DADIVA IPO

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

Francisco Medeiros
Luís Macário
Ricardo Pinto

Orientadores: Filipe Freitas, ISEL
João Pereira, COFIDIS

Relatório do projeto realizado no âmbito de Projecto e Seminário
Licenciatura em Engenharia Informática e de Computadores

Junho 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

Orientadores: Filipe Freitas, ISEL

Joao Pereira, COFIDIS

Relatório do projeto realizado no âmbito de Projecto e Seminário

Licenciatura em Engenharia Informática e de Computadores

Junho 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 to great 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.

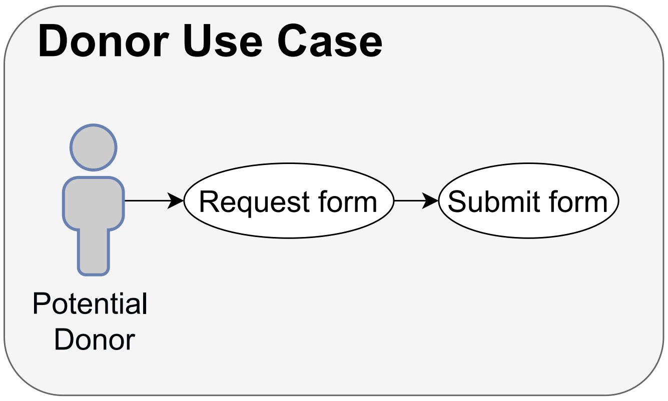


Figure 2.1: Donor use case.

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.

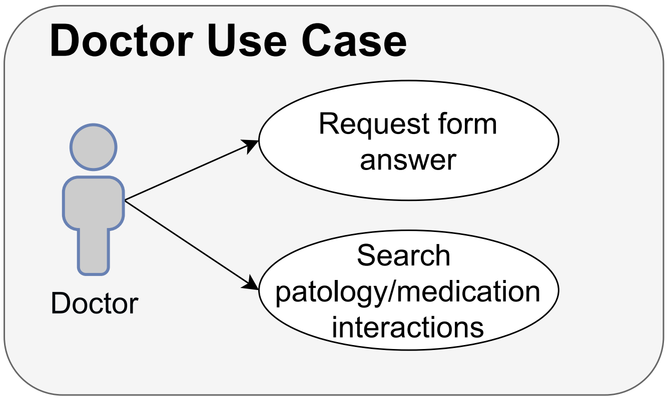


Figure 2.2: Doctor use case.

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.

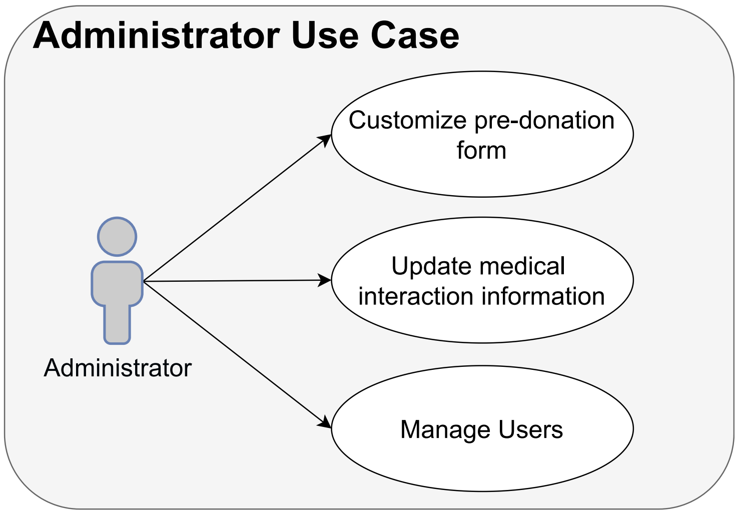

Administrator Use Case
Customize pre-donation form
Update medical interaction information
Manage Users
Administrator

Figure 2.3: Administrator use case.

Chapter 3

Architecture

This chapter provides an overview of the system's components and their interactions. It outlines the capabilities of the project and presents the architecture, entities, and implementation blueprint that have been designed and developed.

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 routes, services, and repositories. The routes handle incoming HTTP requests and call the appropriate service. The services manage data manipulation, validation, and interactions with external APIs or databases. Finally, the processed data is sent to the repositories, which store it in the database.

