BABEŞ-BOLYAI UNIVERSITY CLUJ-NAPOCA

FACULTY OF MATHEMATICS AND COMPUTER SCIENCE

SPECIALIZATION COMPUTER SCIENCE

DIPLOMA THESIS

EasyHelp

The creation of an application suite that aids the process of blood donation

Supervisor

Lector Dr. DRAGOŞ Radu

Author

GEORGESCU Ştefan-Paul

2019

UNIVERSITATEA BABEŞ-BOLYAI CLUJ-NAPOCA

FACULTATEA DE MATEMATICĂ ŞI INFORMATICĂ

SPECIALIZAREA INFORMATICĂ

LUCRARE DE LICENTĂ

EasyHelp

Crearea unei suite de aplicații ce automatizează procesul de donare de sânge

Conducător ştiinţific

Lector Dr. DRAGOŞ Radu

Absolvent

GEORGESCU Ştefan-Paul

2019

Table of Contents

Table of Contents **1**

1. Introduction 3

1.1. Motivation 4

1.2. Required Medical Information and Field Research 4

1.3. Project Description 5

1.4. Existing Attempts at Solving the Proposed Problem 7

1.5. Personal Contribution 7

1.6. Chapter Structure 8

2. Theoretical Aspects 10

2.1. Version Control 10

2.1.1. Git 10

2.1.2. gitHub 11

2.2 Server Application 11

2.2.1 General Presentation of a RESTful API 11

2.2.2. Java Programming Language 12

2.2.3. Spring Framework 13

2.2.4. Spring Boot 13

2.2.5. Spring Security with JWT Token 14

2.2.6. Communicating With the Database 14

2.2.7. PostgreSQL Database 14

2.2.8. Automatic Database Mapping with Hibernate 15

2.2.9. IntelliJ 15

2.2.10. Dependency Managers and Gradle 15

2.2.11. Heroku Online Hosting Platform 16

2.3. Web Client 16

2.3.1. General Presentation of an Angular 2 Application 17

2.3.2. Typescript 17

2.3.3. HTML in Angular 18

2.3.4. CSS and SCSS 19

2.3.5. Angular Routes and Route Guards 19

2.3.6. Angular 2 Dependency Manager 19

2.3.7. Angular Dependency Injection 20

2.3.8. HTTP Library 20

2.3.9. Visual Studio Code 21

2.4. iOS Client 21

2.4.1. General Presentation of the iOS Operating System 21

2.4.2 Swift Programming Language 21

2.4.3. Important Design Patterns in Swift 22

2.4.4. iOS Application Lifecycle 23

2.4.5. UIViewController Lifecycle 24

2.4.6. XCode 25

2.4.7. Testing and XCode Server 25

2.4.8. XCTest and the Page Object Model 26

2.4.9. Cocoapods Dependency Manager 27

3. Application Implementation 28

3.1 Server-Side Application 28

3.1.1 Architecture 28

3.1.2. Application Domain Model 29

3.1.3. Proposed Response Standard 32

3.1.4. Database Creation and Communication 34

3.1.5. Security 35

3.1.6. Endpoint Mapping with Spring 37

3.1.7. Class Instantiation with Spring 38

3.1.8. Using Environment Variables for Local and Distribution Environment 38

3.2 Web Application 38
3.2.1. Functionality 38
3.2.2. Server Communication 39
3.2.3. Managing User Roles and Server Authentication 41
3.2.4. Creating the HTML Views 42
3.2.5. Online Hosting and Environment 44
3.3. iOS Application 44
3.3.1. Functionality 44
3.3.2. Server Communication 45
3.3.3. Implemented Services 46
3.3.4. Individual Targets for Different Application Versions 47
3.3.5. Usage of Important Design Patterns 49
3.3.6. The implementation of UI Tests using XCTest 50
3.3.7. Usage of XCode Server for Continuous Integration 51
4. Conclusion 53
4.1. SWOT Analysis of the Implemented Application 55
4.2. Further Improvements 55
4.2.1. Scalability 55
4.2.2. Security 55
5. Bibliography 55
6. List of Figures 59

1. Introduction

1.1. Motivation

Blood is one of the most important resources of mankind, and it is solely produced by the human body. It cannot be manufactured in a laboratory and it cannot be paid for. In hospitals, blood is used in many of the activities carried out by doctors, therefore, at times, surgery and treatment are postponed due to the scarcity of blood stocks.

