- Submeter:** A button at the bottom of the proposal form.

On the right side, there is a section for 'Equipa de Investigação' with a list of 'Médico 1', 'Médico 2', and 'Médico 3', and an 'Adicionar à equipa' button. Below this is a section for 'Parceiro 1' with a large 'E' icon and fields for 'Nome', 'Website', 'Responsável', 'Contacto', and 'Email'. A button 'Adicionar observações' is located below these fields. At the bottom right, there is a large plus icon and the text 'Adicionar Parceria'.

Figure 3.3: Mock creation of a clinical investigation proposal.

3.1.3 Access control based on different user profiles

The access control within the CASCIFFO application is based on roles. As described in section 1, there are defined roles for each type of user. A user with the role 'UIC' can manage their own clinical investigation, being able to edit and have a hand in advancing the state. In addition, it also has an overview over all ongoing investigations, not being able to edit those that weren't created by said user. Once this user logs in, a dashboard showing overall statistics of the platform, *i.e.* number of active clinical trials, number of submitted proposals, etc. The role 'Financial' is responsible for validating and updating the financial components of the investigations, hence when a user with this role logs in, an appropriate financial management screen will be displayed, whereas the 'Juridical' component validates the juridical component. The role 'CA' is responsible for giving the final decision in whether an investigation proposal can have the go-ahead to begin their clinical trials or observations. A user of role 'CA' once logged in, will be shown a primary screen displaying the overview of clinical investigation proposals awaiting validation. Finally, we have the 'Superuser' role, which

besides having the ability to execute every mentioned action, it can also create new types of services, therapeutic areas and pathologies.

3.1.4 Access by computer, tablet or smartphone

CASCIFFO is a web-application which can be accessed via any device or browser. CASCIFFO offers an extended functionality to browsers that support service workers, since it utilizes the Progressive Web Application (PWA) framework, allowing it to be installed and used as an application. Among the features a PWA allows, the framework was chosen by its ability to let an application run in offline-mode and versatility in that it can be installed via the browser, and used in similarity to a standalone mobile app.

3.1.5 Ability to export information in numerical or graphical mode

CASCIFFO offers the ability to export information via excel, and visualize graphic data within the app. There is a feature, shown in figure 3.4 that allows a user to export data, *e.g.* a selected number of clinical investigation proposals. This feature is present in the screens showing listed data, *e.g.* list of clinical investigation proposals, and the details of each clinical investigation, proposal or trial.

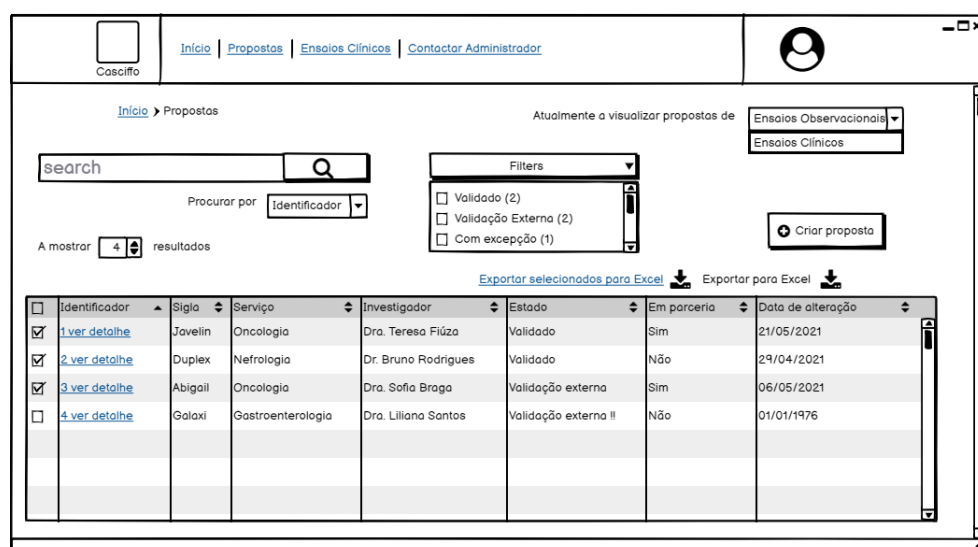


Figure 3.4: Mock screen selecting clinical proposals to export into excel.

3.1.6 Ability to customize the form of visualization

The ability to customize the form of visualization will be available in the initial screen of the app, the Dashboard, where the user will be able to view different types of graphs showing statistics based on the states of clinical investigations.

3.2 Clinical Component Features

The clinical component features of CASCIFFO consist in the following:

- View detailed characteristics and evolution of clinical trials including the tested medicine or technique in question;
- Monitoring the set of patients included in clinical trials and their characteristics;
- Insertion of patient data in face-to-face or tele-consultation;
- Characteristics of the treatment associated with the clinical trial;
- Monitoring of the patient's behavior under trial and its attendance;
- Monitoring of physical and financial assets;
- Monitoring of visits & recording of adverse events.

In this section, all of the mentioned features will be detailed along with the visualization of a mock UI.

3.2.1 View detailed Characteristics and evolution of clinical Trials including the tested medicine or technique in question

To view detailed information about a clinical investigation, one needs to first overview the clinical investigations and then click on the details of a desired clinical investigation. Considering this option, the user will be redirected to a screen detailing the study. The evolution of any clinical investigation consists of its proposal followed the trial activity once the proposal has been fully validated. In figure 3.5, the details of a clinical trial proposal can be viewed. The flow of state of a proposal is shown in the form of a bar in a straight forward manner. Each state corresponds to a division, box, in the bar and has three properties: the name of the State; the date it was completed in, if the

state has otherwise not been completed, then a sequence of dashes will appear in its place; the deadline at which it should be completed and finally the entity responsible for advancing the state. It is possible to click in the box of a state to display information about the events that occurred during the time the proposal was in that state.

The mockup shows a web interface for a clinical investigation proposal. The top navigation bar includes links for 'Início', 'Propostas', 'Ensaios Clínicos', and 'Contactar Administrador'. The main content area is titled 'Proposta 1' and features a tabbed interface. The 'Proposta' tab is selected, displaying a table of states. The table has columns for 'Estado', 'Data', 'Limite', and 'Responsável'. The states listed are 'Submissão', 'Negociação de CF', 'Validação interna', 'Validação externa', 'Submissão ao CA', 'Validação interna', and 'Validado'. Below the table, there are input fields for 'Sigla', 'Serviço', 'Promotor', 'Área terapêutica', 'Patologia', 'Investigador', and 'Equipa de Investigação'. The 'Equipa de Investigação' section includes a list of doctors: 'Médico 1', 'Médico 2', 'Médico 3', 'Médico 4', and 'Médico 5'. At the bottom, there are links for 'Assinaturas do promotor' and 'Documento adicional 1', and a progress bar labeled 'Ensaios Clínicos'.

Figure 3.5: Mock overview of a clinical investigation proposal.

A clinical trial proposal considers five components alongside its principal details displayed as tabs and listed below:

- the Contacts ("Contactos") tab which corresponds to comments related to external communications, viewed in figure 3.6;
- the Observations ("Observações") tab which contains comments made in regard to the proposal, viewed in figure 3.7;
- the Partnerships ("Parcerias") tab, where one can find the partnerships involved in the study proposal, viewed in figure 3.8;
- the validation Protocol ("Protocolo") tab, where it can be viewed the current state of affairs of the process described in section 3.1.1, and in figure 3.9;

- the Chronology ("Cronologia") tab that displays the timeline of events, as in figure 3.10. Events include deadlines introduced by the user and the transition of states. The user will have the ability to specify the scope of the timeline, selecting the time range and type of events to view.

All tabs, except the one pertaining to the Protocol validation, will have present the current state of the proposal.

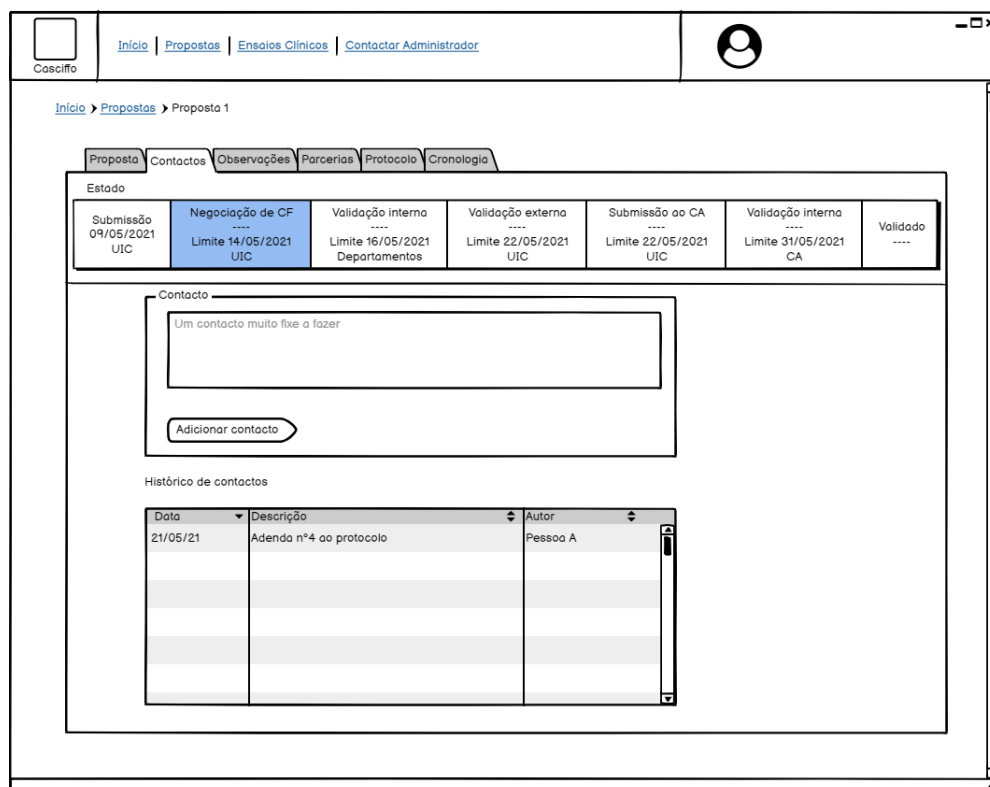


Figure 3.6: Mock overview of the contacts tab.

[Início](#) | [Propostas](#) | [Ensaios Clínicos](#) | [Contactar Administrador](#)

Início > Propostas > Proposta 1

[Proposta](#) | [Contactos](#) | [Observações](#) | [Parcerias](#) | [Protocolo](#) | [Cronologia](#)

Estado

Submissão 09/05/2021 UIC	Negociação de CF Limite 14/05/2021 UIC	Validação interna Limite 16/05/2021 Departamentos	Validação externa Limite 22/05/2021 UIC	Submissão ao CA Limite 22/05/2021 UIC	Validação interna Limite 31/05/2021 CA	Validado
--------------------------------	---	--	--	--	---	-------------------

Observação

Uma observação muito fixe a fazer

Adicionar Observação

Histórico de Observações

Data	Descrição	Autor
data	observação feita	Pessoa A

Figure 3.7: Mock overview of the observations tab.

[Início](#) | [Propostas](#) | [Ensaios Clínicos](#) | [Contactar Administrador](#)

Início > Propostas > Proposta 1

[Proposta](#) | [Contactos](#) | [Observações](#) | [Parcerias](#) | [Protocolo](#) | [Cronologia](#)

Estado

Submissão 09/05/2021 UIC	Negociação de CF Limite 14/05/2021 UIC	Validação interna Limite 16/05/2021 Departamentos	Validação externa Limite 22/05/2021 UIC	Submissão ao CA Limite 22/05/2021 UIC	Validação interna Limite 31/05/2021 CA	Validado
--------------------------------	---	--	--	--	---	-------------------

Parceiro 1

E

Nome:

Website:

Responsável:

Contacto:

Email:

[Ver observações](#)

Figure 3.8: Mock overview of the partnerships tab.

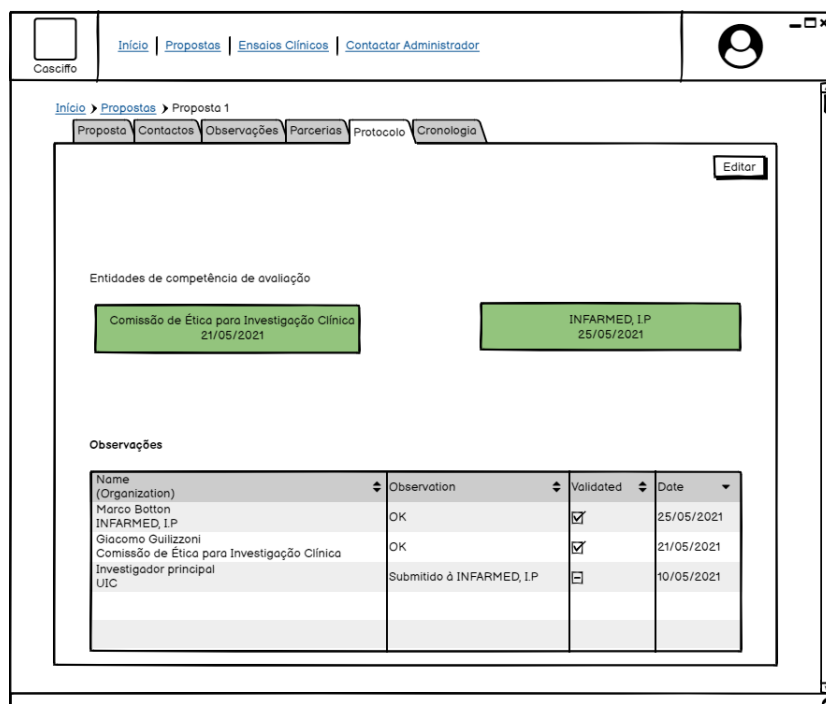


Figure 3.9: Mock overview of the protocol tab.

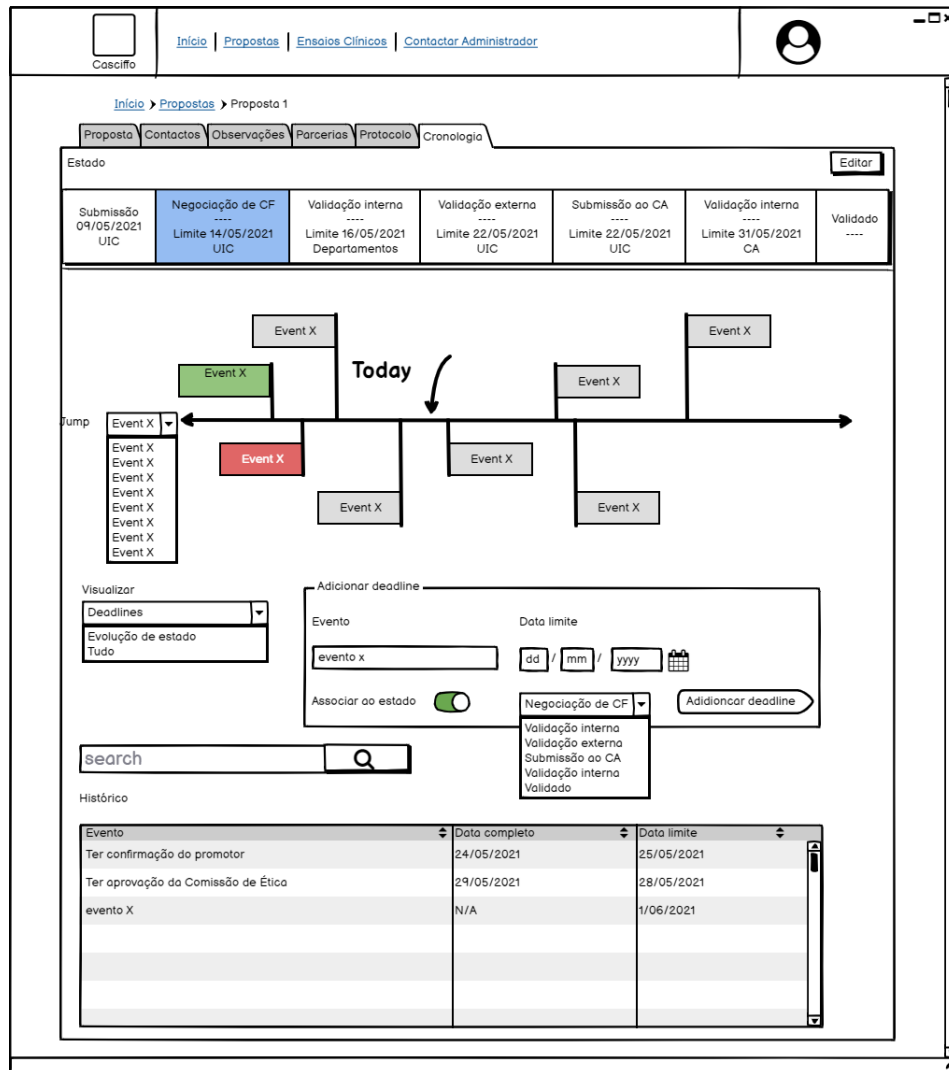


Figure 3.10: Mock overview of the chronology tab.

Once the proposal has reached its successful terminal state, as in section 2.4.1 a clinical trial will be created. To access the newly created trial, the user can either click the button "Ensaio Clínico" from the proposal details screen, as in figure 3.5, or from the overview of clinical trials click on the details of one. In the screen dedicated to the clinical trial and presented in figure 3.11, there are five different tabs:

- the clinical trial tab ("Ensaio Clínico") that displays the characteristics of the study;
- the scientific activities tab ("Atividades científicas"), viewed in figure 3.12, which includes scientific work in the scope of the study, such as articles, thesis, reports, etc.;
- the visits tab ("Visitas") where the investigator team can have an overview of the history and scheduled visits;

- the patients tab ("Pacientes") with the purpose of showing an overview of the participants included in the trial;
- the partnerships tab ("Parcerias"), similar to the proposal's version of partnerships tab, displays the partnerships involved in the study;
- the financial management ("Financiamento"), where one can view the flow of monetary gain from visits and the partition between the investigator team.

[Início](#) | [Propostas](#) | [Ensaios Clínicos](#) | [Contactor Administrador](#)

[Início](#) > [Ensaios Clínicos](#) > Ensaio 1

Ensaio Clínico | Adendas | Atividades científicas | Visitas | Pacientes | Parcerias | Financiamento

Estado
☒ Ativo
☐ Concluído
☐ Cancelado

Sigla
 Javelin

Serviço
 Oncologia

Promotor
 Merck KGaA

Investigador
 Dra. Teresa Flúza

Fases
 Fases 2 e 3

EudraCT
 2017-002195

Iniciativa
 Promotor

Financiamento
 Com Financiamento

Protocolo
 CNT01959CRD3001

Industria
 Javelin Research & Development

Área terapêutica
 Javelin Research & Development

Amostra
 2000

CRO
 Parexel Internacional

Duração
 5 anos e 6 meses

Data início
 01/10/2019

Data prevista conclusão
 31/05/2025

Doentes previstos
 5

Dossiers

Label	Número

Ver Proposta

Figure 3.11: Mock overview of the details of a clinical trial.

