DADIVA IPO

Digital **A**id and **D**onor **I**nformation **V**erification **A**pplication for **IPO**

Francisco Medeiros
Luís Macário
Ricardo Pinto

Supervisors: Filipe Freitas, ISEL
João Pereira, COFIDIS

Final report written for Project and Seminary BSc in Computer Science and Computer Engineering

September de 2024

Instituto Superior de Engenharia de Lisboa

DADIVA IPO

Digital **A**id and **D**onor **I**nformation **V**erification **A**pplication for **IPO**

46331 Francisco Rodrigues Medeiros

47671 Luís Miguel Teixeira Macário

47673 Ricardo Parreira Pinto

Supervisors: Filipe Freitas, ISEL

Joao Pereira, COFIDIS

Final report written for Project and Seminary BSc in Computer Science and Computer Engineering

September de 2024

Resumo

O Instituto Português de Oncologia (IPO) em Lisboa utiliza atualmente um sistema manual para a gestão de informações dos dadores de sangue. Este processo envolve os dadores preencherem um formulário pré-doação em papel, seguido por uma entrevista médica onde um médico avalia a elegibilidade para a doação, com base no formulário e questões verbais adicionais. Este processo manual de gestão e verificação de detalhes médicos e de medicação é altamente ineficiente e pode levar a imprecisões com consequências graves.

O projeto proposto visa digitalizar o processo de doação de sangue no IPO. Isto inclui a criação de uma versão digital do formulário pré-doação e o desenvolvimento de um sistema para gerir e comparar automaticamente dados sobre interações medicamentosas e patológicas com a dádiva. O sistema digital permitirá a fácil atualização, personalização e recuperação de informações. Ao automatizar o formulário e a gestão de dados, o projeto procura reduzir os erros associados à gestão manual dos mesmos, uma vez que a informação sobre medicação e patologia pode ser atualizada regularmente, e diminuir o tempo total necessário para o processo de doação, agilizando assim os procedimentos de triagem, aumentando a ergonomia do processo para dadores e médicos, reduzindo a dependência de papel e tornando possível preencher o formulário fora das instalações do IPO, reduzindo assim a permanência dos dadores no IPO.

Abstract

The Instituto Português de Oncologia (IPO) in Lisbon currently employs a manual system for managing blood donor information. This involves donors completing a pre-donation form on paper, followed by a medical interview where a doctor assesses eligibility based on the form and additional verbal questions. This manual process of handling and verifying pathology and medication details is highly inefficient, and may lead to imprecisions with severe consequences.

The proposed project aims to digitalize the blood donation process at IPO. This includes creating a digital version of the pre-donation form and developing a system to manage and cross-reference medication and pathology data. The digital system will allow for easy updating, customization, and retrieval of information. By automating the form and data handling, the project seeks to reduce errors associated with manual data management, since medication and pathology information can be regularly updated, and decrease the overall time required for the donation process, thereby streamlining triage procedures, increase process ergonomics for both donors and doctors, reduce paper reliance and make it possible to fill out the form outside IPO's installations, hence reducing donor's time commitment per donation.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Blood donation services play a vital role in the healthcare systems of nations worldwide, serving as a cornerstone of public health initiatives. In Portugal, the establishment of the Blood National Institute (Instituto Nacional do Sangue) in 1958 marked the inception of formal coordination of transfusion medicine. This institution, evolving over more than five decades, culminated in the establishment of the Portuguese Blood and Transplantation Institute (Instituto Português do Sangue e da Transplantação, IPST) in 2012 [1].

Throughout this historical trajectory, blood donation services have undergone substantial organizational reforms aimed at ensuring the safety of both donors and recipients. However, the donor screening process has seen limited evolution despite these systemic changes.

The "Council Recommendation of 29 June 1998 on the suitability of blood and plasma donors and the screening of donated blood in the European Community" [2] underscores the importance of gathering information from potential donors through written questionnaires. Although the specifics of these questionnaires may vary among Member States, their primary objective remains consistent: to identify common risk behaviors and diseases.

According to the 2022 Transfusion Activity and the Portuguese Hemovigilance System Report [3], Portugal recorded 306,796 blood donations from 203,287 donors, with 373,209 donor registrations during the same period. Notably, the main reason for the temporary suspension of blood donations is low hemoglobin levels, followed by recent travel to high-risk regions and engagement in behaviors associated with increased health risks.

Institutions like the Portuguese Oncology Institute (Instituto Português de Oncologia, IPO) in Lisbon, which contributed 1.88% of total blood donations in 2022, still rely on traditional, paper-based questionnaires for donor screening. However, this manual process, coupled with the need for cross-referencing against guidelines provided by IPST, is susceptible to inefficiencies and errors. Such inefficiencies may contribute to reduced donor adherence and suboptimal health outcomes.

In partnership with Lisbon's IPO this project endeavors to address these challenges by developing a digital platform. The platform aims to provide donors with a comprehensive digital questionnaire encompassing both standard and relevant sub-questions pertinent to the

screening process. For healthcare professionals, the platform will offer streamlined access to donor responses alongside information regarding potential health risks. Additionally, administrators will have tools to manage user accounts, questionnaire structures, and information regarding drug/disease interactions with blood donation.

By reducing the need for additional questions during screening consultations, this platform seeks to enhance donor participation. This is particularly crucial given the observed decline in donor numbers and donations from 2013 to 2022, amounting to a decrease of over 30,000 donors and 50,000 donations. Through these efforts, we aim to foster greater engagement with blood donation initiatives, thus contributing to the broader health and well-being of our community.

The main challenge with this project is regulatory compliance, particularly given our team's limited expertise in this domain and, to confront this challenge, our development strategy prioritizes the creation of adaptable functionalities designed to meet a broad range of regulatory requirements. Additionally, maintaining close collaboration with Lisbon's IPO will afford us invaluable guidance, ensuring our platform aligns with established frameworks and standards. By taking these proactive measures, we aim to navigate regulatory complexities effectively and develop a robust, compliant solution that can be tailored to the needs of blood donation services.

Chapter 2

Problem Description

Current blood donation workflow faces a set of challenges like screening time for more complex cases, since higher complexity cases may require cross-checking information about drug and pathology interaction, a process that, beyond being time-consuming, may lead to imprecisions with severe consequences. Currently upon form changes the previously printed forms are disregarded, this process can be expedited by supporting a digital form that can be easily updated, helping IPO reduce its paper consumption.

These challenges can be met by employing a dynamic form, in digital format, that shows relevant follow up questions according to the potential donor's answers, thus collecting relevant information, that would otherwise need to be obtained during the medical screening. This solution raises a set of questions such as:

- What data structure is appropriate to describe the form's structure and flow/logic - the questions order, possible answer values, what answers trigger or suppress follow-up questions;
- How will the form's rule be enforced in a way that doesn't force code implementation changes upon form structure changes - the frontend should be able to show and compute various forms and its unfeasible to change the frontend implementation upon every form structure change.

Upon form submission, the information supplied by the potential donor, or automatically obtained, can be automatically cross-checked against IPST guidelines for drug and pathology interaction with blood donation. This solution raises a set of questions such as:

- How are the potential donor's drug and disease information validated - the number of available drugs and possible diseases might be too large for real time validation, when the user is inputting that information into the form;
- Are the IPST guidelines available in a machine readable format that make it feasible to be cross-checked against the form's answers - to our knowledge, the guidelines are

available in pdf and printed format, sometimes drugs/pathologies are individually mentioned and sometimes grouped in a family (ie there's no mention of aspirin in the 2022 manual, being replaced by Non Steroidal Anti Inflammatory, the family of drugs this medication belongs to).

The digital form structure and flow, pathology/drug interaction, and terms of service information should be updatable in the back-office.

This solution raises a set of questions such as:

- How can the form structure and flow be visualized intuitively - the user changing the form shouldn't need to know anything about its implementation but still be able to identify and change its structure and flow;
- How will the drug/pathology interaction be updated - will a user manually insert information in the platform or can this information be requested via a web-service.

Beyond these specific challenges the platform will have to employ multiple types of users, each with a given set of accesses, there are multiple ways of implementing role-based access control, each with pros and cons.

2.1 Proposed Solution

In order to solve the challenges listed above, we have developed DADIVA IPO.

DADIVA IPO is a web platform that allows blood donation services to decrease the screening time of blood donation candidates via a digital, updateable and dynamic form as well as automatic interaction verification.

It is intended as an alternative to the current, and less versatile, paper form used by blood donation services in Portugal, such as Lisbon's IPO.

2.1.1 Functional Requirements

- Donors should be able to quickly fill out a digital pre-donation form. The form should be adequate according to the current law, adaptable, and depend on the donor's answers.
- Doctors should be able to find all relevant data on pathology and/or medication interactions with the donation in a digital format.
- Doctors and administrators should be able to access a back office used for customizing the pre-donation form and for updating the pathology and/or medication interaction information. The back office should also allow for user management.
- Google-like search and results by relevance - Search should be as simple as possible. There may be a need to increase the number of filters, but this complexity should be hidden. The results returned should be sorted based on relevance.

2.1.2 Non-Functional Requirements

- Intuitive user experience through a simple and practical user interface.

- Responsive design that ensures a good user experience both on desktop and mobile.

- Complete and thorough documentation.

- Unit and integration testing with sufficient coverage to ensure confidence that the system is working without flaws.

- Good software engineering practices to ensure the fast development of the system.

2.1.3 Optional Features

- After filling out the pre-donation form, the system could automatically check if the donor had any vaccinations and/or prescriptions that could be medically relevant. It would require integration with the SNS, and/or Infarmed systems.

- The medical interview may be based on a pre-analysis, with the system having already identified possible risk vectors and logical incongruencies that better assist the doctor when deciding on accepting or refusing the donor.

- It is possible that the IPST has already implemented a digital system to maintain pathology and medication interaction information. If so, it would be possible to integrate this into our system, so that this information does not have to be manually updated.

- Users can authenticate using the Digital Mobile Key (CMD). It would require integration with the AMA (Administrative Modernization Agency) systems.

2.1.4 Use Cases

With the requirements listed above, we have identified the use cases that the platform shall support. A use case is a written description of how users will perform tasks on a system. It outlines, from the user's perspective, the behavior of the system as it responds to a request. This approach attempts to predict the users of the platform, their allowed actions and objectives, and how the platform should respond to each action. The use cases are divided into three categories, each representing one type of user. The Donor use case is presented in Figure 2.1. The donor user can request the current form and can submit their form responses.

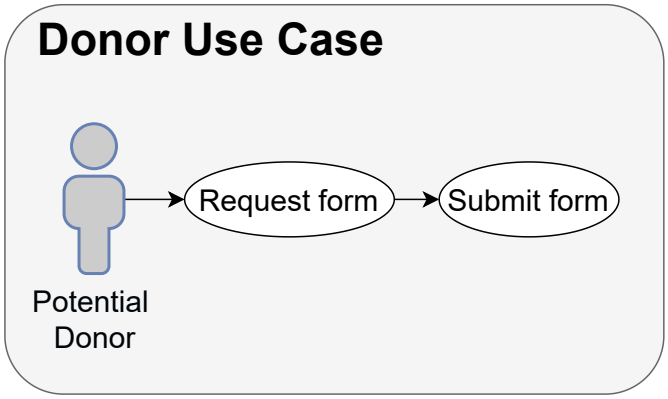

Figure 2.1: Donor use case.

After a donor submits their form responses a doctor user will be able to access their answers by searching by the user's unique id as presented in Figure 2.2.

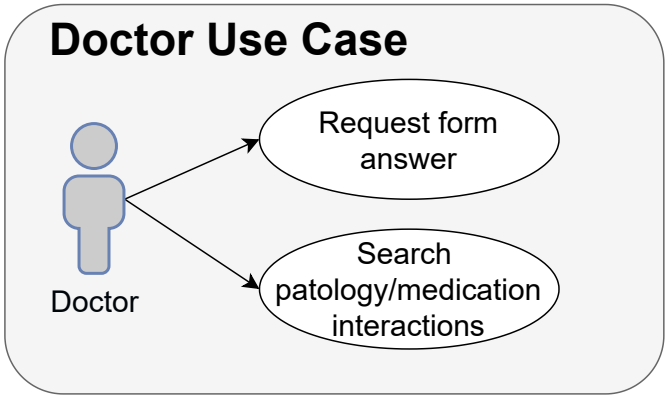

Figure 2.2: Doctor use case.