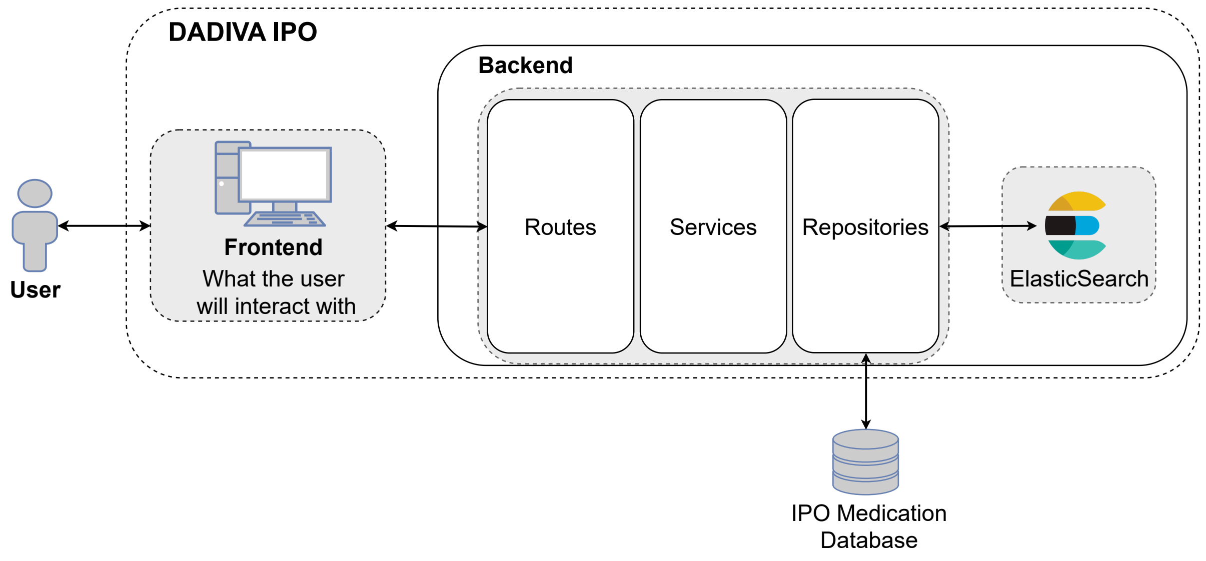


Figure 3.1: Application Architecture, gray squares represent containerized components of the solution.

3.2 Backend Application

The backend application can be abstracted into 3 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;
- repositories: contains the repository layer of the application, uses the ElasticSearch-Client;

The service layer is composed by the following elements:

- form: responsible for form management, such as, creation, requests, submission, editing and deletion;
- search: responsible for medication and pathology interaction information retrieval;
- users: responsible for user management, such as, registration, login, deletion and role management.

The repositories communicate with the database, in which the various data models are divided into specific indexes, such as:

- /form: stores all the form structures;
- /submissions: stores all the user form responses;
- /inconsistencies: stores all the user form inconsistencies;
- /users: stores all the users.

An example of a GET request for the form resource is presented in Figure 3.2

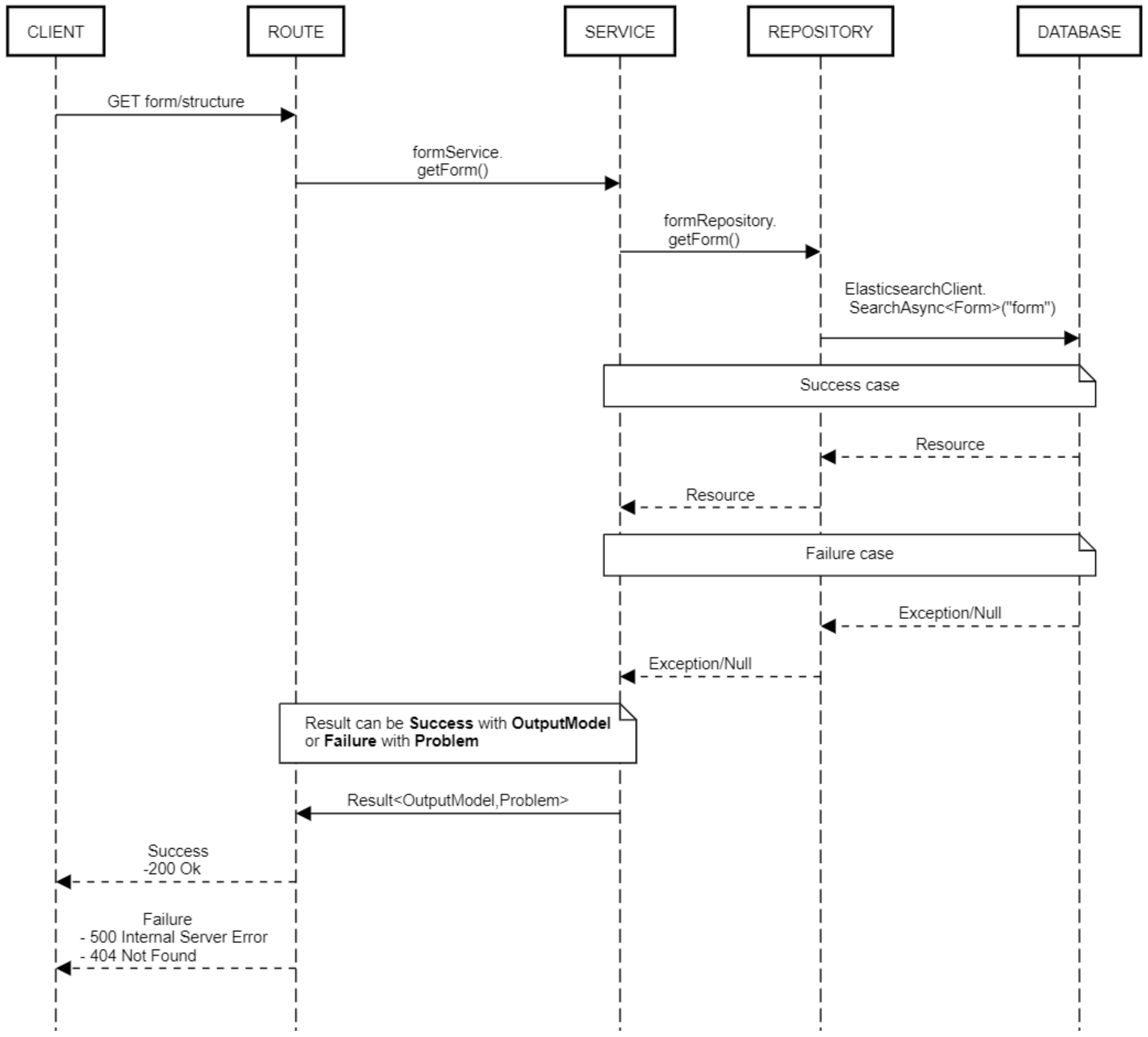


Figure 3.2: Get Form Sequence Diagram.

