



DADIVA IPO

Digital Aid and Donor Information Verification Application 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 Aid and Donor Information Verification Application 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

1	Introduction	1
1.1	Report Organization	2
2	Problem Description And Proposed Solution	3
2.1	Proposed Solution	4
3	Architecture	7
3.1	Overview	7
3.2	Frontend Application	8
3.3	Backend Application	15
4	Data Model	19
5	Frontend Implementation	29
5.1	Structure	30
5.2	Services	30
5.3	Components	32
5.4	Navigation	38
5.5	Testing	38
6	Backend Implementation	39
6.1	Structure	40
6.2	Program.cs	40
6.3	Password Security	51
6.4	Testing	53
7	Conclusions and Future Work	55
7.1	Accomplishments	55
7.2	Challenges	55
7.3	Future Work	55
Referências		57

List of Figures

2.1	Donor use case.	5
2.2	Doctor use case.	6
2.3	Administrator use case.	6
3.1	Application Architecture, gray squares represent containerized components of the solution.	7
3.2	JSON-Rules-Engine example with simple rule and question set.	9
3.3	Login Page Mock.	10
3.4	Form Page Mock.	11
3.5	Form Page Negative Answer Mock.	12
3.6	Form Page Positive Answer Mock.	13
3.7	Backoffice Page Mock.	14
3.8	Get Form Sequence Diagram.	17
4.1	User Entity.	19
4.2	User interactions.	20
4.3	User interactions.	21
4.4	Form Entity.	23
4.5	Condition Entity.	24
4.6	Submission Entity.	25
4.7	Manual Entity.	26
4.8	Lock Entity.	27
5.1	Simplified Component interaction and organization.	29
5.2	Simplified React Component Tree.	32
5.3	Form implementation.	35
5.4	UI navigation.	38
6.1	Block Diagram of our solution.	39
A.1	ER Model.	59

List of Tables

6.1	API endpoints related to the user	50
6.2	API endpoints related to the form	50
6.3	API endpoints related to the form	51
6.4	API endpoints related to the form	51

Listings

5.1	Condition for visible by default questions	34
5.2	Condition generated by and independent questions	36
5.3	Condition generated by and independent questions	36
6.1	Registering Scoped Services in ASP.NET Core Dependency Injection Container.	40
6.2	Registering Scoped Services in ASP.NET Core Dependency Injection Container:	41
6.3	Configuring CORS Policy in ASP.NET Core: Allowing Specific Origin with Full Access Control.	41
6.4	Custom JWT Authentication Middleware in ASP.NET Core: Handling Unauthorized Access with Detailed Problem Responses.	42
6.5	Configuring Role-Based Authorization Policies in ASP.NET Core: Defining Access Control for Donor, Doctor, and Admin Roles.	43
6.6	Custom Extension Methods for Handling HTTP Requests in ASP.NET Core: Simplifying Success and Error Response Handling.	46

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.

1.1 Report Organization

This report is structured into 7 chapters. In chapter 2 we describe the problem and proposed solution. In chapter 3 we go over the architecture of the proposed solution and the technologies used. In chapter 4 we decompose the complete Entity-Relationship diagram into smaller groups and elaborate on what each entity represents and the logic behind their relationships. In chapter 5 and 6 we elaborate the implementation details for the frontend and backend respectively. Finally in chapter 7 we reflect on what was achieved with this project, the encountered challenges, lessons learned and future development.

Chapter 2

Problem Description And Proposed Solution

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 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.

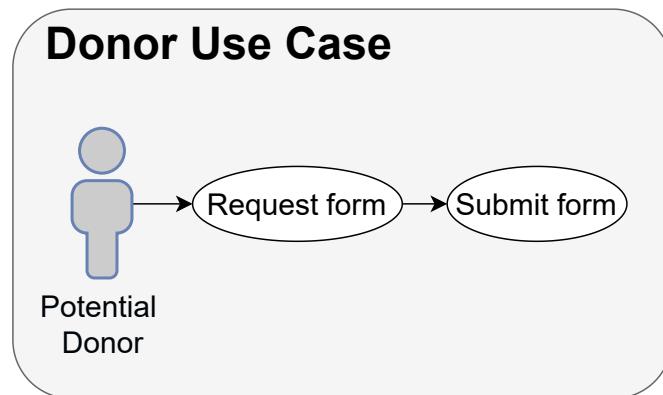


Figure 2.1: Donor use case.

The donor user can request the current form and can submit their form responses.

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.

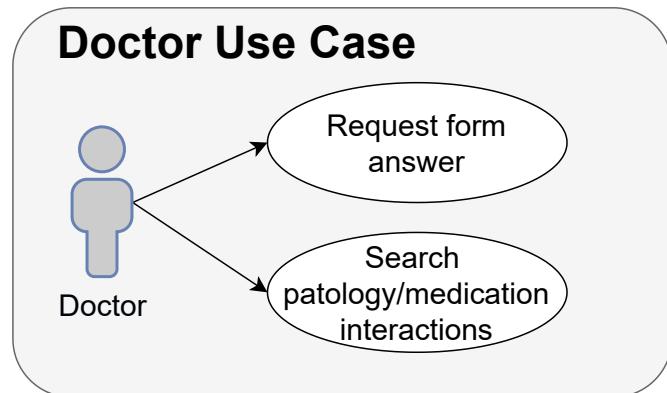


Figure 2.2: Doctor use case.

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.

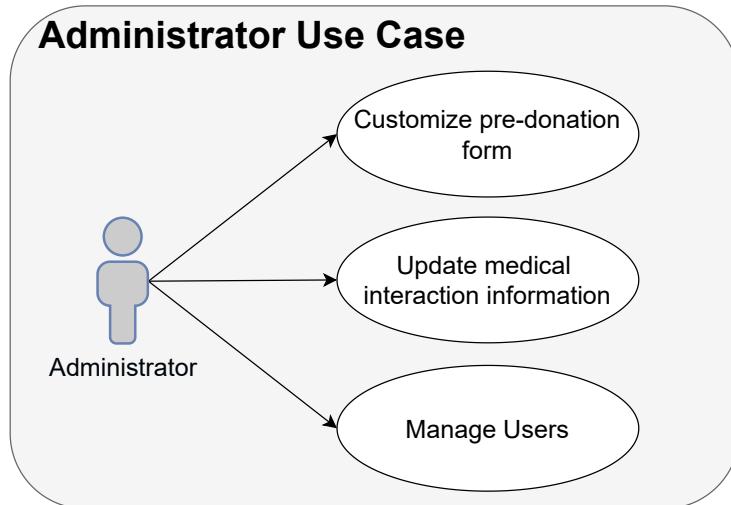


Figure 2.3: Administrator use case.

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.

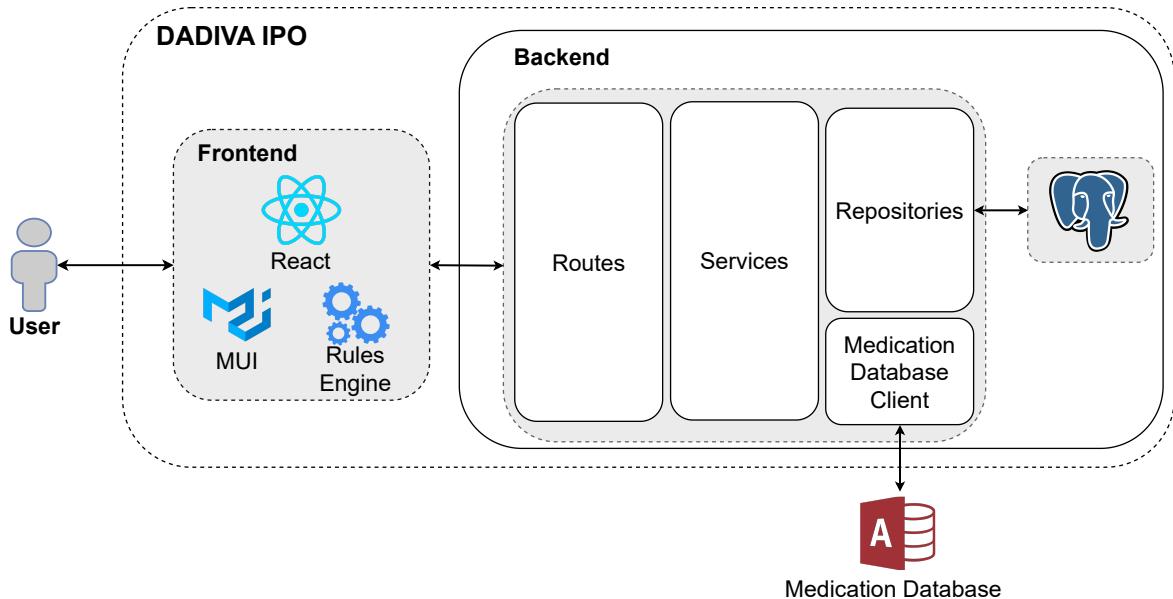


Figure 3.1: Application Architecture, gray squares represent containerized components of the solution.