Furthermore the doctor user is able to search for pathology/medication interactions to

resolve any inquiries that might appear during the screening.

Finally the administrator user can update the form structure and flow, update the interaction information and manage the platform's users. The administrator use case is presented in Figure 2.3.

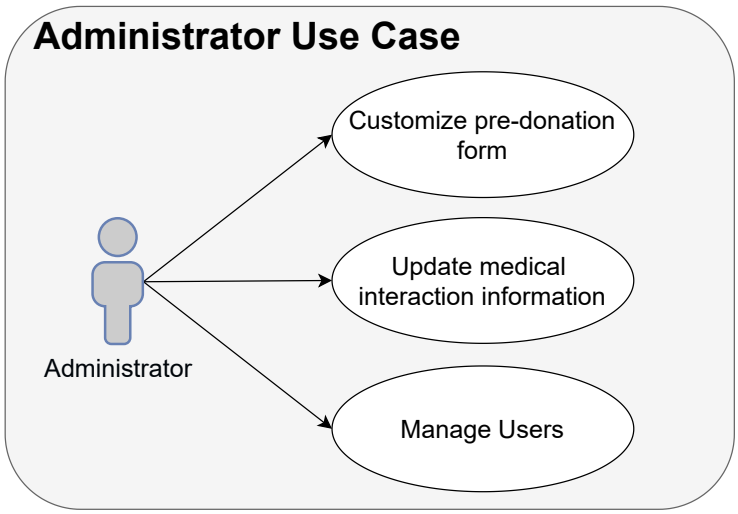

Figure 2.3: Administrator use case.

Chapter 3

Architecture

This chapter offers a comprehensive overview of the system's components and their interactions. It details the project's capabilities and presents the designed and developed architecture, entities, and implementation blueprint.

3.1 Overview

Figure 3.1 presents a diagram illustrating the main components of the system and their interactions. The system consists of a backend application (server-side) and a frontend application (client-side). The backend architecture consists of an authentication server, routes, services, repositories and a medication database client. The authentication server, serves as a placeholder for future integration with AMA's authentication services. The routes expose the backend's endpoints and handle incoming HTTP requests and call the appropriate service. The services manage data manipulation, validation, and interact with the medication database client and the repositories layer. The repository layer stores and retrieves information stored in a PostgreSQL database. The medication database client is responsible for requesting data from IPO's medication database, supplied by Infarmed.

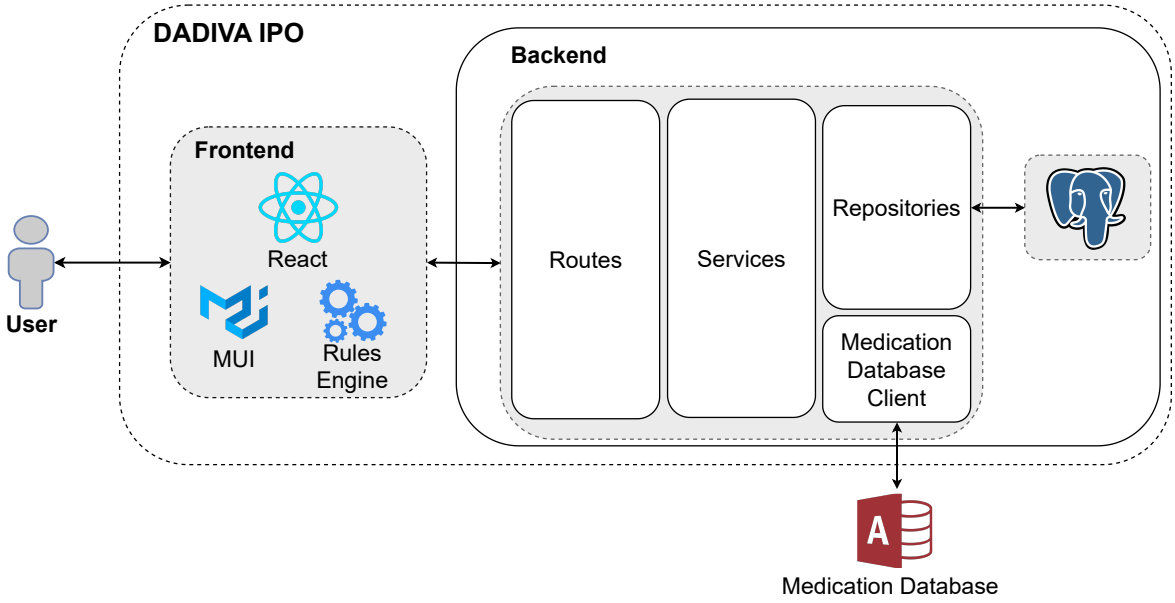

Figure 3.1: Application Architecture, gray squares represent containerized components of the solution.

3.2 Frontend Application

The frontend application is a web-based interface designed to facilitate seamless interaction between users and the backend system. It features a user-friendly and intuitive interface, catering to different types of users with specific functionalities:

- Donor Users: Can fill out the current donation form;
- Doctor Users: Can search for pathology and medication interactions with blood donation and request form answers for specific users;
- Administrator Users: Can customize the current form, update pathology and medication interaction information, update the inconsistencies and manage users.

The application is organized into multiple pages and components, each serving distinct purposes. It includes a service layer responsible for communicating with the backend application through the REST API.

3.2.1 JSON-Rules-Engine

As illustrated in figure 3.1 we used a rules engine in the frontend, more specifically the JSON-Rules-Engine, for the purpose of enforcing form rules. These rules can be abstracted into conditions and events, when a condition is met an event is triggered.

Conditions can be further abstracted into top level conditions or condition properties, as per the types in JSON-Rules-Engine library.

Top level conditions refer to logical operators, more specifically all, any and not, meanwhile condition properties refer to a boolean evaluation, i.e. does a given input equal a certain value.

Furthermore top level conditions can contain other top level conditions, meaning it is possible to describe complex rules using these types.

A graphical representation of these abstractions is presented on section 3.3.5 and section 3.3.6, which elaborate how these rules are stored and interact with the remaining system.

3.2.2 Mocks

During planning some mockups where created of the final result for some pages, the login page is presented in Figure 3.2, the form pages are presented in Figures 3.3, 3.4 and 3.5, the backoffice page is presented in Figure 3.6.

Figure 3.2: Login Page Mock.

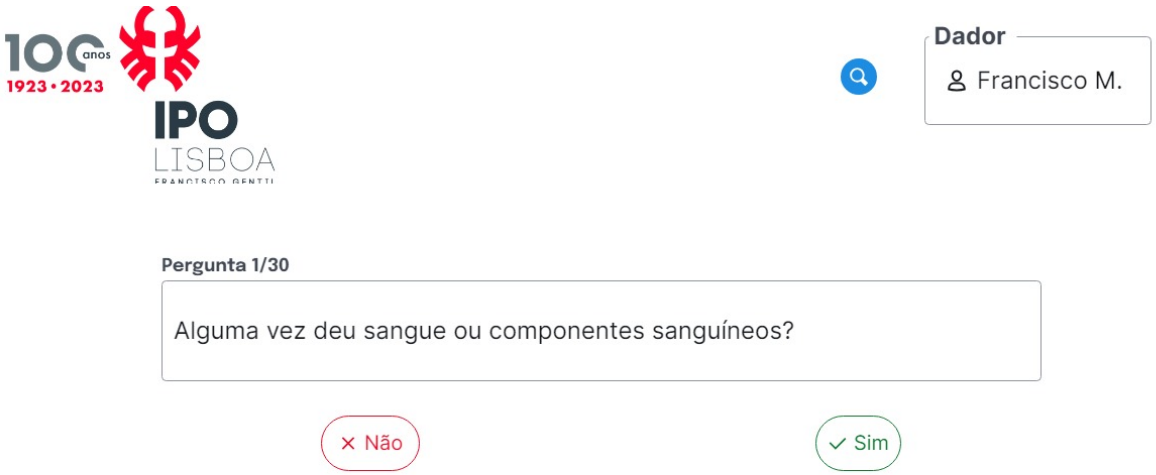

Figure 3.3: Form Page Mock.

Figure 3.4: Form Page Negative Answer Mock.

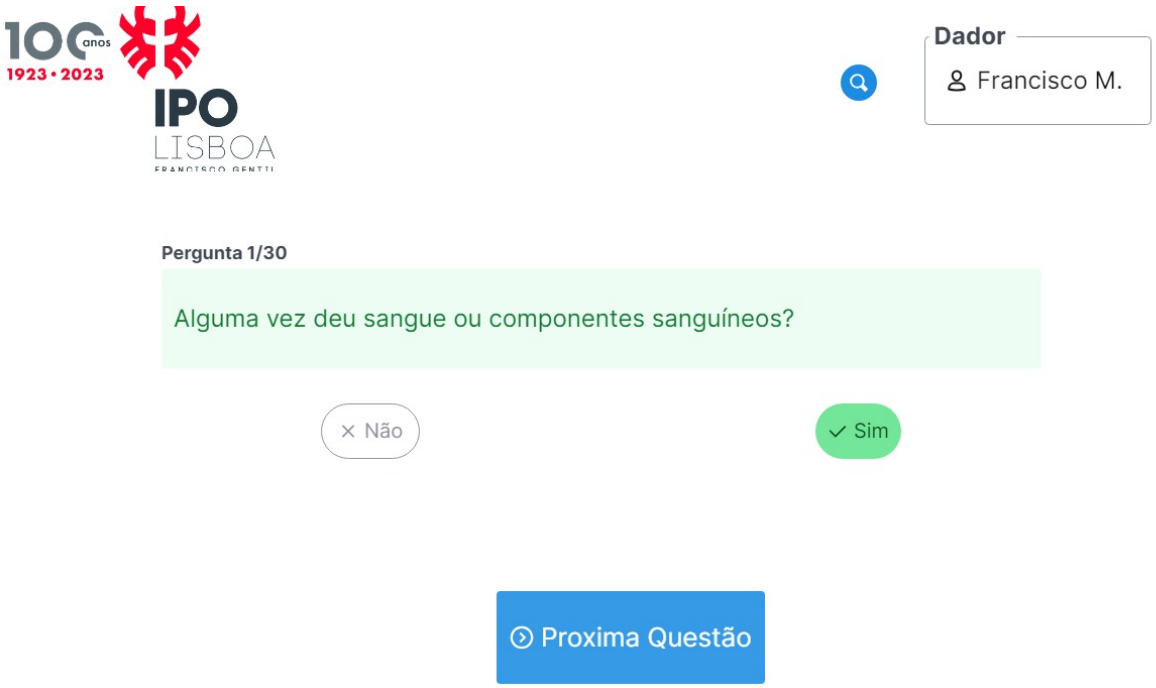

Figure 3.5: Form Page Positive Answer Mock.

Figure 3.6: Backoffice Page Mock.

3.3 Backend Application

The backend application can be abstracted into 4 layers:

- Routes: responsible for receiving the http request and calling the correct service;
- Services: contains the services that manage the business logic of the application;
- Medication Database Client: responsible for requesting information to IPO's medication database;
- Repositories: contains the repository layer of the application;

With each one of these layers being divided by groups of functions that deal with a certain data model, such:

- form;
- manual;
- medications;
- terms;
- submissions;
- review;
- users;

3.3.1 Service Layer Error Handling

When an error occurs in the backend, it is crucial not to expose any sensitive information, such as the exception thrown. Exposing raw exceptions can lead to security vulnerabilities and may leak implementation details that could be exploited by malicious actors. To handle errors safely and effectively, we will utilize a **Result** class.

The **Result** class can represent two states: **Success** or **Problem**.