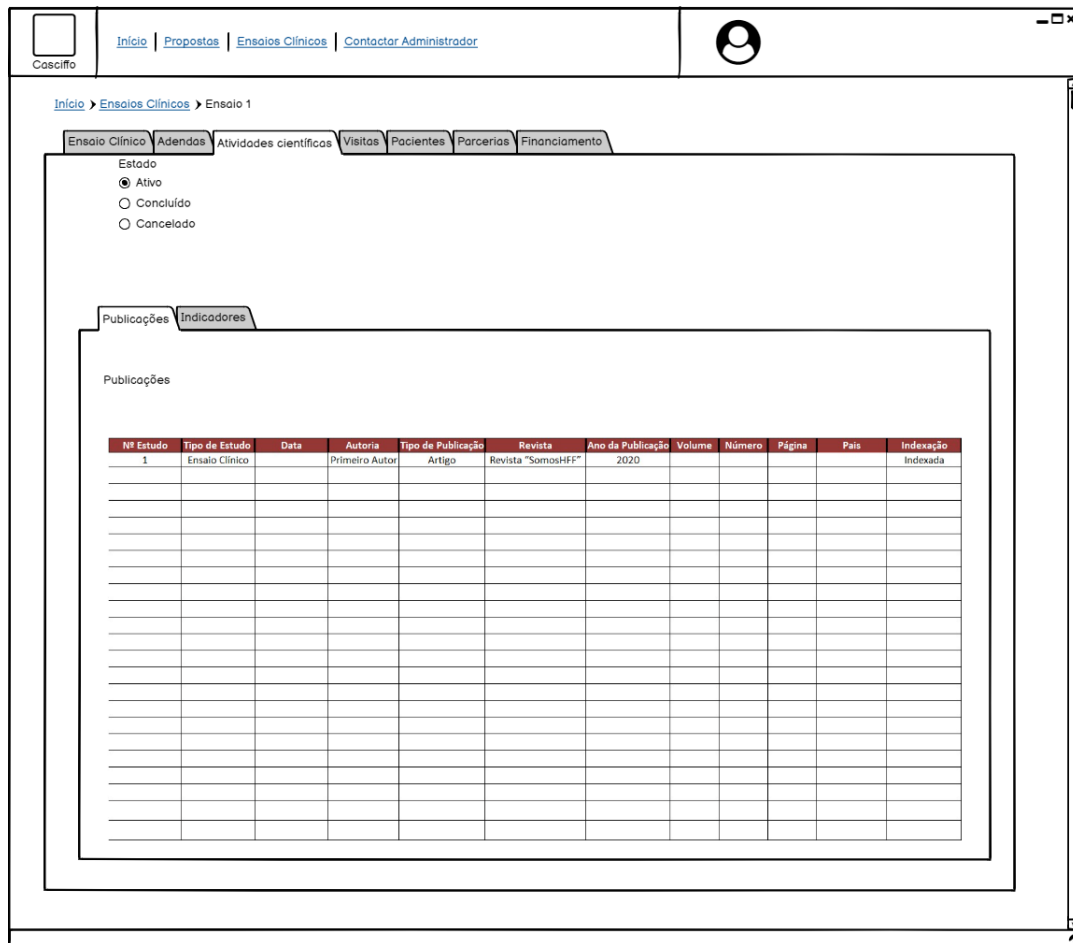


Figure 3.12: Mock overview of scientific activities made within the scope of the investigation.

3.2.2 Monitoring the set of patients included in clinical Trials and their characteristics

The details of the set of patients involved in a clinical trial will be displayed when viewing the details of said clinical trial, under the patients ("Pacientes") tab, illustrated in figure 3.13. This tab displays the set of current participants undergoing the clinical trial and information such as the participant number, used to identify the participant throughout the study, the name of the participant, their age, the treatment branch they were assigned to within the scope of the study, their last and the closest upcoming visit. Included in this screen are the buttons Randomize ("Randomizar") and Add ("Adicionar"). The button "randomize" will randomly assign participants to treatment branches within the study, this one-time procedure is to be used once all the participants have been added to the clinical trial. The button "add", will begin the procedure to add a patient. Upon clicking this button, the user will be shown a small box,

as illustrated in figure 3.14, where he can input the name of the participant and then is also given the chance to immediately schedule several visits.

Ensaios Clínicos

Estado

☒ Ativo

☐ Concluído

☐ Cancelado

Randomizar

Adicionar

Número	Name	Age	Braço de Tratamento	Última Visita
1	João Caldeirão	32	N/A	12-02-2021

Figure 3.13: Mock overview of all participants included in the study.

Figure 3.14: Mock screen of adding a new patient as a participant in the study.

3.2.3 Insertion of patient data in face-to-face or tele-consultation

The insertion of patient data in the context of a visit is made by the investigator associated to the mentioned visit. In order to do this, the investigator has to navigate to the details of the visit, from the overview of clinical trials, to the details of the trial followed by the overview of visits and finally the details of the considered visit. Here the investigator can input observational data into the field "Observações" which will represent the observations made throughout the visit or tele-consultation. He is also asked to mark the attendance of the participant by clicking on a simple button "Marcar presença", as illustrated in figure 3.15.

Figure 3.15: Mock overview of a visit's details.

3.2.4 Characteristics of the treatment associated with the clinical trial

The characteristics of the treatment associated with clinical trials consists of three main factors, the service area it is within, the therapeutic field and the pathology the clinical trial aims to investigate/treat. These details can all be found in the detailed view of a clinical trial under the tab "Ensaio Clínico".

3.2.5 Monitoring of the patient's behavior under trial and its attendance

The monitoring of the patient's behavior and its attendance, can be tracked via the visits and in twofold. The first is to filter the visits, under the visits tab, to show only the visits related to the participant in question. The second is to view the details of this participant in particular, from the overview of total participants in the study, clicking on the details of the participant and then checking the tab "Attendance", as in figure 3.16.

Atividades efetuadas com o participante

search

Procurar Tipo de visita

A visualizar:

A mostrar resultados

Exportar selecionados para Excel Exportar tabela para Excel

<input type="checkbox"/>	Identificador	Tipo de Visita	Data hora	Responsável	Observação	Evento Adverso
<input type="checkbox"/>	1 ver detalhe	Screening	20/05/2021 14:45	Nome Fantastico	Admissão do participante com id 1	Não
<input type="checkbox"/>	2 ver detalhe	Monitorização	21/05/2021 14:45	Nome ainda melhor	OK	Não
<input type="checkbox"/>						
<input type="checkbox"/>						

Figure 3.16: Mock overview of patient details.

3.2.6 Monitoring of physical and financial assets

In regard to the monitoring of physical assets, these can be viewed under the tab "Ensaio Clínico" while in the detail page of the clinical trial in question. The assets to be monitored are documentation archives, consisting of three fields: the Volume, a Label and the total Number. CASCIFFO also offers another type of monitoring, the monetary flow. As described in section 2.4.2, each clinical trial has a financial management section that discriminates the earnings made by visit and the partitions of each team member involved in the study. Under the tab "Financiamento", the general monetary information, such as, total balance ("saldo"), amount per participant, etc., is displayed on a top section of the page. Below this section it can be observed another two tabs, differentiating the income made from the investigation team and the clinical trial itself. The first tab "EC", shown in figure 3.17, shows the income made according to the visits done, whereas the second tab "Equipa", illustrated in figure 3.18, shows the income by team member.

[Início](#) | [Propostas](#) | [Ensaios Clínicos](#) | [Contactar Administrador](#)

[Início](#) > [Ensaios Clínicos](#) > Ensaios 1

Ensaio Clínico | Adendos | Atividades científicas | **Visitas** | Pacientes | Parcerias | Financiamento

Estado: ☒ Ativo
☐ Concluído
☐ Cancelado

Saldo: \$\$\$\$

Valor por participante: \$\$\$\$

Encargo por participante: \$\$\$\$

Nº Participantes: 10

EC | Equipa

search

Procurar por: Data

Tipo de movimento: Responsável

A mostrar: 1 resultados

Nova entrada

também podes procurar por datas!
 A pesquisa 01/02/2020-09/05/2021 demonstra os resultados desse período de tempo!

Data	Tipo	Motivo	Responsável	Quantidade
15-02-2021	Entrada	Visita N	Promotor	\$\$\$\$

Figure 3.17: Mock detailed view of the income flow made in the investigation.

Início | Propostas | Ensaios Clínicos | Contactar Administrador

Início > Ensaios Clínicos > Ensaio 1

Ensaio Clínico | Adendos | Atividades científicas | Visitas | Pacientes | Parcerias | Financiamento

Estado: ☒ Ativo, ☐ Concluído, ☐ Cancelado

Saldo: \$\$\$\$

Valor por participante: \$\$\$\$

Encargo por participante: \$\$\$\$

N° Participantes: 10

EC: Equipa

search

Procurar por: Identificador

também podes procurar por datas! A pesquisa 01/02/2020-09/05/2021 demonstra os resultados desse período de tempo!

A mostrar: 1 resultados

Nova entrada

Data	Tipo de movimento	Responsável	Quantidade	Repartição %	Encargo	Membro de equipa
15-02-2021	Entrada	Promotor	\$\$\$\$	34%	\$\$\$\$	Investigador principal
15-02-2021	Entrada	Promotor	\$\$\$\$	33%	\$\$\$\$	Investigador 1
15-02-2021	Entrada	Promotor	\$\$\$\$	33%	\$\$\$\$	Investigador 2

Figure 3.18: Mock detailed view of income flow divided by the investigator team.

3.2.7 Monitoring of visits & recording of adverse events

The monitoring of visits during a clinical trial can be tracked and viewed under the tab "Visitas", in the detailed screen of a clinical trial. In this setting, any investigator (associated to the clinical trial) is able to view and track the past and scheduled visits made by the team. However, only the investigators associated to a visit are able to manipulate them. When creating a visit, that visit will automatically be associated to its creator, giving also the option to associate other investigators to the same visit, allowing them to freely edit the visit details. Along with this option, the investigator is required to fill in the fields depicted in figure 3.19. These fields are as follows: the periodicity ("Períodicidade") that can be turned ON in case the visit is a reoccurring one with the ability to schedule at a custom interval in days; the type of visit ("Tipo de visita"), that has three possible inputs, Screening, indicating the first visit, Monitoring ("Monitorização") indicating a motoring visit and finally a Closeout visit, which indicates the final visit done to a participant. In case the visit is periodic, the field of 'end date' ("Data fim") becomes mandatory to fill.

Ensaios Clínicos > Ensaios Clínicos > Ensaios 1

Ensaios Clínicos | Adendas | Atividades científicas | Visitas | Pacientes | Parcerias | Financiamento

Estado
☒ Ativo
☐ Concluído
☐ Cancelado

A visualizar: Histórico de visitas | Visitas agendadas

search [Q]

Procurar por: Identificador

tambem podes procurar por datas!
 A pesquisa 01/02/2020-09/05/2021 demonstra as consultas nesse periodo de tempo!

A mostrar: 1 resultados

Visitas agendadas

Agendar visita

Data hora	Investigador	Paciente	Tipo de visita	Observação	Evento Adverso	
15-02-2021 14:45	Giorno Giovanni	Arriverderci	Screening	ok	N/A	ver detalhes
16-02-2021 14:45	Giorno Giovanni	Arriverderci	diaria	ok	N/A	ver detalhes
17-02-2021 14:45	Giorno Giovanni	Arriverderci	diaria	ok	N/A	ver detalhes
18-02-2021 14:45	Giorno Giovanni	Arriverderci	Closeout	ok	N/A	ver detalhes

Figure 3.19: Mock overview of visits scheduled in the clinical trial.

When a visit occurs, the associated investigators will have access to the observations ("Observações") field and adverse event ("Evento adverso") warning. In addition to this, the field "Marcar presença" is now available to mark attendance.

3.3 Non-Functional Requirements

This section quickly lists, in a tabular fashion, the non-functional requirements noted during the analysis and development of the platform.

Table 3.1: Non-Functional Requirements Specification.

Ref.	Description	Label
1	The user should be able to receive and read notifications within the application. Notifications should include:	Required
1.1	Proposal status updates.	Required
1.2	New Contacts.	Optional
1.3	New Observations.	Optional
1.4	When a time line event completes or is overdue.	Optional
1.5	Research status updates.	Required
2	All data presented with tables should have pagination.	Optional
3	Apache hosting.	Required
4	Integration with HFF's patient database.	Optional

4

Architecture

In this chapter we take a look at the architecture developed in the scope of CASCIFFO. The chapter is structured as follows:

- Global architecture 4.1 - describes the global architecture used;
- Data Model 4.2 - presents the conceptual representation of the data structures that the application uses, and it defines how data is organized and related within the application.

4.1 Global architecture

The initial approach taken towards CASCIFFO was made following the monolithic architecture. A monolithic architecture consists in the development of software components within a single codebase. From an operating system's point of view, these components operate within one single application. The advantages brought by this design are centered around simplicity, the application is easier to deploy, test, debug and monitor. Applying this concept into context, the components developed were divided into three layers, the UI, domain and data access layer. The first layer - data access layer - contains the data created and manipulated by the platform CASCIFFO; the second layer - domain layer - responsible for security and business processes within the platform; lastly the UI layer corresponds to the presentation and logic of the user interface where user interaction begins.

The Back-End and Front-End modules have distinct responsibilities within these layers, while the FE contains all of the UI layer, the BE consists of both domain and data access layer, as illustrated by 4.1. Finally, in addition to these layers, there's also a component inherent, although external, to the data access layer, the data base component, which is where all the data is stored.

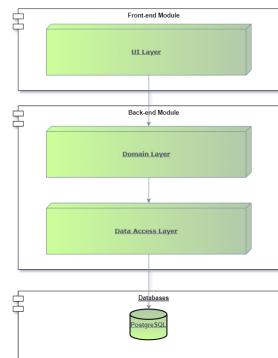


Figure 4.1: Layer Architecture

One of the drawbacks of this architecture is scalability and maintenance; the need of redeploying the entire application when a change is made, for example, a change in the Front-End module should not imply a redeployment of the Back-End module as well. In wake of this, since we want a scalable solution, the layer based architecture was maintained while the monolithic aspect of a single codebase was discarded. The modules were separated, allowing the Front-End module and Back-End module to run independently of each other. Following the segregation of the modules, we also achieve independent deployments.

4.1.1 Back-End module

The BE module consists of the business logic and database access layers. The architecture used follows the Controller - Service - Repository pattern, frequently used in Spring Boot applications, as illustrated in figure 4.2. This pattern consists in having the controllers that manage REST requests and delegate business logic operations to a Service and after having processed the request, an appropriate response shall be created accordingly.

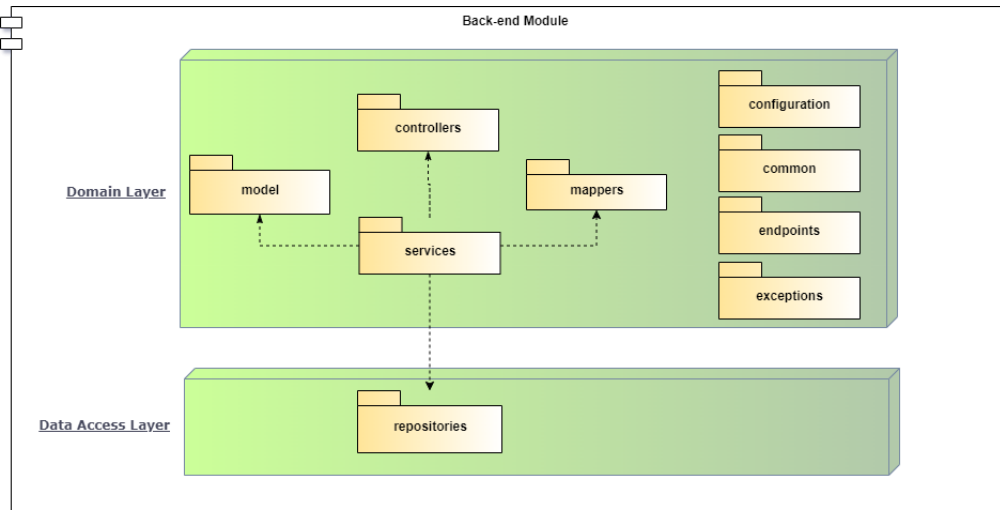


Figure 4.2: Back-end module global architecture.

4.1.2 Front-End module

The FE module consists of the UI layer. The architecture, as viewed in figure 4.3, follows the same principle as the BE module, however, this time having designated Views for each page, each view utilizes a Service for data retrieval and both manipulate Model data. The Services are a stateless abstraction of the domain layer linked to their respective data model.

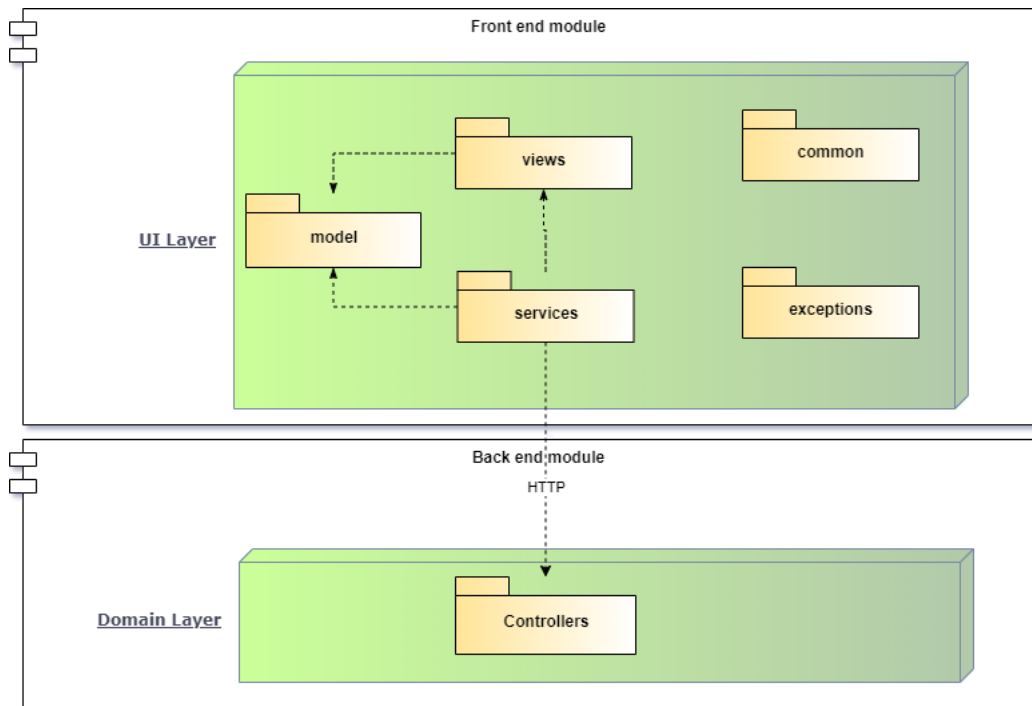


Figure 4.3: Front-end module global architecture.

4.2 Data model

The data model created to satisfy the needs of the platform CASCIFFO was made while having an important limitation of the *R2DBC* driver, originated from Spring Data JDBC, which only allows entities to have an numeric auto generated primary key, not permitting any composite keys, mentioned in the github issue "Add support for composite Id's" ¹ which depends on the issue "Support for Composite Key via @Embedded plus @Id [DATAJDBC-352]" ².

The model is centered around two main entities, the first being the `Proposal` entity which contains information regarding the submission and progress of a proposal of a clinical investigation. Secondly the `Clinical Research` entity which contains information pertaining to a clinical trial or clinical observation that has had its proposal submitted, accepted and validated.

A `Proposal` or `Clinical Research` entity can refer to either a clinical trial or observation study, depending on the field `research_type`. Although a clinical trial and an observational study are different, the information held by both is identical, differentiating only on existence of a financial component inherent to clinical trials. Therefore, these investigations can be stored regardless of their type in the mentioned entities. Aside from the identical data, another reason for choosing only one entity for representing clinical trials and observational studies in each stage, proposal submission and the active stage, is the amount of data stored, which is expected to be hundreds, so it makes it easier to do queries aimed at analyzing statistics.

This section is structured as such:

- Users, Roles and States - Explains the user, roles and states entities as well as their own relationship;
- Proposal Entity - A detailed approach to the proposal entity and the entities surrounding it;
- Research Entity - A detailed approach to the research entity and the entities surrounding it.

¹<https://github.com/spring-projects/spring-data-r2dbc/issues/158>

²<https://github.com/spring-projects/spring-data-relational/issues/574#issue-776914525>

4.2.1 Users, Roles and States

The proposal, clinical research and addenda entity require state to track their evolution over time, thus the `State` entity was created, this entity holds all possible states for any proposal, clinical investigation or addenda. To differentiate between them an entity was created with the goal of interconnecting state chains. A state chain describes the flow from an initial state to a terminal state, i.e in the context of a proposal advancing from the state "Submetido" to "Validado". This entity, called, `Next Possible States`, represents the possible transitions from one state to another, it contains two important fields that describe the chain the current state transition - `state_type` - belongs to (i.e proposal, research or addenda), as well as the general order of the transition in the chain, initial for the first possible transition, progress for any in-between and terminal indicating the end of the chain. To record the transitions, the entity `State Transition` was created, recording the previous state, new state, the date of transition, the type of transition (proposal, research or addenda) and finally the associated reference to what transitioned state. This reference is not a foreign key by choice, since it facilitates recording transitions in one entity rather than three different tables specifying the foreign relationship.

As specified in during the functional requirements, to advance states, a user requires a certain role associated to the state. This is represented through the entity `State Roles`, which associates the entity states with the roles entity. In addition, the data model was created so that a user can have more than one role, hence the relationship entity `User Roles`. These entities can be viewed in figure 4.4.