The system consists of a backend application (server-side) and a frontend application (client-side). The backend architecture consists of routes, services, repositories and a medication database client.

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.

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 utilizes Material UI, a popular React component library, to ensure a consistent and responsive user interface across these components. Material UI provides pre-built components and design elements that streamline the development process and enhance the application's visual appeal.

In addition to the user interface, the application includes a service layer responsible for communicating with the backend application through the REST API.

This chapter serves as an overview, for implementation details refer to chapter 5.

3.2.1 JSON-Rules-Engine and Form structure

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.

Forms consist of a list of questions, divided into groups, and a set of rules, which are added to the engine instance. These rules are defined by conditions and events. When the engine runs all the conditions that are true will return their respective event.

Conditions can be categorized into **top-level conditions** and **condition properties**, following the types specified in the JSON-Rules-Engine library. **Top-level conditions** refer to logical operators, such as all, any, and not, while **condition properties** involve boolean evaluations, such as determining whether a given input equals a specified value.

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.

These rules are described in a JSON format, with an example illustrated in Figure 3.2.

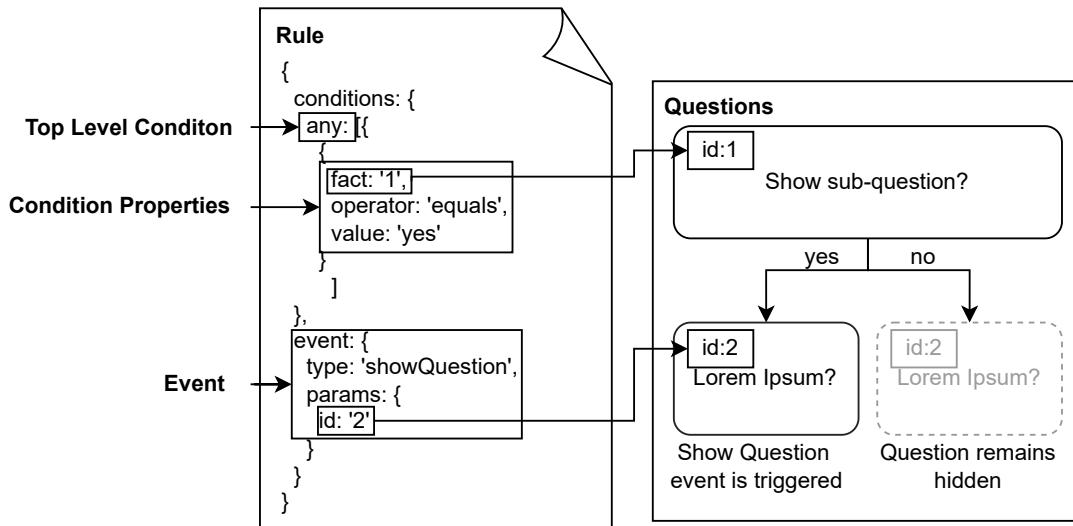


Figure 3.2: JSON-Rules-Engine example with simple rule and question set.

Note that the fact and value fields in the **condition properties** and the type and params fields for the **event** do not have predefined values. The use of a rules engine decouples rules from business logic, meaning the decision to link the fact and params fields to the ID of a question depends on the system implementation. In our case we decided, not only that the before-mentioned fields would pertain to question IDs but also, that the type field for the **event** would have the following values:

- **showQuestion:** show the question with id in the params field;
- **nextGroup:** allow for navigation to next group, is triggered when all the question in current group have been answered;
- **showReview:** allow form review, is triggered when all the answers in the form have been answered and donor can review form.

For implementation details, i.e. how the questions become visible and the rules are added to the engine, refer to section 5.3.3.

3.2.2 Mocks

During planning some mockups where created of the final result for some pages, the login page is presented in Figure 3.3, the form pages are presented in Figures 3.4, 3.5 and 3.6, the backoffice page is presented in Figure 3.7.



Figure 3.3: Login Page Mock.



Dador

g Francisco M.

Pergunta 1/30

Alguma vez deu sangue ou componentes sanguíneos?

✗ Não

✓ Sim

Figure 3.4: Form Page Mock.

Dador



Francisco M.

Pergunta 1/30

Alguma vez deu sangue ou componentes sanguíneos?

Não

Sim

 Proxima Questão

Figure 3.5: Form Page Negative Answer Mock.

Dador 

Pergunta 1/30

Alguma vez deu sangue ou componentes sanguíneos?

✗ Não

✓ Sim

 Proxima Questão

Figure 3.6: Form Page Positive Answer Mock.

 **Dashboard**

 Editar Formulario

 Editar medicação

 Estatísticas

Gestão

 Dadoures

 Doutores

 Notificações

9

 Definições

Administrador
 Francisco M.

Estatísticas Dadoures

 Desaprovedos  Aprovados

Esta semana ▾

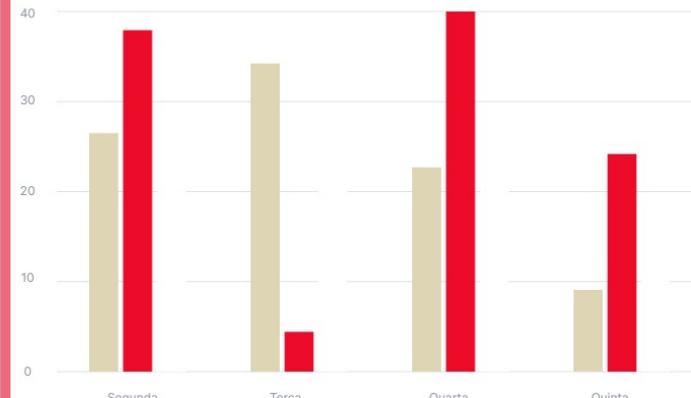


Figure 3.7: Backoffice Page Mock.

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 3.8

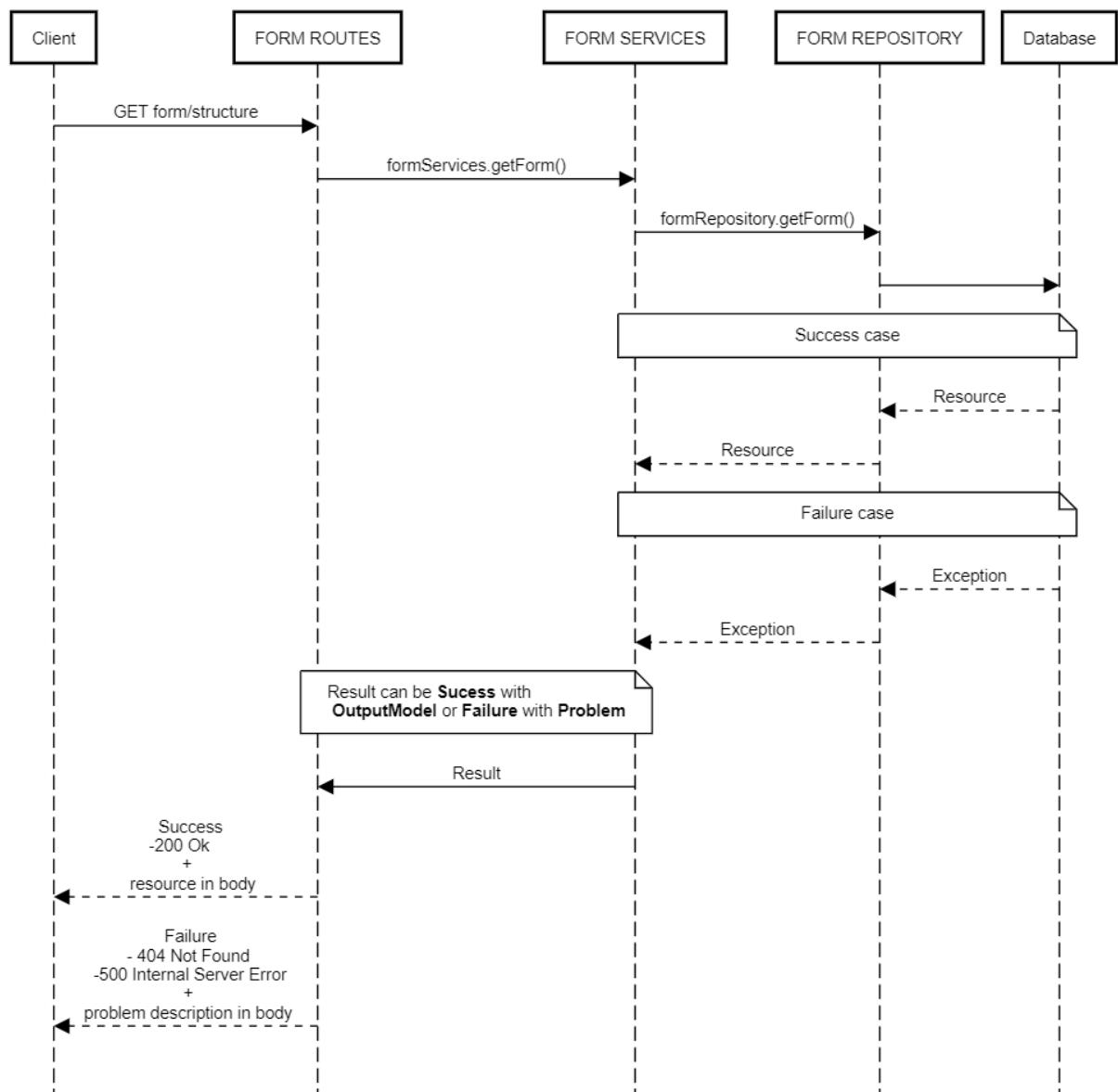


Figure 3.8: Get Form Sequence Diagram.