- **Success**: This state indicates that the request was processed successfully. It encapsulates the result of the operation, ensuring that the expected outcome is communicated clearly to the client.
- **Problem**: This state is defined in accordance with RFC 7807[6]. The **Problem** class provides a standardized way to convey error information. It ensures that detailed error information is supplied without exposing sensitive data. By following the RFC 7807 specification, the Problem class includes the following fields:

– **'type'**: A URI reference that identifies the problem type. This URI is intended to provide human-readable documentation for the specific problem encountered.

– **'title'**: A short, human-readable summary of the problem type. This remains consistent across occurrences of the same problem type, making it easier for developers to recognize recurring issues.

– **'status'**: The HTTP status code generated by the origin server for this particular occurrence of the problem. This aligns with standard HTTP status codes, facilitating easy interpretation by both humans and machines.

– **'details'**: A human-readable explanation specific to this instance of the problem. It provides more context and helps in understanding the error without revealing sensitive information.

The **Problem** class's standardized format ensures that error information is conveyed consistently, which enhances both the readability for humans and the parsability for machines. This consistency is vital for effective debugging, logging, and automated error handling.

When possible the 'title' and 'details' fields should follow some guideline to ensure security, such as owasp's recommendation for authentication error responses [7], which states that "using any of the authentication mechanisms (login, password reset, or password recovery), an application must respond with a generic error message regardless of whether:The user ID or password was incorrect.The account does not exist.The account is locked or disabled."

An example of the flow for GET form resource request is presented in Figure 3.7

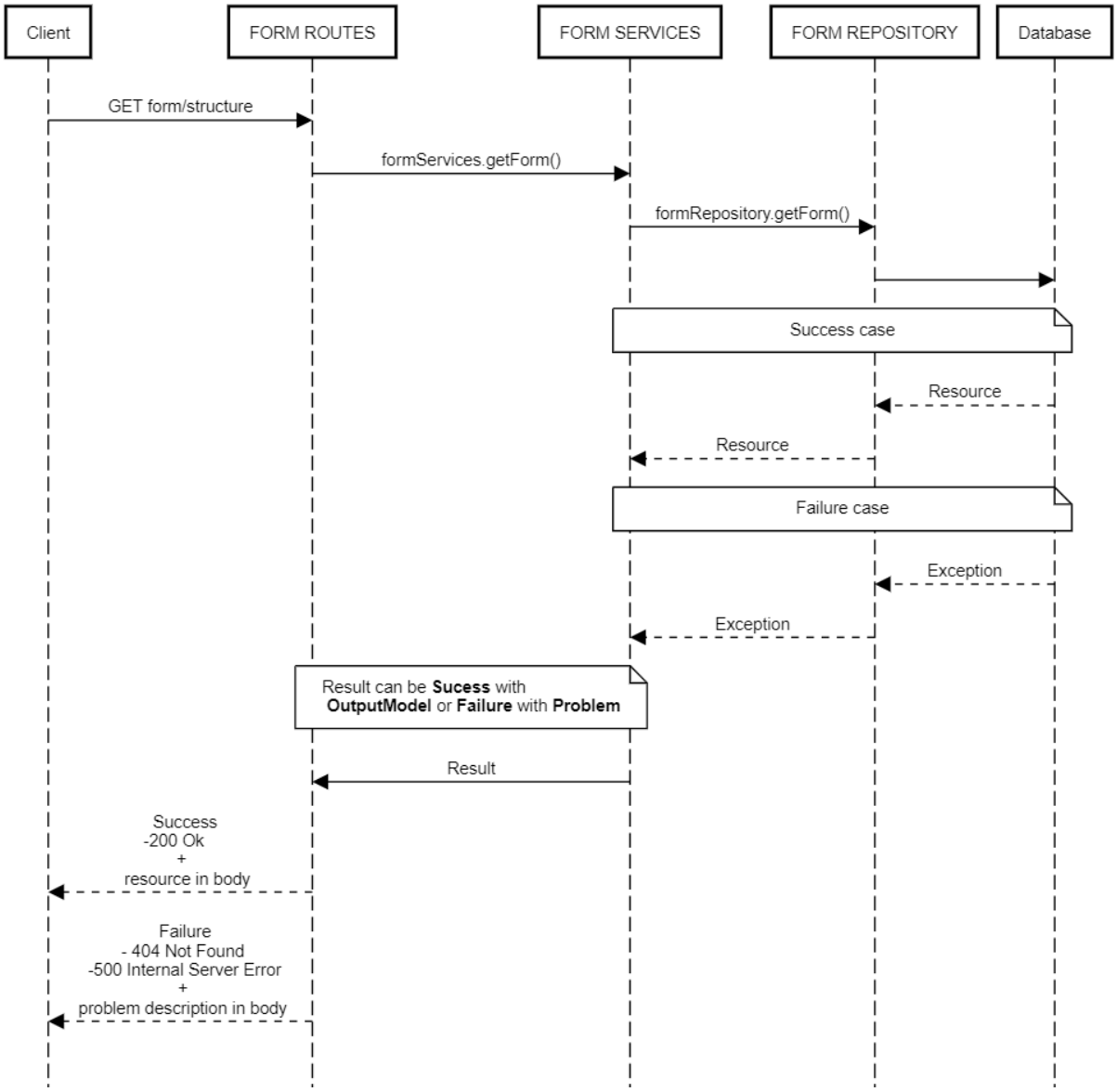

Figure 3.7: Get Form Sequence Diagram.

3.3.2 Data Model

In this section we'll elaborate on the various entities that compose our application and their interactions. The complete entity relationship diagram is present in Appendix A.

3.3.3 User

As outlined in section 2.1.4, our application divides users in roles, with each role being able to perform specific actions, as such, the **User** entity is a supertype, with roles being the overlapping subtypes of user, as a user can occupy multiple roles, as illustrated in Figure 3.8.

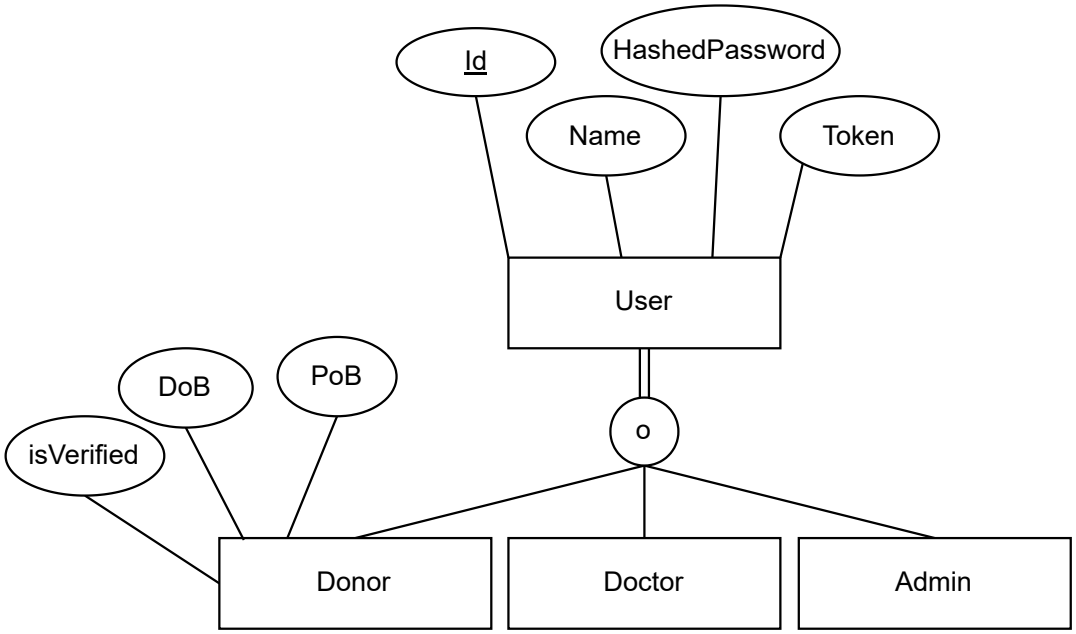

Figure 3.8: User Entity.

The **User** supertype contains the attributes that are common to all roles, which are as follows:

- Id:A unique identifier which can be a passport or civil identification number;
- Name: the user's full name;
- HashedPassword: The user's password, stored securely as a hash, more details in section 3.3.9;
- Token: An authentication token for the user.

The **Donor** subtype contains some specific attributes, which are pertinent to this type, such as:

- isVerified: boolean, indicates if donor supplied proof of identity;
- DoB: the donor's date of birth;
- PoB: the donor's place of birth;

User relationships

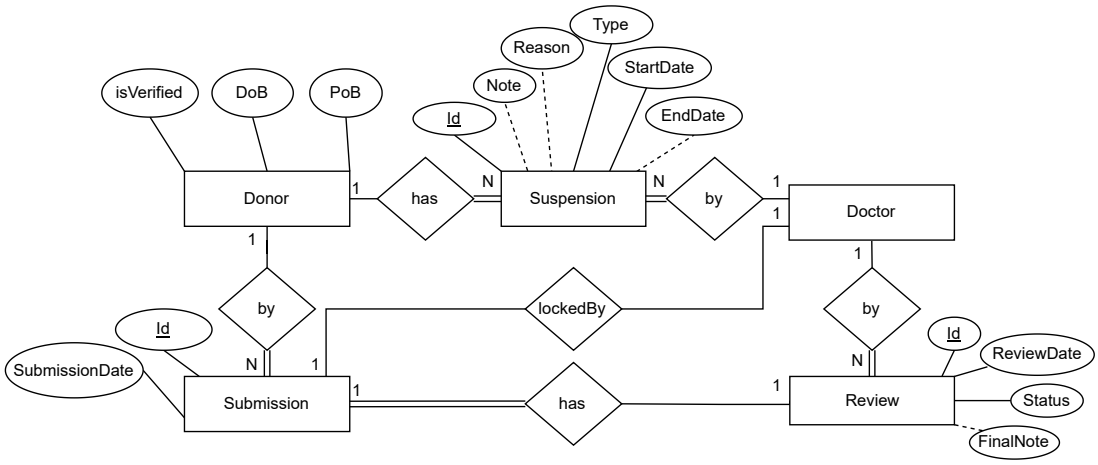

Figure 3.9: User interactions.

A **Donor** can be suspended, ie after a blood donation there's a 2 month waiting period until the next donation, this **Suspension** is created by a **Doctor**. Each **Donor** can have multiple **suspensions**, ie multiples waiting periods after donations, and each **Doctor** can issue multiple **suspensions**.

The attributes used to characterize a **Suspension** are as follows:

- **Id**:A unique identifier for the suspension;
- **Type**: Indicates whether the suspension is temporary or permanent;
- **StartDate**: The date when the suspension begins;
- **EndDate**: The date when the suspension ends, optional as permanent suspensions don't have an end date;
- **Reason**: An optional field to specify the reason for the suspension;
- **Note**: An optional note related to the suspension.

A **Donor** performs various pre-donation form **submissions**, which need to be reviewed by a **Doctor**, whom can **review** multiple **submissions**, with each **Submission** having a single **Review**. The **Submission** entity's relationships are elaborated on in section 3.3.7.

A **Submission** is characterized by the date in which it was performed, **SubmissionDate**, and a unique identifier.