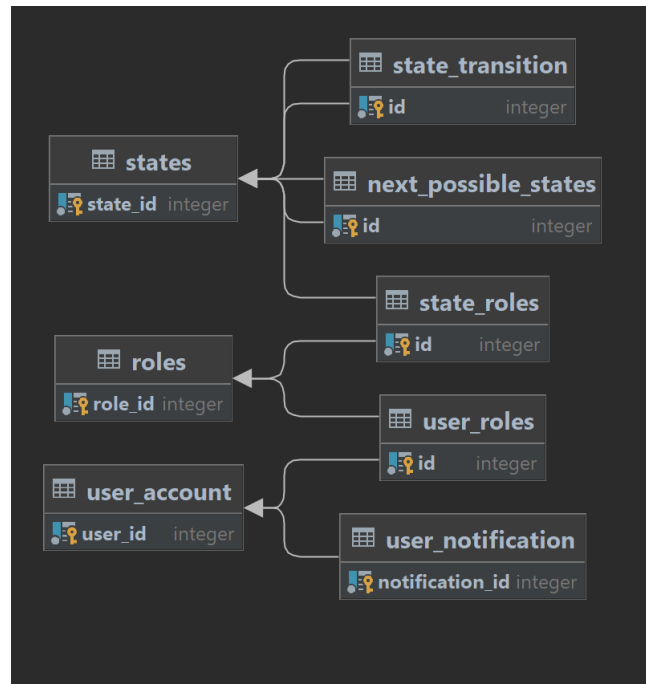


Figure 4.4: Entity relationship diagram of users, roles and states.

4.2.2 Proposal Entity

Associated to a proposal entity is the type of service it's included in, the therapeutic area and type of pathology, these three fields correspond to a foreign key to their own entities. Each of these entities consist of two fields, an identifier and a name. They were made into entities rather than simple fields to allow for consistency in the data model. This approach makes use of built-in foreign-key validations, which facilitates maintenance and CRUD operations on the entities. In addition to these fields, the proposal entity also includes a principal investigator responsible for submitting the proposal, an investigation team with all the investigators that can edit the proposal, a set of proposal comments made within the scope of the proposal, a financial component in case its a proposal for a clinical trial, a set of timeline events, the files associated to the proposal and most importantly the state, which corresponds to its current state.

In addition to the need of tracking state, a proposal can have multiple file resources. These files are stored on the operating system of the back-end module to avoid bloating the database with files. The information stored within the database through the entity `Files` is the full-path, file size and name.

The diagram involving the entities centered around the proposal entity can be viewed in figure 4.5

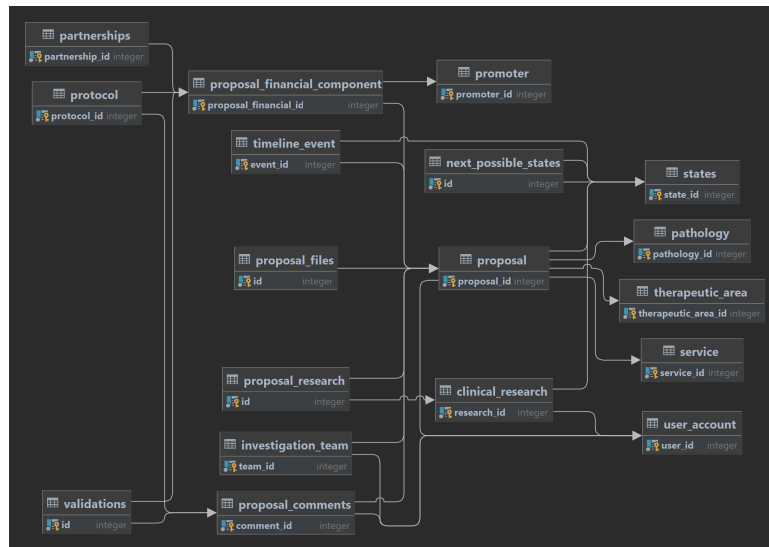


Figure 4.5: Entity relationship diagram centered on the Proposal Entity.

4.2.3 Clinical Research

A clinical research entity represents a clinical trial or observational trial and is associated to its proposal through the `Proposal Research` entity which stores the association of each proposal to its clinical trial. A clinical research, in similarity to a proposal, also has a state to track its current status. In addition to this, it also has a list of visits that occur in the context of the investigation, a list of several addenda, a team of investigators, equal to the team submitted during its proposal stage and in case its a clinical trial it also has a financial component associated to it. It is to note that the financial component associated to a clinical research is different than a proposal's financial component.

The research financial component is divided into two different sub categories, the team finance, which discriminates the revenue made by each team member and the clinical research finance, which represents the financial transactions made within the scope of the clinical research itself.

The addenda entity, associated to a clinical research, can have a file associated to it pertaining to the change being made, a state since it must undergo a validation process before it has an affect on the clinical research and a list of comments.

The diagram involving these entities centered around the clinical research entity can be viewed in figure 4.6.

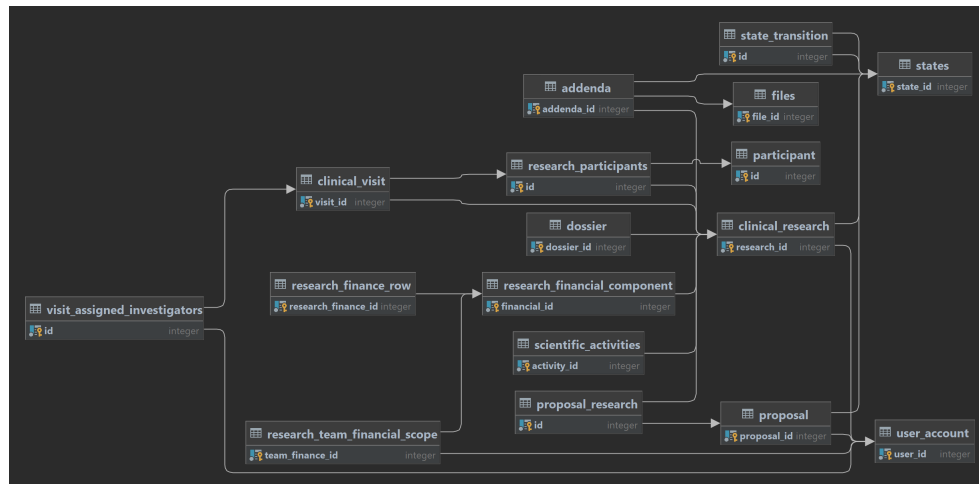


Figure 4.6: Entity relationship diagram centered on the Clinical Research Entity.

5

Implementation

This chapter presents the solution developed to create CASCIFFO given the functional requirements detailed previously. It describes the techniques used and choices made throughout the implementation of the solution.

It is structured with the following sections:

- Back-end 5.1: Describes the back-end module.
- Front-end 5.2: Describes the front-end module.
- Heroku deployment 5.3.2: Usage of cloud-based platform Heroku for continuous development.
- On-Premises Deployment 5.3.3: Deploying the application on-premises hosted by Apache.

5.1 Back-end module

This section describes the implementation choices and details of the Back-end module. It first introduces the development of the back-end module and follows into more details regarding the repositories, services, controllers, mappers and authentication respectively.

This section is structured as follows:

- Framework Stack 5.1.1 - Describes the technology used to created the framework stack of the BE module;
- PostgreSQL database 5.1.2 - Explains the configuration a creation of the PostgreSQL database used;
- Initial Development 5.1.3 - Describes in detail the steps initially taken for the implementation of the back-end module;
- Spring Configuration 5.1.4 - Details the Spring configuration to allow the module to run as expected;
- Repositories 5.1.5 - Explains and details how the repositories function and their creation;
- Services 5.1.6 - Details the functionality of the services in the module and shows three examples of services used in the implementation;
- Controllers 5.1.7 - Describes how controllers work and their role in the module;
- Mappers 5.1.8 - Details how the role of mappers in the module, how they are used and how we can take advantage of Spring features to make a sustainable and maintainable codebase;
- Authentication 5.1.9 - Describes how the authentication works and it's implemented;
- Exception Handling 5.1.10 - Describes the steps taken to raise and handle exceptions.

5.1.1 Framework Stack

The back-end module was developed using *Spring Webflux* [20] over Spring Web due to the reactive nature. *Spring Webflux* [20] brings a fully non-blocking programming environment and provides asynchronous calls to the database through the *R2DBC* driver. This allows for a completely non-blocking experience from the moment an HTTP request is received.

The module consists of three layers, the Repository layer, which is responsible for database access, the Service layer, which is responsible for business logic, and finally the Controller layer, which receives and responds to requests. The layers communicate only with the layers below them, preventing circular dependencies. The controller

layer communicates with the service layer, the service layer communicates with the repository layer and the latter communicates directly with the database. Across these layers there are exception handlers to handle any foreseen and unforeseen exceptions.

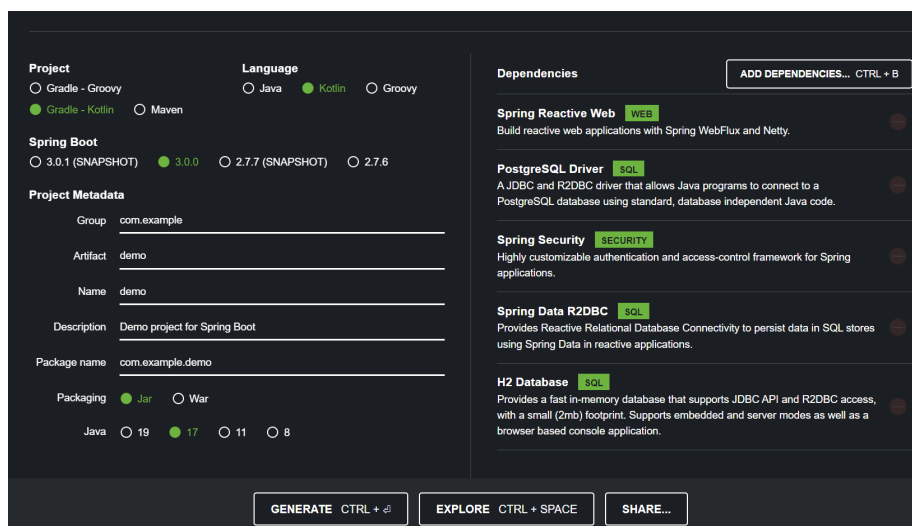
5.1.2 PostgreSQL database

Following the detailed data model described in 4.2, an SQL script, written in PostgreSQL language was created to entail the creation of the database, the user ownership and initial data of the database. The script is over 2 thousand lines long, thus not available as an appendix, however, it is available on the Github Repository Casciffo¹ along with the rest of the essentials to run the platform locally.

The only configuration required is the existence of a user `vp` who's credentials are be used in the Spring configuration explained further on.

5.1.3 Initial Development

By using the tool *Spring Initializr*² provided by *Spring* we're able to start with a project containing the necessary dependencies ready to launch. The project type was chosen to be Gradle-Kotlin, developed in Kotlin language, as illustrated in figure 5.1.

The image shows the Spring Initializr web interface. On the left, under 'Project', 'Gradle - Kotlin' is selected. Under 'Language', 'Kotlin' is selected. Under 'Spring Boot', '3.0.0' is selected. The 'Project Metadata' section includes fields for Group (com.example), Artifact (demo), Name (demo), Description (Demo project for Spring Boot), and Package name (com.example.demo). The 'Packaging' is set to 'Jar' and the 'Java' version is '17'. On the right, the 'Dependencies' section lists 'Spring Reactive Web' (WEB), 'PostgreSQL Driver' (SQL), 'Spring Security' (SECURITY), 'Spring Data R2DBC' (SQL), and 'H2 Database' (SQL). At the bottom, there are buttons for 'GENERATE', 'EXPLORE', and 'SHARE...'.

Project	Language	Spring Boot	Project Metadata	Dependencies
<input checked="" type="radio"/> Gradle - Kotlin	<input checked="" type="radio"/> Kotlin	<input checked="" type="radio"/> 3.0.0	Group: com.example Artifact: demo Name: demo Description: Demo project for Spring Boot Package name: com.example.demo	Spring Reactive Web (WEB) PostgreSQL Driver (SQL) Spring Security (SECURITY) Spring Data R2DBC (SQL) H2 Database (SQL)

Figure 5.1: Initial project structure using *Spring Initializr*.

After downloading the project template, the project was organized by folders corresponding to each entity, this was chosen for the convenience of navigation due to the

¹<https://github.com/ValdemarPCAntunes/casciffo>

²<https://start.spring.io/>

amount of entities.

In each entity folder, a class representing the controller, service and repository layer was created alongside the entity model itself.

The next step in development was the configuration of the spring application.

5.1.4 Spring Configuration

Given the starting project we first need to set the database credentials so that Spring knows how handle database access requests. This was achieved by utilizing the file `application.properties` inside the resources folder, in the case it doesn't exist, create it. The properties: `spring.r2dbc.url` - refers to the full URL of the database location, in the following format `r2dbc:postgresql://[HOST]:[PORT]/[DATABASE_NAME]`, in the exceptional case that the PostgreSQL service runs on a different port from the default 5432, the attribute `PORT` in the mentioned format should be changed accordingly; `spring.r2dbc.username` and `spring.r2dbc.password` correspond to the database credentials used for each connection made during a database access operation. Other fields can be set to make use of the potential Spring offers, such as:

- `spring.jackson.locale` — Serves to indicate what type of *JSON* serialization to use, the locale that should be considered for formatting dates, the used value was `pt_PT`;
- `spring.jackson.default-property-inclusion` — Setting this property to `non_null` indicates that Spring's serializer that only properties with non-null values are to be included in REST requests and responses.
- `server.port` — Sets the port on which the Spring application will be started on, defaults to 8080.
- `server.error.include-message` — Indicates whether an error message should be attached to an error response, defaults to `never` to possibly prevent sensitive information leaking, in the context of *CASCIFFO*, this wasn't a worry so it was set to `always` as a way to not only debug but also better handle exceptions in the FE.
- `spring.web.resources.static-locations` — Indicates the location of static resources to be served by Spring. This property is crucial to the development and deployment of the application as the static resources include the FE module. It was set with `file : /webapp`, the prefix `file :` serves to point out to Spring that the following argument

corresponds to a file path. In this specific case, the prefix dot before `/webapp` means relative path from the directory that the app was started in.

- `spring.r2dbc.properties.sslMode` — This property defines whether the connection to the database requires SSL mode to successfully establish a connection. During continuous development with the platform *Heroku* this setting was set to `REQUIRED`, since the database was hosted via HTTPS. During deployment and local development this property was commented out.

Another type of properties which are substantially helpful during development are the logging options, prefixed with `logging.level`. By default they have the value of `INFO`, and by changing them to `DEBUG` the information displayed during run-time reflects the execution of each operation in a very transparent manner which was most useful for testing purposes. During development the value of the properties `logging.level.web` - which refers to incoming and outgoing requests; `logging.level.org.springframework.security` - which refers to security web filters and finally `logging.level.org.springframework.r2dbc` - which refers to the database queries being executed.

To compile and execute the spring application without an integrated development environment (IDE) like IntelliJ³ we can use the files present inside the project directory, namely the file `gradlew` and `build.gradle.kts`. The former executes *gradle tasks* while the latter is where we configure said tasks and explicitly state the dependencies of the project. To compile the project we can run a terminal like using `gradlew build -x test` which will compile and bundle the project sources into a *JAR* file which we can later execute using *Java* commands. The specification `-x test` is useful for a production build since it tells Gradle to skip the tests task.

The dependencies required for the implementation of this module are present in the listing 5.1. This configuration snippet includes the dependencies for *Spring Security*, *Spring Webflux* and *Kotlin coroutines*, all of which will be mentioned further on this document.

```
1 plugins {
2     id("org.springframework.boot") version "2.6.3" //Spring boot
3     id("io.spring.dependency-management") version "1.0.11.RELEASE"
4     kotlin("jvm") version "1.6.10" //Kotlin jvm
5     kotlin("plugin.spring") version "1.6.10" //Support for spring boot in a
6         kotlin project
7 }
8
```

³<https://www.jetbrains.com/help/idea/discover-intellij-idea.html>

```

9 dependencies {
10 implementation("org.springframework.boot:spring-boot-starter-security") //
    Spring Security
11 implementation("org.springframework.boot:spring-boot-starter-data-r2dbc
    :3.0.0") //Spring Data R2DBC
12 implementation("org.springframework.boot:spring-boot-starter-webflux") //
    Spring Webflux
13 implementation("io.projectreactor.kotlin:reactor-kotlin-extensions:1.1.7")
    //Kotlin Flow
14 implementation("org.jetbrains.kotlinx:kotlinx-coroutines-reactor:1.6.4") //
    Kotlin coroutines
15 implementation("org.jetbrains.kotlin:kotlin-reflect") //Kotlin reflection
16 implementation("org.jetbrains.kotlin:kotlin-stdlib-jdk8")
17
18 //PostgreSQL
19 runtimeOnly("io.r2dbc:r2dbc-postgresql:0.8.13.RELEASE")
20 runtimeOnly("org.postgresql:postgresql:42.5.1")
21
22 //Json Web Tokens
23 implementation("io.jsonwebtoken:jjwt-api:0.11.5")
24 runtimeOnly("io.jsonwebtoken:jjwt-impl:0.11.5")
25 runtimeOnly("io.jsonwebtoken:jjwt-jackson:0.11.5")
26 }

```

Listing 5.1: Gradle configurations for production build.

From this snippet we can observe the keywords `runtimeOnly` and `implementation`, these keywords refer as to when the libraries are required, as the name implies, the former keyword means that the library will only be included during run-time whereas the `implementation` dependencies are required for the implementation, compilation of the project and subsequently run-time as well. This helps to keep the compile *classpath* smaller and faster.

The `plugins` property is used to specified the plugins required to run the project. In a brief manner, as per the official Gradle documentation on Using Gradle Plugins ⁴ a plugin is a set of tasks and configurations that can be added to a project to extend its functionality.

5.1.5 Repositories

The repository layer is responsible for directly accessing and manipulating data in the database.

⁴<https://docs.gradle.org/current/userguide/plugins.html>

The repositories are interfaces annotated with Spring's `@Repository`, indicating they behave as a repository, each repository should implement an appropriate public interface, in the reactive context, it can either be `ReactiveCrudRepository` or `ReactiveSortingRepository` followed by the type of entity and type of identifier (such as `ReactiveCrudRepository<Entity, EntityIdType>`), where the former allows for basic CRUD operations (Create, Read, Update, Delete) without needing to implement further functionality, while the latter allows for built-in sorting functionalities. Spring also allows custom functions to be implemented either through naming conventions, such as `findAllByIdAndFieldIs(id, field)`, and through custom queries. For more complex queries another option is available, by annotating a method with `@Query(sql: String)` and passing a string argument with the SQL statement defined at design time. All repository methods return a type encapsulated by either `Mono` or `Flux`. The type `Mono` should be returned when the method is expected to return between zero and a single result, while `Flux` should be used a result varying between zero and n elements are expected. These features can be seen in the listing 5.2 below.

```

1 @Repository
2 interface ProposalRepository : ReactiveSortingRepository<ProposalModel, Int> {
3
4     @Query(
5         "SELECT COUNT(*) " +
6         "FROM proposal p " +
7         "WHERE p.proposal_type = :type"
8     )
9     fun getCountByType(type: ResearchType): Mono<Int>
10
11     @Query(
12         "SELECT COUNT(*) filter ( where p.proposal_type = 'OBSERVATIONAL_STUDY' ) as studies, " +
13         "COUNT(*) filter ( where p.proposal_type = 'CLINICAL_TRIAL' ) as trials " +
14         "FROM proposal p"
15     )
16     fun countTypes(): Mono<CountHolder>
17
18     fun findAllBySiglaLike(sigla: String): Flux<ProposalModel>
19
20 }

```

Listing 5.2: Repository used for the Proposal entity.

A key factor to let the *Spring Webflux* [20] framework know a data class is an entity, for the main purposes of saving and updating, is annotating said class with `@Table("table_name")` where `"table_name"` refers directly to the database table, and the identifier field with `@Id` which also directly corresponds to the identifier present in the mentioned table. It is worth noting that the identifier of a database table must be numeric. As briefly mentioned in section 4.2, text identifiers and compound keys don't function properly with this framework, in order to circumvent this, each database table has its

own unique numeric identifier.

When creating an entity, the property annotated with `@Id` must a value of `null`, otherwise, the framework will attempt to update an nonexistent entity. To work around this, we can either implement a custom repository and override the `save()` method or design the entities used to have an automatically generated identifier. The approach taken while building CASCIFFO's data model was the latter.

It should also be noted that naming conventions are taken into account between the database and the entity, the *Snake Case* naming convention used by *PostgreSQL*, is automatically converted into *Camel Case* in *Spring*, for example a table field named `created_date` is automatically converted into `createdDate`. This automation is very useful when the mapped object contains the property `createdDate`, however, what if the property was named `createdOn`?