Chapter 4

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.

4.0.1 User

As outlined in section 2.1.3, 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 4.1.

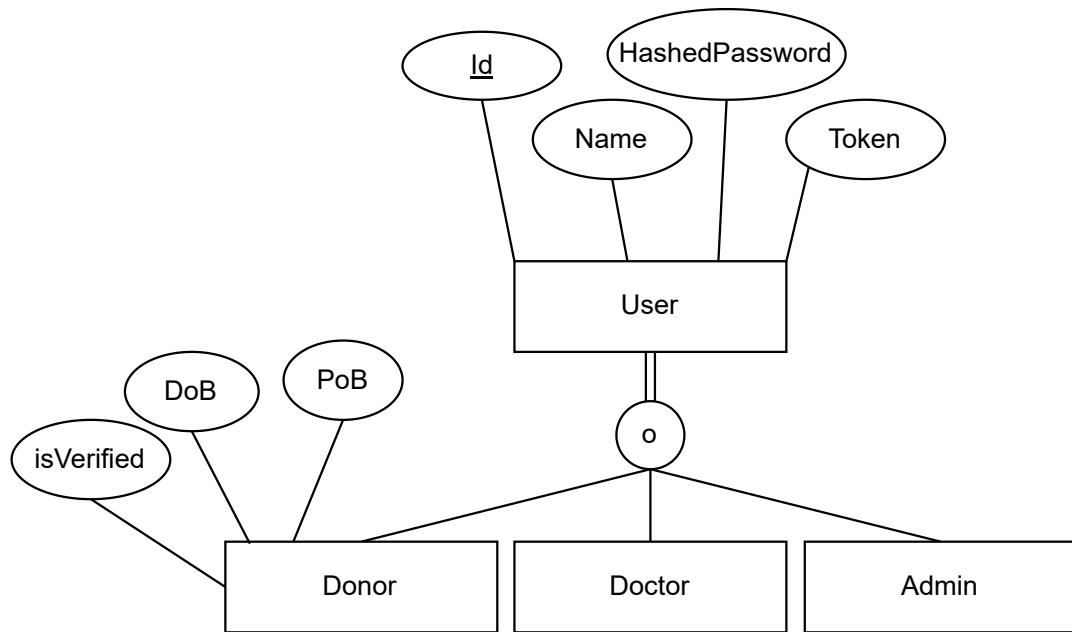


Figure 4.1: User Entity.

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 6.3;
- **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

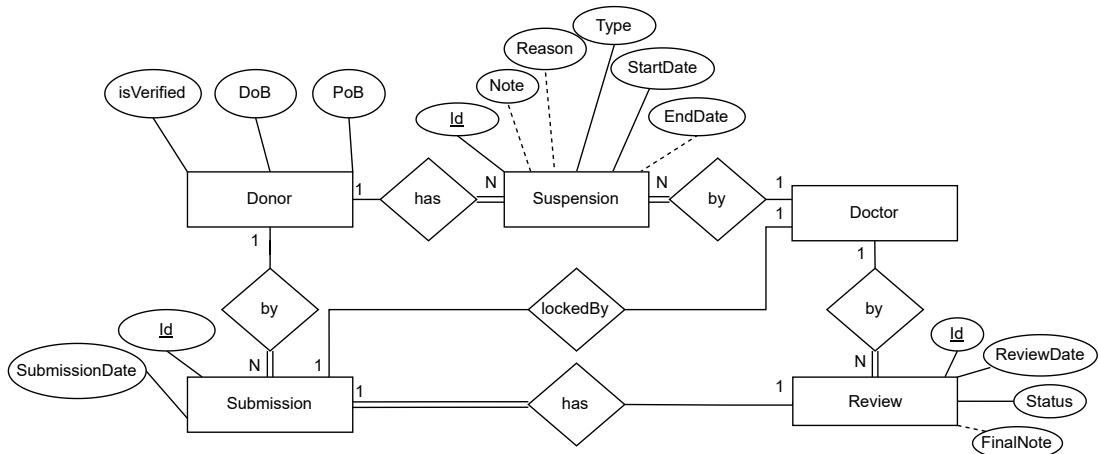


Figure 4.2: User interactions.

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 4.0.5.

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;

4.0.2 Admin

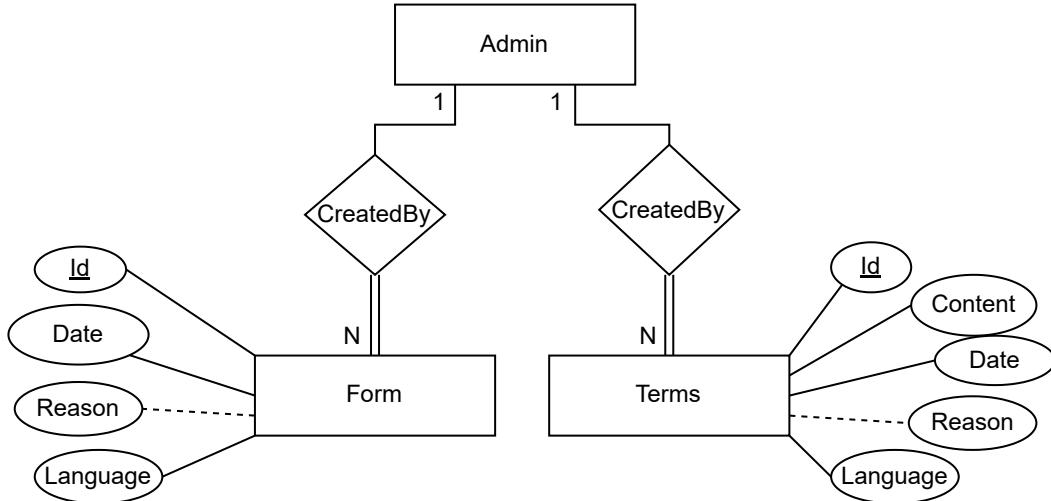


Figure 4.3: User interactions.

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 4.0.3.

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;

4.0.3 Form relationships

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 4.4, 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 4.0.4.

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** has 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;