3.3 Form Data Model and Inconsistencies Data Model

The first approach to solve the dynamic form challenge was to use a data structure formed by pairs of main questions and sub-questions, example presented in Figure 3.3, where a main question can only be answered with boolean values, and one of those values triggers the display of a sub-question which has a certain type of response, such as boolean, dropdown for known multiple answers, and text, to accept user text input.

This approach has several drawbacks. Firstly, it prevents the suppression of further questions, which contradicts the goal of creating a flexible and adaptable solution. Additionally, it conflates questions and rules within sub-questions, leading to a lack of clarity and potential confusion in implementation.

Upon further discussion we settled on using a more complex data structure , exemplified

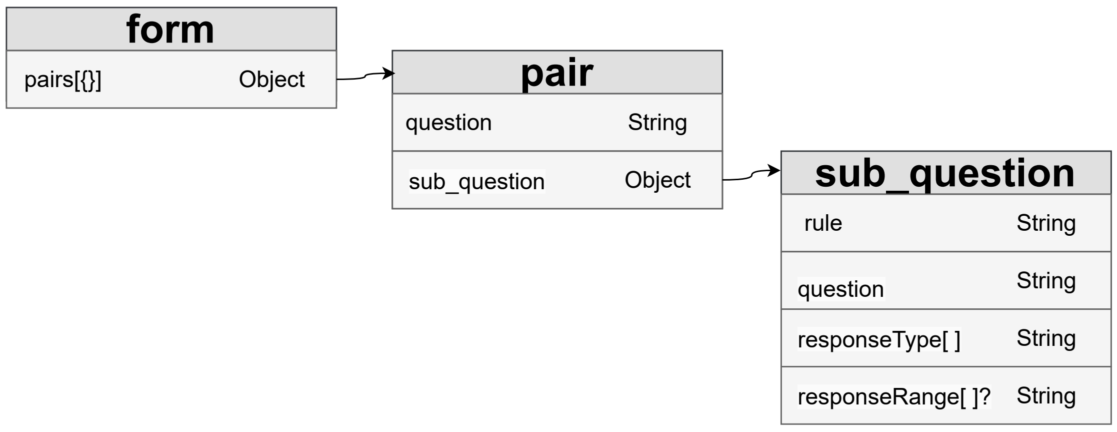


Figure 3.3: First Form Data Structure.

in Figure 3.4, composed by a list of questions and a list of rules.

Each question has an id, the text that composes it, the type of response (boolean, text and dropdown) and can have options that lists all the possibles values for a multiple(dropdown) response.

Each rule has conditions, which can be ”any”,”all” or ”not”, so that, when any, all or none of the conditions are met an event is triggered. Each condition type will have a fact, an operator and a value. In essence, when a question, which is identified by the fact field via it’s id, is answered a condition can true or false depending on the logical operator used, ie equal or notEqual, and the value of the answer. If the condition is true an event is triggered, this event can be to show or hide a subsequent question, this targeted question is identified via the id, supplied in the params field.

This model, specifically the rules field, was chosen as it is part of the JSON-Rules-Engine specification, which is presented in more detail in Chapter 4 and is easily stored and retrieved in a ElasticSearch database.

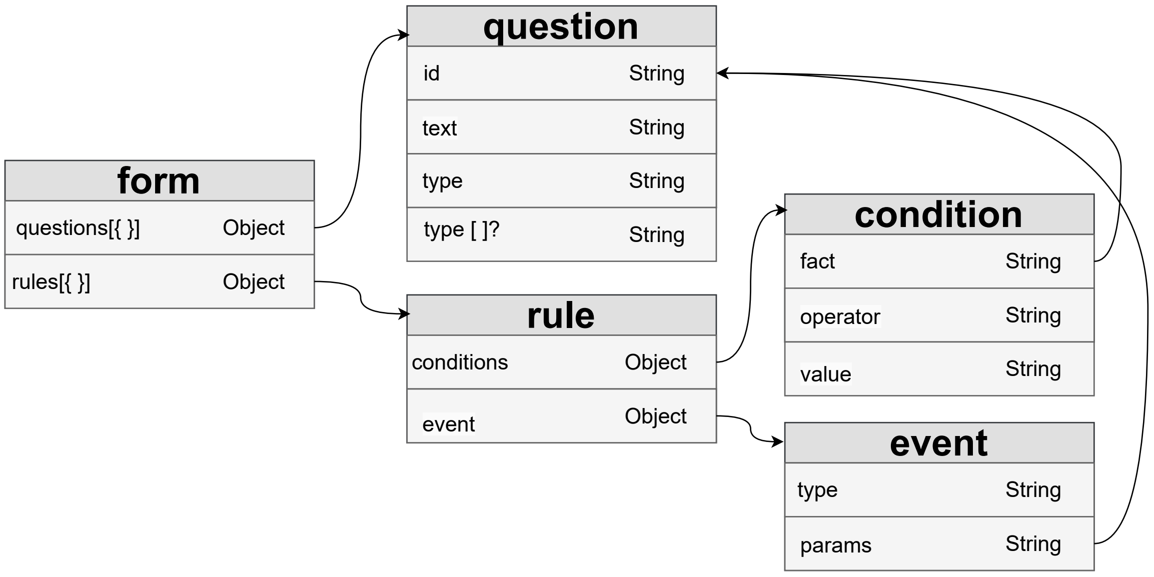


Figure 3.4: Final Form Data Structure.