In such cases, two different approaches can be used to answer this question. Firstly we can customize the repository fetch method by including the annotation `@Query("sql_statement")` followed up with writing a select that renames `created_date` to `created_on`, this takes advantage of the auto naming conversion but requires effort to customize the query. The other approach that was heavily relied on what annotating the property with `@Column("entity_column")` which indicates what column Spring should map to the property.

By taking the second option, the property `createdOn` will be annotated with `@Column("created_date")`.

One noticeable downside of using *Spring Webflux* [20] and *R2DBC*, is that its current version doesn't provide mechanism of designing complex relationships with annotations, a feature available in *Spring Web*. For example given two entities, A and B, with a relationship of 1-to-1, when fetching entity B, there isn't an automatic mechanism to build the associated entity A, after the fetch, the data will come as-is in the database, as opposed to *Spring Data JPA* where such relationships can be modeled and built by the *Spring Framework*. While nested entities can exist in a data class representing a database table, since they can't be loaded through automatic means, they must be annotated with `@Transient` alongside `@Value(defaultValue)`. The former annotation tells *Spring's* framework to ignore the annotated property when saving a new entity while the latter annotation comes into play during the mapping after a fetch, this annotation defines the default value to assign to the annotated property, in the development context, the value assigned was always `@Value("null")`.

This means that for complex entities with multiple foreign keys, with each foreign key corresponding to a nested object that needs to be manually instantiated, in order

to fetch the data to build these objects would require to make that as database calls as there are nested entities which leads to performance issues. To counter this, entities were created to serve as containers of data, by aggregating all the fields required to build not only the base entity but also the nested ones. In conjunction with this, for each of these containers a repository was created, however, since the data that we want is split between tables, a custom query is needed to fetch the desired results. By using this approach we can specify all the necessary data manually. To illustrate this, the listing 5.3 shows a selection of fields from several different database entities, which are then mapped to the aggregate of type `ProposalAggregate`. The container of this aggregation doesn't need to be annotated with `@Table()` or have `@Id` on one of its fields, since it will be exclusively used for fetching data.

```

1 @Query(
2     "SELECT p.proposal_id, p.sigla, p.created_date, p.last_modified, p.proposal_type, " +
3         "pfc.proposal_financial_id, pfc.financial_contract_id, pfc.promoter_id, pfc.has_
4         partnerships, " +
5         "state.state_name, st.service_name, pl.pathology_name, ta.therapeutic_area_name, " +
6         "promoter_name, user_name " +
7     "FROM proposal p " +
8     "JOIN states state ON p.state_id = state.state_id " +
9     "JOIN pathology pl ON p.pathology_id = pl.pathology_id " +
10    "JOIN service st ON st.service_id = p.service_id " +
11    "JOIN therapeutic_area ta ON ta.therapeutic_area_id = p.therapeutic_area_id " +
12    "JOIN user_account pinv ON p.principal_investigator_id = pinv.user_id " +
13    "LEFT JOIN proposal_financial_component pfc ON p.proposal_id = pfc.proposal_id " +
14    "LEFT JOIN promoter pr ON pr.promoter_id = pfc.promoter_id " +
15    "ORDER BY p.last_modified DESC " +
16    "LIMIT :n"
17 )
18 fun findLastModifiedProposals(n: Int): Flux<ProposalAggregate>

```

Listing 5.3: A proposal aggregate repository function that fetches data from multiple sources.

The entities associated to each repository are classes which only contain data, as such, *Data classes* ⁵, available in Kotlin. The data classes were created using the described annotations and followed the mentioned rules. Below is the listing of the proposal data class (5.4) which includes everything mentioned in regards to entities.

```

1 @Table(value = "proposal")
2 data class ProposalModel(
3     @Id
4     @Column(value = "proposal_id")
5     var id: Int? = null,
6     @Column(value = "sigla")
7     var sigla: String? = null,
8     @Column(value = "proposal_type")
9     var type: ResearchType? = null,

```

⁵<https://kotlinlang.org/docs/data-classes.html>

```

10  @Column(value = "created_date")
11  var createdDate: LocalDate? = null,
12  @Column(value = "last_modified")
13  var lastModified: LocalDateTime? = null,
14  @Column(value = "state_id")
15  var stateId: Int? = null,
16  @Column(value = "service_id")
17  var serviceTypeId: Int? = null,
18  @Column(value = "therapeutic_area_id")
19  var therapeuticAreaId: Int? = null,
20  @Column(value = "pathology_id")
21  var pathologyId: Int? = null,
22  @Column(value = "principal_investigator_id")
23  var principalInvestigatorId: Int? = null,
24  @Transient
25  @Value("null")
26  var researchId: Int? = null,
27  @Transient
28  @Value("null")
29  var state : State? = null,
30  @Transient
31  @Value("null")
32  var serviceType: ServiceType? = null,
33  @Transient
34  @Value("null")
35  var therapeuticArea: TherapeuticArea? = null,
36  @Transient
37  @Value("null")
38  var pathology: Pathology? = null,
39  @Transient
40  @Value("null")
41  var principalInvestigator: UserModel? = null,
42  @Transient
43  @Value("null")
44  var financialComponent: ProposalFinancialComponent? = null,
45  @Transient
46  @Value("null")
47  var comments: Flow<ProposalComment>? = null,
48  @Transient
49  @Value("null")
50  var stateTransitions: Flow<StateTransition>? = null,
51  @Transient
52  @Value("null")
53  var timelineEvents: Flux<TimelineEventModel>? = null,
54  @Transient
55  @Value("null")
56  var investigationTeam: Flux<InvestigationTeamModel>? = null
57 )

```

Listing 5.4: Data class representing the entity Proposal.

5.1.6 Services

The service layer is where all business logic is handled. The approach taken to implement this layer was with interfaces together with Spring's dependency injection, all services have an interface that describes their operations as briefly shown in figure 5.2. The implementation of these services must then be annotated with `@Component` so Spring can easily scan and auto inject the implementations to where they're used. This facilitates maintenance and rearranging of code, as well as building and swapping existing implementations of a service.

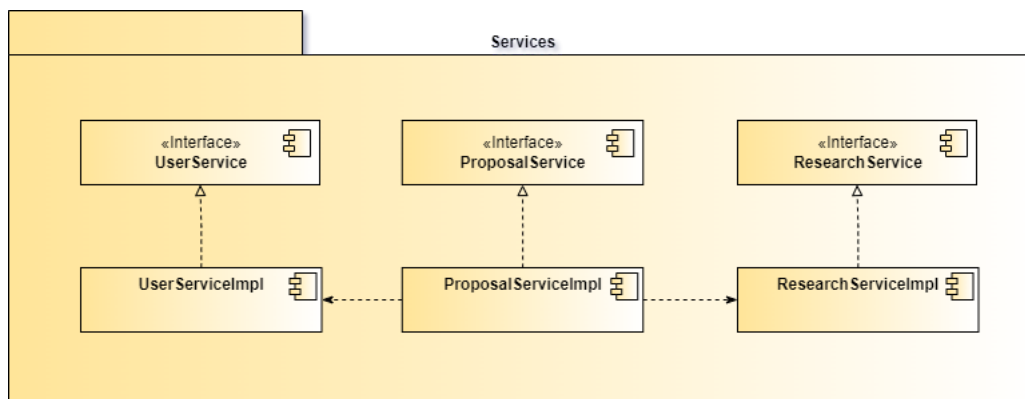


Figure 5.2: Example of three existing services in CASCIFFO.

Each service can communicate with their own associated repositories and other services, in addition to business logic, the request data validation and other exceptions are raised in this layer.

Most services have common similarities in terms of logic, replacing the repository used and data passed. Thus we will focus on three services: the proposal service, visit service and the user service. The first service is a pivotal point of the platform, interacting directly with one of the 'big' entities - the proposal entity, while the second one shows more common functionality to other services.

5.1.6.1 User service

The user service is to be mainly used by the user controller, the service itself utilizes other services as well as its dedicated repository - user repository.

The development of this service started with the design of the interface, from there, we ask the question of what operations are required from this type of service. Each client of the CASCIFFO platform requires a user account to be able to navigate through

its features and use its capabilities. From this simple observation, the need for the registration and login operations arises. Doing a complete analysis of the features will find us with the following declaration of the user service interface as shown in the listing 5.5

```

1 interface UserService {
2     //create operations
3     suspend fun registerUser(userModel: UserModel) : BearerTokenWrapper
4     //get operations
5     suspend fun loginUser(userModel: UserModel): BearerTokenWrapper
6     suspend fun getAllUsers() : Flow<UserModel?>
7     suspend fun getUser(id: Int) : UserModel?
8     suspend fun getAllUsersByRoleNames(roles: List<String>): Flow<UserModel>
9     suspend fun searchUsers(name: String, roles: List<String>): Flow<UserModel?>
10    suspend fun findUserByEmail(email: String): UserModel?
11    //update operations
12    suspend fun updateUser(user: UserModel): UserModel
13    suspend fun updateUserRoles(roles: List<Int>, userId: Int): UserModel
14    suspend fun notifyRoles(roles: List<Roles>, notificationModel: NotificationModel)
15    //delete operations
16    suspend fun deleteUser(userId: Int): UserModel
17    suspend fun deleteUserRoles(roleIds: List<Int>, userId: Int): UserModel
18 }

```

Listing 5.5: User service interface.

This first iteration of the user service interface shows methods with the responsibility of registration, login, searching, updating, notifying and deleting user related data. Another question that can be raised with this declaration is why do we have notification and user role manipulation here when their own services exist? This can be explained by seeing the implementation of the user service. The class implementing this interface that is used across the module is as named UserServiceImpl and it's declaration can be viewed in the listing 5.6

```

1 @Service
2 class UserServiceImpl(
3     @Autowired val userRepository: UserRepository,
4     @Autowired val userRolesRepo: UserRolesRepo,
5     @Autowired val userRolesAggregateRepo: UserRolesAggregateRepo,
6     @Autowired val roleService: RoleService,
7     @Autowired val notificationService: NotificationService,
8     @Autowired private val encoder: PasswordEncoder,
9     @Autowired val jwtSupport: JwtSupport
10 ) : UserService { /*...*/ }

```

Listing 5.6: User service implementation declaration.

Through this short code, there are many details that require some attention, namely the new annotation @Service. This annotation is a general-purpose stereotype ⁶ where

⁶<https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/stereotype/Service.html>

its semantics are left up to the developers and its context. It is also considered a specialization of `@Component`, allowing the allowing for implementation classes to be auto detected through *classpath* scanning.

Moving to the class arguments it's observable that the service communicates with other parts of the module. Besides the service and repository arguments, the arguments `encoder` and `jwtSupport` are used for authentication purposes and will be further discussed in the authentication section 5.1.9.

The implementation of this class consists on overriding each method declared in the mentioned interface. Here, we take a look at the implementation of the method `updateUserRoles` seen below in the listing 5.7.

```

1 override suspend fun updateUserRoles(roles: List<Int>, userId: Int): UserModel {
2     val user = getUser(userId) ?: throw ResponseStatusException(HttpStatus.BAD_REQUEST, "Utilizador
3     com id [$userId] não existe!")
4     if (roles.isEmpty()) return user
5     val currentRoles = user.roles?.collectList()?.awaitFirstOrNull()
6     val rolesToAdd = if (currentRoles.isNullOrEmpty()) roles
7                     else roles.filter { r -> !currentRoles.any { cr -> r == cr.roleId!! } }
8     if (rolesToAdd.isEmpty()) return user
9     userRolesRepo
10        .saveAll(rolesToAdd.map { UserRoles(userId = userId, roleId = it) })
11        .doOnError {
12            logger.info { it }
13        }.subscribe()
14    val updatedRoles = roleService.findByUserId(userId).map { it.roleName!! }.collectList().
15        awaitSingle()
16    notificationService.createNotification(userId,
17        NotificationModel(
18            title = "Novos papéis adicionados.",
19            description = "Papéis novos:\n\t " + updatedRoles.joinToString(", ") + ". ",
20            notificationType = NotificationType.USER_NEW_ROLES
21        )
22    )
23    return getUser(userId)!!
24 }

```

Listing 5.7: Login method implementation.

This method is interesting because we can see the usage of other services in play as well as data validation and depending on validity whether to raise an exception. The usage of `ResponseStatusException` will be discussed further in section 5.1.9.

5.1.6.2 Proposal service

The proposal service is one of the most complex services in terms of business logic due to validation restrictions of the module. Just like every other service, the proposal

service consists of two parts, the interface and implementation, the interface, shown in listing 5.8, displays operations being exposed to consumers of this service.

```

1 interface ProposalService {
2     //get operations
3     suspend fun getProposalCount(): CountHolder
4     suspend fun getAllProposals(type: ResearchType, pageRequest: PageRequest? = null): Flow<
        ProposalModel>
5     suspend fun getProposalById(id: Int, loadDetails: Boolean = true): ProposalModel
6     suspend fun getProposalStats(): Flow<ProposalStats>
7     suspend fun getLatestModifiedProposals(n: Int): Flow<ProposalModel>
8     //create operations
9     suspend fun create(proposal: ProposalModel) : ProposalModel
10    //update operations
11    suspend fun updateProposal(proposal: ProposalModel) : ProposalModel
12    suspend fun transitionState(proposalId: Int, nextStateId: Int, request: ServerHttpRequest):
        ProposalModel
13    //delete operations
14    suspend fun deleteProposal(proposalId: Int): ProposalModel
15 }

```

Listing 5.8: Proposal service interface.

The complexity of this service lies mainly on the method `transitionState` which is where the proposal transitions from its current state to a new one. To begin to analyze this method, we must first detail the constructor of the implementing class, displayed in listing 5.9, as to know what we can work with.

```

1 @Service
2 class ProposalServiceImpl(
3     @Autowired val proposalAggregateMapper: Mapper<ProposalModel, ProposalAggregate>,
4     @Autowired val proposalResearchRepository: ProposalResearchRepository
5     @Autowired val proposalAggregateRepo: ProposalAggregateRepo,
6     @Autowired val proposalRepository: ProposalRepository,
7     @Autowired val proposalStatsRepo: ProposalStatsRepo,
8     @Autowired val userService: UserService,
9     @Autowired val stateService: StateService,
10    @Autowired val researchService: ResearchService,
11    @Autowired val partnershipService: PartnershipService,
12    @Autowired val validationsService: ValidationsService,
13    @Autowired val commentsService: ProposalCommentsService,
14    @Autowired val notificationService: NotificationService,
15    @Autowired val timelineEventService: TimelineEventService,
16    @Autowired val stateTransitionService: StateTransitionService,
17    @Autowired val investigationTeamService: InvestigationTeamService,
18    @Autowired val proposalFinancialService: ProposalFinancialService,
19    @Autowired val jwtSupport: JwtSupport,
20 ) : ProposalService { /* ... */ }

```

Listing 5.9: Proposal service implementation declaration.

From the declaration, we can observe the amount of services and repositories this service communicates with. Focusing on the implementation of the method `transitionState`, shown in the listing 5.10.

```

1 @Transactional(rollbackFor = [ResponseStatusException::class])
2 override suspend fun transitionState(
3     proposalId: Int,
4     nextStateId: Int,
5     request: ServerHttpRequest
6 ): ProposalModel {
7     val token = request.headers.getFirst(HttpHeaders.AUTHORIZATION)!!
8     val bearer = BearerToken(token.substringAfter("Bearer "))
9     val userEmail = jwtSupport.getUserEmail(bearer)
10    val user = userService.findUserByEmail(userEmail)!!
11    val userRoles = user.roles!!
12    val prop = getProposalById(proposalId, false)
13    return handleStateTransition(prop, nextStateId, userRoles)
14 }

```

Listing 5.10: Signature and implementation of the transitionState method.

In the signature of this method we can observe a new annotation `@Transactional`. This annotation describes a transaction attribute on a method or class⁷. It's present on this method and all others who's execution alters data in the database, through create, update or delete operations. The addition of the attribute `rollbackFor = [ResponseStatusException::class]` specifies the type of exception, in this case `ResponseStatusException`, on which the entire transaction should trigger a rollback operation. By default the rollback is triggered via run time exceptions. A rollback operation is one of reverting the changes made within the context of a given transaction.

In the method `transitionState`, up until the last line of `return`, the user roles are fetched and passed to `handleStateTransition()` where the validation occurs and actual transition is made. To dissect this method, we start with the validations, these can be observed in 5.11, the first validation call is made to the `stateService` to verify whether `nextStateId` corresponds to a possible and valid state. To briefly explain the method `verifyNextStateValid`, it makes a database access to query the `NextPossibleStates` table and verifies whether the aforementioned state id matches the column `nextStateId` on any row. The following validation, in case the proposal's type is a clinical trial and its current state is `VALIDACAO_CF` which corresponds to the state validation of financial contract, in this context we must verify whether the financial contract is fully validated by both the financial and juridical department.

Lastly we validate whether the next state corresponds to the final state in a proposal, in case it is, we perform a check on whether the financial component (includes both the protocol and financial contract validation) is fully validated. Assuming this is correct the research is automatically created followed with a notification to the investigation team.

⁷<https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/transaction/annotation/>

In case any of these validations fail, an exception of type `ResponseStatusException` is triggered which will then trigger the rollback mentioned previously.

```

1 suspend fun handleStateTransition(
2     proposal: ProposalModel,
3     nextStateId: Int,
4     role: List<String>
5 ): ProposalModel {
6     //initial variables
7     val currState = stateService.findById(proposal.stateId!!)
8     val nextState = stateService.findById(nextStateId)
9     val isClinicalTrial = proposal.type === ResearchType.CLINICAL_TRIAL
10    val stateType = if (isClinicalTrial) StateType.FINANCE_PROPOSAL
11    else StateType.STUDY_PROPOSAL
12    //validations
13    nextState.roles = stateService.verifyNextStateValid(currState.id!!, nextStateId, stateType,
14    role).toFlux()
15
16    if (isClinicalTrial && currState.name == States.VALIDACAO_CF.name) {
17        val check = validationsRepository.isPfcValidatedByProposalId(proposal.id!!).awaitSingle
18        ()
19        if (!check) {
20            throw ResponseStatusException(HttpStatus.BAD_REQUEST, "O contrato financeiro ainda
21            não foi completamente validado!")
22        }
23    }
24    if (stateService.isTerminalState(nextStateId, stateType)) {
25        if (isClinicalTrial) {
26            val fullyValidated = validationsRepository.isPfcFullyValidated(proposal.
27            financialComponent!!.id!!).awaitSingle()
28            if (!fullyValidated)
29                throw ResponseStatusException(
30                    HttpStatus.BAD_REQUEST,
31                    "O contrato financeiro ainda não foi completamente validado!"
32                )
33        }
34        createResearch(proposal, isClinicalTrial)
35        notifyTeam(proposal.id!!,
36            NotificationModel(
37                title = if (isClinicalTrial) "Ensaio Clínico criado!" else "Estudo observacional
38                criado!",
39                description = "A proposta com a sigla ${proposal.sigla} foi validada!\n" +
40                "Como tal, o ${if (isClinicalTrial) "ensaio clínico" else "estudo
41                observacional"} foi criado" +
42                " com sucesso, visita a página para preencheres os detalhes!",
43                notificationType = NotificationType.RESEARCH_DETAILS,
44                ids = convertToJson(listOf(Pair("researchId", proposal.researchId!!)))
45            )
46        )
47    }
48    // ...
49 }

```

Listing 5.11: Implementation of the `handleStateTransition` method 1 of 2.

Upon successfully passing the validations, the time line events are checked in case

there's any associated to a state that can now be validated followed by the creation of the transition and a notification of progress to the team members, as shown in listing 5.12. To conclude, the proposal is updated with the new state, saved with the repository call `proposalRepository.save(proposal)` and returned.

```

1 suspend fun handleStateTransition (
2     proposal: ProposalModel,
3     nextStateId: Int,
4     role: List<String>
5 ): ProposalModel {
6     // ...
7     if (proposal.timelineEvents != null) {
8         updateTimelineEvent(proposal.timelineEvents!!, nextState.name!!)
9     }
10    stateTransitionService.newTransition(proposal.stateId!!, nextState.id!!, stateType, proposal.id!!)
11    if (nextState.stateFlowType != StateFlowType.TERMINAL) {
12        notifyTeam(proposal.id!!,
13            NotificationModel(
14                title = "Progresso no estado de Proposta",
15                description = "Proposta com sigla ${proposal.sigla!!} avançou para o estado ${
16                    nextState.name!!}",
17                notificationType = NotificationType.RESEARCH_DETAILS,
18                ids = convertToJson(listOf(Pair("proposalId", proposal.id!!))),
19                viewed = false
20            )
21        )
22        proposal.stateId = nextState.id
23        return proposalRepository.save(proposal).awaitSingle()
24    }

```

Listing 5.12: Implementation of the `handleStateTransition` method 2 of 2.