Even given this fact, Romania continues to stay at the bottom of the list when it comes to donating blood; only 2% of the adult population contributes [45]. Besides the small number of donors, a general issue is the fact that not all of these donors give blood regularly. Blood has a limited shelf-life, and if donations do not happen regularly, blood stocks diminish.

Regular donations are vital for the transfusion system, which is, in its turn, vital for numerous individuals with health problems. Given all of this, the necessity of implementing a solution which would help people understand the issue at hand while offering aid in carrying out the process is obvious.

1.2. Required Medical Information and Field Research

In order to replicate the blood donation system of Romania as accurately as possible, research has been conducted in this field. The following information was used when developing the application.

The Blood Donation Process

Initially, each donor must complete a form containing questions about one's lifestyle and medical history. Afterwards, they will go through a screening process where a doctor will establish the donor's aptitude for donation. If they are deemed qualified, the medical staff will proceed with collecting the blood. If the donor is donating for the first time, the blood group is determined at this point before the harvesting process begins.

During the harvesting process, a small sample of the donor's blood is kept for running tests. On each blood sample, two tests are run: immunohematology and blood transmissible diseases (in Romania, HIV, Hepatitis B, Hepatitis C, Syphilis, HTLV, ALT). If any sample fails a test it is marked as unusable and the donor is privately notified.

If all tests came out with good results, then the harvested blood will go through a process of filtration, centrifuge and separation to obtain the three blood components: red blood cells, plasma and platelets. These separated components are then distributed to transfusion centres throughout the country.

The Blood Components Lifetime

The blood components cannot be kept in storage forever, their shelf life being limited. Platelets can be kept for up to five days, red blood cells for 42 days, while plasma for up to one year.

Donation Restrictions

Certain real-life restrictions were considered when creating the project. First of all, a donor must wait at least 72 days before donating again. Secondly, a limit of donations one gives each year exists: four donations for women, and five for men.

Besides the factual information obtained, field research provided useful insight into the needs of the personnel in charge of harvesting and distributing the blood. The personnel from the Cluj-Napoca Donation Centre have been extremely helpful in the designing phase of the project, providing information regarding their needs which allowed the author to properly reflect reality in this software application.

1.3. Project Description

When planning the functionalities of the product, the key objective followed was the full integration of the blood process within the system, covering all the steps from donor to patient. In order to make this a possibility, multiple user types had to be taken into account when designing the solution.

Donors

Donors interact with the system via a mobile application available for iOS. In the application they have the possibility of registering, logging in and configuring their profile. Once logged in the application, donors can book a donation and fill in the pre-screening form normally available at the donation centre. Donors can view their donation history in order to review blood test results.

Doctors

Within the web application available to them, doctors can manage their current patients with the purpose of requesting blood for them. They can then manage the blood requests by accepting commitments from donation centres, or marking a certain blood commitment as arrived at their hospital. The blood commitments feature is described in the Personal Contributions chapter.

Donation Centre Personnel

The Donation Centre Personnel (DCP) is the middleman in the blood donation process, ensuring that blood given by donors reaches the patients. The features available to them are also implemented via a web application. Part of the features accessible to a DCP are related to the interaction with donors. A DCP can manage donation bookings at his donation centre and can mark a donor as arrived for donation. Afterwards, a DCP can input blood test results and the results from the process of dividing whole blood into components.

The other features available for DCP represent the management of blood requests made by doctors. A DCP can commit to a blood request, and after receiving confirmation from a doctor that the commitment is accepted, can mark blood as shipped from the donation centre. A DCP is also able to manage the blood stocks within his donation centre by discarding blood which has spent too much time without being used.