The inconsistencies data model is compromised of rules, and describes answer combinations that are logical fallacies, ie a donor answering that they're healthy in one question and that they have a chronic disease in another question.

3.4 Submission Data Model

The submission data structure represents and answered form, has such it contains a list of answered questions, each answered question is composed by the question id and the answer, as referenced in Figure 3.5.

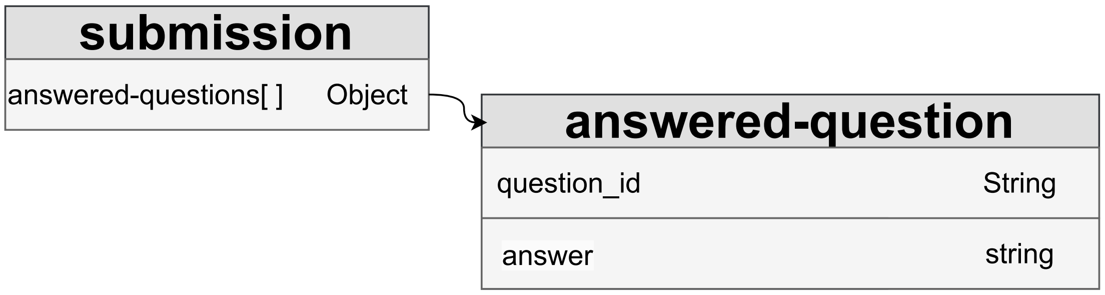


Figure 3.5: Submission Data Structure.

3.5 User Data Model

The user data structure is composed on an unique identifier, the nic, and the user hashed password, as referenced in Figure 3.6.

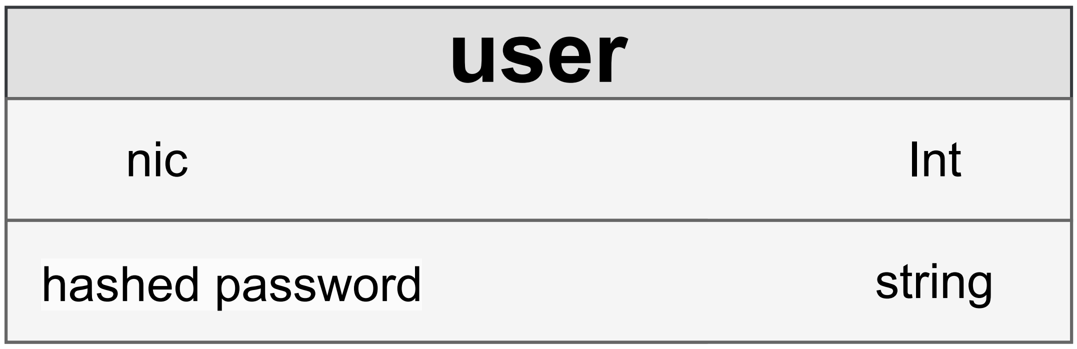


Figure 3.6: User Data Structure.

3.6 IPO Medication Portal

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 where 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 information. However the terms used in the guidelines don't always reflect the available classifications, and, as such, the platform will employ a dictionary that associates the terms used in the guidelines to pharmacotherapeutic classifications. This dictionary will be updatable by the platform's administrators. The fact that the IPST guidelines aren't available in a machine readable format persists.

3.7 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, 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.

During planning some mockups where created of the final result for some pages, the login page is presented in Figure 3.7, the form pages are presented in Figures 3.8, 3.9 and 3.10, the backoffice page is presented in Figure 3.11.

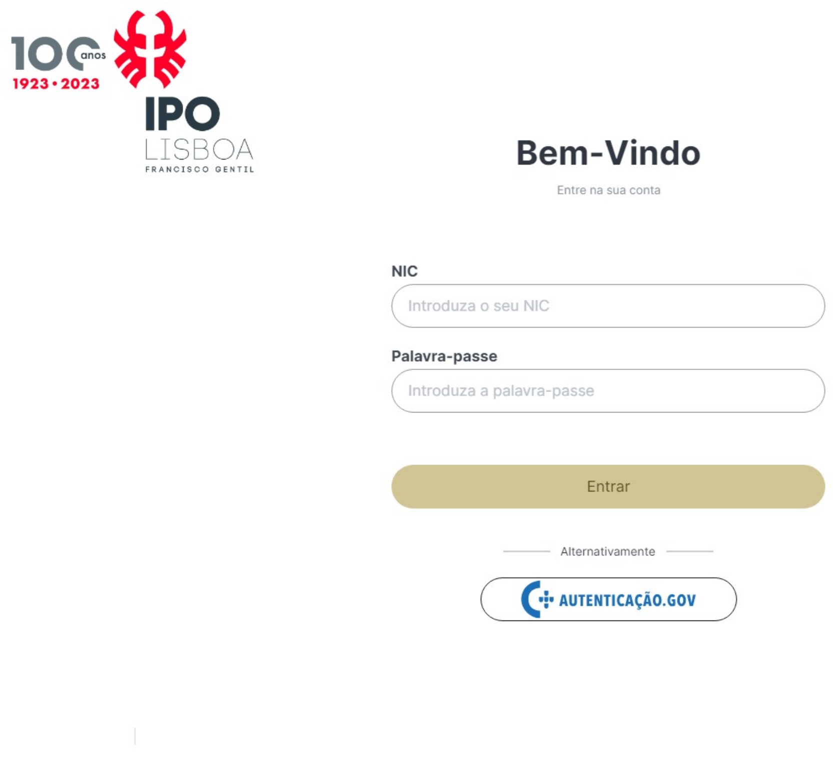


Figure 3.7: Login Page Mock.