5.1.7 Controllers

The controller layer is where the HTTPS requests are handled, the routing of requests is handled by Spring's framework with the help of annotations that distinguish what controller handles which route and operation. To build a controller class, the class needs to be annotated with `@Controller` or in this case of a *REST API* module, `@RestController`. The main difference between the two annotations is that `@RestController` indicates that the class is a controller where `@RequestMapping` methods assume `@ResponseBody` semantics by default. The `@RequestMapping` accepts a route path to serve as the base of the controller class, `@ResponseBody` indicates the type of response, in this case, JSON is used in all responses.

When building the operations of a controller class, the methods that handle route requests, depending on the type of operation, need to be annotated with one of the following `@Post`, `@Put`, `@Get`, `@Patch` or `@Delete`, each of these annotations take in an optional argument that represents the path of the route taking into account the base path if specified in `@RequestMapping(path)` on the class.

Each of the methods in a controller class can return an object or a `ResponseEntity<Type>` for more flexibility in the structure of the response. In the reactive environment that *Spring Webflux* [20] provides, each return type should be inside a `Mono` or `Flux` stream. The former represents a stream of one object, whereas the latter a stream of objects, from 0 to N objects. `Mono` and `Flux` represent streams of data using *Reactor*, these data types can be chained together like building blocks in a declarative manner of programming. First the developer explicitly declares the behavior of the chain, with functions such as `map()`, `filter()`, `groupBy()`, `flatMap()`. The execution of this chain only begins once the stream is subscribed to.

Returning a `Mono` or `Flux` lets the Spring framework handle when its subscribed and structures the response data, whenever a conversion to JSON happens the streams are automatically subscribed to. While `Mono` and `Flux` are *Spring Webflux* [20]'s standard since it operates over *Reactor*, *Kotlin* also offers another way of dealing with streams, while implementing the back-end module *Kotlin Coroutines* [7] were preferred over `Mono` and `Flux` due to its `async-await` nature and ease of use. To use *Kotlin Coroutines* the functions need to have the `suspend` keyword, i.e `suspend fun functionName(params)`, in this way, instead of returning a `Mono<Type>` we can return the `Type` directly. When dealing with `Flux` data types, *Kotlin* offers an alternative called `Flow<Type>`, the type `Flow` has a similar set of operations and behavior to `Flux`. To convert a stream of type `Flux` to a `Flow`, a simple call to the method `asFlow()` applied to the `Flux` stream will do the job.

A caveat of `Flux` and `Flow` occurs when returning objects with nested streams, (i.e returning a `Flow` of object A that has a nested `Flow` of object B in its instance), during *Spring*'s automatic conversion from object to JSON, the nested streams aren't represented properly, showing only a fetch status instead of the stream itself. In order to address this issue, a Data Transfer Object (DTO) representation of each entity model was created to hold lists instead of reactive streams. In conjunction with this, several mappers were created for the conversion of entity models to their respective DTO representations.

5.1.7.1 Mono/Flux syntax vs Suspend function syntax

To understand the concept of a suspend function we must first understand what are *Kotlin coroutines*, since a suspend function is a coroutine itself.

A *Kotlin coroutine*, taken to its literal name, co and routine, can be seen as a cooperative routine, meaning a conjunction of instructions (routine) that runs along (co) with other tasks. In other words, a coroutine is an instance of suspendable computation that can execute blocks of code in concurrency with other coroutines. In this context of concurrent programming, we can find another and very common way of dealing with concurrent tasks; threads.

Kotlin coroutines and threads are both ways of concurrent programming, but they have some significant differences. A coroutine is conceptually similar to a thread, in the sense that it takes a block of code to run concurrently with the rest of the code. However, a coroutine is not bound to any particular thread. It may suspend its execution in one thread and resume in another one. [8]

According to the article by Kumar Chandrakant one of the main differences between coroutines and threads is that coroutines are light-weight, while threads are heavy-weight. This means that coroutines use much less memory and computational resources than threads, and can therefore be more efficient when dealing with large numbers of concurrent tasks.

Another difference pointed out in the same article, is that coroutines are cooperatively scheduled, while threads are preemptively scheduled. This means that in a coroutine, the currently running coroutine must explicitly yield control to another coroutine, while in a thread, the scheduler can interrupt the currently running thread at any time and give control to another thread. We can say that unlike threads which are usually controlled by the operating system, coroutines are managed by the user in conjunction with the programming language. This makes coroutines easier to reason about and debug, since the control flow is more predictable.

Additionally, coroutines can be easily composed and used in a non-blocking manner, while threads require explicit synchronization mechanisms such as locks or semaphores to avoid race conditions and deadlocks.

Now, with the concept of *Kotlin coroutine* defined, we can move on to creating *suspending functions*. To do this, we simply need to add the keyword `suspend` before the declaration of the function, thus making it a *suspending function*.

On the topic of syntax choice, *Kotlin Coroutines* [7] was favored due to the simplicity and clarity of the written code. To illustrate the difference in syntax between the usage of *Spring's* `Mono/Flux` and *Kotlin coroutines* we'll use a small portion of CAS-CIFFO's data model focusing on a proposal, its principal investigator and the states of a proposal. The entities in play will be `Proposal`, `User` - representing investigators - and `Comment`. Following the context given, a proposal has one principal investigator and can have several comments made by other investigators. The data of a proposal is split in the database, when wanting to get the full details of a proposal, a query must also be made to fetch the `User` and another to fetch the `Comment` associated to the proposal. Assuming a repository exists for each of these entities their names are their respective entities suffixed with `Repo` we can write the following code to query the database and fetch the entities needed to build a proposal with nested entity of investigator and a list of comments. Below is the representation of the method `getProposalDetails` using `Mono/Flux` in listing 5.13 and using a *Kotlin Coroutine* in listing 5.14.

```

1 fun getProposalDetails(pId: Int): Mono<Proposal> {
2     return personRepo
3         //findById returns Mono<Proposal>
4         .findById(pId)
5         // zipWith returns Mono<Tuple2<Proposal, User>>
6         .zipWith(userRepo.findById(pId))
7         .flatMap {
8             it.t1.principalInvestigator = it.t2.get()
9             it.t1 //Proposal
10        }
11        // zipWith returns Mono<Tuple2<Proposal, (Mutable) List<Comment>>
12        .zipWith(commentsRepo.findAllByProposalId(pId).collectList())
13        .flatMap {
14            it.t1.comments = it.t2.get().map { e -> commentsMapper.mapToDto(e) }
15            it.t1
16        }
17    }

```

Listing 5.13: Loading nested entities in an object using `Mono/Flux` syntax

```

1 suspend fun getProposalDetails(pId: Int): Person {
2     val proposal = proposalRepo.findById(pId).awaitSingle()
3     person.principalInvestigator = userRepo.findById(pId).awaitSingle()
4     //here comments was changed from type Flux<Comment> to Flow<Comment>
5     person.comments = commentsRepo.findAllByPersonId(pId).asFlow()
6     return proposal
7 }

```

Listing 5.14: Loading nested entities in an object using suspend function syntax

5.1.8 Mappers

Mappers are crucial tool in any application, allowing the conversion of one data type into another and hiding the complexity of such operation.

A single generic interface was created with two simple methods which can be observed in listing 5.15, from which all other mappers implement.

```

1 interface Mapper<Model, DTO> {
2     suspend fun mapDTOtoModel(dto: DTO?): Model
3     suspend fun mapModelToDTO(model: Model?): DTO
4 }

```

Listing 5.15: Mapper interface.

An implementation of this interface can be seen in the mapper below in listing 5.16. By comparing both the interface and implementation of the mapper we can observe that only the implementation has the annotation `@Component`. To give a brief explanation, since it will be explained further during subsection 5.1.9, it tells Spring which classes can be scanned and used for dependency injection.

```

1 @Component
2 class StateMapper: Mapper<State, StateDTO> {
3     override suspend fun mapDTOtoModel(dto: StateDTO?): State {
4         throw NotImplementedError("Left blank on purpose.")
5     }
6
7     override suspend fun mapModelToDTO(model: State?): StateDTO {
8         if(model == null) return StateDTO()
9         return StateDTO(
10             id = model.id,
11             name = model.name,
12             nextInChain = model.nextStates?.collectList()?.awaitSingleOrNull(),
13             roles = model.roles!!.collectList().awaitSingleOrNull(),
14             stateFlowType = model.stateFlowType!!
15         )
16     }
17 }

```

Listing 5.16: Mapper for state entity.

Using Spring's powerful dependency injection, the services and controllers which use mappers don't need to know their specific name, we only need to set the correct types for Model and DTO. For example if we want to use the state mapper implementation 5.16, we pass it as a constructor argument to a class as such `@Autowired val stateMapper: Mapper<State, StateDTO>`. The `@Autowired` annotation indicates that a property should be injected via Spring's dependency injection mechanic.

During the mapping phase, nested objects of type Flux or Flow are subscribed to and subsequently transformed into lists that can be represented in JSON.

Each mapping conversion from DTO to Model and vice versa are called within the controllers.

5.1.9 Authentication

On top of these layers lies Spring's own framework, and in it the Web Filters. Web Filters are methods that run during an incoming request. Through Spring's framework, they intercept and perform operations with the request data before control is given to a controller handler. These Web Filters are especially useful to perform authentication operations on the incoming requests.

The authentication was implemented with Json Web Token (JWT), each request that requires authentication must have a valid JWT in the header 'Authentication' of the request. This JWT in addition to the standard fields, also holds user information, namely the user email that is used to uniquely identify each user. To further security, a role-based access control was defined as mentioned previously in the document. This can be implemented in a few steps, first and foremost is to implement *Spring Webflux* [20]'s `ReactiveUserDetailsService`, which is responsible for fetching a user based on a passed argument `username`, this method will be called during the interception of a request that requires authentication. The next step is to create an authentication manager that implements `ReactiveAuthenticationManager` and overriding the method `authenticate`, this manager is responsible for extracting and validating the information from a JWT. Note that both the classes that implement these interfaces need to be annotated with `@Component`. The need for the annotations comes from a useful feature of *Spring's* framework, dependency injection, by annotating these classes with `@Component` or methods with `@Bean` allows *Spring* to scan and detect the classes and auto inject them into other parts of the code where they are passed as arguments for methods or constructors. This classes then become *candidates* for injection depending on the context.

Spring taking control of this aspect of the code, can be referred as Inversion of Control (IoC) where a framework takes control of a portion of the program or code, instead of the programmer explicitly controlling it.

To make use of this newly created components we need to add the authentication manager to the `WebFilter` chain that can be accessed and configurable from a class annotated with `@Configuration` and implementing the method `springSecurityFilterChain`, annotated with `@Bean`. This method receives an argument `http` of type `ServerHttpSecurity` and should also receive the created authentication manager component as an argument. Finally to add it to the `WebFilter` chain, the method `addFilterAt()` is used, applied to the argument `http`.

This describes how a JWT's information is extracted and validated, to implement a role-based access control, one can choose Spring's annotation-based authentication, by annotating the required methods or controller classes with `@PreAuthorize(string)`. The passed argument is a function name alongside its arguments, in this context, the most adequate ones would be `hasAuthority()` or `hasAnyAuthority()`. Another possible way of implementing this requirement is by utilizing the `http` argument mentioned previously inside the method `springSecurityFilterChain` and programmatically add these rules to each route, as shown in the coding listing 5.17. By utilizing the methods `authorizeExchange()` followed by `pathMatchers(route)` and applying `hasAnyAuthority(authorities)` or `hasAuthority(authority)`. An authority is simply a constant with a string value specifying a role, for context, some of the authorities used were `SUPERUSER_AUTHORITY` and `UIC_AUTHORITY`.

```

1 @Bean
2 fun springSecurityFilterChain (
3     converter: JwtServerAuthenticationConverter ,
4     http: ServerHttpSecurity ,
5     authManager: JwtAuthenticationManager
6 ): SecurityWebFilterChain {
7
8     val jwtFilter = AuthenticationWebFilter(authManager)
9     jwtFilter.setServerAuthenticationConverter(converter)
10
11     http
12         .addFilterAt(jwtFilter , SecurityWebFiltersOrder.AUTHENTICATION)
13
14     //example of routing configuration
15     http
16         .authorizeExchange ()
17         .pathMatchers (HttpMethod.GET, ENDPOINTS_URL)
18             .permitAll ()
19         .pathMatchers (HttpMethod.POST, ENDPOINTS_URL)
20             .hasAnyAuthority (SUPERUSER_AUTHORITY, UIC_AUTHORITY)
21         .pathMatchers (HttpMethod.DELETE, ENDPOINTS_URL)
22             .hasAuthority (SUPERUSER_AUTHORITY)
23
24     //other http configurations
25     ...
26
27     return http.build ()
28 }

```

Listing 5.17: Configuration of spring security filter chain

Both the discussed approaches were used, having settled with programmatically adding new rules to the routes themselves. This choice was made based on performance issues with the annotation-based authentication, the annotation `@PreAuthorize()` was adding an overhead of up to 1 second per request, whereas programmatically this issue was not observed. The cause can be due to *Spring Webflux* [20], since the annotation `@PreAuthorize` is more commonly used with Spring Web.

5.1.10 Exception handling

On the topic of business rules, in case one is broken, an exception must be raised with information regarding the cause. Business rules are defined in the service layer and conversely it is where incoming requests are validated before performing the requested operation. If the information passed along the request isn't valid, an exception is manually thrown with an appropriate `HTTP status` and a reason, since the back-end module exposes a REST API.

With exceptions being raised whenever there's invalid data on incoming requests, the response must come with appropriate information on why the request failed or was rejected. For example, when requesting a resource that requires authentication yet the request sent none, the response will provide sufficient information so the one that made the request knows that for accessing that specific resource, it is also required to send authentication and how to send it.

To handle these exceptions and others that can occur during run-time, such as a database integrity violation or in case the database is unreachable among other possible run-time exceptions we can make use of great features provided by *Spring's* framework. We first looked at utilizing custom exceptions annotated with `@ResponseStatus`, by annotating a class that extends `Exception`, we can define the response status code and the reason that caused the exception to be thrown, as illustrated by the listing 5.18.

```
1 @ResponseStatus(  
2     code = HttpStatus.NOT_FOUND,  
3     reason = "Specified resource Id doesn't exist."  
4 )  
5 class ResourceNotFoundException() : Exception()
```

Listing 5.18: Example of an exception annotated with `@ResponseStatus`

This approach is simple and makes use of *Spring's* automatic converter to *JSON*. The issue lies in its simplicity, it doesn't allow us to change anything other than reason and status code, the body itself is handled by *Spring*, an example of an answer can be observed in the code listing 5.19. We would also need to create several exception classes for each edge case that requires an exception to be thrown.

```
1 "{  
2     "timestamp": "2022-11-09T12:24:21.609+00:00",  
3     "path": "/proposals/999",  
4     "status": 404,  
5     "error": "Not Found",  
6     "message": "Specified resource Id doesn't exist.",  
7     "requestId": "c0d35eab-2"
```



```
8 } "
```

Listing 5.19: Example response using *Spring's* automatic response converter.

One other possible solution was to create specific exception handlers, using the annotation `@ExceptionHandler(SampleException::class)` on a method and passing the class type of exception to catch as its argument, we can process specific exceptions locally within the controller classes themselves, as shown in listing 5.20. This resolves issue of the first approach in the sense that we can make use of the incoming request and outgoing response to write our own defined response body. The trade offs, however, come at a cost of maintenance, the exception handlers will be tightly coupled to their controllers and one exception handler can only process one type of exception for that particular controller.

```
1 class FooController{
2     // ...
3     @ExceptionHandler({ CustomException1.class, CustomException2.class })
4     fun handleException() {
5         // logic
6     }
7 }
```

Listing 5.20: Sample controller with exception handler

Another analyzed approach was the implementation of `ErrorWebExceptionHandler` and overriding the method `override fun handle(ServerWebExchange, Throwable): Mono<Void>`. The purpose of this method is to catch any exceptions not handled by the exception handlers mentioned earlier. In the context of this method we have access to the argument of type `ServerWebExchange` which includes in its properties the request, the response and other useful information about the this particular connection that resulted in an exception. Here we are also able to access and customize the body property of the response as we wish.

Yet another studied approach was the usage of classes annotated with `@RestControllerAdvice`, these classes can intercept exceptions when controllers, respectively annotated with `@RestController` raise exceptions. With this class we can centralize all exception handlers in one place, decoupling the controllers of their exception handlers and moving the latter here.

The last studied solution was the usage of the exception class `ResponseStatusException`, which on the contrary to the first mentioned approach, we define raise the exceptions and set their status and reason locally when the exception is thrown. This reduces the amount of exception classes since one type of exception can be used for any issue which great for prototyping and testing. A trade off of this approach will be the increased

difficulty of code management and duplicated code inside the code blocks that raise exceptions.

The final solution taken was a mix and match of the described approaches. A global exception handler was created by utilizing the creation of a class annotated with `@RestControllerAdvice`. For specific exceptions, the usage of a method annotated with `@ExceptionHandler` and accepting the argument `ResponseStatusException::class` facilitated in manipulating the response body. With this solution, we benefit from global exception handling and decoupled controllers. This solution can be observed in listing 5.21.

```

1 @RestControllerAdvice
2 class GlobalExceptionHandler {
3
4     val logger = KotlinLogging.logger { }
5
6     @ExceptionHandler(ResponseStatusException::class)
7     suspend fun handleResponseStatus(ex: ResponseStatusException, req: ServerHttpRequest, rsp:
8         ServerHttpResponse): GenericExceptionDTO {
9         val exDTO = GenericExceptionDTO(
10             path = req.path.toString(),
11             uri = req.uri.toString(),
12             reason = ex.reason ?: "Internal Error",
13             status = ex.rawStatusCode
14         )
15         logger.error {exDTO}
16         rsp.statusCode = ex.status
17         rsp.headers.contentType = MediaType.APPLICATION_JSON
18         return exDTO
19     }
20 }

```

Listing 5.21: Implementation of a global exception handler.

5.2 Front-end module

In this section, an examination of the Front-End module development is made. Starting with the reasons for choosing each tool in this process, then the inner workings of each layer in this module, and concluding with how functional requirements are met.

It is structured as follows:

- Framework Stack 5.2.1 - describes the frameworks used for the development of the Front-End module;
- Configuration & Required Modules 5.2.2 - Describes the configuration and used modules to build the Front-End module;

- Model Layer 5.2.3 - Describes the model layer;
- Services Layer 5.2.4 - Describes the service layer;
- View Layer 5.2.5 - Describes the view layer;
- Security 5.2.5.1 - describes the steps taken to implement authentication and role based permission in the Front-End module;
- Meeting the Functional Requirements 5.2.6 - describes how each functional requirement was achieved.

5.2.1 Framework Stack

The FE module was developed using *ReactJs* [16], version 18, mainly due to its popularity and ease of use alongside a super set of *JavaScript; TypeScript* [22], version 4. The module is built using a node package manager, the popular *npm* [12].

During development the tool *npx webpack* was used to deploy the module, benefiting from its Watch⁸ feature where whenever a change is made on a source file the code is compiled without needing to manually compile and deploy.

Using the tool *npx* and *create-react-app*⁹ we can obtain a boiler template with some required settings for this module, namely, having the project in *TypeScript* and having it adapted to be a PWA. This is achieved through the command `npx create-react-app front-end --template cra-template-pwa-typescript`, which creates a ready to go application in the folder `front-end` using *TypeScript* and *ReactJs* with the requirements to be a PWA.

This module was developed to be a Single Page Application (SPA) and consists of three main components, the View, Model and Service.

5.2.2 Configuration & Required Modules

With the ready to application from the mentioned command, we are met with two important files for configuration and setting required modules - `tsconfig.js` (viewed in listing 5.22) and `package.json` (a brief showcase can viewed in listing 5.23 with

⁸<https://webpack.js.org/configuration/watch/#watch>

⁹<https://create-react-app.dev/>

the entire file available in the GitHub repository). The former refers to the typescript compiler options while the latter includes the required modules for development and production.

```
1 {
2   "compilerOptions": {
3     "target": "es6",
4     "lib": [
5       "dom",
6       "dom.iterable",
7       "esnext"
8     ],
9     "allowJs": true,
10    "skipLibCheck": true,
11    "esModuleInterop": true,
12    "allowSyntheticDefaultImports": true,
13    "strict": true,
14    "forceConsistentCasingInFileNames": true,
15    "noFallthroughCasesInSwitch": true,
16    "module": "esnext",
17    "moduleResolution": "node",
18    "resolveJsonModule": true,
19    "isolatedModules": true,
20    "noEmit": true,
21    "jsx": "react-jsx"
22  },
23  "include": [
24    "src"
25  ]
26 }
```

Listing 5.22: TypeScript compiler configuration file.