Admin

The admins of the system are in charge of approving doctor and DCP accounts such that no fake accounts find their way into the system. They are also responsible with managing the

hospitals and the donation centres registered in the application. An admin also has the possibility of banning a doctor or a DCP account in the case of illicit use.

A server application ensures that all the client applications mentioned above have access to the system.

1.4. Existing Attempts at Solving the Proposed Problem

Currently in the Romanian market there are only two applications that fall in the same problem space.

The first one, Donorium, is an application for donors in which they can track their donation history. At the time of writing, you could not book donations to donation centres, but it was planned as an upcoming feature. You can register a donation by scanning a QR Code within selected donation centres. By donating you obtain points, allowing you to rank against your friends.

The second application is called Donez450 and provides functionality similar to Donorium, not including the verification step for registering donations. At any point you can add a donation to your donation history, as long as the 72 days have passed from the previous donation.

Neither of these two applications provide coverage for the whole blood donation process, hence are limited in usability at a large scale.

1.5. Personal Contribution

The solution proposed differentiates itself from existing implementations by integrating the blood's whole journey within the system, while at the same time attempting to innovate the way doctors and DCPs interact when requesting blood.

Normally, a doctor will forward an on-paper request to a donation centre, whose personnel will coordinate the dispatch of blood either from his donation centre, if the whole quantity requested can be delivered, or will try to get help from a different donation centre.

The way the proposed solution tackles this problem is the following. Doctors can forward a request which can be seen by any DCP in the country. Then, DCPs can forward commitments for that blood request - either enough to fulfil the request or partially - translating to them promising the doctor that amount of blood. Doctors can then evaluate the available commitments and choose the better one, based on distance, blood type compatibility and other factors, such as time before expiry. It is up to the doctor to decide which donation commitment to accept. After that, it is just a matter of sending and receiving the blood.

A side-feature of this mechanism is enhanced control over where blood goes to waste. All the data that the application stores can be used to create an analysis of all the blood flowing through the country, making sure that none goes to waste.

Having this whole process monitored by the application also allows for different ways of incentivising people to donate. On the donor application, if the blood type of the donor is known, he will see how many patients need his exact blood type in the country.

1.6. Chapter Structure

In the following chapters the process of implementing the applications will be presented, followed by the conclusion and outcome of this project.

Firstly, the theoretical aspects which were required to be known before the development process are outlined. Information about RESTful APIs, Token Authorisation and the Spring Framework were required for developing the server application, and is outlined in **Chapter 2**. In the same chapter, the Angular 2 Framework and its particularities are introduced, together with the basics of developing an application on iOS. Since all code was kept under version control, an introduction to Git is also provided.

Secondly, **Chapter 3** will present the process of implementing the three software products required for the system to work. An analysis of the architecture and domain model of the server application is available. The aspects of managing multiple users and restricting features are covered for the Angular 2 Framework. Implementing and managing multiple application versions, together with continuous integration and automated UI tests, are presented for the iOS platform.

Finally, in the **conclusions chapter**, the outcomes of the project will be discussed, the implemented software application will be evaluated and possible improvements will be outlined.

2. Theoretical Aspects

2.1. Version Control

Version control is a category of software tools with the purpose of managing changes to source code over time. A version control system (VCS) keeps track of every modification brought to any tracked file, allowing developers to compare versions of code, which eases the process of identifying mistakes. This is only one of the features provided by a modern VCS. More functionalities will be introduced in the following subchapter, describing the chosen VCS.

2.1.1. Git

Git is an open source Distributed VCS. Control System means that Git acts as a content tracker. It can be used to store any type of content, but it is usually used to store code due to its features. It is a VCS, since Git provides users with a history of file changes over time. It is Distributed due to the fact that Git stores a remote repository on a server and every user has a copy of the repository on their machine.