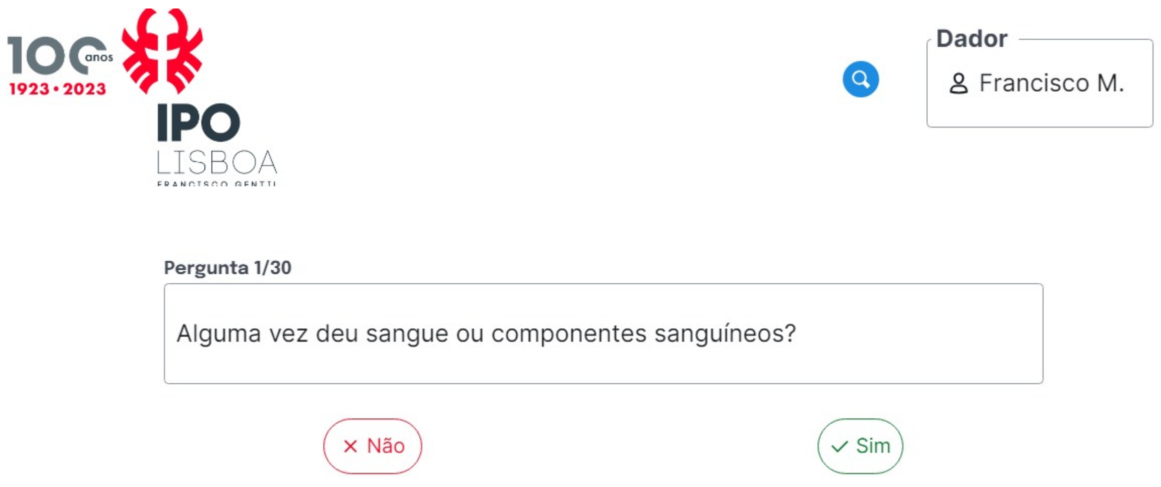


Figure 3.8: Form Page Mock.

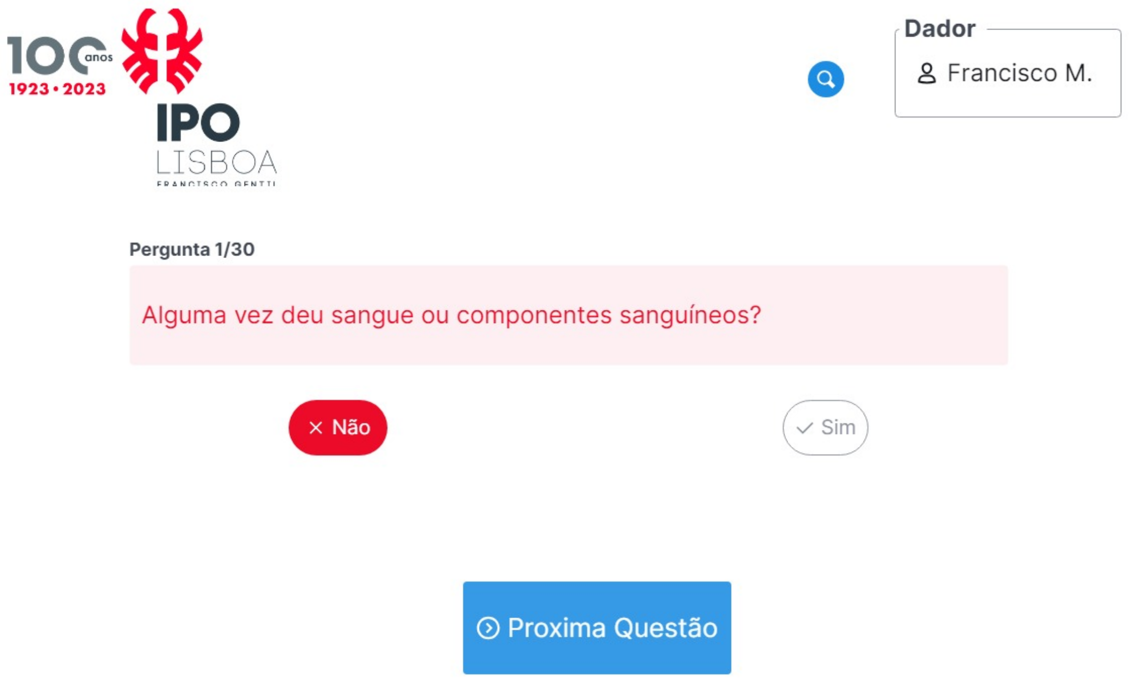
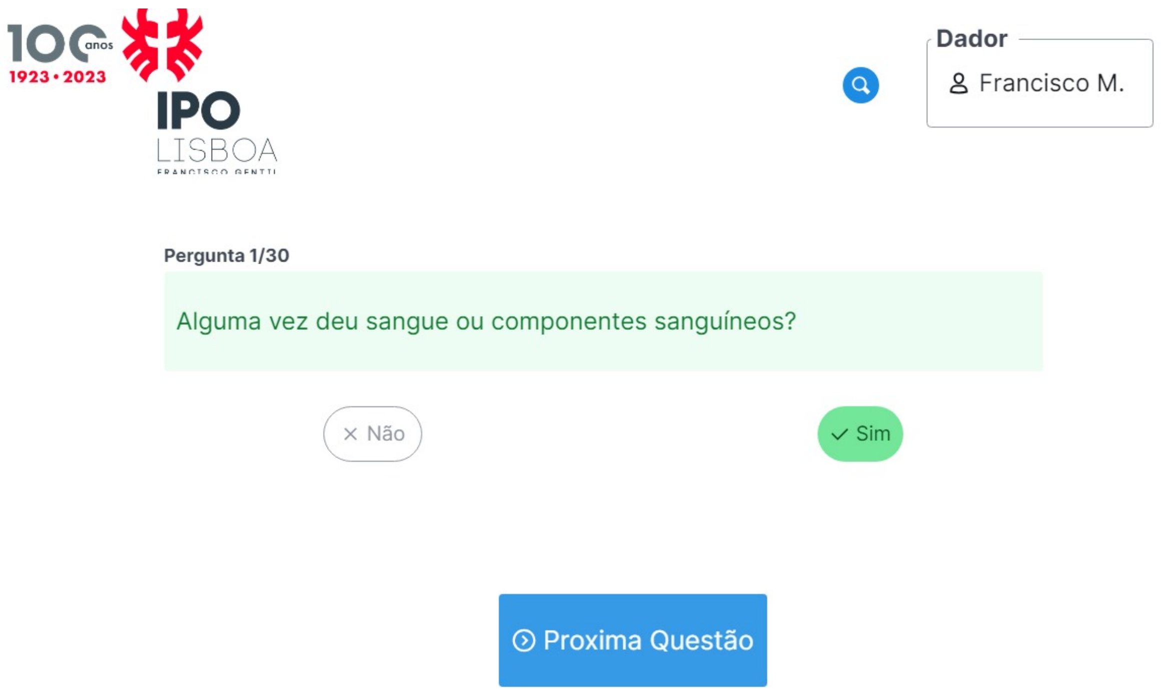