Out of the properties present in the `tsconfig.json` file 5.22, some of the most impactful are the following:

- `"target": "es6"` - This option tells the *TypeScript* compiler to generate JavaScript code that is compatible with *ECMAScript 6*, a widely supported version of the JavaScript language, allowing it to be compatible with most modern web browsers;
- `"strict": true` - This option enables a number of strict checks in the *TypeScript* compiler, amongst which is the stricter type checking and other checks that help detect errors in the code. It helps to improve the quality and maintainability of the developed code;
- `"include": ["src"]` - This option specifies that the *TypeScript* compiler should only include files inside the `src` folder and its sub-folders. It takes in an array of strings, each denoting what folder should be included in the compilation.

The `package.json` file, displayed in listing 5.23, is a plain *JSON* file and it contains information about the project, its dependencies, scripts, and version. The `"dependencies"` property lists the packages that the project depends on in order to function properly. These are packages that are required to run the project in production. Each dependency is represented by the package's name and version number. The version number can be specified in a variety of ways, such as an exact version number (e.g. `"1.2.3"`), a range of versions (e.g. `"^1.2.3"` or `"1.2.3"`), or the keyword `"latest"` to always use the latest version of the package. To make sure the project stays consistent we used the pattern `"^1.2.3"` so the dependency can be updated according to its major version, the use of a caret `"^"` version prefix indicates that any version that is compatible with the specified version can be used. For example, if a package has the version number `"^1.0.0"`, it will match any version starting with `"1."` Such as `"1.0.0"`, `"1.1.0"`, `"1.2.0"` etc, but not `"2.0.0"` or `"0.9.0"`.

The `"devDependencies"` property lists the packages that are required for development only, such as testing and building libraries, which in this case contains the `react-scripts` used for building and compiling the application. Once compiled this dependency is no longer required, thus having it included in this property makes it so the production build isn't bloated with this package.

When the command `run npm install` is run from the project directory, it will install all the packages listed in both mentioned property fields and create a `node_modules` folder that contains all the installed packages, along with their own dependencies.

During development the mentioned command was used with no other options, however, to deploy using a production build, we can pass the flag `--only=production` which will tell Node to skip the dependencies listed in the property `devDependencies`.

Along with these two properties, have another two that restrict the environment the project can run on. These are as follows:

- `"engines"` - Indicates the version of *NodeJs* and *npm* that the project requires to run properly.
- `"enginesStrict"` - Specifies whether the package should error if the version of *NodeJs* or *npm* is not compatible with the version specified in the `"engines"` field.

```
1 {  
2   "name": "cascaffo-front-end",  
3   "version": "1.0.0",  
4   "private": true,  
5   "dependencies": {  
6     "axios": "^1.2.1",
```

```
7   "bootstrap": "^5.2.3",
8   "chart.js": "^4.1.1",
9   "file-saver": "^2.0.5",
10  "react": "^18.2.0",
11  "react-bootstrap": "^2.7.0",
12  "react-chartjs-2": "^5.1.0",
13  "react-chrono": "^1.20.0",
14  "react-csv": "^2.2.2",
15  "react-data-table-component": "^7.5.3",
16  "react-dom": "^18.2.0",
17  "react-router-bootstrap": "^0.26.2",
18  "react-router-dom": "^6.6.0",
19  "react-toastify": "^9.1.1",
20  "serve": "^14.1.2",
21  "typescript": "^4.9.4",
22  //other dependencies
23 },
24 "devDependencies": {
25   "react-scripts": "^5.0.1"
26 },
27 "scripts": {
28   "start": "react-scripts start",
29   "build": "react-scripts build",
30   "test": "react-scripts test",
31   "eject": "react-scripts eject"
32 }
33 "engines": {
34   "node": "18.x",
35   "npm": "8.x"
36 },
37 "engineStrict": true,
38 //other configurations
39 }
```

Listing 5.23: Package.json configuration file.

In addition to these we also need a way to specify whether the current build is for development or production, to indicate which base URL the FE should use. This is relevant when the project is run during development with the BE module running on another URL, such as in two different *Heroku* containers. This can be done by creating two files, `.env.development` and `.env.production`, where we set the needed variables during on the type of build. Since we created the project using the command `npm create-react-app`, the project uses an environment variable `NODE_ENV`, will look for these two environments depending on the type of command being used to start the build. For example the command `npm start` makes it so the variable `NODE_ENV` will always have the value of "development" and therefore look for `.env.development` while the command `npm run build` will have `NODE_ENV` with the value of "production" and thus make it scan for `.env.production`. The variable `NODE_ENV` cannot be changed manually, however, in case we need to other variables, according to the official documentation on Adding

Custom Environment Variables ¹⁰ , we can set them prior to running the command, granted that the variable must be prefixed with `REACT_APP_`.

5.2.3 Model Layer

The model layer is where all model entities are held. The entities are represented through interfaces, since we only need to specify the shape of the object being worked on and don't expect any type of internal behavior. An example of these interfaces can be seen in listing 5.24. Interfaces were chosen over classes since we don't want any behavior attached to a model entity. In contrast, type alias and interfaces are very similar in *TypeScript* with the key distinction being that a type cannot be re-opened to add new properties versus an interface which is always extendable, as specified in the official documentation Everyday Types ¹¹ .

```
1 export interface UserModel {  
2     userId?: string,  
3     name?: string,  
4     email?: string,  
5     password?: string,  
6     roles?: UserRoleModel[],  
7 }
```

Listing 5.24: User Model Interface.

These model interfaces are used by the service and view layer.

5.2.4 Service Layer

The service layer is where data is requested via HTTP to the BE module.

All services are stateless and responsible for requesting data related to certain entities. Services may also communicate with each other to form an aggregate service, for the ease and convenience of centralizing the dependencies of a view component into one single aggregate.

Requests made over HTTP use the built-in `fetch` ¹² function. This function takes in one mandatory argument, the URL, and optional arguments specifying the options to

¹⁰<https://create-react-app.dev/docs/adding-custom-environment-variables/>

¹¹<https://www.typescriptlang.org/docs/handbook/2/everyday-types.html#differences-between-type-aliases-and-interfaces>

¹²https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API

be considered, such as whether the request being made is a POST or GET, as well as defining the headers and body of said request.

The services utilize this function indirectly through a manually made utility class which several utility methods. In the listing 5.25 we can observe the utility methods used to fetch data.

```

1  const APPLICATION_CONTENT_TYPE = 'application/json'
2  const HEADER_ACCEPT_JSON: [string,string] = ['Accept', APPLICATION_CONTENT_TYPE]
3  const HEADER_CONTENT_TYPE: [string,string] = ['Content-Type', APPLICATION_CONTENT_TYPE]
4  const HEADER_AUTHORIZATION = (token: string): [string,string] => ['Authorization', 'Bearer ' +
   token]
5  const HTTP_STATUS_NO_CONTENT = 204
6
7  export function httpGet<T>(url: string) : Promise<T> {
8      return _httpFetch(url, 'GET')
9  }
10 export function httpDelete<T>(url: string, body: unknown = null) : Promise<T> {
11     return _httpFetch(url, 'DELETE', [HEADER_CONTENT_TYPE], body)
12 }
13 export function httpPost<T>(url: string, body: unknown): Promise<T> {
14     return _httpFetch(url, 'POST', [HEADER_CONTENT_TYPE], body)
15 }
16 export function httpPostNoBody<T>(url: string): Promise<T> {
17     return _httpFetch(url, 'POST')
18 }
19 export function httpPut<T>(url: string, body: unknown = null): Promise<T> {
20     return _httpFetch(url, 'PUT', [HEADER_CONTENT_TYPE], body)
21 }

```

Listing 5.25: Utility fetch methods.

As observed in the listing, each of these methods use the function `_httpFetch`, as viewed in listing 5.26, this is where the built-in `fetch` function is wrapped. This function first prepares adds to the headers array the content type of the request, the type being `application/json` as observed in the previous listing(5.25). The communication between the FE and BE over HTTP consist in requests where data is passed in `JSON` format.

Afterwards, the local storage is accessed to retrieve a token, this token is the token sent back by the BE module upon a successful login, upon which it is stored in the browser's local storage. The token is required to be present in the header `Authorization`, prefixed by `Bearer` since it is a bearer token, to able to perform a number of operations within the platform, especially those that involve database write operations. Following the flow of control of the method, the options, named `opt`, is created with the previously set headers and the argument `url`, in case the body is not null, it is included in the options of the request, and finally the call to the BE is made through the function `fetch`. We can explicitly note that this function (`fetch`) is asynchronous as it returns a `Promise` of data. As a final note upon receiving a response from the BE, the response is then checked

for errors, in which case we retrieve the content of the message through `rsp.text()` and not `rsp.json()`. The former method reads the body as if it were a string thus being better suited to handle any type of possible error where the response body is not certain while the latter method expects the body to be formatted in *JSON*.

```

1 async function checkAndRaiseError(rsp: Response): Promise<Response> {
2     if(!rsp.ok) {
3         const status = rsp.status
4         const body = await rsp.text()
5         throw new MyError(
6             JSON.parse(body).reason, status)
7     }
8     return rsp
9 }
10
11 function _httpFetch<T>(  
12     url: string,  
13     method: string,  
14     headers: [string, string][] = [],  
15     body: unknown = null  
16 ): Promise<T> {  
17     headers.push(HEADER_ACCEPT_JSON)  
18     const token = localStorage.getItem(TOKEN_KEY)  
19     if(token != null) {  
20         const userToken = JSON.parse(token) as UserToken  
21         headers.push(HEADER_AUTHORIZATION(userToken?.token))  
22     }  
23     const opt : RequestInit = {  
24         headers: headers,  
25         method: method  
26     }  
27     if(body != null) {  
28         opt.body = JSON.stringify(body)  
29     }  
30     return fetch(url, opt)  
31         .then(checkAndRaiseError)  
32         .then(rsp => rsp.status === HTTP_STATUS_NO_CONTENT ? rsp : rsp.json())  
33 }

```

Listing 5.26: Utility fetch wrapper method implementation.

With the utility methods explained, the services can be easily described with the short piece of code observable in listing 5.27. All the methods in each service behaves in a similar way to it, prepare the request URL and call the utility function.

```

1 export class UserService {  
2     fetchAll() : Promise<Array<UserModel>> {  
3         return httpGet(ApiUrls.usersUrl)  
4     }  
5     fetchUser(userId: string): Promise<UserModel> {  
6         const url = ApiUrls.userDetailsUrl(userId)  
7         return httpGet(url)  
8     }  
9     login(userModel: UserModel) : Promise<UserToken> {  
10        return httpPost(ApiUrls.userLoginUrl, userModel)  


```

```
11     }
12     register(userModel: UserModel): Promise<UserToken> {
13         return httpPost(ApiUrls.userRegisterUrl, userModel)
14     }
15     addRolesToUser(userId: string, roleIds: number[]): Promise<UserModel> {
16         const url = ApiUrls.userRolesUrl(userId)
17         return httpPut(url, roleIds)
18     }
19     removeUserRoles(userId: string, roleIds: number[]): Promise<UserModel> {
20         const url = ApiUrls.userRolesUrl(userId)
21         return httpDelete(url, roleIds)
22     }
23     //other request methods ...
24 }
```

Listing 5.27: User Service implementation snippet.

5.2.5 View Layer

The view layer is where the user interaction begins, with the framework *ReactJs* [16], the UI is built with Components. A component can be understood as a building block of a page, it is reusable and provides ways to handle state and logic in an isolated manner. Conceptually, components are like JavaScript functions. They accept arbitrary inputs, props, and return React elements describing what should appear on the screen. Furthermore, a component can also be divided into two categories: state-full and state-less components, depending on whether they manage state or not. The usage of state in components with state can be represented with the react hook `useState<Type>(initialValue)`, whereas a stateless component, commonly referred to as functional component, does not handle state and only encapsulates logic and/or returns a React element based on the arguments it receives within the props property. Functional components are useful for re-usability and code readability.

Each component is created in the virtual DOM, a concept that represents the UI in memory as a lightweight, abstract tree of nodes. When the state of a React component changes, React updates the virtual DOM instead of the actual DOM. Then, React calculates the difference between the previous virtual DOM and the new virtual DOM, and applies the smallest possible updates to the actual DOM. This behavior allows pages to only re-render components that have been updated instead of the whole page, thus reducing the load on the browser and leading to an increased performance in page response time.

Each component goes through a lifecycle of events, beginning when the component

is created, followed when it's data is updated and finally when it's destroyed. These events in a class component correspond, respectively, to the functions:

- `componentDidMount()` - Called when the component is mounted and its initial render has completed;
- `componentDidUpdate()` - Called whenever there is a change in the data of the component which causes a re-render;
- `componentWillUnmount()` - Called before the component is unmounted, meaning removed from the DOM.

In the context of the FE module, function based components were used and in such components, instead of these methods, *React hooks* are used instead. The most notable *React hooks* are: `useState()` - used to declare "state variables", a call to this method returns the state variable and a function to set a new value to the variable that will trigger a re-render of the component; `useEffect()` - used to perform logic after the component has rendered, this method takes in a callback as first argument that will be executed upon a successful render of the component and can be customized to trigger only on certain state variable changes. This is achieved by passing a dependencies array as a second argument, where each array value corresponds directly to the variable to observe, in the case the dependencies array is empty or depends only on props, the provided callback will only execute on the initial render of the component.

In conjunction with React, other JavaScript libraries that offer a base of components and styles were used to build the UI, namely *React Bootstrap* for components and *Mazer UI*¹³ for styling.

With how each UI page is created explained, we need to address how we can navigate between pages, which leads us to *React Router*. *React Router* is a library that provides the capability of mapping URLs to specific components, making it easier to keep track of and manage the application as it grows. Additionally, *React Router* also supports dynamic routing, allowing the creation of routes with variables, such as `/proposal/:pId` where `:pId` identifies a variable named `pId`, this variable can then be accessed by using the *React* function `useParams()`. This function returns an object with all variables mapped as properties, as such, to access the previous `pId` we need only to retrieve the property `pId` from the object returned by the function in question.

To create a route using JSX syntax, we can utilize the following snippet `<Route path = "/" element=Component />`, where the property `path` indicates the URL and the element

¹³<https://github.com/zuramai/mazer>

property the component that should be displayed once the designated URL is in place. This feature is heavily used in the routing management of the FE module, as observed in listing 5.28 where a snippet of the creation of routes is shown.

```

1 export function CreateRoutes() {
2
3 return (
4   <Routes>
5     <Route path={"/"} element={RequiresAuth(<Dashboard statisticsService={new
StatisticsService()}/>)} />
6     <Route path={"/login"} element={<Login UserService={new UserService()}/>} />
7     <Route path={"/logout"} element={<Logout/ >} />
8     <Route path={"/propostas/:proposalId"}
9       element={RequiresAuth(<ProposalDetailsPage proposalService={new
ProposalAggregateService()}/>)} />
10    />
11    <Route path={"/ensaios/:researchId"}
12      element={RequiresAuth(<ResearchDetailsPage researchService={new
ResearchAggregateService()}/>)} />
13    />
14    <Route path={"/utilizadores/:userId/notificacoes"}
15      element={RequiresAuth(<NotificationsView service={new NotificationService()}/>)} />
16    />
17    <Route path={"/utilizadores/:userId"}
18      element={RequiresAuth(<UserDetails service={new UserService()}/>)} />
19    //other routes...
20  </Routes>

```

Listing 5.28: Routes snippet for FE module.

Another noteworthy aspect of React Router is how it handles URL navigation. Upon a URL change for example triggered by the user clicking on a link, it automatically displays the corresponding component.

With *React Router* handling the navigation client-side, we encounter a problem when this application is hosted via Spring. The *React Router* is loaded via JavaScript when the user goes to the homepage, from there, the library takes control of navigation, however, in the case where the user goes to an URL different from the homepage as it's first request or upon a hard refresh (made with Ctrl + F5 in a Windows OS), the router is not loaded thus causing an error page to be shown. To address this server side rendering issue, inside the BE module, the following snippet 5.29 was introduced to redirect any request that isn't the homepage URL towards it.

```

1 @Component
2 class ServeStaticContent : WebFilter {
3   override fun filter(exchange: ServerWebExchange, chain: WebFilterChain): Mono<Void> {
4     // server side rendering
5     // when a request is made while the client app is not loaded, like a hard refresh,
6     // send redirect the request to /index in order to first load the required javascript and
7     // let the front end handle it
8     return if (

```

```
9      !exchange.request.uri.path.startsWith("/api")
10      && exchange.request.uri.path != "/"
11      && !exchange.request.uri.path.startsWith("/static")
12      && !exchange.request.uri.path.startsWith("/img/")
13      && !exchange.request.uri.path.startsWith("/asset-manifest.json")
14      && !exchange.request.uri.path.startsWith("/manifest.json")
15    ) {
16      chain.filter(exchange.mutate().request(exchange.request.mutate().path("/index.html").
17        build()).build())
18    } else chain.filter(exchange)
19  }
```

Listing 5.29: Configuration snippet of Spring static content routing.

5.2.5.1 Security

Moving to security in the front-end module, role based access-control is a requirement, in order to satisfy this, a login system was developed using the concept of Higher Order Component (HOC), these components are the same as any other, however, they have the responsibility to perform validations before rendering other components. To implement this functionality, any page that requires a user being authenticated or having a specific role will be chained with a HOC, first the HOC is rendered, which performs validations on the user info available on the browser session storage and makes a decision on whether to render the page requested or display an error. Upon a successful login, the back-end module sends a valid JWT, the user identifier and the roles associated to the logged in user that is then stored within the browser's session storage.

5.2.6 Meeting the Functional Requirements

According to the functional requirements the screens were developed based on the mock ups detailed in the report. In addition, the module follows the standard of a Progressive Web Application (PWA), allowing the application to be downloaded from the browser, to work as a mobile app and offline functionality albeit limited. The dashboard screen currently displays several useful stats through graphs and tables.

Each page corresponds to a route that is handled by *React.Router* as mentioned previously in the document. Most of the requirements were fulfilled exactly as detailed in the mock-ups, as such, here we will focus on the ones that ended up with significant differences, namely:

- The addition of the financial contract tab when viewing the details of a clinical trial proposal;

- A user management page accessible only by the admin;
- Data management page to delete/create/update several data types, such as service type, therapeutic area and pathology.
- The implementation of the dashboard;

Along side these main changes, the filter options on general screens such as the proposal and research overview screens were removed due to the ability of filtering by property together with sorting by column.

5.2.6.1 Financial contract tab

The development of this additional tab, viewed in figure 5.3 was to centralize the validation of the financial contract in one place, since the state where the financial contract awaits validation is a two-step process involving both juridical and financial departments of HFF. The refusal on one's side implicates a new financial contract submission. In addition, this tab also facilitates viewing the history of approvals/refusals.

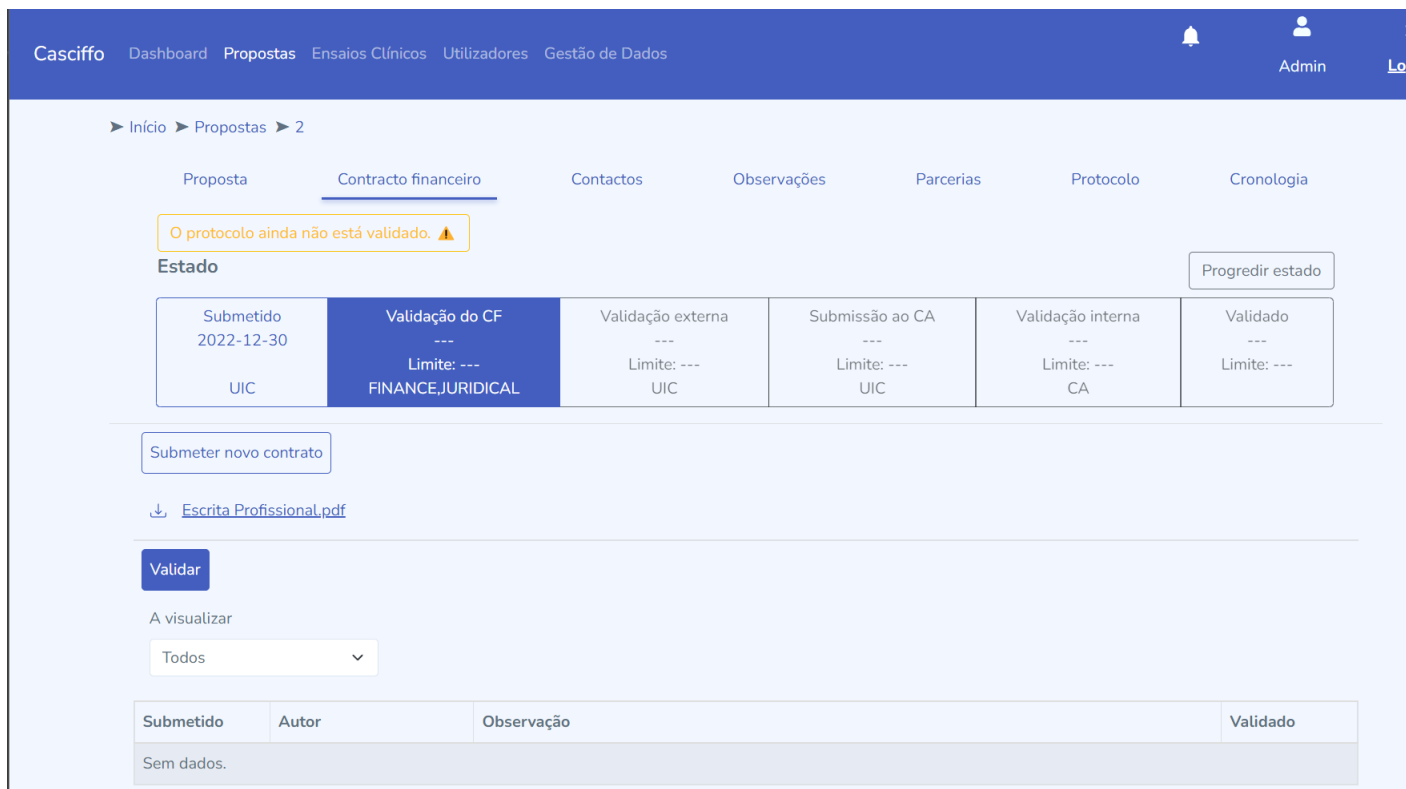


Figure 5.3: Financial contract tab UI.