A review is characterize by the following attribute:

- **Id**:A unique identifier for the Review;

- **ReviewDate**: The date in which the review was performed;
- **Status**: ;
- **FinalNote**: Optional, potential notes;

3.3.4 Admin

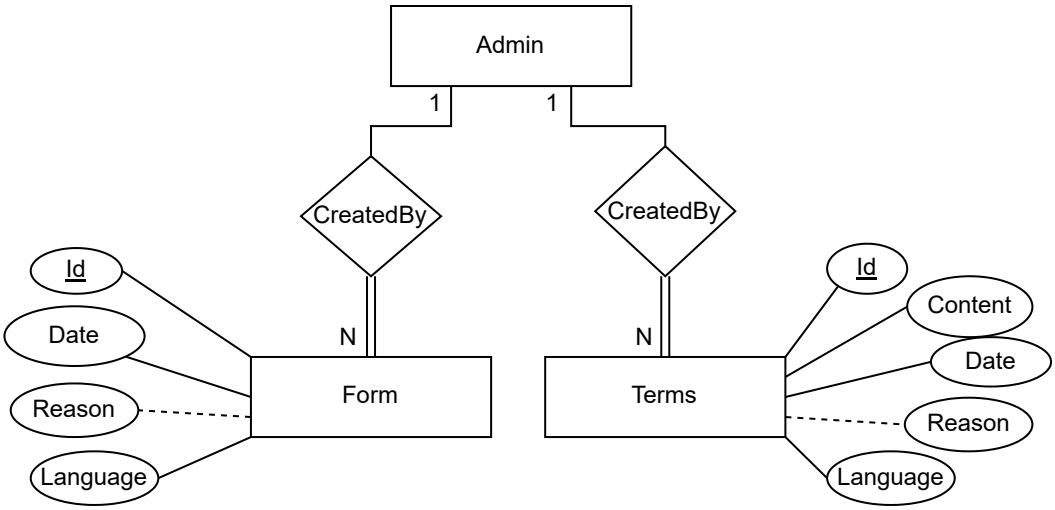

Figure 3.10: User interactions.

An **Admin** can create multiple pre-donation **forms** and multiple legal **terms** for the donation. The **Form** entity has more relationships, but to preserve this section's scope that information is omitted but is available in section 3.3.5.

The **Form** entity is characterized by the following attributes:

- **Id**:A unique identifier for the form;
- **CreatedAt**: The date in which the form was created;
- **Title**: The title of the form, for ease of identification;
- **Language**: The language of the form stored in ISO 639-3;
- **isActive**: Indicates if this form is the one being presented to the donors, only one form can be active at any time for a given language;

The **Terms** entity is characterized by the following attributes:

- **Id**:A unique identifier for the terms;
- **Content**: The actual terms;

- **CreatedAt**: The date in which the terms were created;
- **Title**: The title of the terms, for ease of identification;
- **Language**: The language of the terms;
- **isActive**: Indicates if these terms are being presented to the donors, only one of these entities can be active at any time;

3.3.5 Form relationships

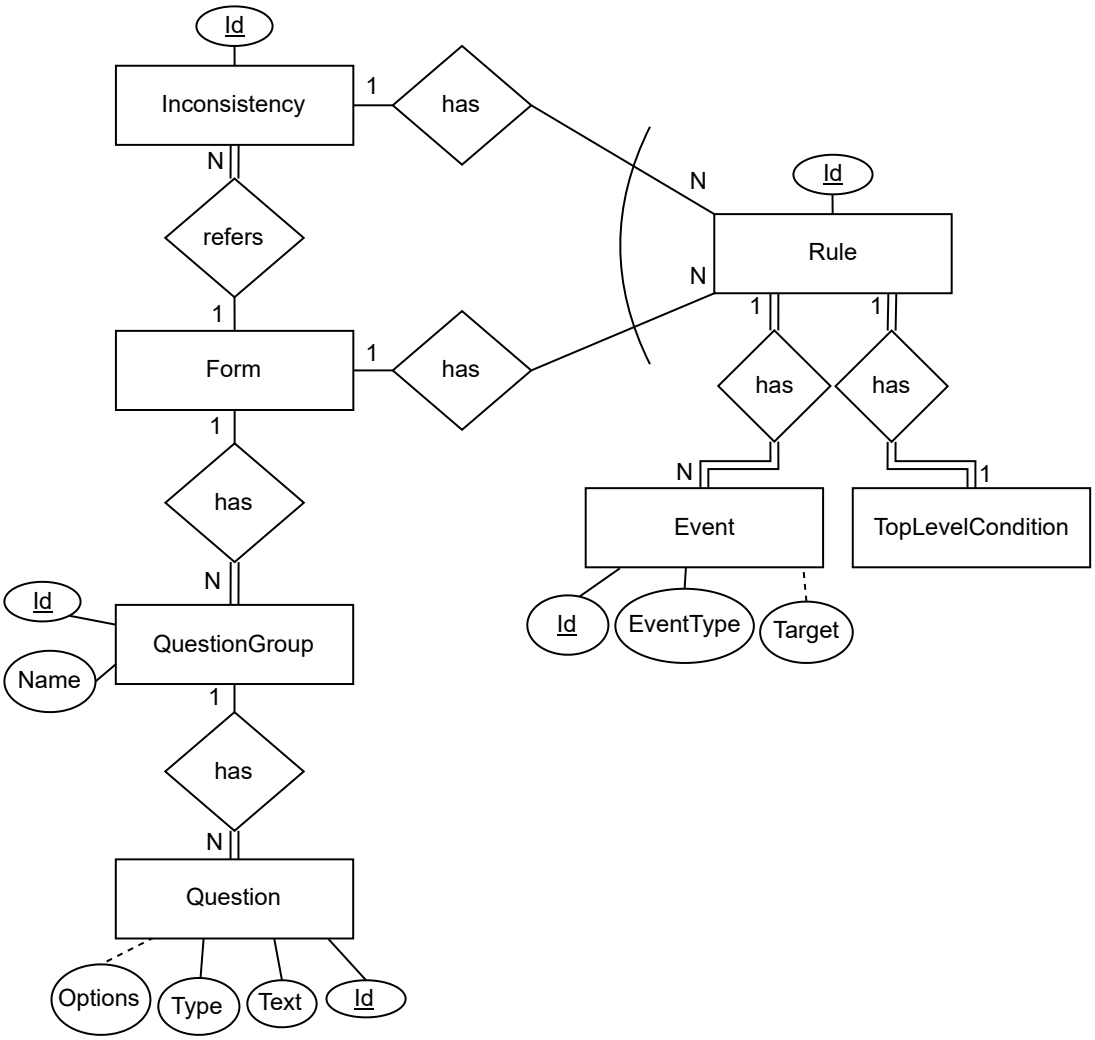

Figure 3.11: Form Entity.

A **Form** is composed of a group of **questions**, this group represents a theme, and a set of **rules**, as such it has a one to many relationship with these entities. For the sake of simplicity

a rule can be defined as a logical condition, the **TopLevelCondition** entity in Figure 3.11, that triggers an event when met,ie when a donor answers that he has traveled abroad a subsequent question appears asking to which country, a further explanation of the entities that make up these conditions is presented in section 3.3.6.

A **QuestionGroup** entity has the following attributes:

- **Id**: The group's unique identifier;
- **Name**: The theme of the group, ie travel, health, previous donations, etc;

Logically, a **QuestionGroup** as a one to many relationship with the **Question** entity. A **Question** entity is characterized by the following attributes:

- **Id**: The question's unique identifier;
- **Text**: The actual question;
- **Type**: The type of accepted answer, ie boolean, text, multiple values, etc;

An **Event** entity is characterized by the following attributes:

- **Id**: The event's unique identifier;
- **EventType**: The action the event performs, ie hide/show a question, allow navigation to next group, etc ;
- **Target**: Optional, the target of the action, ie the question to be hidden or displayed;

The **Inconsistency** entity represents a logical fallacy in sets of answers, eg a donor stating the they never traveled outside of Portugal but also stating that they've resided outside of Portugal. This entity is compromised of a single identifying attribute,Id, and has a one to many relationship with the **Rule** entity.

3.3.6 Conditions

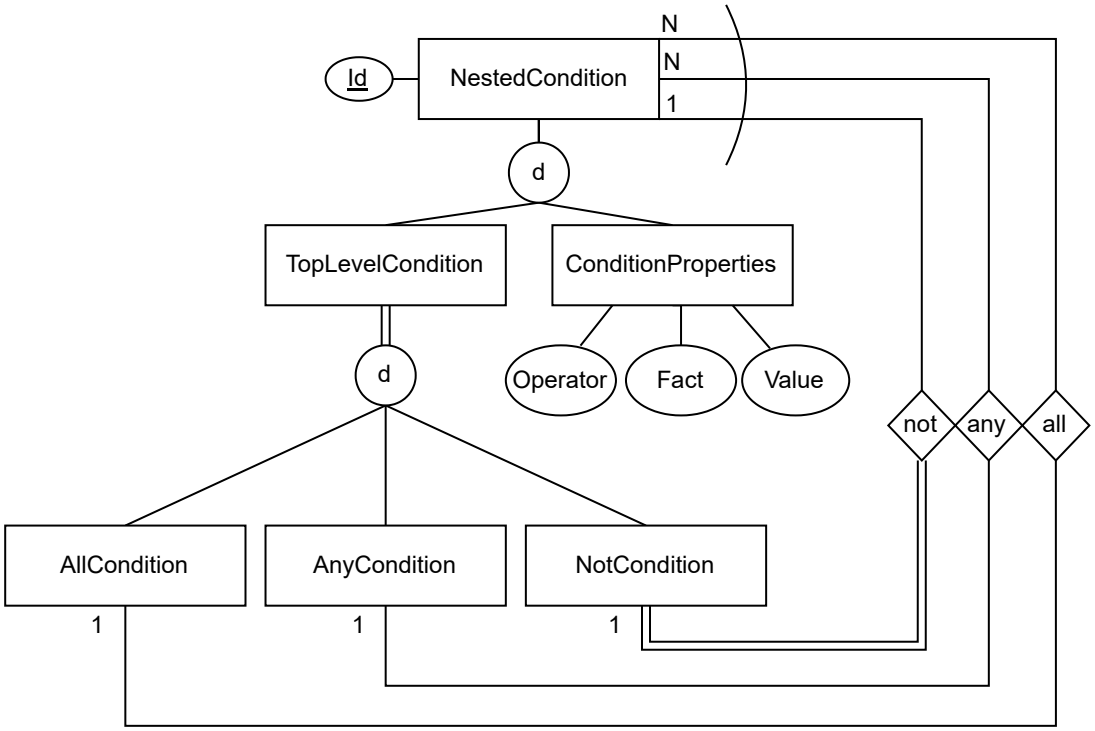

Figure 3.12: Condition Entity.

The entities and relationships mentioned in this section reflect the types belonging to the **JSON-Rules-Engine**. The **NestedCondition** entity is a supertype of the **TopLevelCondition** entity and the **ConditionProperties** entity. As the **TopLevelCondition** entity is a supertype of the **AllCondition**, **AnyCondition** a **NotCondition** entities, it can be seen as a representation of logical operators. As illustrated in Figure 3.12, the **AllCondition** and **AnyCondition** entities have a one to many relationship with the **NestedCondition**, while the **NotCondition** as a one to one relationship, this means the **all** and **any** conditions can have multiple conditions nested inside them while the **not** condition can have a single condition nested inside it, which allows for the creation of complex boolean expressions.

The **ConditionProperties** entity represents a logical evaluation and has the following attributes:

- **Operator**: The logical operator of the evaluation, ie equal, less than, greater than, etc;
- **Fact**: The id of the question being evaluated;
- **Value**: The expected value of the question being referenced.