Figure 3.9: Form Page Negative Answer Mock.



Alguma vez deu sangue ou componentes sanguíneos?

Figure 3.10: Form Page Positive Answer Mock.

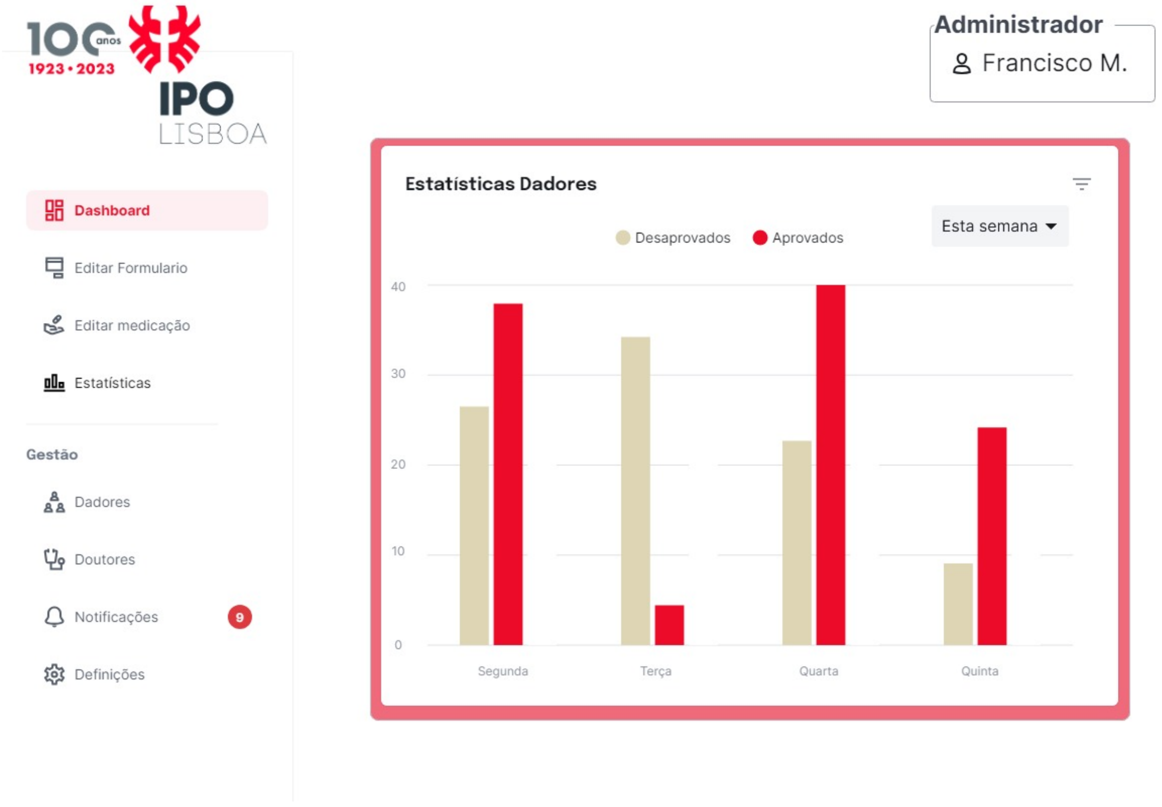


Figure 3.11: Backoffice Page Mock.

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:

- Introduction to Web Programming (Introdução à Programação na Web) - Express.js;
- 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). This framework also allows to create Minimal API's which are controller free API's similar to the Express.js API created during Introduction to Web Programming.

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 [4] 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 [5], 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 [6] 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 3.7.