There are multiple workflows which can be applied to Git, the most common in everyday usage being a Subversion-Style Workflow [22]. Git will not allow developers to push changes to the remote repository if there are new changes that have not yet been fetched. This ensures consistency within the distributed system.

The features of Git revolve around development teams due to its history. It was designed specifically for the team developing the Linux kernel in 2005 [38]. Git's unique characteristic compared to other Source Control Management systems is branching and merging. It's branching model encourages development teams to have multiple branches, each serving a different purpose. You can have one branch that is only for production code and one branch in which features are developed. Each feature can have its own branch, which will be deleted once the feature is merged into the development branch. This was just an example, since Git allows for a multitude of scenarios.

Git was chosen over other version control systems due to its popularity and previous knowledge of the author on how to use it.

2.1.2. gitHub

GitHub is an online service for hosting Git remote repositories. The first commit was pushed in October 2017, and at the time of writing hosted over 100 million repositories owned by over 36 million users [23]. It is one of the go-to websites when a developer wants to host version control repositories online due to its features focused on collaboration.

2.2 Server Application

In the following subchapters the theoretical aspects of implementing a RESTful API with Java are presented.

2.2.1 General Presentation of a RESTful API

Representational State Transfer, or REST, has been introduced by Roy Fielding in his Ph. D. dissertation in the year 2000, titled "Architectural Styles and the Design of Network-based Software Architectures" [40]. According to the thesis, "REST is intended to evoke an image of how a well-designed Web application behaves: a network of web pages (a virtual state-machine), where the user progresses through an application by selecting links (state transitions), resulting in the next page (representing the next state of the application) being transferred to the user and rendered for their use." The constraints introduced by REST are uniform interface, statelessness, client server, cacheable, layered system and code on demand, the latter being optional. In order to better understand REST, we will explain each constraint.

The principal concern regarding the client server constraint is the separation of concerns. By separating the interface from the data storage, we improve portability across multiple platforms.

Statelessness means that each request from client to server must contain all the necessary information for processing the request. The server should not use any stored context.

By adding cache constraints, we ensure that the server load is manageable, improving efficiency and scalability. Data within a request can be marked as cacheable or non-cacheable, letting the client know if said data is reusable.

Uniform interface constraint states that all the components within a REST architecture must share a single, prevailing interface. This means that the interface for a component needs to be as generic as possible (usually HTTP). It simplifies and decouples the architecture, which enables each part of the architecture to evolve independently.

The layered system style allows an architecture to be composed of layers arranged in a hierarchy. This is achieved by constraining each component's behaviour so that each component's knowledge of other components in the system is limited to the immediate layer they communicate with.

The code on demand constraint is optional. It allows for the REST client functionality to be extended by downloading and executing code. This can be used to add features after deployment.

API stands for Application Programming Interface. It is a set of functions and procedures allowing the creation of applications that access the features or data of an operating system, application, or other service [19]. Since RESTful services usually communicate via HTTP, a REST API is a library based on the HTTP standard. The CRUD operations (Create, Read, Update, Delete) are mapped to HTTP methods Post, Get, Put and Delete.

2.2.2. Java Programming Language

Java is a programming language first released by Sun Microsystems (later acquired by Oracle) in 1995. It is one of the most powerful programming languages, presenting the following characteristics: high-level, object-oriented, portable and open source. The most interesting of these characteristics is the portability, achieved by the fact that programs written in Java do not run on the operating system, but on a virtual machine called Java Virtual Machine (JVM). Since JVM is available on every operating system, Java implements the "write once, run anywhere" concept, hence supporting the portability characteristic.

This makes Java one of the desired languages when implementing services that need to be hosted online, and it is also the reason it was chosen for this project.

2.2.3. Spring Framework

Spring Framework is a Java platform which provides developers with an infrastructure in order to support the development of Java applications.

The framework itself is divided into modules, allowing developers to choose which modules they need. The modules of the core container include configuration classes and a dependency injection mechanism. Beyond that, Spring Framework provides foundational support for multiple architectures, including transactional data and persistence, web, servlet-based Spring Model-View-Controller (MVC) and others [42].