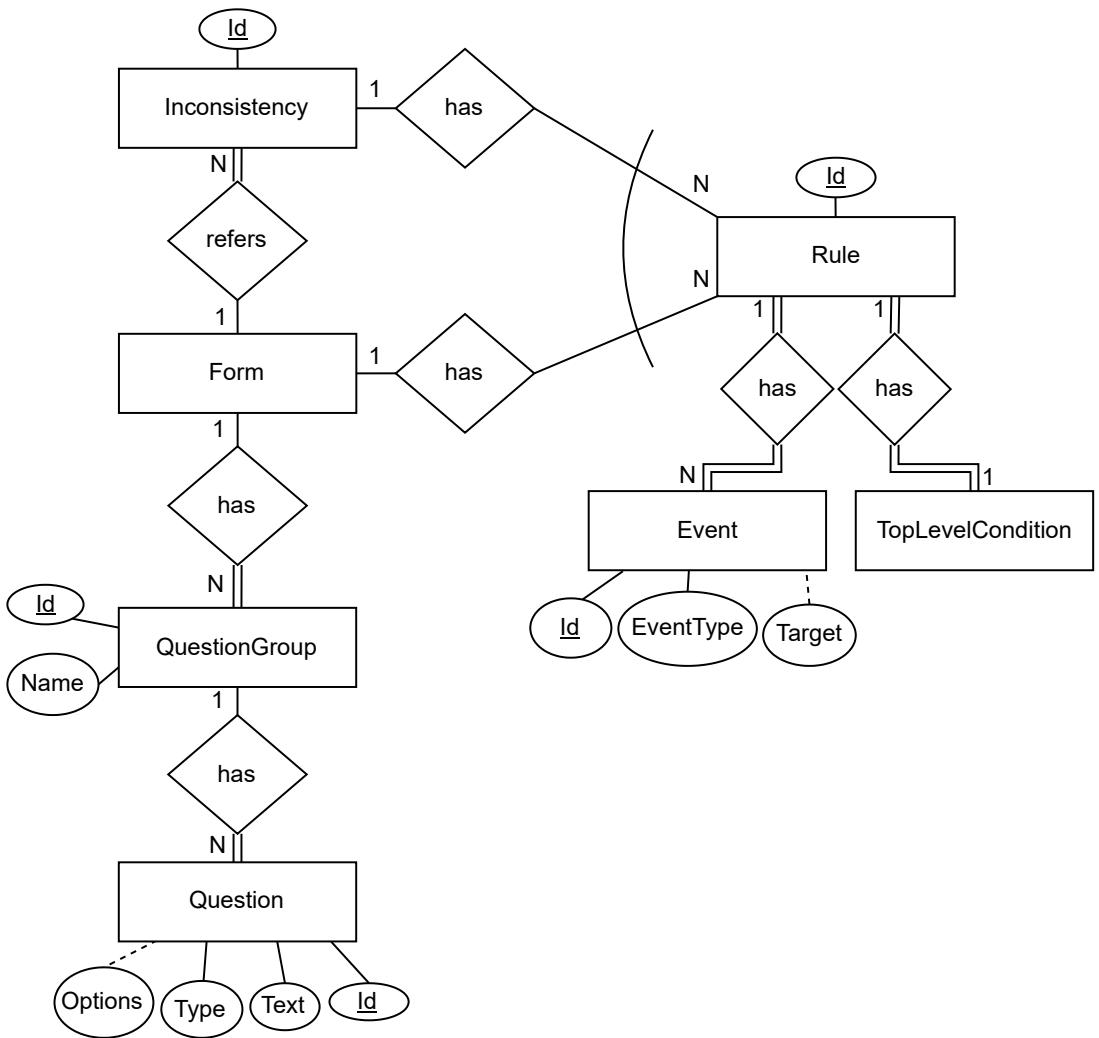


Figure 4.4: Form Entity.

- **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 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.

4.0.4 Conditions

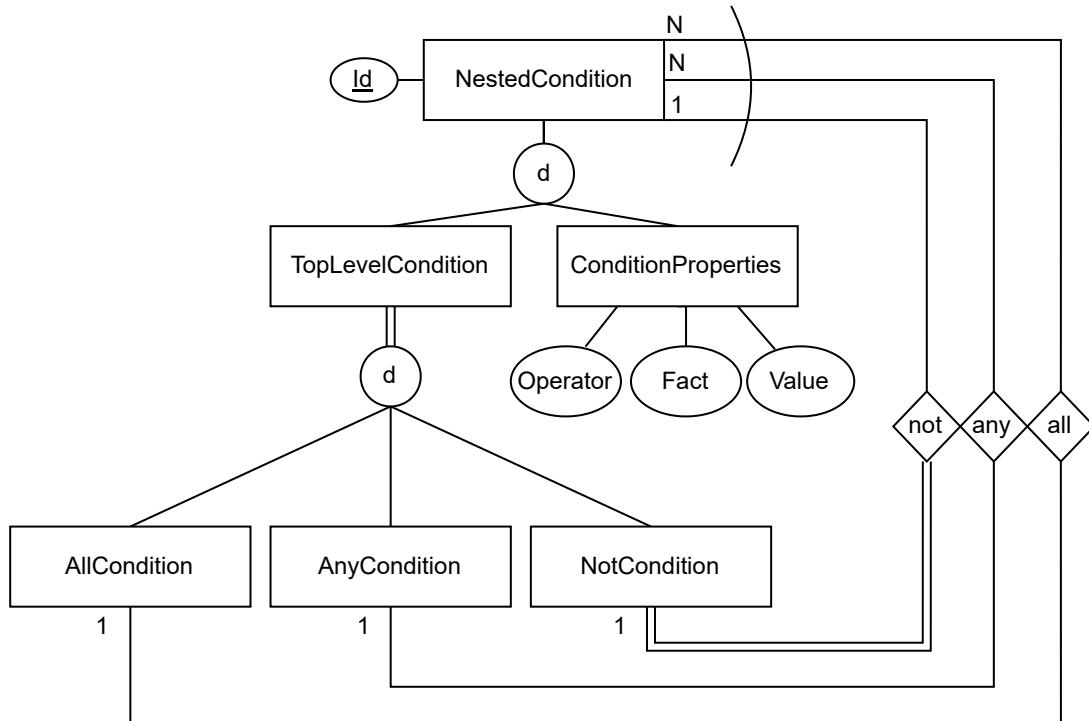


Figure 4.5: Condition Entity.

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** and **NotCondition** entities, it can be seen as a representation of logical operators. As illustrated in Figure 4.5, 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.

4.0.5 Submission

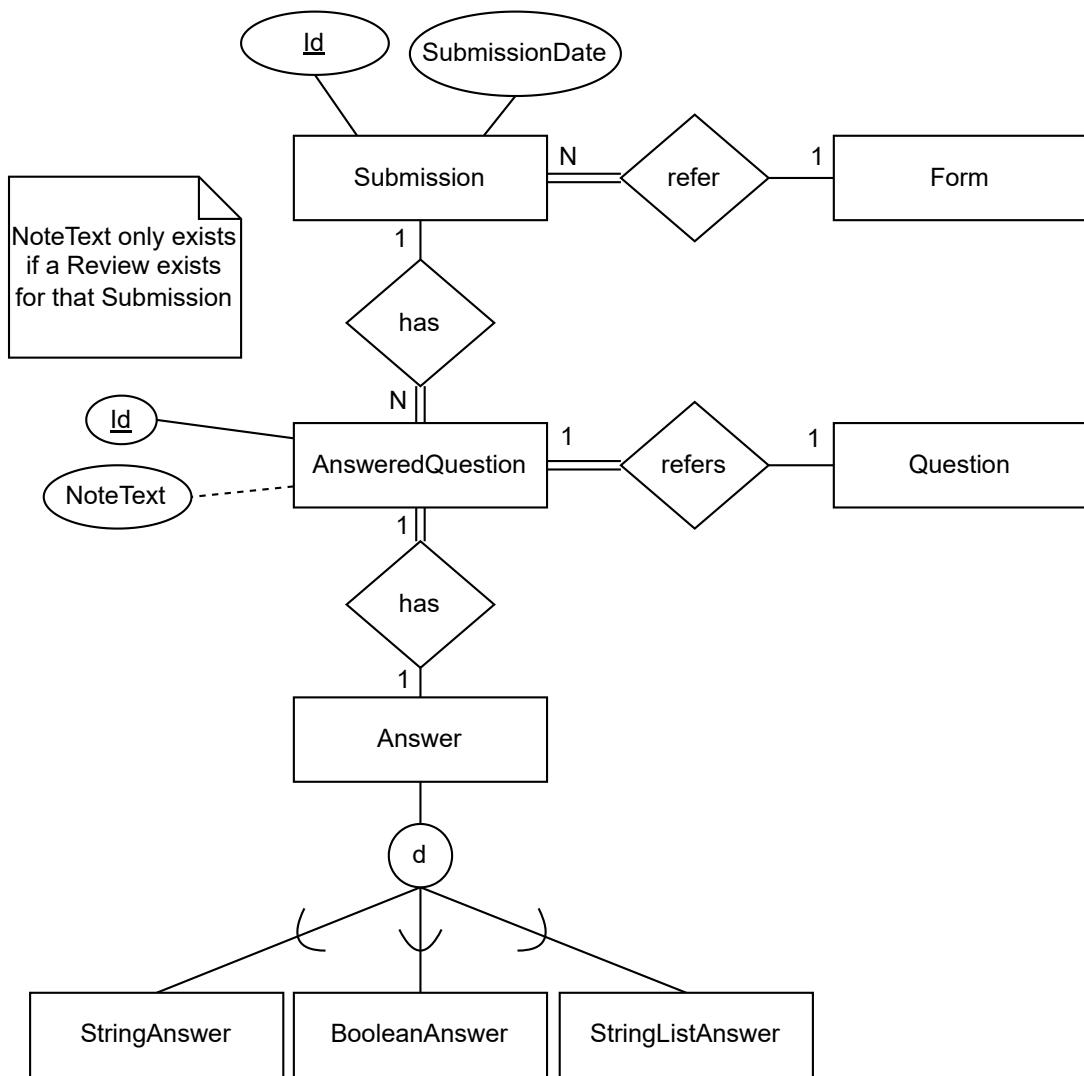


Figure 4.6: Submission Entity.

As illustrated in Figure 4.6, the **Submission** entity has 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.

4.0.6 Manual Information

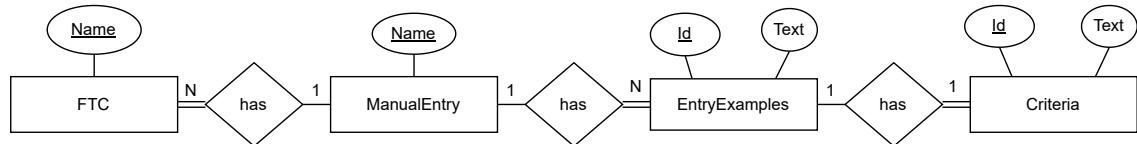


Figure 4.7: Manual Entity.

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 has 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.