3.3.7 Submission

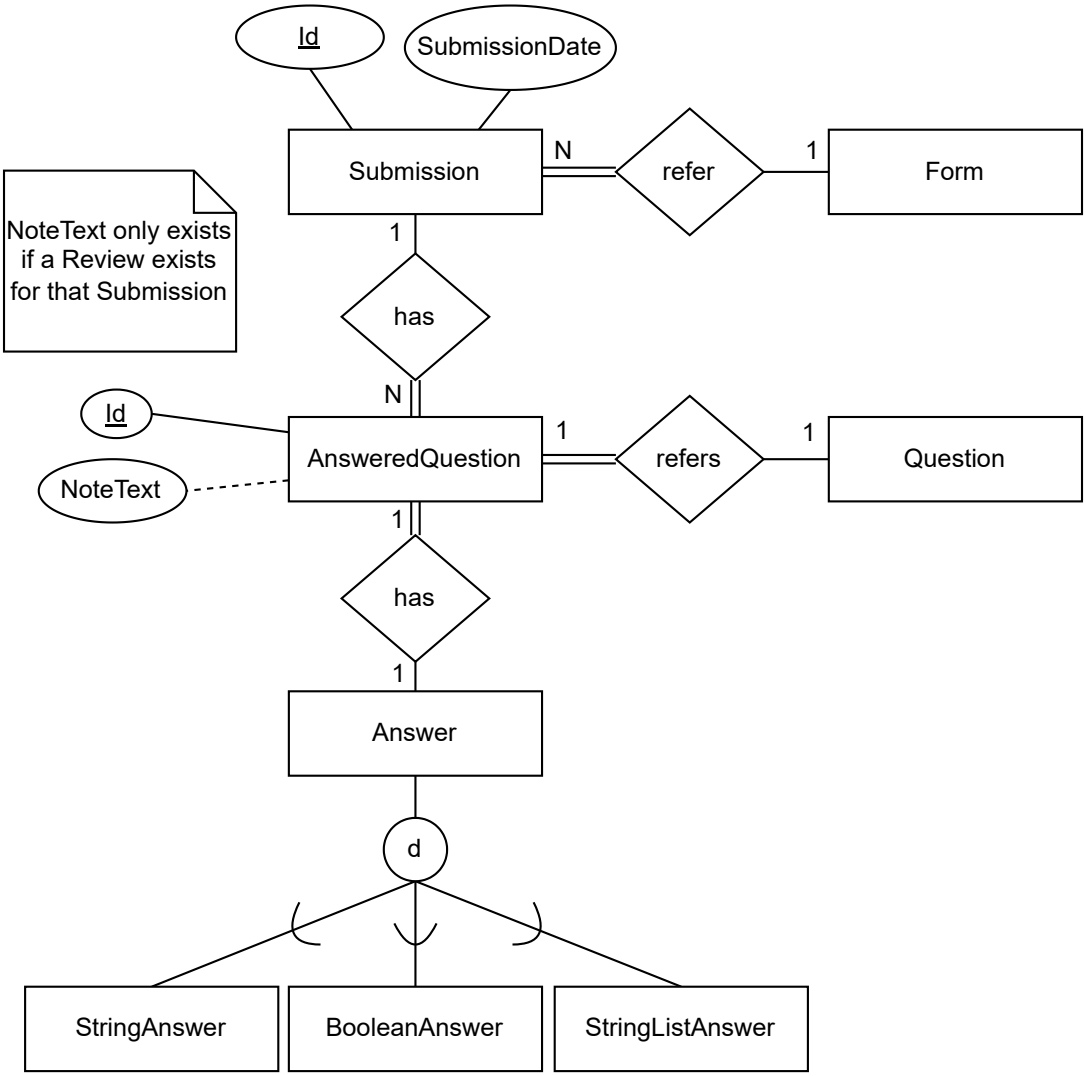

Figure 3.13: Submission Entity.

As illustrated in Figure 3.13, the **Submission** entity as a relationship with the **Form** entity, since a given submission pertains to a certain version of the form which changes overtime, and with the **AnsweredQuestion** entity,this entity has a **NoteText** attribute, which is optional, and represents a doctor note about the answer provided and a relationship with the **Question** entity, since every answer must refer to a question in the form. The **AnsweredQuestion** entity also has a relationship with the **Answer** entity which is a supertype representing the possible values for the form's answers.

3.3.8 Manual Information

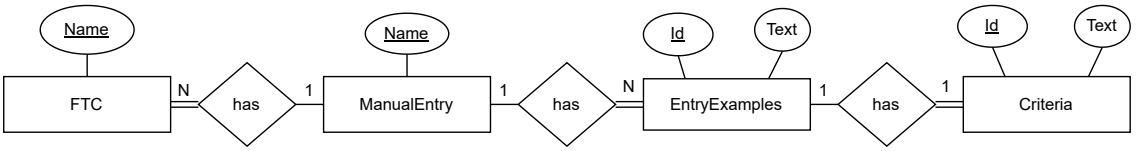

Figure 3.14: Manual Entity.

To allow doctors to search for medication interactions with blood donations and to perform the risk vector analysis we created the **FTC** entity which is the pharmaco-therapeutic classification of the medication. This entity has an identifying **Name** attribute, which is the name for this group of medications, e.g. analgesics and antipyretics, non steroidal anti inflammatories, etc.

The **ManualEntry** entity has an identifying **Name** attribute, which is the name used to group medications in the manual.

The **FTC** and **ManualEntry** entities have a many to one relationship as the names used in the manual might refer to more than one classification, and will possibly not have a match with any **FTC**.

The **EntryExamples** entity represents the examples presented in the manual for each entry, ie in the 2022 manual the analgesics entry contains two examples "Paracetamol, Ben U Ron, Tramadol..." and "Opioid Analgesics", so this entity has a many to one relationship with the **ManualEntry** entity and an attribute **Text**, with the examples.

Finally the **Criteria** entity which refers to whether a donor is able to donate blood or should be suspended, and whether that suspension is temporary or permanent if he's taken this medication. This entity as a one to one relationship with **EntryExamples** entity, as the evaluation of blood donation capabilities is dependent on these examples, since different medications within the same classification can lead to distinct outcomes.

3.3.9 Password Security

Two common password attacks are brute force attacks, and side-channel attacks.

Brute force attacks leverage the high computational power of Graphics Processing Units (GPUs) to paralelize password hashing tasks.

Side-channel attacks try get indirect information leaked during the execution of cryptographic algorithms, ie the time or power it takes for a system to hash a password.

Mitigation

Password hashing is a crucial security measure used to protect stored passwords. Instead of saving passwords in plaintext, which can be easily compromised, passwords are transformed

into a hashed format using a hashing algorithm.

The work factor is the number of iterations of the hashing algorithm that are performed for each password. The work factor is typically stored in the hash output. It makes calculating the hash more computationally expensive, which in turn reduces the speed and/or increases the cost for which an attacker can attempt to crack the password hash [4], which increases the brute-force attack further. Choosing a work factor requires a compromise between security and performance, since, if to much computing power is required to hash a password the system becomes targetable to denial of service attacks.

Hashing algorithms that employ constant time operations and parallelism whenever possible help mitigate the risk of side channel attacks, as these factors increases the difficulty to extract information form side-channels.

Argon2id

Argon2[5] is a state-of-the-art password hashing algorithm that won the Password Hashing Competition in 2015. It comes in three variants: Argon2d, Argon2i, and Argon2id. Argon2id is a hybrid version that combines the benefits of both Argon2d (which provides resistance against brute-force attacks) and Argon2i (which is designed for side-channel attack resistance), and will be the algorithm used to secure the passwords for our platform.

Argon2id takes in configurable parameter, such as:

- **Memory Cost**: The amount of memory (in kilobytes) used by the algorithm;
- **Time Cost** : The number of iterations the algorithm runs, which affects the computation time;
- **Parallelism** : The number of parallel threads used to process the hash.

Since these parameters are configurable it is possible to adjust them throughout the lifetime of an application, for example increase the time cost as the hosting hardware's capacity increases, and decrease it if concurrent accesses increase.

Chapter 4

Technologies

In this chapter, we introduce the key technologies that underpin the various components of the DADIVA IPO Platform. We will explain each technology's purpose and relevance within the context of our project. The discussion is organized into three main sections: backend technologies, frontend technologies, and version control tools. Our choices were significantly influenced by the experience and knowledge gained during our bachelor's degree in computer science:

- Web Application Development (Desenvolvimento de Aplicações Web)- Docker, React, Material-UI, Webpack, Spring;
- Systems Virtualization Techniques (Técnicas de Virtualização de Sistema) - Docker;
- Software Laboratory (Laboratório de Software) - Docker;
- Informatic Security (Segurança Informática) - RBAC;
- Git and Github were used during most of the course.

4.1 Backend Technologies

The backend technologies used in the DADIVA IPO platform share conceptual similarities with those we encountered in courses such as Web Application Development and Introduction to Web Programming. These technologies form the server-side components responsible for data processing, database management, and API integrations. In this section, we will provide a comprehensive overview of the key backend technologies employed in our project, highlighting their functionalities, benefits, and relevance to our system. We will also explore how these technologies collaborate to ensure the seamless operation and performance of the DADIVA IPO platform.

The programming language used for the backend is C#. This general-purpose, object-oriented language was selected for several reasons:

- Industry Relevance: C# is part of the technology stack at Cofidis, aligning our project with industry standards;
- Learning Opportunity: Using C# in this project provided us with valuable experience in a new language, preparing us for diverse development environments post-graduation;
- Robustness: C# is well-suited for building scalable and maintainable backend systems, offering strong type safety and extensive libraries.

While C# was our final choice, we considered other languages based on our coursework experience:

- Kotlin: Extensively used during our course, Kotlin is known for its modern syntax and interoperability with Java;
- Java: A staple language for backend development, Java shares many characteristics with C#, making it another viable option.

However, C# presented a unique and interesting challenge, offering a fresh perspective compared to the more familiar alternatives.

4.1.1 .NET

.NET is a comprehensive development framework created by Microsoft. It serves as the backbone for building a variety of applications, including web, mobile, desktop, gaming, and Internet of Things (IoT) applications.

.NET provides a built-in dependency injection container that is straightforward to use. This container is integrated into various application types, including ASP.NET Core, and is conceptually similar to Spring, as well as a Role Based Access Control(RBAC).

4.1.2 Docker

Docker is an open-source project which wraps and extends Linux containers technology to create a complete solution for the creation and distribution of containers. The Docker platform provides a vast number of commands to conveniently manipulate containers.

A container is an isolated, yet interactive, environment configured with all the dependencies necessary to execute an application. The use of containers brings advantages such as:

- Having little to no overhead compared to running an application natively, as it interacts directly with the host OS kernel and no layer exists between the application running and the OS;

- Providing high portability since the application runs in the environment provided by the container; bugs related to runtime environment configurations will almost certainly not occur;

- Running dozens of containers at the same time, thanks to their lightweight nature;

- Executing an application by downloading the container and running it, avoiding going through possible complex installations and setup.

To easily configure the virtual environment that the container hosts, Docker provides Docker images. Images are snapshots of all the necessary tools and files to execute an application. Containers can be started from images, the same way virtual machines run snapshots. To effortlessly distribute images, Docker provides registries. These are public or private stores where users may upload or download images. Docker provides a cloud-based registry service called DockerHub.

In addition to the Docker platform, we use Docker Compose to orchestrate the containers. Docker Compose is a tool for defining and running multi-container Docker applications. With Compose, we can define a multi-container application in a single file, then spin it up in a single command which does everything that needs to be done to get it running. Compose is especially useful in development environments, testing environments.

We use Docker as an alternative for software containerization because it is the most well known, actively developed and supported in the area. Many frameworks already support it or are starting to support it.

4.2 Frontend Technologies

4.2.1 React

React is a JavaScript library for building user interfaces. It is maintained by Facebook and a community of individual developers and companies. React can be used as a base in the development of single-page or mobile applications.

React is a component-based library, which means that the application is built by assembling components. Each component is a small piece of code that can be reused in different parts of the application. React is also declarative, which means that it is possible to describe the user interface without specifying how the user interface should be updated.

In DADIVA IPO, we use React to create the user interface. We also use React Router to manage the routing of the application. React Router is a collection of navigational components that compose declaratively with your application.

4.2.2 JSON-Rules-Engine

JSON-Rules-Engine is a library that enables the evaluation of business rules based on data inputs, providing a way to separate business logic from the core application code. Rules are defined in JSON format, making them easy to read, write, and maintain. The engine evaluates these rules against provided facts (data inputs) and triggers actions based on the results.

4.2.3 Material-UI

In addition to React, we also use Material-UI to create the user interface. Material-UI is a React component library that implements Google's Material Design, which is a design language that combines the classic principles of successful design along with innovation and technology. Material-UI provides a set of components that can be used to create a user interface that follows the Material Design guidelines, such as buttons, cards, and tables. This makes it easier to create a consistent user interface.

4.2.4 Webpack

Webpack is a module bundler. It takes modules with dependencies and generates static assets representing those modules. Webpack is used to bundle JavaScript files for usage in a browser. It also provides a set of plugins that can be used to optimize the application, such as minification and code splitting.

In DADIVA IPO, we use Webpack to bundle the JavaScript files of the application, optimizing them for production. The technology also provides a development server, which is used to serve the application during development. We also use ts-loader to compile TypeScript files into JavaScript.