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

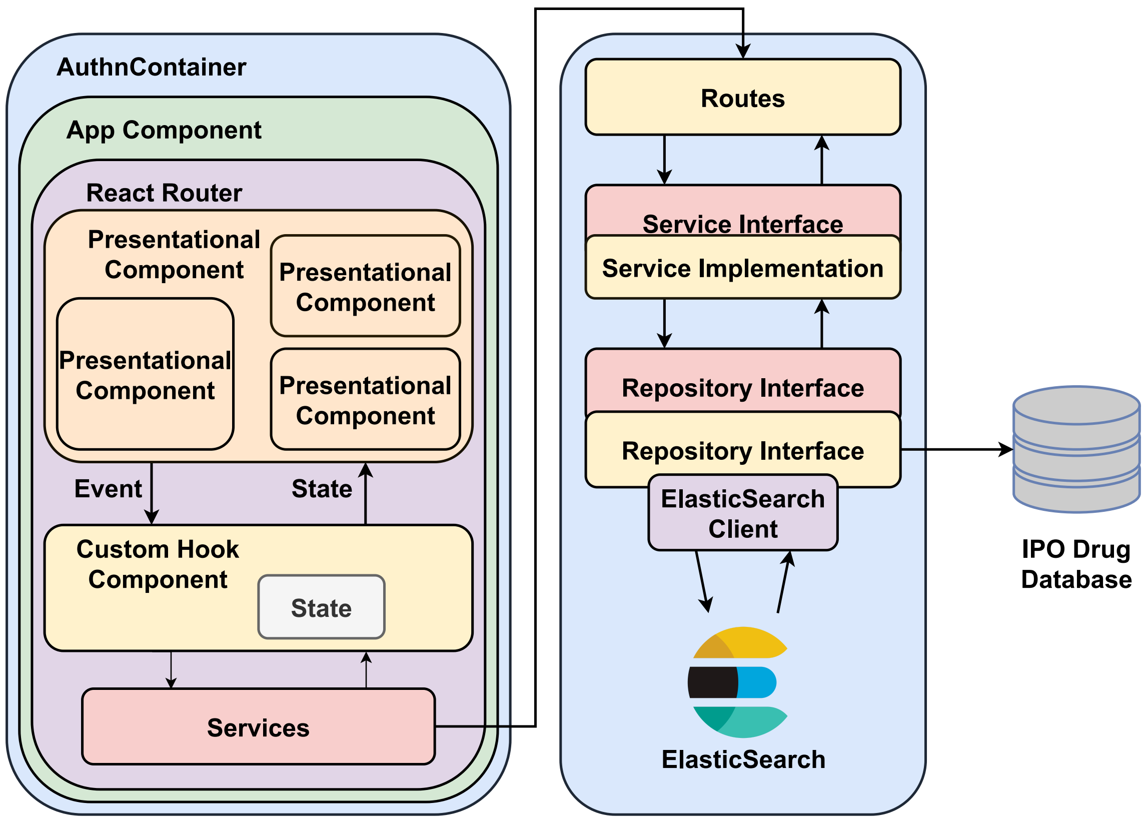


Figure 5.1: Block Diagram of our solution.

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 ElasticSearch database;
- 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 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.

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 Singleton scope, as follows:

```
    builder.Services.AddSingleton<IUsersService, UsersService>();
    builder.Services.AddSingleton<IFormService, FormService>();
    builder.Services.AddSingleton<ISearchService, SearchService>();

    builder.Services.AddSingleton<IUsersRepository, UsersRepositoryES>();
    builder.Services.AddSingleton<IFormRepository, FormRepositoryES>();
    builder.Services.AddSingleton<ISearchRepository,
        SearchRepositoryMemory>();
```

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.

Beyond these, we also used an ElasticClient, however in this service registration, we use a different pattern, instead of using a type and implementation, only the latter was used, this method doesn't allow for multiple implementation, but since there was no need to mock the elastic search database, this feature wasn't needed.

```
var nodePool = new SingleNodePool(new Uri("http://localhost:9200"));
var settings = new ElasticsearchClientSettings(
 nodePool,
 sourceSerializer: (_, settings) =>
  {
    return new DefaultSourceSerializer(settings, options =>
      {
        options.Converters.Add(new AnswerConverter());
        options.Converters.Add(new ConditionConverter());
      });
  });
builder.Services.AddSingleton(new ElasticsearchClient(settings));
```

According to the Elastic Search documentation, as long as the client instance is a singleton, our application database is thread safe.

5.1.3 Repositories

The main idea in the repositories is to allow for CRUD operations, ie create users/forms, read users/forms, update users/forms and delete users/forms, on data that is or will be stored in an Elastic Search database. As such the repositories have a dependency on the ElasticsearchClient mentioned before, and they'll use the same client but target different indexes.

Form Repository

In the case of the form repository the target indexes will be:

1. form : when performing an operation for the form structure, ie the questions and rules;
2. submissions: when performing an operation for the form's answers;
3. inconsistencies: when performing an operation for the form's logical fallacies, described in the form of rules;

And this repository will have the following methods:

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

    public Task<Form> EditForm(Form form);

    public Task<bool> SubmitForm(Submission submission, int nic);

    public Task<Dictionary<int, Submission>> GetSubmissions();

    public Task<Inconsistencies> GetInconsistencies();

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

User Repository

In the case of the user repository the only target index will be users.

And this repository will have the following methods:

```
public interface IUsersRepository
{
    public Task<bool> CheckUserByNicAndPassword(int nic, string
        hashedPassword);

    public Task<bool> AddUser(User user);

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

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

    public Task<Boolean> DeleteUser(int nic);
}
```

5.1.4 Services

Each service is responsible for managing a certain group of requests, ie 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.