4.0.7 Locks

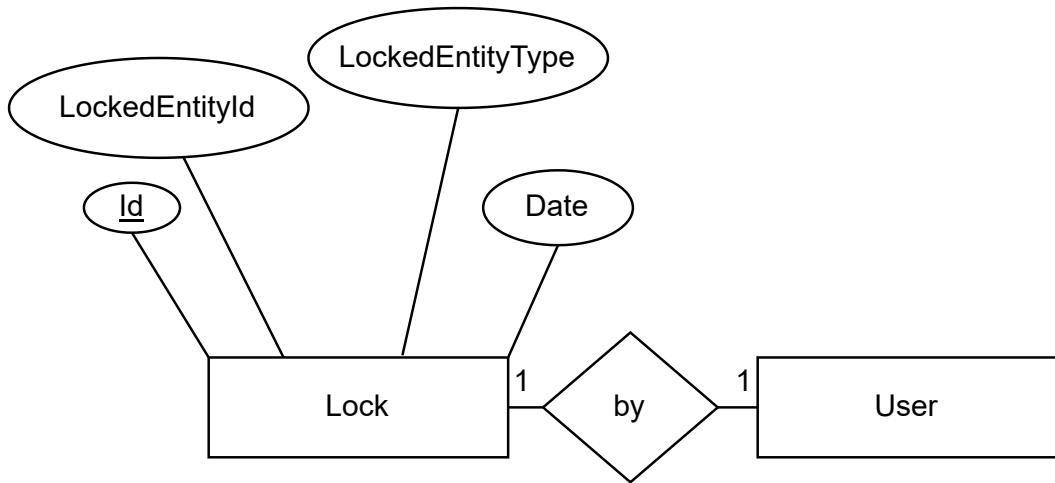


Figure 4.8: Lock Entity.

To avoid concurrent manipulation of resources we created the **Lock** entity. This entity has a one to one relationship with the **User** holding the lock for this resource and has the following attributes:

- **Id**: The unique identifier for this lock
- **LockedEntityId**: The identifier for the resource being locked;
- **LockedEntityType**: The type of resource being locked, i.e. submission, form, terms;
- **Date**: The date when the resource was locked .

Chapter 5

Frontend Implementation

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 2.1.3.

The frontend is implemented in Typescript and React, and makes use of JSON-Rules-Engine to enforce the form's rules.

A simplified overview of the component interaction and organization is presented in figure 5.1.

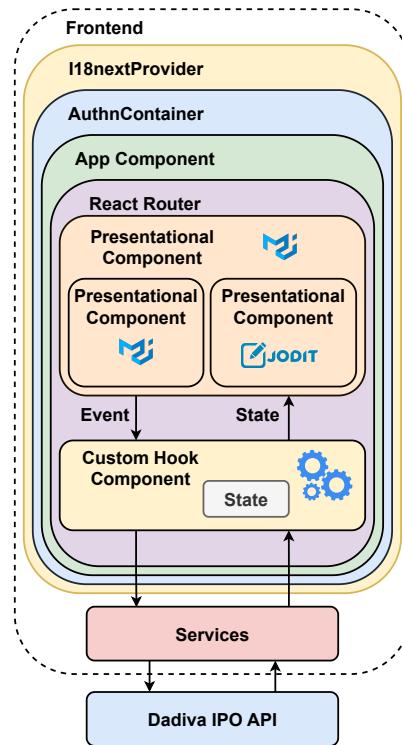


Figure 5.1: Simplified Component interaction and organization.

5.1 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: contains the dependencies of the application;
- tsconfig.json: contains the TypeScript configuration;
- 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.

5.2 Services

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

5.3 Components

A simplified view of the component tree is presented in Figure 5.2. In this chapter we'll mainly focus on the:

- **I18nextProvider**: manages the language preferences of the user;
- **AuthnContainer**: manages the session information;
- **FormPage & useNewForm**: used to display form to the donor and manage its state using JSON-Rules-Engine;
- **EditFormPage & useEditFormPage**: displays form in the backoffice, to the administrators, and allows to perform changes to the structure, also manage its state;
- **Editor**: used to display the terms to the administrators and allows to change them.

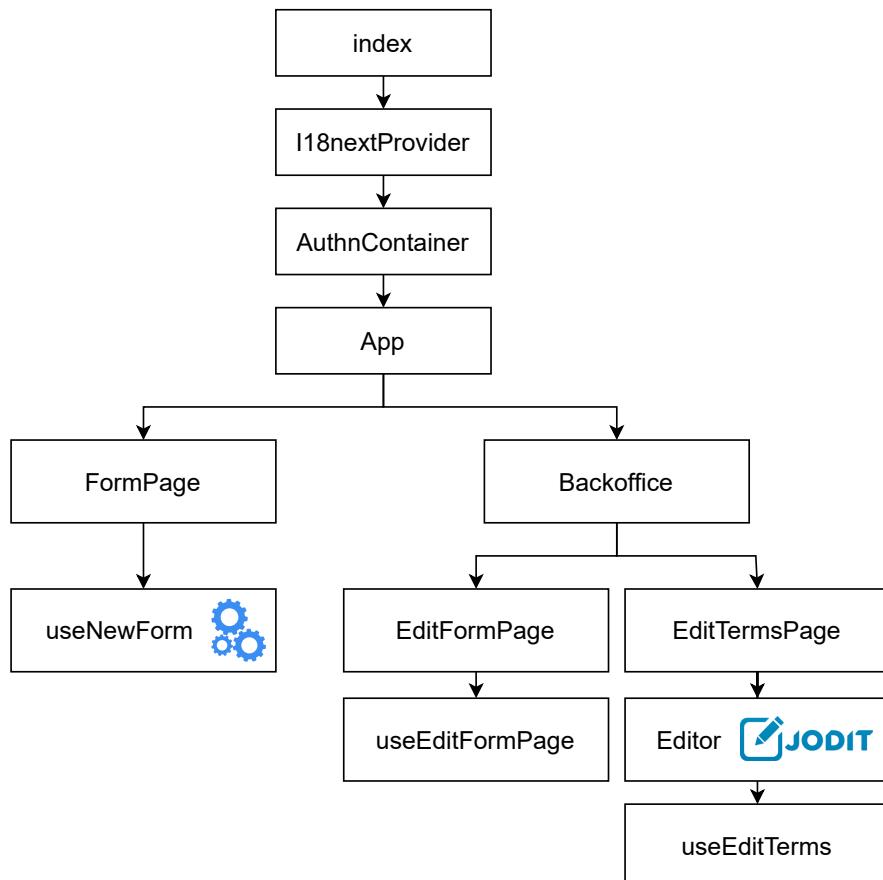


Figure 5.2: Simplified React Component Tree.

5.3.1 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.

5.3.2 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 through the component tree without having to pass props down manually at every level, thus enabling seamless data sharing between components.

The Session type describes 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 its child components within its LoggedInContext.Provider, providing the session manager instance and its methods as the context value, making the authentication information available throughout the component tree.

5.3.3 FormPage & useNewForm

The concepts in this section build upon those discussed in Section 3.2.1. It is advised to refer to that section for a thorough understanding.

The FormPage is responsible for form display to donor users. It leverages the useNewForm hook.

The useNewForm hook is a custom React hook designed to manage the state and logic of the form within the frontend application. This hook integrates several important functionalities, including handling form data, managing user interactions, and executing business rules, making it a critical part of the form-handling process.

To do this the hook makes use of the JSON-Rules-Engine. After the form is fetched using the frontend service layer the initial state of the engine is set, an illustration of the following example is presented in Figure 5.3. Firstly the facts, i.e. the answers, are added to the engine, which state that all the questions haven't been answered, and secondly the rules, which contain the navigation and display logic.

To make sure all questions that should show by default are visible the form has a rule, per question, in which the top level condition, no matter the type, is empty with an event of type showQuestion and referring to the question that should be visible by default, as illustrated in Listing 5.1. Once the engine is run, these conditions are met and the respective events are returned.

```
1  conditions: {
2      any: [],
3  },
4  event: {
5      type: 'showQuestion',
6      params: {
7          id: 'visibleByDefault'
8      },
9 }
```

Listing 5.1: Condition for visible by default questions

The hook's states are then updated, mainly, the showQuestions, canGoNext, canGoReview and formAnswers. The showQuestions state stores the questions that are being displayed. The canGoNext state stores if the next group should be available, by default false when initializing. The canGoReview state stores if all questions where answered and the form can be reviewed, by default false when initializing. The formAnswers state stores all the answers given at any point. As the user answers the form's questions the formAnswers state is updated and is fed to the engine, which runs again, returning the relevant events.