4.3 DevOps Technologies

4.3.1 Git and GitHub

Git and GitHub are widely used version control tools that play a critical role in modern DevOps practices. Git is a distributed version control system that allows teams to efficiently manage changes to source code, track them over time and streamlines developer collaboration. GitHub, on the other hand, is a web-based hosting service for Git repositories that provides additional collaboration and project management features. Both of these technologies where extensively used trough our course.

4.3.2 Swagger

Swagger [8] is an open-source framework that simplifies the design, documentation, and consumption of RESTful web services. It is widely used in the software development industry

to create, visualize, and interact with API specifications. Swagger's comprehensive suite of tools and features enhances the development workflow, making it easier for developers to build and maintain APIs.

4.3.3 diagrams.net

In the development of this report, the majority of the diagrams were created using diagrams.net [9], also known as draw.io. Diagrams.net is a highly popular online diagramming tool that offers users the ability to design a wide variety of diagrams with ease and precision. Some of diagrams.net's key features are:

- **User-Friendly Interface**: Diagrams.net boasts an intuitive and user-friendly interface, making it accessible to both beginners and experienced users. The drag-and-drop functionality allows for quick creation and editing of diagrams.
- **Wide Range of Diagram Types**: The platform supports a diverse array of diagram types, including flowcharts, organizational charts, mind maps, network diagrams, UML diagrams, ER diagrams, and more. This versatility makes it a one-stop solution for most diagramming needs.
- **Customization Options**: Users can customize diagrams extensively with a variety of shapes, connectors, and styles. The tool offers a rich library of predefined shapes and templates that can be tailored to specific requirements.
- **Collaboration Capabilities**: Diagrams.net supports real-time collaboration, allowing multiple users to work on the same diagram simultaneously. This feature is particularly useful for team projects and collaborative work environments.
- **Integration and Compatibility**: The tool integrates seamlessly with popular cloud storage services like Google Drive, OneDrive, Dropbox, and GitHub. This ensures that diagrams can be easily saved, shared, and accessed from anywhere.

4.3.4 visily.ai

Visily.ai [10] is a powerful design tool that leverages artificial intelligence to facilitate the creation of wireframes and prototypes for web and mobile applications. It is designed to help both designers and non-designers quickly produce high-quality visual representations of their ideas, and was used to produce the mockups show in section 3.2.

Chapter 5

Implementation

In this chapter, we will discuss DADIVA IPO platform's components in more detail, how they interact, the development approach, project structure and implementation details.

An overview of the projects implementation is presented in Figure 5.1

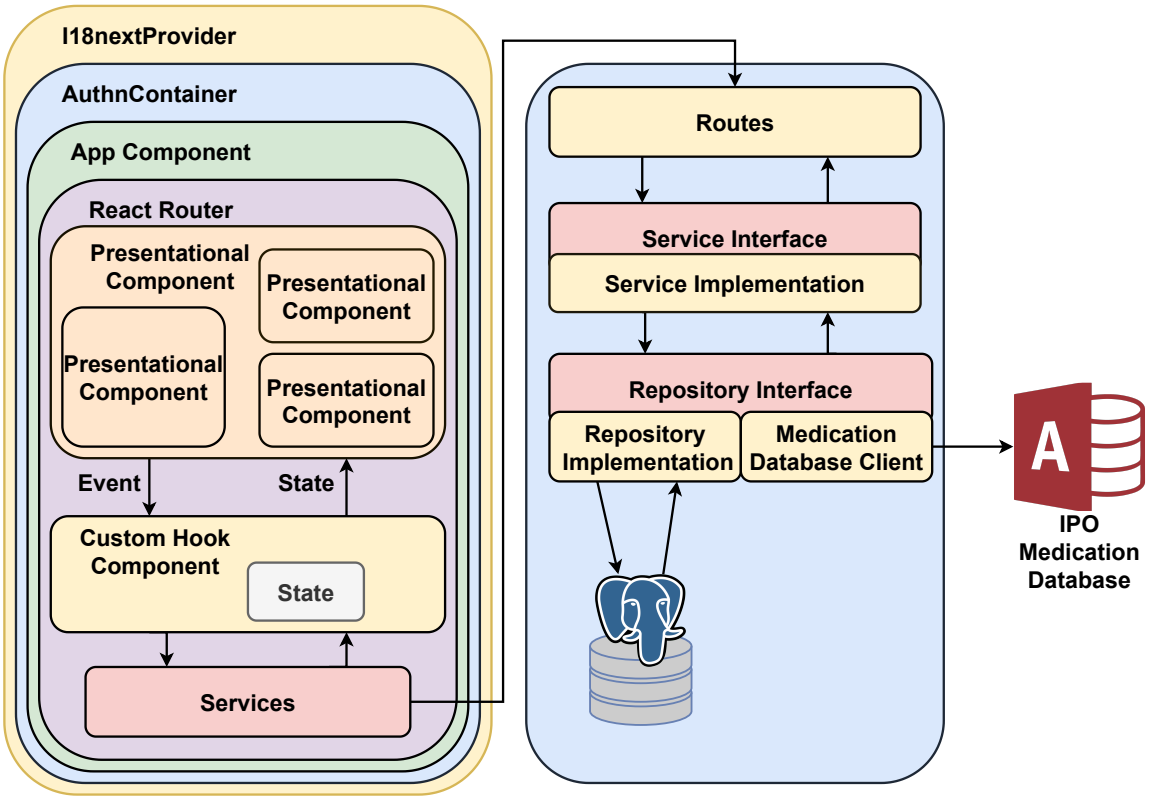

Figure 5.1: Block Diagram of our solution.

5.1 Backend

In this section we will describe the backend.

5.1.1 Structure

The structure for the backend application is as follows:

- Program.cs: the entry point of the application;
- domain: contains all the domain classes;
- repositories: contains the backend repositories that communicate with the postgreSQL database;
  - sql: the methods to perform the requests;
  - entities: contains the entities outlined in section 3.3.2.
- services: contains all the services that, validate and manipulate data, that is received or sent to the routes and repository layer;
- routes: contains all the routes of the API which call the adequate service.
- utils: contains auxiliary classes and methods.

5.1.2 Program.cs

As mentioned before since we're creating a Minimal API, similar to apis created with Express.JS, and, as such, server creation is streamlined and the application's functionality is based on middlewares and routing.

The .NET framework makes use of a dependency injection container, aka the service container. As with the Spring Framework, dependencies can have various lifetimes, which in the .NET framework are as follows:

- Transient: the dependency is created when needed and disposed thereafter;
- Scoped: the dependency is created and maintained in a per request basis;
- Singleton: once the dependency is created it's maintained throughout the application's lifetime.

Beyond this, the framework also makes use of the builder pattern, meaning to build a web application we first instantiate a builder, i.e. a class that "knows" how to build a web application, and then supply the needed middlewares to build it, with the desired lifetime. The aforementioned middlewares, which refer to the services and repositories, are registered as services in the container. In our application these services were registered with the Scoped scope, as follows:

```
    builder.Services.AddScoped<IUsersService, UsersService>();
    builder.Services.AddScoped<IFormService, FormService>();
    builder.Services.AddScoped<ITermsService, TermsService>();
    builder.Services.AddScoped<IMedicationsService, MedicationsService>();
    builder.Services.AddScoped<IManualService, ManualService>();


    builder.Services.AddScoped<IRepository, Repository>();
```

Listing 5.1: Adding middlewares using dependency injection and inversion of control

Using this registration method, for example, when a dependency of type IUsersService is needed, the service container creates a UsersService object to fufill that dependency, hence the inversion of control.

To allow for cross-origin resource sharing, i.e. to allow the frontend client to access the resources in the backend, we also had to create CORS policy, as follows:

```
builder.Services.AddCors(options =>
{
    options.AddPolicy("MyCorsPolicy",
    policy =>
    {
        policy.WithOrigins("http://localhost:8000")
        .AllowAnyHeader()
        .AllowAnyMethod()
        .AllowCredentials();
    });
});
```

Listing 5.2: Configuring CORS Policy in ASP.NET Core: Allowing Specific Origin with Full Access Control.

Notice that request originating from port 8000 of the localhost ip can have any type of header, any HTTP method and can include credentials, such as cookies.

Authentication JWT

Our platform uses JSON Web Tokens, **JWT** [11], to represent user claims. These token's issuer, audience and key, refered to as jwtIssuer, jwtAudience and jwtKey in listing 5.3 below, are stored in the appsettings.json file.

```
builder.Services.AddAuthentication(x =>
{
    x.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;
    x.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;
}).AddJwtBearer(options =>
{
    options.Events = new JwtBearerEvents
    {
        ...
        OnChallenge = context =>
        {
            context.HandleResponse();

            context.Response.ContentType = "application/problem+json";
            context.Response.StatusCode = StatusCodes.Status401Unauthorized;
            var problemDetails = new
            {
                type = "https://localhost:8000/errors/unauthorized",
                title = "Unauthorized",
                detail = "You are not authorized to access this resource.
                    Please provide valid credentials.",
                status = StatusCodes.Status401Unauthorized
            };
            var problemJson = JsonSerializer.Serialize(problemDetails);
            return context.Response.WriteAsync(problemJson);
        }
    };
    options.SaveToken = true;
    options.TokenValidationParameters = new TokenValidationParameters
    {
        ValidateIssuer = true,
        ValidateAudience = true,
        ValidateLifetime = true,
        ValidateIssuerSigningKey = true,
        ValidIssuer = jwtIssuer,
        ValidAudience = jwtAudience,
        IssuerSigningKey = new
            SymmetricSecurityKey(Encoding.UTF8.GetBytes(jwtKey))
    };
});
```

Listing 5.3: Custom JWT Authentication Middleware in ASP.NET Core: Handling Unauthorized Access with Detailed Problem Responses.

Notice that in case the **JWT** isn't valid the default behavior of responding with a 401 status code and an empty body is suppressed by "context.HandleResponse()" and instead a Problem response is sent with more details.

5.1.3 Repositories

The Repository layer acts as a vital intermediary for data access within the application. It ensures the database and service layer remain independent from one another. This separation allows the service layer to access data without being tightly coupled to the database, facilitating easy database switching by merely replacing this module.

By defining a well-known interface contract, we can abstract the implementation details.

Utilizing a transaction manager in the service layer enables our solution to handle multiple concurrent accesses across various resources and provides rollback capabilities for effective error handling.

Form Repository

The form repository is responsible for the Form, Submission, Review, Inconsistency entities.

And this repository will have the following methods:

```
public interface IFormRepository
{
    public Task<Form> GetForm();

    public Task<Form?> GetFormWithVersion(int version);

    public Task<Form> EditForm(Form form);

    public Task<bool> SubmitForm(Submission submission);

    public Task<List<Submission>?> GetPendingSubmissions();

    public Task<Submission> GetSubmission(int nic);

    public Task<Submission?> GetSubmissionById(int id);

    public Task<Submission?> GetLatestPendingSubmissionByUser(int userNic);

    public Task<(List<SubmissionHistoryDto>? Submissions, bool
        HasMoreSubmissions)> GetSubmissionHistoryByNic(int nic, int limit,
        int skip);

    public Task<Inconsistencies> GetInconsistencies();

    public Task<bool> LockSubmission(int submissionId, int doctorId);

    public Task<bool> UnlockSubmission(int submissionId, int doctorId);

    public Task<List<SubmissionLock>> GetExpiredLocks(TimeSpan timeout);

    public Task<bool> SubmissionExists(int id);

    public Task<Review> AddReview(Review review);

    public Task<bool> AddNote(Note note);

    public Task<bool> EditInconsistencies(Inconsistencies inconsistencies);
}
```

User Repository

The user repository will be responsible for the User, UserAccountStatus and UserSuspension entities

And this repository will have the following methods:

```
public interface IUsersRepository
{
    public Task<bool> AddUser(User user);

    public Task<List<User>?> GetUsers();

    public Task<User?> GetUserByNic(int nic);

    public Task<Boolean> DeleteUser(int nic);

    public Task<UserAccountStatus?> GetUserAccountStatus(int userNic);

    public Task<Boolean> UpdateUserAccountStatus(UserAccountStatus
        userAccountStatus);

    public Task<bool> AddSuspension(UserSuspension suspension);
    public Task<bool> UpdateSuspension(UserSuspension suspension);
    public Task<UserSuspension?> GetSuspension(int userNic);
    public Task<bool> DeleteSuspension(int userNic);

}
```

5.1.4 Services

Each service is responsible for managing a certain group of requests, i.e. the logic to fulfill a request to the /users endpoint will be in the user services. Each service as a dependency on their corresponding repository, which is handled by the service container.