Form Service

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

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

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

    public Task<Result<Dictionary<int, Submission>, Problem>>
        GetSubmissions();

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

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

User Service

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

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

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

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

5.1.5 Routes

Form Routes

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

Endpoint	**HTTP Method**	**Description**
/form/structure	GET	Retrieve the last form structure
/form/structure	PUT	Updates the form structure
/form/submissions	GET	Retrieve all the form answers
/form/submissions	POST	Submit form answers
/form/inconsistencies	GET	Retrieve the last form's inconsistencies
/form/inconsistencies	PUT	Updates the form's inconsistencies

Table 5.1: API endpoints related to the form

User Routes

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

Endpoint	**HTTP Method**	**Description**
/users	POST	Creates a new user
/users	GET	Retrieves all the users
/users/login	POST	Creates a new access token
/users/nic	DELETE	Deletes the user with the specified nic

Table 5.2: API endpoints related to the user

These routes are then added to the application as follows:

```
var group = app.MapGroup("/api");

group.AddUsersRoutes();
group.AddFormRoutes();
```

Meaning that the added routes extend the /api endpoint.

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 [7], 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 AuthnContainer, which manages the session information, the Form, Backoffice and EditFormPage, which are presentational components and the useNewForm and useEditFormPage which are custom hooks.

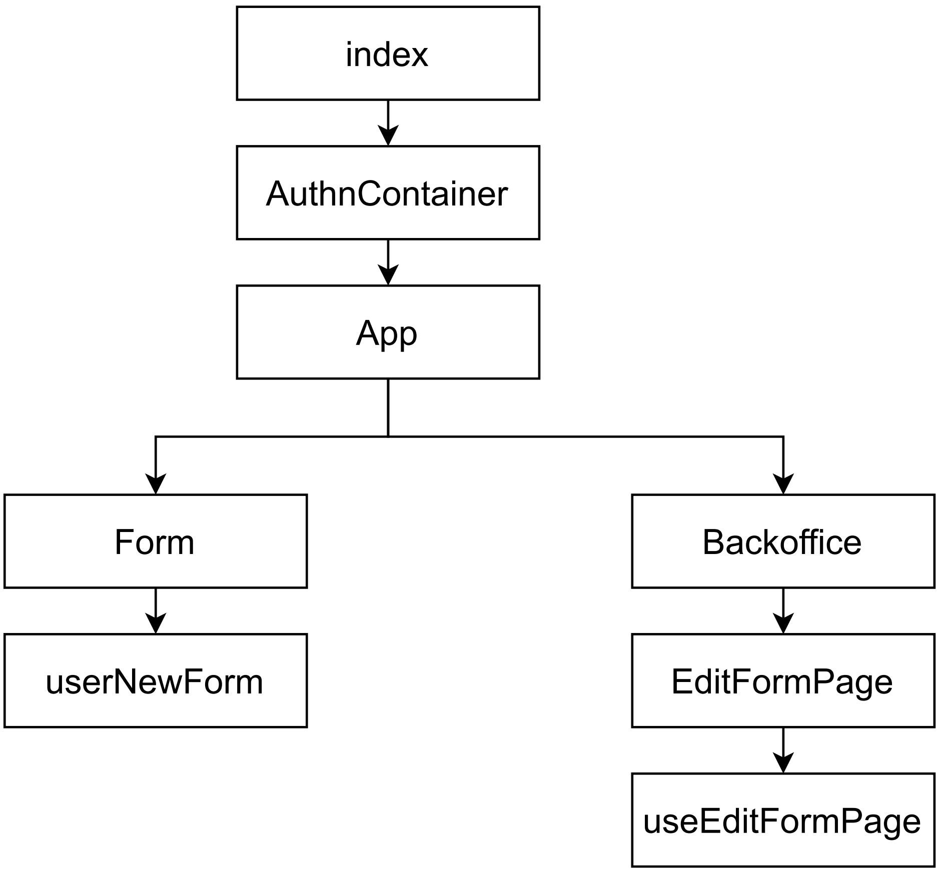


Figure 5.2: Simplified React Component Tree.

5.4.1 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 [8], 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, ie their name and nic. The SessionManager type acts as a wrapper, containing both the session and the methods to manage it, ie 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.

5.4.2 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.

5.4.3 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, ie it's answer causes another question to appear, and a child, ie 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.

Chapter 6

Project's current state

The solution currently offers functionalities to:

- Register and login users.
- Fill out a digital pre-donation form, that encompasses both the main questions of the form as well as relevant follow-up questions, that are triggered according to the user's responses.
- Manage the form's questions as well as the form's rules and manage users.

The project is currently experiencing some delays. These setbacks are primarily due to the inherent complexity of the project and our lack of knowledge in this particular domain.

One of the significant hurdles we have faced is the development of the backoffice's UI. Creating an intuitive and simple visual representation of both the form's questions and the underlying rules has proven to be particularly difficult. Ensuring that the user interface is both user-friendly and accurately reflects the complexity of the rules involved has required substantial effort and ongoing iterative refinement.

Furthermore, we remain committed to delivering a solution that will automatically verify drug and pathology interactions with blood donations. Although this feature is optional, it has the potential to significantly enhance the blood donation procedure. However, the lack of easily accessible, machine-readable information from the IPST manual has posed a significant challenge. Extracting and integrating this critical data in a way that ensures accuracy and reliability has been a complex task.

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

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

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

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

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