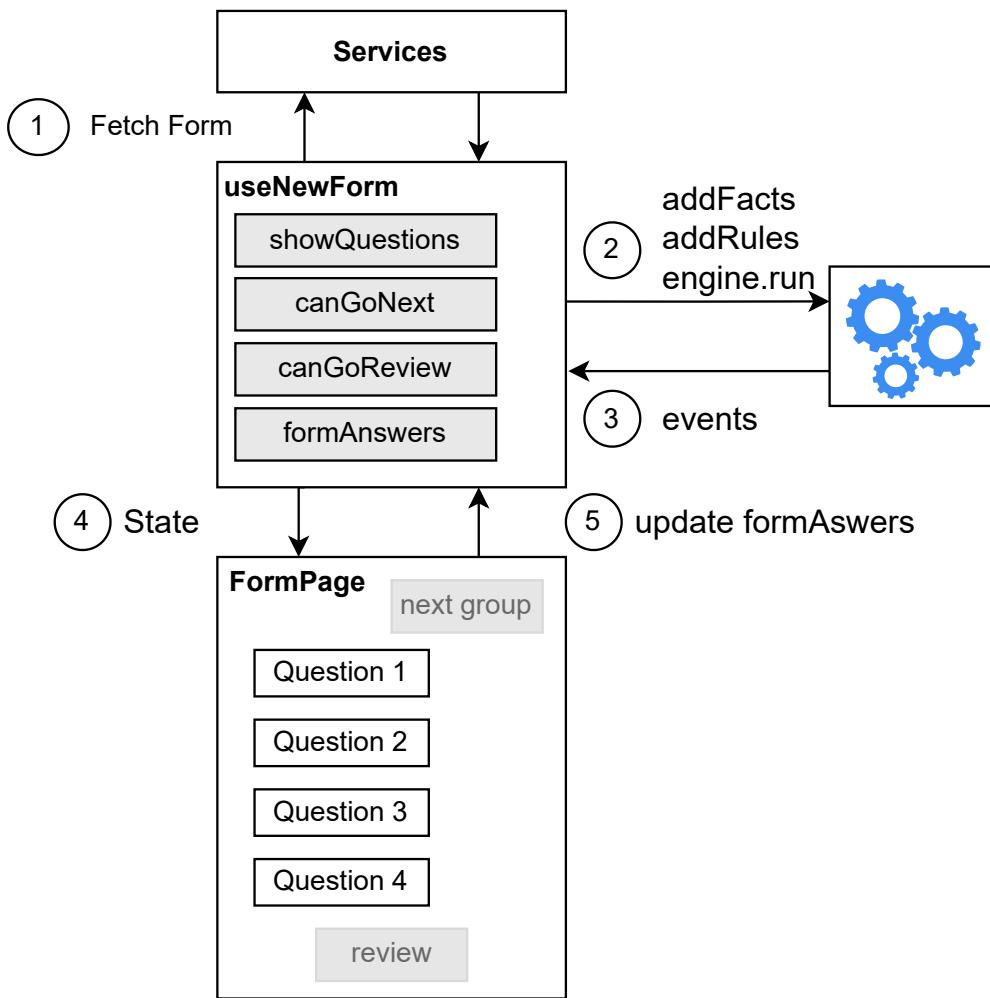


Figure 5.3: Form implementation.

5.3.4 EditFormPage & useEditFormPage

The `EditFormPage` component acts as an outlet for the backoffice component page. It leverages the `useEditFormPage` hook to retrieve the form structure from the backend and manage its state.

As mentioned in Section 3.2.1, the type of form events in our system are:

- **showQuestion:** show the question with id in the params field;
- **nextGroup:** allow for navigation to next group, is triggered when all the questions in current group have been answered;
- **showReview:** allow form review, is triggered when all the answers in the form have

been answered and donor can review form.

To generate the nextGroup event we iterate over all the group of questions and for each create a top level condition of the all type and then iterate over every question in that group to create more conditions within this top level condition. These questions can be divided into two types, which generate different conditions:

- **Independent questions:** These generate the simplest conditions, since they just need to have an answer;
- **Child questions:** If the question is a child of another question there are two possible scenarios, either the question is visible and it needs an answer or it isn't displayed and should be ignored.

The first scenario generates a condition as illustrated in Listing 5.3.

```
1 conditions: {
2     all: [
3         {
4             fact: 'IndependentQuestion1',
5             operator: 'notEqual',
6             value: ''
7         },
8         {
9             fact: 'IndependentQuestion2',
10            operator: 'notEqual',
11            value: ''
12        }
13    ],
14 }
```

Listing 5.2: Condition generated by and independent questions

Meanwhile the second scenario generates a condition as illustrated in Listing ?? the all top level condition type is used as a question can have multiple parent questions.

```
1 conditions: {
2     all: [
3         any: [
4             {
5                 fact: 'ChildQuestion',
6                 operator: 'notEqual',
7                 value: ''
8             },
9             all:{ 
10                 fact: 'ParentQuestion',
11                 operator: 'notEqual',
12                 value: 'value that trigger child question display'
13             }
14         ]
15     ],
16 }
```

Listing 5.3: Condition generated by and independent questions

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.

5.3.5 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.

5.4 Navigation

This chapter features a navigation graph, presented in Figure 5.4, 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.

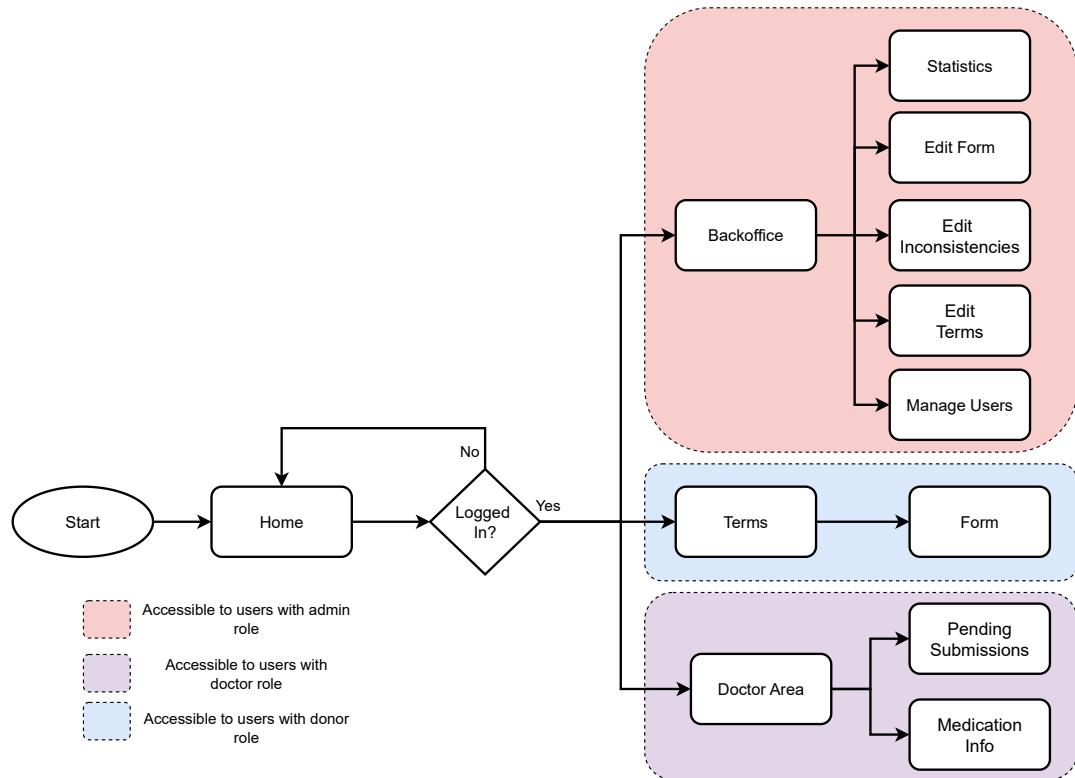


Figure 5.4: UI navigation.

5.5 Testing

TO-DO

Chapter 6

Backend Implementation

The backend application is tasked with handling http requests, business logic and data persistence.

The backend logic is implemented in C# and data is persisted in a PostgreSQL database, interaction with this database is done through EF CORE.

An overview of the backend implementation is presented in Figure 6.1.

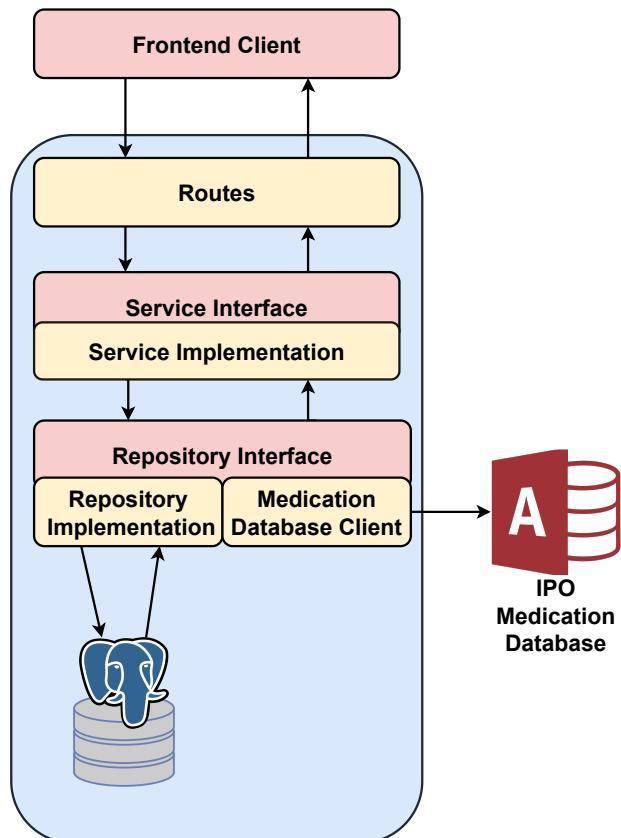


Figure 6.1: Block Diagram of our solution.