The methods within each service will reflect the possible actions outlined in Chapter 2's section on uses cases.

User Service

```
public interface IUsersService
{
    public Task<Result<Token, Problem>> CreateToken(int nic, string
        password);

    public Task<Result<UserExternalInfo, Problem>> CreateUser(int nic,
        string name, string password, Role role);

    public Task<Result<List<UserExternalInfo>, Problem>> GetUsers(string
        token);

    public Task<Result<Boolean, Problem>> DeleteUser(int nic);

    public Task<Result<UserAccountStatus?, Problem>>
        GetUserAccountStatus(int userNic);

    public Task<Result<Boolean, Problem>>
        UpdateUserAccountStatus(UserAccountStatus userAccountStatus);

    public Task<Result<UserWithNameExternalInfo?, Problem>>
        CheckNicExistence(int nic);

    public Task<Result<bool, Problem>> AddSuspension(UserSuspensionRequest
        suspension);

    public Task<Result<bool, Problem>> UpdateSuspension(UserSuspension
        suspension);

    public Task<Result<UserSuspension?, Problem>> GetSuspension(int userNic);

    public Task<Result<bool, Problem>> DeleteSuspension(int userNic);
}
```

Terms Service

```
public interface ITermsService
{
    public Task<Result<List<Terms>, Problem>> GetTerms();
    public Task<Result<Terms, Problem>> GetActiveTerms();
    public Task<Result<bool, Problem>> SubmitTerms(Terms terms);

    public Task<Result<bool, Problem>> UpdateTerms(int termId, int
        updatedBy, string newContent);

    public Task<Result<List<TermsChangeLog>?, Problem>>
        GetTermsChangeLog(int termId);
}
```

Form Service

```
public interface IFormService
{
    public Task<Result<GetFormOutputModel, Problem>> GetForm();

    public Task<Result<GetFormWithVersionOutputModel, Problem>>
        GetFormWithVersion(int version);

    public Task<Result<Form, Problem>> EditForm(List<QuestionGroupModel>
        groups, List<RuleModel> rules, User user);

    public Task<Result<SubmitFormOutputModel, Problem>>
        SubmitForm(Dictionary<string,IAnswer> answers, int nic, int
        formVersion);

    public Task<Result<Submission, Problem>> GetSubmission(int id);

    public Task<Result<bool, Problem>> LockSubmission(int submissionId, int
        doctorId);

    public Task<Result<bool, Problem>> UnlockSubmission(int submissionId,
        int doctorId);

    public Task UnlockExpiredSubmissions(TimeSpan lockTimeout);

    public Task<Result<Review, Problem>> ReviewForm(int submissionId, int
        doctorNic, string status, string? finalNote, List<NoteModel>?
        noteModels = null);

    public Task<Result<List<Submission>, Problem>> GetPendingSubmissions();

    public Task<Result<Inconsistencies, Problem>> GetInconsistencies();

    public Task<Result<Submission?, Problem>>
        GetPendingSubmissionsByUserNic(int userNic);

    public Task<Result<SubmissionHistoryOutputModel, Problem>>
        GetSubmissionHistoryByNic(int nic, int limit, int skip);

    public Task<Result<bool, Problem>> EditInconsistencies(Inconsistencies
        inconsistencies);
}
```

Manual Service

```
public interface IManualService
{
    public Task<Result<List<ManualInformation>, Problem>>
        GetManualInformation(string productName);
}
```

Medication Service

```
public interface IMedicationsService
{
    Task<Result<List<string>, Problem>> SearchMedications(string query);
}
```

5.1.5 Routes

User Routes

The available endpoints, HTTP method and corresponding operation for all the user routes are available in Table 5.1.

Endpoint	HTTP Method	Description
/users	POST	Creates a new user
/users	GET	Retrieves all users
/users/{nic}	GET	Checks the existence of a user with the specified NIC
/users/{nic}	DELETE	Deletes the user with the specified NIC
/users/login	POST	Creates a new access token
/users/status/{nic}	GET	Retrieves the status of the user account with the specified NIC
/update-status	POST	Updates the status of a user account
/users/suspension	POST	Adds a new suspension
/users/suspension/update	POST	Updates an existing suspension
/users/suspension/{nic}	GET	Retrieves the suspension details for the specified NIC
/users/suspension/{nic}	DELETE	Deletes the suspension for the specified NIC

Table 5.1: API endpoints related to the user

Form Routes

The available endpoints, HTTP method and corresponding operation for all the form routes are available in Table 5.2.

Endpoint	HTTP Method	Description
/forms/structure	GET	Retrieves the form structure
/forms/structure	PUT	Edits the form structure
/forms/structure/{version:int}	GET	Retrieves the form structure for the specified version
/forms/submissions	GET	Retrieves pending submissions
/forms/submissions/{nic:int}	POST	Submits a form
/forms/submissions/{nic:int}	GET	Retrieves a pending submission for the specified NIC
/forms/submissions/history/{nic:int}	GET	Retrieves submission history for the specified NIC
/forms/submissions/{submissionId:int}/lock	POST	Locks a submission
/forms/submissions/{submissionId:int}/unlock	POST	Unlocks a submission
/forms/inconsistencies	GET	Retrieves inconsistencies
/forms/inconsistencies	PUT	Edits inconsistencies
/forms/review/{submissionId:int}	POST	Reviews a form submission
/forms/notifications	GET	Sets up server-sent event notifications

Table 5.2: API endpoints related to the form

Terms Routes

Endpoint	HTTP Method	Description
/terms	GET	Retrieves all the terms
/terms/active	GET	Retrieves the active terms
/terms	POST	Submits terms
/terms/{termsId:int}	PUT	Updates terms with the specified termsId
/terms/change-log/{termsId:int}	GET	Gets the change-logs for the terms with specified termsId

Table 5.3: API endpoints related to the form

Medication and Manual Routes

Endpoint	HTTP Method	Description
/medications/search	GET	Retrieves medication list according to a query s
/manual/{product:string}	GET	Retrieves the blood donation information relevant to the s

Table 5.4: API endpoints related to the form

5.2 Frontend

The frontend application is tasked with data visualization and user interaction. The objective is for the application to provide an intuitive interface for the various users to be able to fulfill the set use cases outlined in section 2.1.4.

The frontend is implemented in Typescript and React, and makes use of JSON-Rules-Engine to enforce the form's rules, for further information about these technologies refer to 4.2.

5.2.1 Structure

The frontend structure is as follows:

- src: contains the source code of the application;
- public: contains the static files of the application;
- package.json: package.json;
- tsconfig.json: contains the TypeScript configuration;
- routes: contains all the routes of the API which call the adequate service.
- webpack.config.js: contains the Webpack configuration.

The src folder is then subdivided into multiple folders/files, each being responsible for a different functionality of the application:

- components: contains the components of the pages;
- domain: contains the domain objects;
- pages: contains the application's pages, each containing various components;
- services: contains the services that communicate with the backend application;
- session: contains the code needed to maintain a user session.
- utils: contains general utility functions.

It should be noted that, folders within the components folder may contain further utils, containing utility functions that are specially pertinent for that component and shouldn't necessarily be in the general utils.

Furthermore, besides the aforementioned folders, the src folder also contains an index.tsx and App.tsx files. The index.tsx file is the entry point for the application meanwhile the App.tsx is the main component of the React application.

5.3 Services

The frontend services are responsible for communicating with the backend application and, as such, each frontend service as a backend counterpart.

To facilitate this communication we used the Fetch API [12], which enables asynchronous resource requests by returning a promise that resolves to a response for that request.

To expedite, and reduce the code for, the api resource requests we created a fetchAPI function that accepts the request parameters and return a promise that will resolve to the requests response, this function can also handle errors if the response's status code isn't in the 200 family.

To further abstract the api calls, the fetchAPI function was encapsulated within functions that represent specific HTTP methods, such as GET, POST, PUT and DELETE.

5.4 Components

A simplified view of the component tree is presented in Figure 5.2, in this chapter we'll mainly focus on the I18nextProvider which manages the language preferences of the user, AuthnContainer, which manages the session information, the Form, Backoffice, EditFormPage, EditTermsPage and Editor which are presentational components and the useNewForm, useEditFormPage, useEditTerms which are custom hooks.

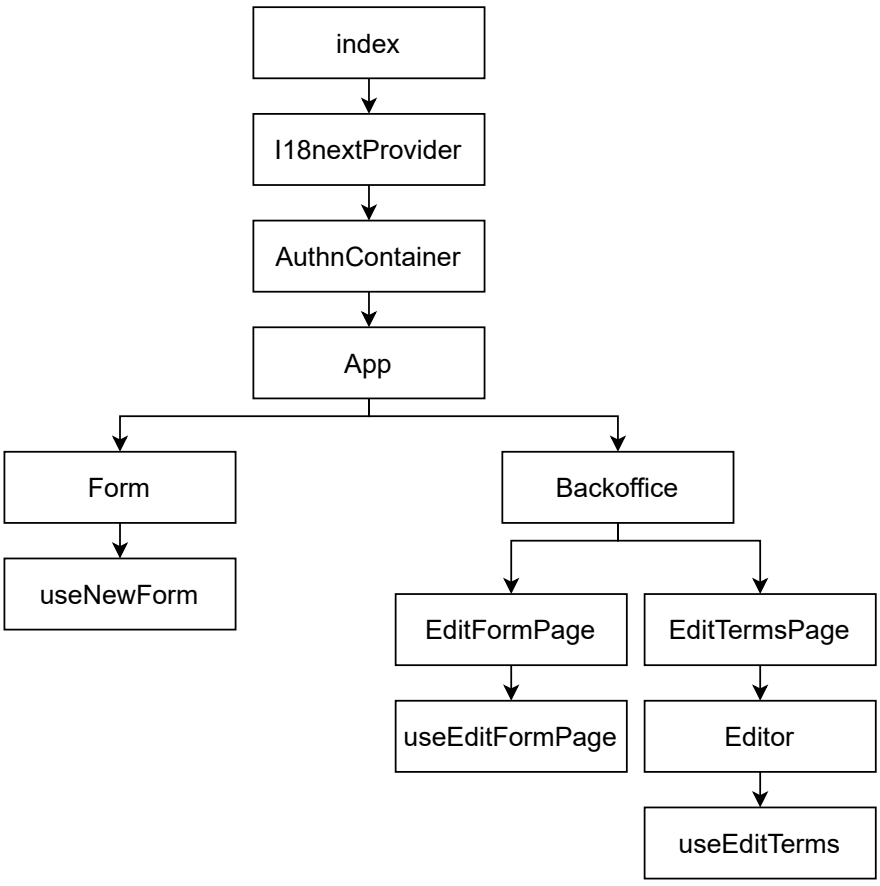

Figure 5.2: Simplified React Component Tree.

5.4.1 I18nextProvider

The I18nextProvider component is part of the i18next library ecosystem, which is used for internationalization (i18n) in JavaScript applications. The purpose of the I18nextProvider component is to integrate i18next into React applications by providing the i18n instance to the component tree.

By accessing the i18n instance, we can set the desired language for the application. This is useful for both the frontend and to request certain resources from the backend, i.e. requesting

the resource with the user's language preferences, or default to a certain resource if that isn't available.

5.4.2 AuthnContainer

The AuthnContainer component plays a crucial role in enabling user authentication and consequent storage of the authentication information within the application state.

To do so it uses the React Context API [13], which allows to pass data trough the component tree without having to pass props down manually at every level, thus enabling seamless data sharing between components.