5.2.6.2 User management page

Without a defined way of automatically pulling investigator data, this has to be introduced manually, thus the increased need for this screen, viewed in figure 5.4, arouse. In this screen, available only to administrator level users, users can be created by defining their name and email, afterwards the user themselves can define their own password. In conjunction with this feature, administrators can also grant and revoke permissions to other users.

The screenshot displays the 'Utilizadores' (Users) management page. At the top, a navigation bar includes the 'Casciffo' logo and links to 'Dashboard', 'Propostas', 'Ensaio Clínicos', 'Utilizadores', and 'Gestão de Dados'. A user profile icon labeled 'Admin' is in the top right. Below the navigation bar, a breadcrumb trail shows 'Início > Utilizadores'. The main content area features a 'Criar novo utilizador' (Create new user) form with input fields for 'Nome' (Name) and 'Email', both marked as required with red asterisks. Below these fields are 'Cancelar' and 'Criar' buttons. Underneath the form is a table of existing users:

Id	Nome	Email	Permissões	Ações
1	admin	casciffo.admin@admin.pt	Administrador	<button>Adicionar permissões</button> <button>Remover permissões</button>

Figure 5.4: User Management UI.

5.2.6.3 Data management page

The page of data management, in similarity to the user management page, is only available to administrator level users, allowing the creation, edit and deletion of data that can't otherwise be created normally in the application. This page, as viewed in figure 5.5, is organized by several tabs with their own purpose. The data for service types, therapeutic areas, pathologies, as well as patient data is manually created here.

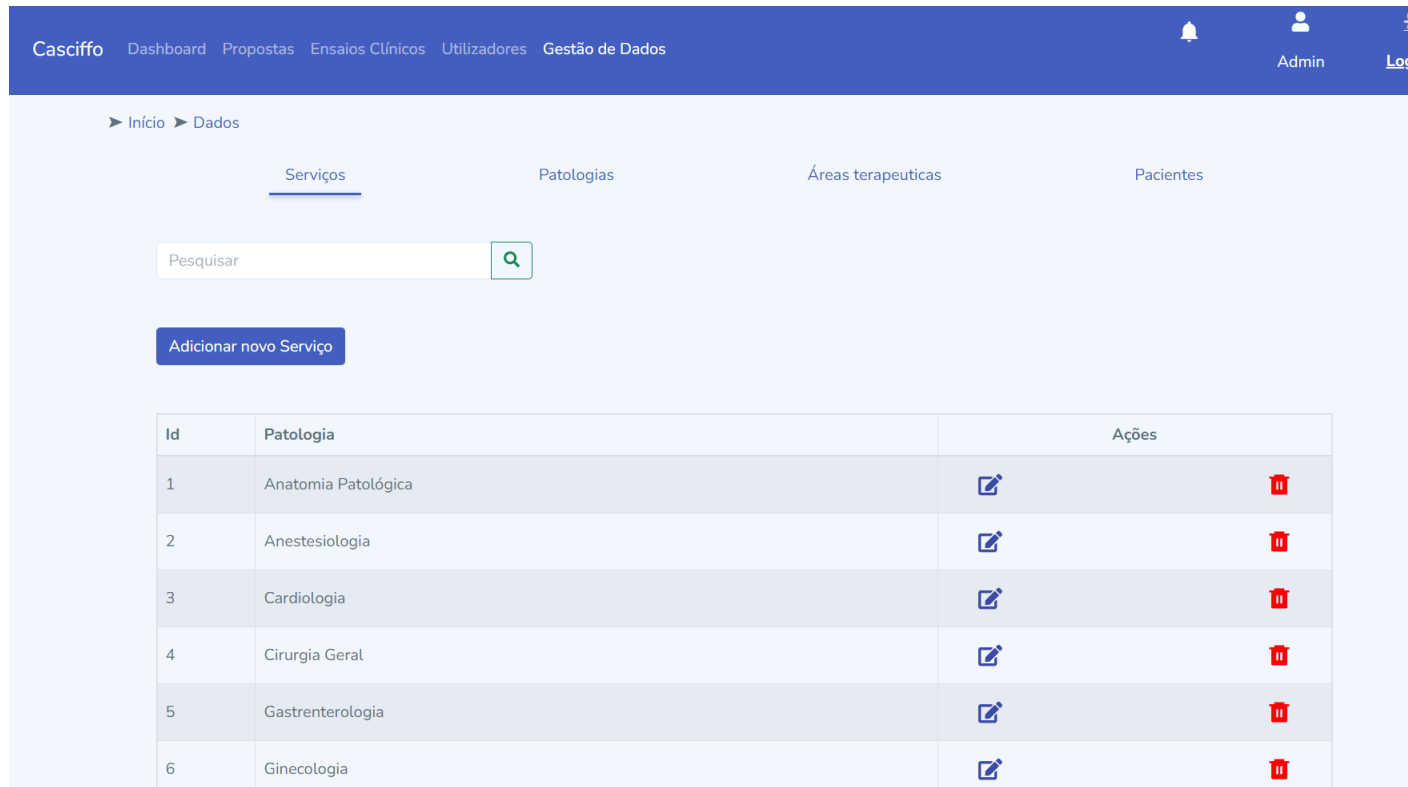


Figure 5.5: Data management UI.

5.2.6.4 Dashboard

The development of the dashboard was made with the goal of being able to view relevant data displayed in graphs and tables. Making an observation of this screen as viewed in figure 5.6, the data displayed in the first row of graphs refers to the amount of completed, on-going and canceled clinical and observational trials on the left hand side, while on the right hand side the amount of submitted, on-going validation and fully validated proposals is shown. Moving to the section table data section, the five most recently updated trials and proposals are shown. Below these two tables, the most events occurring on the current week are shown in a tabled manner as well as viewed in the figure 5.7.

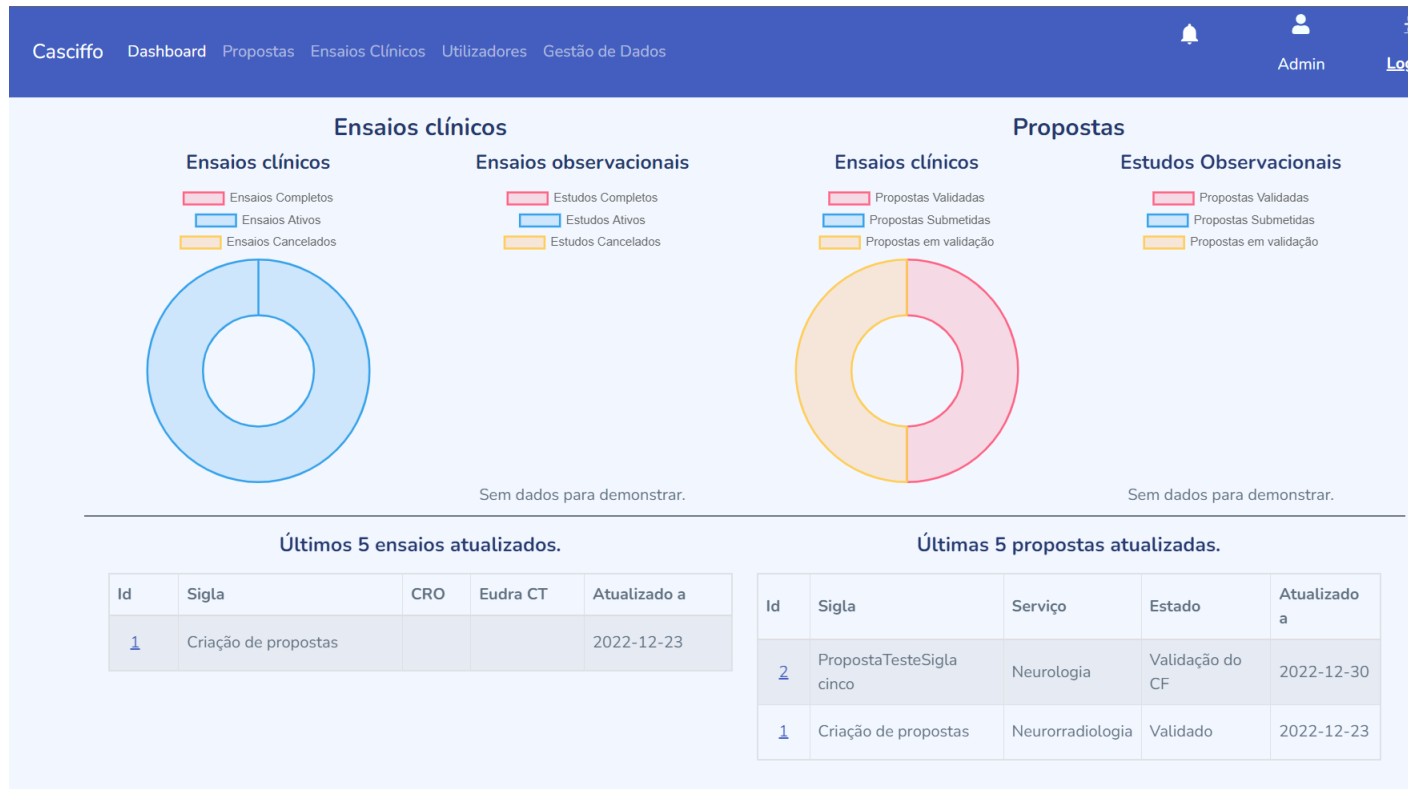


Figure 5.6: Dashboard UI 1/2.



Figure 5.7: Dashboard UI 2/2.

5.3 Installation & Deployments

This section describes the process of installation and deployment in two different settings, on Heroku and within an on-premises Ubuntu server. This section is structured

as follows:

- Installation 5.3.1
- Heroku Deployment 5.3.2 - Describes the requirements and deployment process to Heroku.
- On-Premises Deployment 5.3.3 - Describes the requirements and deployment process to the on-premises server in HFF.

5.3.1 Installation

Before installing the platform CASCIFFO, the prerequisites must first be met. The prerequisites refer to the technologies required to run the platform, namely, *PostgreSQL* version 12, *Java* version 17, *NodeJs* version 18 and finally *git tools* to fetch the project from its *GitHub* repository. Having these applications installed, we can pull the project to a local folder, which we'll name `casciffo` for convenience. Navigating to this folder, we will see two folders containing each module and two types of scripts that build and run the application. These types are *Batch* and *Bash* scripts, the former bundles and runs the application on Windows OS while the latter was made with Linux OS in mind.

The scripts were made to launch the application as a whole, by executing process to create a production build in the FE module and moving the resulting folder into the BE module's directory in a folder called `webapp`, after which the BE is started. In addition to this, there are configurable settings for the scripts, namely setting the port for which the BE module should attach to and the database user password.

5.3.2 Heroku Deployment

Heroku is a cloud-based Platform as a Service (PaaS) that enables users to deploy and manage web applications through the use of proprietary abstraction called dynos. These dynos are managed by the *Heroku* platform and provide an isolated environment for running the application. They can then be accessed via a unique URL based on the application's name (e.g. `example-app.herokuapp.com`).

This platform is a powerful tool for continuous delivery and development, offering a range of both premium and free services. The free services, used during development, offered by *Heroku* have the dynos put to sleep after a certain period of inactivity to conserve resources. When a request is sent to the app, the dyno wakes up and starts running again. This causes a delay as the dyno needs time to boot up and initialize all the necessary resources before it can start handling requests. As such a temporary lag can be felt in the performance of the application at on the first request.

Each dyno runs a specific process for the application as defined in a file called Procfile. This file contains commands that launch a dyno per command, allowing the app to be built and executed effectively.

With the goal of having continuous delivery of development, both modules were deployed to *Heroku*, each as their own separate application to serve as a testing environment. With each new feature a new deployment could be released per module without compromising the other.

A Procfile was created for both modules to start the application upon deployment in *Heroku*. In addition to this, the node module *express* was added to the FE in conjunction with a JavaScript file with the purpose of serving the static files through the use of the newly added module. On the BE module, a new task, viewed in listing 5.30, was added to the `build.gradle.kt` configuration file that cleans the project, compiles and bundles the application into a JAR file.

```
1 tasks {  
2     register("stage") {  
3         dependsOn(clean, bootJar)  
4         mustRunAfter(clean)  
5     }  
6 }
```

Listing 5.30: Gradle task added for *Heroku* deployment.

The Procfiles consist of a single command on each module, the FE has the command `web: node server.js` while the BE module contains the command `web: java -jar build/libs/casciffo-spring-backend-1.0.0-SNAPSHOT.jar`. In order to specify the *Java* version of this module, a file named `system.properties` was created with the only line `java.runtime.version=17`. This file indicates what version should be used for deploying the process in the dyno.

Another caveat to address was that since to connect to *Heroku* apps the HTTPS protocol is utilized, an additional setting needs to be added to the Spring application properties, that is `spring.r2dbc.properties.sslMode=REQUIRED`. This configuration specifies that SSL

mode connections are required to establish a successful connection to the database, hosted in the *Heroku* app.

Since the FE module was developed as a PWA, if a user has already been to the website, it will cache that version to permit offline use. Considering the hibernation by *Heroku* as well, resulted in some delays in testing the application. These were later solved by using a hard refresh (using the shortcut Ctrl + F5 on a Windows OS) while on the web page which clears the cache and forces the application to update to its most recent version.

As of November 28th, 2022, it was announced the Deprecation of *Heroku* free resources¹⁴. These resources included the previously free *PostgreSQL* database hosting, which rendered this continued development null.

5.3.3 On-Premises Deployment

The on-premises deployment consists in deploying CASCIFFO within a fresh Ubuntu Server using the 20.04 Long Term Support (LTS) version. After deployment the platform is to be hosted on Apache and it is important to note that upon entering the server we verified that the Apache service was already installed and available.

The first step in this process was setting up the Ubuntu server so that it can actually run the platform. These services are:

- *Gradle* - To compile the BE module;
- *Java* version 17 - To run the BE module;
- *PostgreSQL* version 12 - To integrate CASCIFFO's data base in the server;
- *Node* version 18 & *npm* version 8 - To run the FE module.
- *Git* - To be able to pull the project from its GitHub repository.

To install these services we also require the usage of the command `curl`, as such, the services were installed in the following order:

1. `sudo apt-get update` - Updates every current package to its latest version;
2. `sudo apt-get install git` - Installs *git*;

¹⁴<https://devcenter.heroku.com/changelog-items/2461>

3. `sudo apt-get install curl` - Installs *curl*;
4. `sudo apt-get install postgresql-12` - Installs *ProgreSQL* 12;
5. `sudo apt-get install openjdk-17-jdk` - Installs *Java* 17;
6. `curl -sL https://deb.nodesource.com/setup_18.x | sudo -E bash - && sudo apt-get install -y nodejs` - Fetches the setup for *Node*'s major version 18 and installs it in conjunction with *npm*'s major version 8.

To verify the correct versions of *Node* and *Java* we can use the command `node --version` for the former and `java --version` for the latter. Upon the successful installation of these services we can move onto the deployment of the platform.

5.3.4 Deploying the Application

The process to deploy the application to the server consists in fetching it from the GitHub repository and executing. To pull the project we used token based authentication to be able to perform the pull operation. The GitHub access token is a personal token generated by the user on request, it grants access to the user's account and repositories and can be used to authenticate the user's actions on the GitHub API and command-line operations. The token was generated with a set of permissions specific for the project's repository. To create the token, the steps detailed in the official documentation on Creating a personal access token¹⁵ were followed.

The directory where the project is pulled was chosen to be within the `/srv/` folder. With the git command to pull from a repository we can specify the directory where the pulled content should go. In this context the folder name used was `casciffo` resulting in the following directory structure `/srv/casciffo/`. Inside the project directory, there are several scripts inside with the purpose of setting up the *PostgreSQL* database (mentioned in section 5.1.2), and running the application. The database script should be the first in order of execution, as the application depends on it, however, there is a requirement that must be fulfilled beforehand; the *PostgreSQL* user `vp` must exist. To achieve this requirement, we used the command `sudo -u postgres create user -s vp -W` which prompts a password specification, which will later be used in the database credentials for the script running the app, for this example we can use 123456 as password. Upon creation, the database creation script can be executed by passing it as an

¹⁵<https://docs.github.com/en/authentication/keeping-your-account-and-data-secure/creating-a-personal-access-token>

argument to the `psql` command, like so `sudo -u postgres psql < db-creation.sql`. The script creates the database with the user `vp` as owner.

The script to run the application will need execution and read/write permissions, this can be done with the command `sudo chmod +xrw run.sh`. The flags `x r w` correspond to **e**xecute, **R**ead and **W**rite permissions. This also applies to the *JAR* file containing the bundled application. Once this is done, within the directory of the *JAR* file we can execute it by using the run script specifying a password and port like so `./run.sh --port=9000 --db_pwd=123456`.

Once we execute it, we should see the application starting via the command line.

5.3.5 Apache Hosting

With the application deployed, it now needs to be hosted by Apache to allow users to connect to it. To start this service through the command line, we ran the command `sudo systemctl start apache2`. In order to check the status of the service the command `sudo systemctl status apache2` can be used and will display the details of the current status. When the service is active and running, accessing the URL `http://localhost` should result in the default Apache web page.

We can verify that all required services are running with the command `sudo lsof -i -P -n | grep LISTEN`, this command lists all the currently listening network connections and their associated processes, the result of this command can be seen in figure 5.8. From the figure we can observe the service *PostgreSQL* listening on port 5432, the *Java* process, which corresponds to the *CASCIFFO* application, listening on port 9000 and finally the Apache service, under the name `apache2` listening on port 80.

```
alocal@HFFWVPRJISEL:/etc/apache2/sites-available$ sudo lsof -i -P -n | grep LISTEN
[sudo] password for alocal:
systemd-r 657 systemd-resolve 13u IPv4 22790      0t0  TCP 127.0.0.53:53 (LISTEN)
xrdp-sesm 822      root      7u  IPv6 26075      0t0  TCP [::1]:3350 (LISTEN)
xrdp      842      root     11u  IPv6 28244      0t0  TCP *:3389 (LISTEN)
sshd      843      root      3u  IPv4 27560      0t0  TCP *:22 (LISTEN)
sshd      843      root      4u  IPv6 27562      0t0  TCP *:22 (LISTEN)
smbd     248329   root     44u  IPv6 5500533     0t0  TCP *:445 (LISTEN)
smbd     248329   root     45u  IPv6 5500534     0t0  TCP *:139 (LISTEN)
smbd     248329   root     46u  IPv4 5500535     0t0  TCP *:445 (LISTEN)
smbd     248329   root     47u  IPv4 5500536     0t0  TCP *:139 (LISTEN)
postgres 450407   postgres 3u  IPv4 9410108     0t0  TCP 127.0.0.1:5432 (LISTEN)
java     1523758 root     23u  IPv6 33976507    0t0  TCP *:9000 (LISTEN)
apache2  1523804 root      4u  IPv6 33976028    0t0  TCP *:80 (LISTEN)
apache2  1526292 www-data 4u  IPv6 33976028    0t0  TCP *:80 (LISTEN)
apache2  1526293 www-data 4u  IPv6 33976028    0t0  TCP *:80 (LISTEN)
cupsd    1526357 root      6u  IPv6 34030256    0t0  TCP [::1]:631 (LISTEN)
cupsd    1526357 root      7u  IPv4 34030257    0t0  TCP 127.0.0.1:631 (LISTEN)
```

Figure 5.8: Command to list currently listening processes.