6.1 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;
 - entities: contains the entities outlined in chapter 4.
- 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.

6.2 Program.cs

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 most of these services were registered with the Scoped scope, as illustrated in Listing 6.1.

```
1 builder.Services.AddScoped<IUserService, UserService>();
2 builder.Services.AddScoped<IFormService, FormService>();
3 builder.Services.AddScoped<ITermsService, TermsService>();
4 builder.Services.AddScoped<IMedicationsService, MedicationsService>();
5 builder.Services.AddScoped<IManualService, ManualService>();
6 builder.Services.AddScoped<ISubmissionService, SubmissionService>();
7 builder.Services.AddScoped<IReviewsService, ReviewsService>();
8
```

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

Listing 6.1: Registering Scoped Services in ASP.NET Core Dependency Injection Container.

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

The services that don't make use of the scoped lifetime are associated with the lock management and server sent events, are illustrated in 6.2, these services are elaborated on section ??.

```
1 builder.Services.AddSingleton< INotificationService , NotificationService>();
2 builder.Services.AddSingleton< NotificationEndpoint >();
3 builder.Services.AddHostedService< UnlockExpiredSubmissionsService >();
```

Listing 6.2: Registering Scoped Services in ASP.NET Core Dependency Injection Container:

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:

```
1 builder.Services.AddCors(options =>
2 {
3     options.AddPolicy("MyCorsPolicy",
4         policy =>
5     {
6         policy.WithOrigins("http://localhost:8000")
7             .AllowAnyHeader()
8             .AllowAnyMethod()
9             .AllowCredentials();
10    });
11});
```

Listing 6.3: 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, referred to as jwtIssuer, jwtAudience and jwtKey in listing 6.4 below, are stored in the appsettings.json file.

```
1 builder.Services.AddAuthentication(x =>
2 {
3     x.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;
4     x.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;
5 }).AddJwtBearer(options =>
6 {
7     options.Events = new JwtBearerEvents
8     {
9         ...
10        OnChallenge = context =>
11        {
12            context.HandleResponse();
13
14            context.Response.ContentType = "application/problem+json";
15            context.Response.StatusCode = StatusCodes.Status401Unauthorized;
16            var problemDetails = new
17            {
18                type = "https://localhost:8000/errors/unauthorized",
19                title = "Unauthorized",
20                detail = "You are not authorized to access this resource.
21                    Please provide valid credentials.",
22                status = StatusCodes.Status401Unauthorized
23            };
24            var problemJson = JsonSerializer.Serialize(problemDetails);
25            return context.Response.WriteAsync(problemJson);
26        }
27        options.SaveToken = true;
28        options.TokenValidationParameters = new TokenValidationParameters
29        {
30            ValidateIssuer = true,
31            ValidateAudience = true,
32            ValidateLifetime = true,
33            ValidateIssuerSigningKey = true,
34            ValidIssuer = jwtIssuer,
35            ValidAudience = jwtAudience,
36            IssuerSigningKey = new
37                SymmetricSecurityKey(Encoding.UTF8.GetBytes(jwtKey))
38        };
39 });
40 }
```

Listing 6.4: 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.

Role Based Access Control

In our system, users are assigned one of three roles: donor, doctor, or admin. Each of these roles grants access to a specific set of endpoints, ensuring that users can only interact with

the parts of the application that are relevant to their role. To enforce this, we have configured authorization policies within the ASP.NET Core framework, which restrict access based on the role claims present in the user's JSON Web Token (JWT).

As shown in Listing 6.5, we define three authorization policies—one for each role. The AddAuthorization method is used to add these policies to the service container. Each policy requires that the user's JWT contains a claim of type ClaimTypes.Role with a corresponding value of either "donor", "doctor", or "admin".

```
1 builder.Services.AddAuthorization(options =>
2 {
3     options.AddPolicy("donor", policy =>
4         policy.RequireClaim(ClaimTypes.Role, "donor"));
5     options.AddPolicy("doctor", policy =>
6         policy.RequireClaim(ClaimTypes.Role, "doctor"));
7     options.AddPolicy("admin", policy =>
8         policy.RequireClaim(ClaimTypes.Role, "admin"));
9 });
10
```

Listing 6.5: Configuring Role-Based Authorization Policies in ASP.NET Core: Defining Access Control for Donor, Doctor, and Admin Roles.

When defining an endpoint, we can specify which authorization policy should be applied. For instance, an endpoint that should only be accessible to doctors can be protected by the "doctor" policy.

By leveraging these role-based authorization policies, we enforce a clear and secure access control mechanism, ensuring that users can only perform actions that are appropriate for their role. This approach not only enhances security but also simplifies the management of user permissions across the application.

6.2.1 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:

```
1  public interface IFormRepository
2  {
3      public Task<Form> GetForm();
4
5      public Task<Form?> GetFormWithVersion(int version);
6
7      public Task<Form> EditForm(Form form);
8
9      public Task<bool> SubmitForm(Submission submission);
10
11     public Task<List<Submission>?> GetPendingSubmissions();
12
13     public Task<Submission> GetSubmission(int nic);
14
15     public Task<Submission?> GetSubmissionById(int id);
16
17     public Task<Submission?> GetLatestPendingSubmissionByUser(int
18         userNic);
19
19     public Task<(List<SubmissionHistoryDto>? Submissions, bool
20         HasMoreSubmissions)> GetSubmissionHistoryByNic(int nic, int
21         limit, int skip);
22
22     public Task<Inconsistencies> GetInconsistencies();
23
23     public Task<bool> LockSubmission(int submissionId, int doctorId);
24
24     public Task<bool> UnlockSubmission(int submissionId, int doctorId);
25
25     public Task<List<SubmissionLock>> GetExpiredLocks(TimeSpan timeout);
26
26     public Task<bool> SubmissionExists(int id);
27
27     public Task<Review> AddReview(Review review);
28
28     public Task<bool> AddNote(Note note);
29
29     public Task<bool> EditInconsistencies(Inconsistencies
30         inconsistencies);
31
31 }
```

User Repository

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

And this repository will have the following methods:

```
1  public interface IUsersRepository
2  {
3      public Task<bool> AddUser(User user);
4
5      public Task<List<User>?> GetUsers();
6
7      public Task<User?> GetUserByNic(int nic);
8
9      public Task<Boolean> DeleteUser(int nic);
10
11     public Task<UserAccountStatus?> GetUserAccountStatus(int userNic);
12
13     public Task<Boolean> UpdateUserAccountStatus(UserAccountStatus
14         userAccountStatus);
15
16     public Task<bool> AddSuspension(UserSuspension suspension);
17     public Task<bool> UpdateSuspension(UserSuspension suspension);
18     public Task<UserSuspension?> GetSuspension(int userNic);
19     public Task<bool> DeleteSuspension(int userNic);
20 }
```

6.2.2 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.

Error Handling Implementation

As mentioned in section 3.3.1 we will utilize a **Result** class which can encapsulate either a successful outcome or an error.

To enforce this practice, every service method returns a Result, and each service call in the route layer is wrapped by a HandleRequest method. This method belongs to the HttpExtensions class, as illustrated in Listing 6.6.

```
1 public static class HttpExtensions
2 {
3     public static IResult HandleRequest<TIn>(
4         this Result<TIn> result, Func<TIn, IResult> onSuccess)
5     {
6         return result.IsSuccess ? onSuccess(result.Value) :
7             Results.Problem(ErrorToProblem(result.Errors[0]));
8     }
9
10    public static IResult HandleRequest(
11        this Result result, Func<IResult> onSuccess)
12    {
13        return result.IsSuccess ? onSuccess() :
14            Results.Problem(ErrorToProblem(result.Errors[0]));
15    }
16
17    private static ProblemDetails ErrorToProblem(IError error)
18    {
19        Console.Out.WriteLine("Error:" + error);
20        return new ProblemDetails();
21    }
22}
```

Listing 6.6: Custom Extension Methods for Handling HTTP Requests in ASP.NET Core: Simplifying Success and Error Response Handling.

The HandleRequest method is an extension method designed to streamline the handling of HTTP requests by automatically processing the result of a service call.

The first HandleRequest method, defined on lines 3-7, handles cases where the service call returns a value. If the operation is successful (IsSuccess is true), it invokes the onSuccess function, passing the successful result value (result.Value) to generate the appropriate HTTP response. If the operation fails, it calls the Results.Problem method to generate a standardized error response, using the first error in the Errors collection to populate the ProblemDetails object.