The Session type describe a user's session, i.e. their name and nic. The SessionManager type acts as a wrapper, containing both the session and the methods to manage it, i.e. set a session and delete it.

The AuthContainer component wraps it's child components within it's LoggedInContext.Provider, providing the session manager instance and it's methods as the context value, making the authentication information available throughout the component tree.

5.4.3 Form

The Form component is the central component of the application serving as the form page. It leverages the useNewForm hook, which is responsible for retrieving the form structure from the backend and manage it within the application state.

As the user answers the form's the JSON-Rules-Engine is run and, according to the rules in the form, events are triggered, showing or hiding questions, allowing the user to answer the next group of questions or reviewing their answers.

5.4.4 EditFormPage

The EditFormPage component acts a an outlet for the backoffice component page. It leverages the useEditFormPage hook to retrieve the form structure from the backend and manage it's state.

As the user edit's the form's structure the changes are reflected in the hook's state, which is specially difficult given that a question can be a parent, i.e. it's answer causes another question to appear, and a child, i.e. it appears as a result of another question's answer.

To solve this issue the hook can reassign questions upon deletion, by finding the group with the parent question and setting the show condition of it's child question as undefined, which means they're automatically shown.

5.4.5 EditTermsPage

The EditTermsPage component acts as an outlet for the backoffice component page. It provides an interface for editing the terms and conditions. This component leverages the

Editor component, which integrates the Jodit WYSIWYG editor, offering a rich text editing experience. The state management is handled using the useEditTerms custom hook, which ensures seamless interaction with the backend. The content of the editor is stored as HTML allowing for flexible presentation and formatting when displayed to end-users.

5.5 Navigation

This chapter features a navigation graph, presented in Figure 5.3, that describes the UI flow of our platform. The primary entry point for all users is the Home page, from which navigation diverges based on the user's login status and role. With admins being able to navigate throughout the platform, while doctors lack backoffice access and finally donors only having access to the terms and form page.

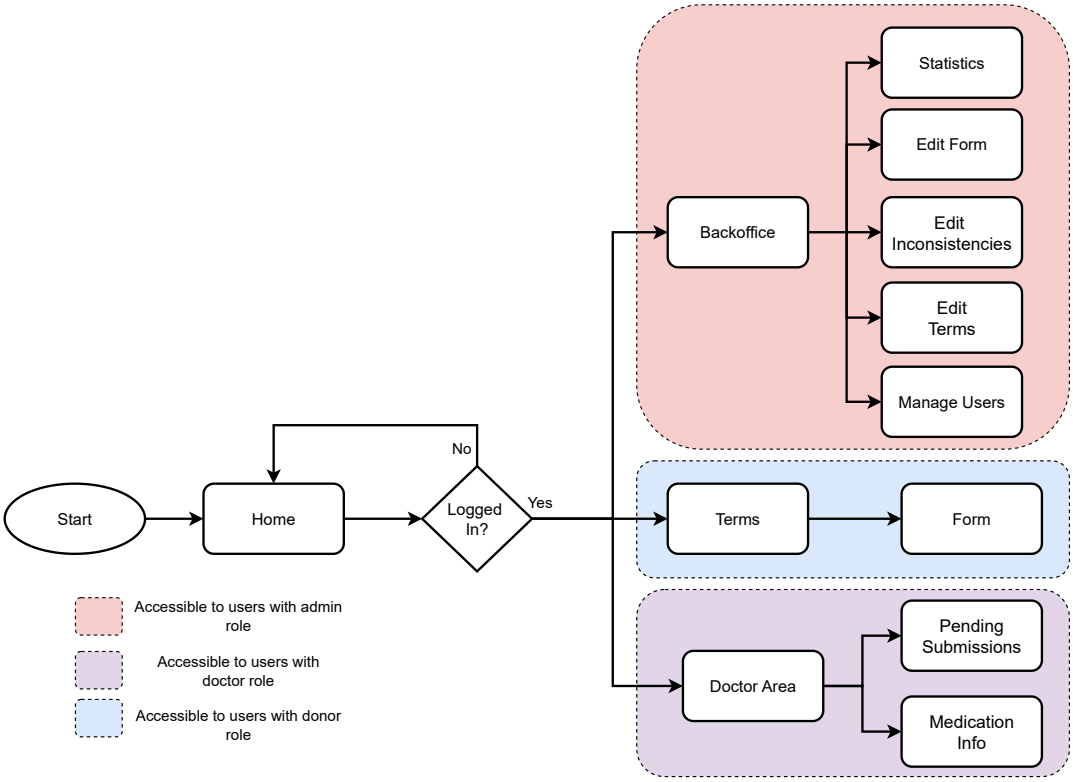

Figure 5.3: UI navigation.

Chapter 6

Tests

Testing is a crucial step in ensuring the reliability and functionality of our application. This chapter outlines the approaches and tools used to validate the correctness of our code.

6.1 Manual Testing

During the development of the application, particularly on the backend, manual testing was employed to verify that the system behaved as expected. The primary tools used for manual testing were:

- **Swagger:** Swagger was utilized to test the API through HTTP requests. Although Swagger is primarily an API documentation and automation tool, it provides the necessary features for effective manual testing of APIs.
- **Postman:** Postman was occasionally used to interact with ElasticSearch, specifically for data retrieval and storage operations.

6.2 Programmatic Testing

Programmatic testing involves the use of specialized software tools to ensure the correctness and robustness of the application. This approach allows for repetitive and comprehensive testing of the codebase, improving efficiency and coverage.

6.2.1 Unit Tests

Unit tests are designed to verify the behavior of individual components or modules in isolation. We followed the Arrange-Act-Assert (AAA) pattern to structure our unit tests:

- **Arrange:** Prepare the necessary preconditions and inputs for the test.
- **Act:** Execute the operation or function being tested.
- **Assert:** Verify that the outcome matches the expected result.

We employed xUnit.net, a free and open-source unit testing tool for the .NET framework, to execute our unit tests.

6.2.2 Integration Tests

TODO

Chapter 7

Future Work

The IPST medication guideline are organized in a table like manner, in the following column layout:

1. Class/Group of Medication: This column categorizes medications.
2. Active Substance/Commercial Name: This column lists either the active ingredient or the brand name of the medication.
3. Criteria: This column specifies if a particular class or group of medications affects eligibility for blood donation, including details such as the duration of ineligibility and other relevant conditions.

The terms used in the first column are, from what we can access, similar to the available pharmacotherapeutic classifications. A reliable source of a drug's pharmacotherapeutic classification is a portal provided by Infarmed to it's partner organizations, such as Lisbon's IPO. As such, upon a donor's form submission, assuming they were taking some medication, our application would perform requests to said portal, get the appropriate pharmacotherapeutic classification and, by cross-checking the classification with the term used in the first column of the guidelines, return the relevant interaction information. However, the terms used in the guidelines don't always reflect the available classifications, and, as such, the platform would need to employ some form of automated categorization, and allow for manual manipulation of these associations by the administrators.

Bibliography

[1] IPST. IPST história. Accessed: 01-05-2024.

[2] Council of European Union. 98/463/ec: Council recommendation of 29 june 1998 on the suitability of blood and plasma donors and the screening of donated blood in the european community, 1998. https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX

[3] Ana Paula Sousa Augusto Ramoa Cristina Caeiro Eugénia Vasconcelos Isabel Miranda Maria Antónia Escoval, Jorge Condeço and Mário Chin. Relatório de atividade transfusional e sistema português de hemovigilância 2022.

[4] OWASP Foundation. Password storage cheat sheet. `https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html`, n.d. Accessed: 2024-07-25.

[5] Alex Biryukov, Daniel Dinu, Dmitry Khovratovich, and Simon Josefsson. Argon2 Memory-Hard Function for Password Hashing and Proof-of-Work Applications. RFC 9106, September 2021.

[6] Mark Nottingham and Erik Wilde. Problem details for http apis. Internet Engineering Task Force (IETF), March 2016.

[7] OWASP Foundation. Authentication cheat sheet. `https://cheatsheetseries.owasp.org/cheatsheets/Authentication_Cheat_Sheet.html`, n.d. Accessed: 2024-07-25.

[8] SMARTBEAR. Api documentation & design tools for teams. Accessed: 02-06-2024.

[9] draw.io. Security-first diagramming for teams.

[10] Visily. Visily - ai-powered ui design software. Accessed: 02-06-2024.

[11] Michael B. Jones, John Bradley, and Nat Sakimura. JSON Web Token (JWT). RFC 7519, May 2015.

[12] mdn.developer.mozilla.org. Fetch api - web apis. Accessed: 30-05-2024.

[13] Meta Open Source. Passing data deeply with context - react. Accessed: 30-05-2024.

Appendix A

ER MODEL

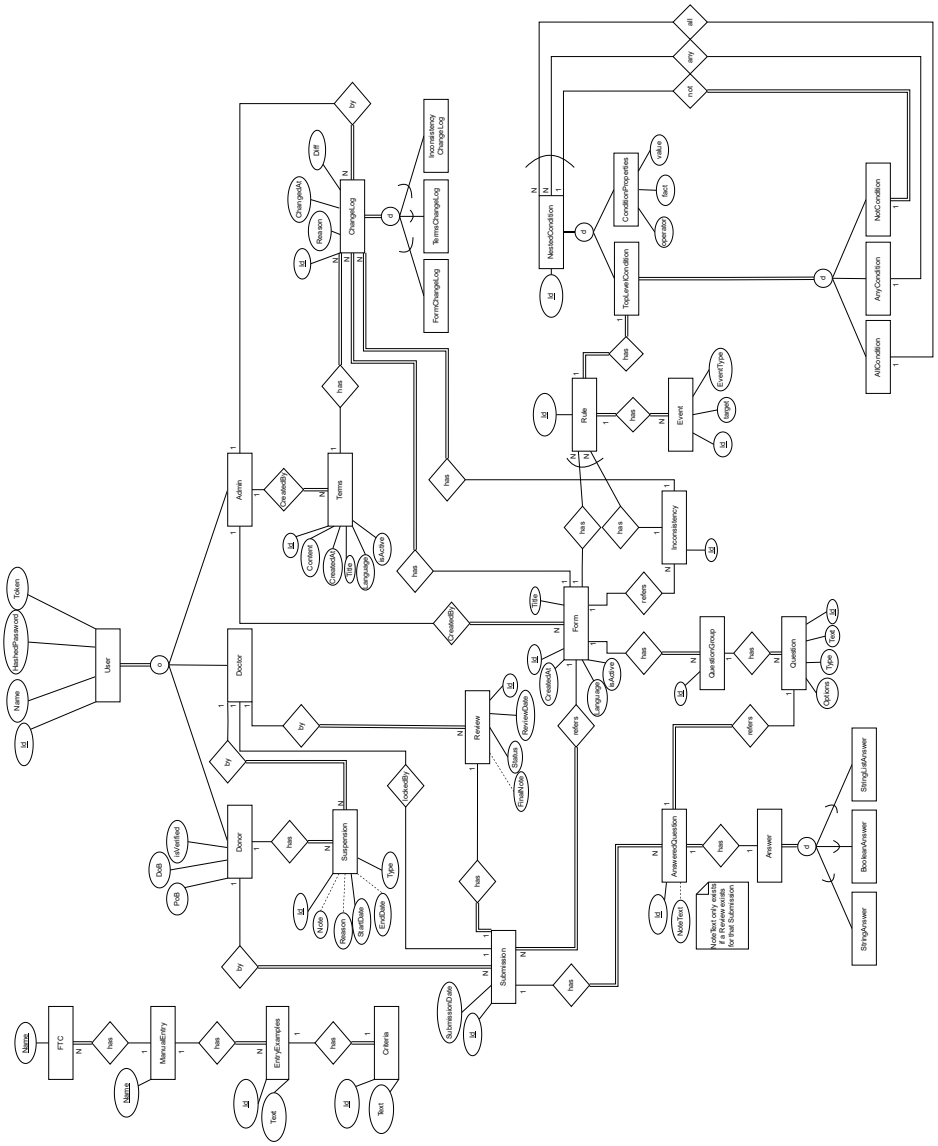

Figure A.1: ER Model.