To host the application within the running Apache service, we need to create a configuration file made of Apache Directives tailored to the needs of the application. In

conjunction to this file used by Apache it's also needed to enable the necessary modules for the configuration to work. Firstly, the default location for the configurations of each web site hosted on Apache is in `/etc/apache2/sites-available`, from there, we can create our custom configuration in a file created via the command line with `sudo nano casciffo.conf`.

The configuration consists in setting up a virtual host listening on port 80, the default port of HTTP, since the application is supposed to be available over HTTP. The term Virtual Host¹⁶ refers to the practice of running multiple web sites on a single machine. These web sites can be IP-based meaning that what differentiates each web sites is their IP-address, or, as in this case, name-based meaning that each web site has a different host name. The users of this application need a way of accessing it, as we're using a name-based approach, the name of the virtual host needs to be defined, this can be achieved with the properties `ServerName` and `ServerAlias`, which were both set to `casciffo.hff.corp`. This allows the user to connect to the virtual host by navigating to the URL `http://casciffo.hff.corp`, however it will currently only show the Apache web server default page.

To follow-up, to have the application be hosted the property `DocumentRoot` is required to specify the location of the root directory of this virtual host, and as mentioned previously, the directory is `/srv/casciffo`, which is the value of the property in question.

Furthermore it is a best practice to include error and custom log files of the virtual host somewhere within the same directory, as such the properties `ErrorLog` and `CustomLog` were set to `/srv/casciffo/error.log` and `/srv/casciffo/custom.log`, respectively. The combined keyword is a format used by Apache web that includes several pieces of information about each logged request. This way we can perform monitoring operations on the application while being hosted and several other data analytics on the requests made.

Finally, although the user can reach the URL `http://casciffo.hff.corp`, the application is not currently receiving any requests, to reroute the requests in order for the application to be able to respond to user requests we utilize the `ProxyPass` and `ProxyPassReverse` properties, which take in two argument, the first being the requested URL on the virtual host, followed by the target URL Apache web should proxy it towards. Since we want to proxy every request to the application, the first argument is set to `/` while the target will be the URL of the running application. Since the application is running on localhost and we know its port, the second argument is set to `http://localhost:9000/`, as per our previous example on running the application indicated.

With the configurations set, we can proceed to the creation of the file `casciffo.conf` as

¹⁶<https://httpd.apache.org/docs/2.4/vhosts/>

viewed in listing 5.31. Having the configuration file created, all that's left is enabling the configuration through the command `a2ensite casciffo` and reloading the Apache service with `systemctl reload apache2`. However, doing this will result in an error due to the properties used, `ProxyPass` and `ProxyPassReverse` require the modules `proxy` and `proxy_http` both of which can be enabled with the command `a2enmod`. With the purpose completing these commands, we can chain them and use the following command chain snippet:

- `sudo a2enmod proxy && sudo a2enmod proxy_http && sudo a2ensite casciffo.conf && systemctl reload apache2`

```
1 <VirtualHost *:80>
2     ServerName casciffo.hff.corp
3     ServerAlias casciffo.hff.corp
4
5     DocumentRoot /srv/casciffo
6     ErrorLog /srv/casciffo/error.log
7     CustomLog /srv/casciffo/custom.log combined
8
9     ProxyPass / http://localhost:9000/
10    ProxyPassReverse / http://localhost:9000/
11 </VirtualHost>
```

Listing 5.31: Apache configuration file.

Upon completing these steps, we can verify the correct functioning of the application through the virtual host, by accessing the site `http://casciffo.hff.corp` from a different machine on the same network.

6

Module Evaluation

In this chapter, the evaluation and testing methodology of each module that make up CASCIFFO are detailed. The chapter has the following structure:

1. Back-End Evaluation 6.1;
2. Front-End Evaluation 6.2.

6.1 Back-End Evaluation

The back-end module evaluation consisted in assuring that each layer (controller, service and repository) was functioning properly. To achieve this, several integration tests were made utilizing the *Spring Unit* [19] testing framework, the integration tests used an in-memory database based on *H2* [4] so that they could run without affecting the development database. The configurations required to set up this in-memory database require a different set of configurations compared to when using the main database. This can be achieved by creating another `.properties` file with its name set to `application-test`, thus creating the file `application-test.properties`, a snippet of which can be viewed in listing 6.1. The striking differences are apparent in the properties `spring.r2dbc.url`, `spring.r2dbc.username` and `spring.sql.init.data-locations`. The first two are now set to the configurations of the *H2*, while the last one indicates where the testing data is located.

```
1 # Database settings
2 spring.r2dbc.url=r2dbc:h2:mem:///~/test;MODE=PostgreSQL;
3 spring.r2dbc.username=SA
4 spring.sql.init.schema-locations=classpath:sql/schema.sql
5 spring.sql.init.data-locations=classpath:sql/data-h2-test.sql
```

Listing 6.1: Spring properties configurations with H2 database.

In addition to the changes of the configuration file, we also require new testing libraries on the project, namely *H2*, *Spring Boot* and coroutines testing libraries. With this goal in mind, the following dependencies in listing 6.2 were added to the `build.gradle.kt` file.

```
1 dependencies {
2     // implementation and runtimeOnly dependencies ...
3     // test dependencies.
4     testImplementation("io.r2dbc:r2dbc-spi:1.0.0.RELEASE")
5     testImplementation("com.h2database:h2:2.1.214")
6     testImplementation("io.r2dbc:r2dbc-h2:1.0.0.RELEASE")
7     testImplementation("org.springframework.boot:spring-boot-starter-test
8     :3.0.0")
9     testImplementation("io.projectreactor:reactor-test:3.4.24")
10 }
11 testImplementation("org.jetbrains.kotlinx:kotlinx-coroutines-test:1.6.4")
```

Listing 6.2: Spring test dependencies snippet.

In order to start *H2* [4], assuming it is installed, the service needs to be launched and afterwards it should automatically open a browser and direct the user to a localhost URL that initially may display an unsafe warning. This warning can be ignored since it is running on the local machine, not remotely. Upon advancing, the user is met with a login screen, here we can configure the credentials according to the previously created configurations, resulting in the settings which can be observed in figure 6.1.

[scale=0.5]

Figure 6.1: H2 configurations.

With the database created and configured, the next step consists in creating the test classes. In order for these classes to use the different application properties, the annotation `@ActiveProfiles(value = ["test"])` was used to declare which Spring profile the application will use when loading the application context for test classes ¹. In addition, to launch the tests with the required application context with Spring Boot and provide other features, the annotation `@SpringBootTest` ² was also used. A test file was created for each component with several methods testing the control flow and logic of their respective component. Since the application environment is non-blocking, two approaches were used to implement the tests. The first consists in using the `StepVerifier` utility class, provided in the Project Reactor library that includes utility test methods that assert the behavior of reactive data streams of `Mono` and `Flux`. This is especially useful for testing the repository component, for example, in the listing 6.3, we verify the correct verification of a proposal entity. Through several methods available in the utility class `StepVerifier`, the entire behavior of the data stream can be tested. It is important to have each `StepVerifier` block run the method `.expectSubscription()` to subscribe to the stream passed as argument.

```

1 @Test
2 fun whenProposalCreated_thenFindIdInRepository () {
3     val proposal = ProposalModel(
4         sigla = "Created for test",
5         type = ResearchType.OBSERVATIONAL_STUDY,

```

¹<https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/test/context/ActiveProfiles.html>

²<https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/test/context/SpringBootTest.html>

```

6      stateId = 1,
7      principalInvestigatorId = 1,
8      therapeuticAreaId = 1,
9      serviceTypeId = 1,
10     pathologyId = 1
11 )
12
13 StepVerifier
14     .create(proposalRepository.save(proposal))
15     .expectSubscription()
16     .as`("Create proposal")
17     .consumeNextWith {
18         assert(it.id != null)
19         proposal.id = it.id
20     }
21     .expectComplete()
22     .log()
23     .verifyThenAssertThat()
24
25 StepVerifier
26     .create(proposalRepository.findById(pId = proposal.id!!))
27     .expectSubscription()
28     .as`("Find created proposal with id ${proposal.id}")
29     .thenAwait()
30     .assertNext {
31         it.id == proposal.id
32     }
33     .expectComplete()
34     .log()
35     .verifyThenAssertThat()
36 }

```

Listing 6.3: Test method for the Proposal Repository.

This approach was used to test the repository layer, as each repository returned reactive data streams.

The second approach which was used to test the service layer, consisted in realizing the test inside a coroutine with the keyword `runBlocking` which runs a coroutine in a blocking manner, meaning that the code inside it will be executed synchronously and will block the current thread until the coroutine completes, an example of this can be observed in listing 6.4. It is important to note that this approach was used since the service layer heavily relies on *Kotlin coroutines*.

```

1 @Test
2 fun testServiceAddPatientToResearch() {
3     var patient = PatientModel(
4         processId = 102,
5         fullName = "Manuel Santos",
6         gender = "m",
7         age = 50
8     )
9     runBlocking {
10         patient = patientService.save(patient)

```

```

11      val addedPatient = participantService.addPatientToResearch(patientId = patient.id!!,
researchId = 1)
12      val patients = patientService.findAllByResearchId(researchId = 1)
13      assertDoesNotThrow {
14          patients.first { assert(it.id === addedPatient.id) }
15      }
16  }
17 }

```

Listing 6.4: Unit integration test for adding a patient to an on-going research.

Finally the controller layer was tested using an external tool called *Postman* [14]. This platform allows the user to verify the responses and facilitate the ability to manipulate the header requests as well as viewing the response headers. The Postman request collection is available within the *Github* repository. Within Postman there are several fields we can define to build a request and receive a response, as viewed in the figure 6.2 which depicts a login request. This is a POST request made to the application running in localhost, hence the localhost URL which is defined with the variable localhost surrounded by curly braces, along with the request body specifying the credentials of the user. Upon executing the request, the response received is also detailed the figure, consisting of the access token of the user, their roles, user id and user name.

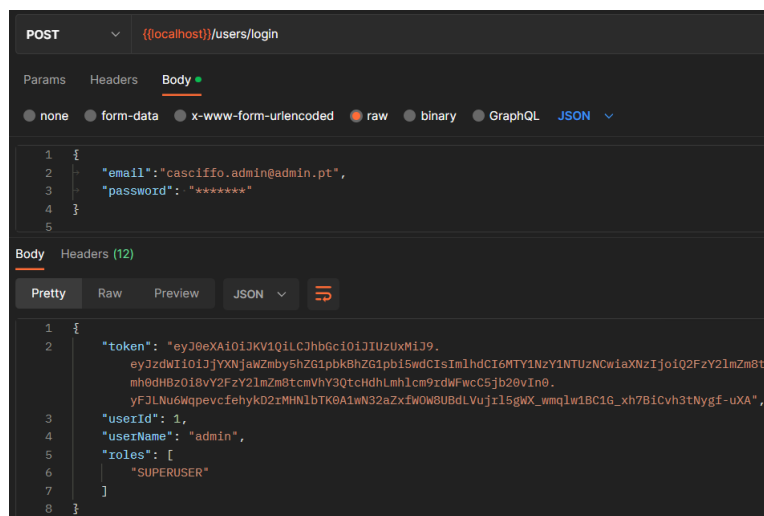


Figure 6.2: Authentication request made in Postman.

6.2 Front-End Evaluation

The FE module evaluation consisted in the frequent use of the application with the usage of the continuous development methodology. To measure browser performance,

the developer tool *Lighthouse* was utilized to point out possible issues.



Conclusions and Future Work

The platform CASCIFFO was developed with the main goal of modernizing HFF's clinical investigation unit, bringing new simple and effective ways of managing clinical trials. This management includes tracking the progress of proposals and clinical trials; asset management in clinical trials; financial management by keeping track of budget use; scheduling visits transparent to their physical location (face-to-face or teleconsultation); management and tracking of patients who are added to a clinical trial as participants and finally allowing data to be exported into Excel files.

Defining the platform's functional requirements was the first step in developing it. This allows us to gain a clear understanding of the problems that need to be addressed and create a suitable solution for each problem according to their needs. Additionally, by delineating the functional requirements, the business logic flow will be derived naturally from their analysis.

To help with visualization of the requirements, several User Interface (UI) mock-ups were created with the goal for providing a guideline on addressing each functional requirement.

Upon completing this step, we began an analysis of the technology and frameworks that will be used in CASCIFFO's implementation. The platform was planned to have 2 main modules, the Back-End (BE) module that manages business logic, and the Front-End (FE) module where user interactive begins and presents the platform's capabilities to the user. The interaction of these modules was defined to be over HTTP, by exposing the BE as a REST API to be consumed by the FE.

With a mind set on innovative technology, the framework chosen for the BE was *Spring WebFlux* [20], in conjuncture with *Gradle* and written in *Kotlin*.

The choice for *Spring WebFlux* [20] resides in the non-blocking environment offered by the framework. Not only is it fully capable of receiving and responding over HTTP in a reactive manner, while also providing support for the *R2DBC driver* [15]; a database driver capable of making non-blocking access calls to a certain database. In the scope of this platform, the database was written in *PostgreSQL* [13], chosen for its capabilities and the support available through the community and documentation.

Moving onto the technology that makes up the FE, we chose a *Node.js* [11] environment along with the framework *React.js* [16] in conjunction with the language *TypeScript* [22]. Since the FE consists in the development of the website of the platform CASCIFFO, choosing *Node.js* [11] was an immediate choice, *React.js* [16] was chosen due to its simplicity in creating websites by using a reactive and component based approach; incredible popularity and support; the community and documentation available makes it an alluring framework to choose. Finally, with *TypeScript* [22] being a super set of *JavaScript*, offering type safety and many other features that allow detection of errors at compile time was the first and foremost choice and final decision for the programming language of the FE module.

Once the second phase of development - choosing the technology for each module - finished, the third phase began. This phase consists in the implementation of both modules accompanied by continuous testing for quality assurance. To facilitate continuous development, the *Heroku* cloud-based service was used, which allowed for testing, rapid maintenance and feature delivery. Having both modules in separate *Heroku* instances allowed either one to work independently, which enabled the deployment of newer versions on either module without implying a deployment on the one other as well. Although the FE relies heavily on the BE module for data, it can function on its own as a Progressive Web Application (PWA). As a plus, the database was also stored in conjunction with the BE module in its *Heroku* instance.

Although the use of *Heroku* allows for testing by simply accessing the URL of the instance in question, since it's using their free plan, the instance will hibernate after a set timeout. Accessing the instance URL will wake it up and make it ready for use. This process takes time and causes a delay on startup. As of November 28th of 2022, *Heroku* has deprecated ¹ some of it's free resources, including *PostgreSQL* hosting, thus not allowing the recreation of this process without spending resources.

When it came to testing each module, the BE module was tested using unit tests for

¹<https://devcenter.heroku.com/changelog-items/2461>

each non-controller component. The controllers, which consume the HTTP requests, were tested with the help of the framework *Postman* [14]. The tests made for the FE module consisted on two main aspects, the performance evaluation made by the *Developer Tool Lighthouse* [10] and the feedback given by the team at HFF accompanying us.

Once the implementation was finished, the platform was deployed on-premises on a dedicated server instance running on Ubuntu and served by the Apache service. Based on the needs of the HFF, the application was hosted via HTTP, not HTTPS, which deters the possibility for the FE to be a PWA. This could be a future next step, as the module is fully ready to operate as a PWA needing only to be hosted via HTTPS.

Looking at the developed solution, we can conclude that the functional requirements were met, though some deviated from their original UI mock-up design due to sudden business logic changes or previously uncounted obstacles. As to the non-functional requirements, the integration of the HFF/UIC database - "Admission", was not completed in time, having been replaced with the possibility of adding patient and investigator data manually through the platform. Although it is early to conclude whether the platform CASCIFFO will achieve its desired goal in full, it is certain that it's a step into the right direction.

Setting the sight on the future of CASCIFFO, an interesting take on the next steps of development could be the addition of new functionalities and quality of life improvements, such as:

- Having a screen dedicated to the finance and juridical roles to display proposals pending validation;
- Adding a timeline with events on the financial aspect of a clinical trial;
- Having push notifications over site notifications;
- Integrating with a finance module to provide improved ways of finance management of clinical trials;
- Integration with a calendar platform to merge the scheduled visits in that calendar;
- Implementation of a Publisher Subscribe ² methodology to allow users to 'Subscribe' to updates on certain clinical trials;

²<https://learn.microsoft.com/en-us/azure/architecture/patterns/publisher-subscriber>

- Standardize the responses made by the Back-End module. For example using HATEOAS.
- Analysis of hosting via HTTP vs HTTPS, as the latter allows the FE module to be considered a PWA.
- Integration with the HFF's internal database to pull patient and researcher data over having to manually add each of them.

References

- [1] *About advarra*, Mar. 2022. [Online]. Available: <https://www.advarra.com/about/>.
- [2] Comissão de Ética para a Investigação Clínica, *Apresentação*, Jan. 2022. [Online]. Available: <https://www.ceic.pt/missao>.
- [3] *Revolutionize the way you conduct clinical research*, Mar. 2022. [Online]. Available: <https://info.advarra.com/clinical-conductor-demo-request-capterra.html>.
- [4] *H2 - quick start*, Nov. 2022. [Online]. Available: <https://www.h2database.com/html/quickstart.html>.
- [5] Hospital Professor Doutor Fernando Fonseca, *Introduction of the hospital professor doutor fernando fonseca*, Jan. 2022. [Online]. Available: <https://hff.min-saude.pt/>.
- [6] Portugal Clinical Trials, *Hospital professor doutor fernando fonseca*, Jan. 2022. [Online]. Available: <https://portugalclinicaltrials.com/pt/centros-de-investigacao-clinica/hospital-professor-doutor-fernando-fonseca/>.
- [7] JetBrains, *Coroutines*, Nov. 2022. [Online]. Available: <https://kotlinlang.org/docs/coroutines-overview.html>.
- [8] JetBrains, *Coroutines basics*, Nov. 2022. [Online]. Available: <https://kotlinlang.org/docs/coroutines-basics.html#your-first-coroutine>.
- [9] Kumar Chandrakant, *Light-weight concurrency in java and kotlin*, Nov. 2022. [Online]. Available: <https://www.baeldung.com/kotlin/java-kotlin-lightweight-concurrency>.

- [10] Jeremy Wagner, *Using lighthouse to improve page load performance*, May 2018. [Online]. Available: <https://developer.chrome.com/blog/lighthouse-load-performance/>.
- [11] Node Inc., *About node.js*, 2022. [Online]. Available: <https://nodejs.org/en/about/>.
- [12] *About npm*, Jan. 2022. [Online]. Available: <https://docs.npmjs.com/about-npm>.
- [13] *What is postgresql?*, Jan. 2022. [Online]. Available: <https://www.postgresql.org/about/>.
- [14] Inc. Postman, *About postman*, Nov. 2022. [Online]. Available: <https://www.postman.com/company/about-postman/>.
- [15] *Spring data r2dbc*, Mar. 2022. [Online]. Available: <https://spring.io/projects/spring-data-r2dbc>.
- [16] *React*, Jan. 2022. [Online]. Available: <https://reactjs.org/>.
- [17] *Revolutionize the way you conduct clinical research*, Mar. 2022. [Online]. Available: <https://realtime-ctms.com/clinical-research-software/>.
- [18] *Spring boot*, Mar. 2022. [Online]. Available: <https://spring.io/projects/spring-boot>.
- [19] *Unit testing*, Mar. 2022. [Online]. Available: <https://docs.spring.io/spring-framework/docs/current/reference/html/testing.html#unit-testing>.
- [20] *Web on reactive stack*, Mar. 2022. [Online]. Available: <https://docs.spring.io/spring-framework/docs/current/reference/html/web-reactive.html#webflux-new-framework>.
- [21] Portugal Clinical Trials, *O que é um estudo clínico?*, Jan. 2022. [Online]. Available: <https://portugalclinicaltrials.com/pt/porque-e-que-os-estudos-clinicos-sao-importantes/>.
- [22] Microsoft, *Why does typescript exist?*, 2022. [Online]. Available: <https://www.typescriptlang.org/why-create-typescript>.
- [23] *Yarn*, Jan. 2022. [Online]. Available: <https://yarnpkg.com/getting-started>.