Similarly, the second HandleRequest method, defined on lines 9-13, handles cases where the service method returns a Result without a value. It follows the same logic: if the operation is successful, it executes the onSuccess function to produce the HTTP response; otherwise, it generates a ProblemDetails response to represent the error.

The private ErrorToProblem method (lines 15-19) converts an IError object into a ProblemDetails object. This method is responsible for translating internal error details into a standardized format that can be returned as part of an HTTP problem response. The method currently includes a line to output the error details to the console for debugging purposes.

By employing these extension methods, we simplify the process of handling success and error scenarios across the entire application. This ensures that all HTTP responses are consistent, well-structured, and easy to manage, which enhances both the maintainability and reliability of the codebase.

User Service

```
1  public interface IUsersService
2  {
3      public Task<Result<Token, Problem>> CreateToken(int nic, string
4          password);
5
6      public Task<Result<UserExternalInfo, Problem>> CreateUser(int nic,
7          string name, string password, Role role);
8
9      public Task<Result<List<UserExternalInfo>, Problem>> GetUsers(string
10         token);
11
12     public Task<Result<Boolean, Problem>> DeleteUser(int nic);
13
14     public Task<Result<UserAccountStatus?, Problem>>
15         GetUserAccountStatus(int userNic);
16
17     public Task<Result<Boolean, Problem>>
18         UpdateUserAccountStatus(UserAccountStatus userAccountStatus);
19
20     public Task<Result<UserWithSimpleNameExternalInfo?, Problem>>
21         CheckNicExistence(int nic);
22
23     public Task<Result<bool, Problem>>
24         AddSuspension(UserSuspensionRequest suspension);
25
26     public Task<Result<bool, Problem>> UpdateSuspension(UserSuspension
27         suspension);
28
29     public Task<Result<UserSuspension?, Problem>> GetSuspension(int
30         userNic);
31
32     public Task<Result<bool, Problem>> DeleteSuspension(int userNic);
33 }
```

Terms Service

```
1  public interface ITermsService
2  {
3      public Task<Result<List<Terms>, Problem>> GetTerms();
4      public Task<Result<Terms, Problem>> GetActiveTerms();
5      public Task<Result<bool, Problem>> SubmitTerms(Terms terms);
6
7      public Task<Result<bool, Problem>> UpdateTerms(int termId, int
8          updatedBy, string newContent);
9
10     public Task<Result<List<TermsChangeLog>?, Problem>>
11         GetTermsChangeLog(int termId);
12 }
```

Form Service

```
1  public interface IFormService
2  {
3      public Task<Result<GetFormOutputModel, Problem>> GetForm();
4
5      public Task<Result<GetFormWithVersionOutputModel, Problem>>
6          GetFormWithVersion(int version);
7
8      public Task<Result<Form, Problem>> EditForm(List<QuestionGroupModel>
9          groups, List<RuleModel> rules, User user);
10
11     public Task<Result<SubmitFormOutputModel, Problem>>
12         SubmitForm(Dictionary<string, IAnswer> answers, int nic, int
13             formVersion);
14
15     public Task<Result<Submission, Problem>> GetSubmission(int id);
16
17     public Task<Result<bool, Problem>> LockSubmission(int submissionId,
18                 int doctorId);
19
20     public Task<Result<bool, Problem>> UnlockSubmission(int
21                 submissionId, int doctorId);
22
23     public Task<UnlockExpiredSubmissions> UnlockExpiredSubmissions(TimeSpan lockTimeout);
24
25     public Task<Result<Review, Problem>> ReviewForm(int submissionId,
26                 int doctorNic, string status, string? finalNote, List<NoteModel>?
27                     noteModels = null);
28
29     public Task<Result<List<Submission>, Problem>>
30         GetPendingSubmissions();
31
32     public Task<Result<Inconsistencies, Problem>> GetInconsistencies();
33
34     public Task<Result<Submission?, Problem>>
35         GetPendingSubmissionsByUserNic(int userNic);
36
37     public Task<Result<SubmissionHistoryOutputModel, Problem>>
38         GetSubmissionHistoryByNic(int nic, int limit, int skip);
39
40     public Task<Result<bool, Problem>>
41         EditInconsistencies(Inconsistencies inconsistencies);
42 }
```

Manual Service

```
1  public interface IManualService
2  {
3      public Task<Result<List<ManualInformation>, Problem>>
4          GetManualInformation(string productName);
}
```

Medication Service

```
1  public interface IMedicationsService
2  {
3      Task<Result<List<string>, Problem>> SearchMedications(string query);
4 }
```

6.2.3 Routes

User Routes

The available endpoints, HTTP method and corresponding operation for all the user routes are available in Table 6.1.

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 6.1: 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 6.2.

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 6.2: 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 6.3: API endpoints related to the form

Medication and Manual Routes

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

Table 6.4: API endpoints related to the form

6.3 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 parallelize password hashing tasks.

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

6.3.1 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 too 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 from side-channels.

6.3.2 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.

6.4 Testing

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.

6.4.1 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.

6.4.2 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.

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.

Integration Tests

TODO

Chapter 7

Conclusions and Future Work

In the final chapter of this report, we will summarize the accomplishments of this project, discuss the challenges encountered, and outline potential improvements and features that were envisioned but not implemented due to time constraints.

7.1 Accomplishments

7.2 Challenges

7.3 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.

It would also be a valuable feature to have the platform 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.

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

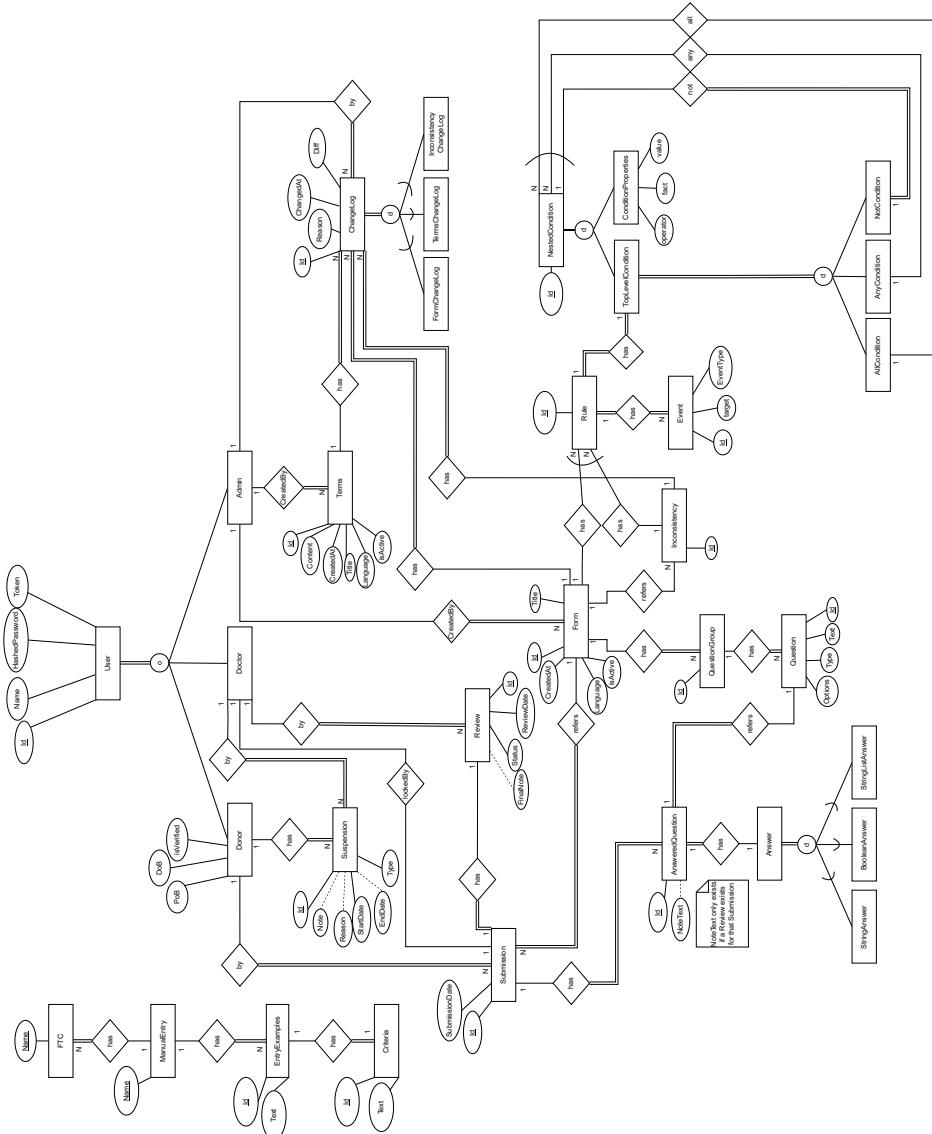


Figure A.1: ER Model.