2.2.4. Spring Boot

Spring Boot is one of the projects Spring offers. It takes away the infrastructure configuration tasks from the developers. The following terms are key concepts used in Spring Boot: bean, autowiring, dependency injection, inversion of control, and application context.

Bean is a term used to define different objects that are managed by the spring framework.

Dependency injection is injecting some class as a dependency for another class, used when instantiating objects.

Autowiring is the process of identifying the required dependencies, finding matches for those dependencies in the beans repository and populating them. It is the way the Spring Framework handles dependency injection.

Inversion of Control is a process in which the framework takes control over the objects that would otherwise need to be controlled with custom code written by the developer.

The application context is the part of spring boot where all the beans are created and managed, hence being one of the most important modules.

2.2.5. Spring Security with JWT Token

Spring Security is a module within the Spring Framework tasked with authentication and access-control. The default behaviour is suited for web applications, as it uses cookie-based authentication. When building a REST API, Spring Security can be configured to work with JSON Web Tokens (JWT).

JWT is an open standard that defines a compact and self-contained way for securely transmitting information as a JSON object. It is secure since it can be verified that it was signed with the HMAC algorithm [17]. It is called self-contained due to the fact that it holds all the information required to identify the owner of the token and its properties. It is often used in authentication, since it can be used as a header on a HTTP request, due to its small size.

2.2.6. Communicating with the Database

In order to communicate with the database, Java uses Java Database Connectivity (JDBC) to connect and execute queries [30]. It is the lowest level API used when working with databases and Java. JDBC uses drivers to connect to the database and also provides support for managing transactions. It provides an interface through which statements can be created, modified and executed. If a developer only uses JDBC, then all queries must be written in the query language correspondent to the database used.

The Java Persistence API (JPA) is a specification for accessing, persisting and managing Java objects. It does not provide implementation, but an interface through which developers can query the database using Java.

2.2.7. PostgreSQL Database

PostgreSQL is an object-relational database management system available on all major operating systems. It is a very mature DBMS including conformance with a large part of the SQL standard and a support from extension modules [35].

In order to connect to a PostgreSQL database, a Java application will use the PostgreSQL JDBC Driver [36] in order to perform database operations using Java code.

2.2.8. Automatic Database Mapping with Hibernate

Hibernate is a JPA certified Object Relational Mapping (ORM) framework that utilizes Hibernate Query Language in place of SQL to provide high level querying capabilities allowing users to interact directly with the database without writing queries. Hibernate provides support for performance tuning such as lazy loading of properties and various fetching and cascading strategies for handling relationships [29].

Being an ORM framework, its main purpose, besides implementing the JPA interface, is the automatic mapping of objects to tables in the database. By using a set of annotations, developers can specify relationships between objects and choose a table creation strategy. Hibernate interprets those annotations and creates the database scheme accordingly.

2.2.9. IntelliJ

IntelliJ IDEA is a popular Java Integrated Development Environment (IDE). It provides a robust combination of development tools, such as intelligent coding assistance, code analysis and numerous refactoring options. The IDE’s functionality is continuously extended by users and third parties via plugins, offering support for Spring and Hibernate [31]. The support it offers for these two frameworks made it a good choice for this project.

2.2.10. Dependency Managers and Gradle

Using third party libraries is a common practice in development, motivated by the fact that in many occasions one part of an application, such as networking, is common with many other applications. Hence, recreating the same functionality and performance as a third-party software is often impossible.

Dependency managers make use of third-party software accessible. Developers do not have to manually add the source code of the library in their projects. Adding the name of the

library within the dependency manager will automatically make that library available in the project.

Gradle is an open-source build automation tool designed to be flexible enough to build almost any type of software. It runs on JVM, making it a good match for Java applications, and also has great IDE support in IntelliJ.

Its build process is based on tasks which the build tool organises in Directed Acyclic Graphs, representing the dependencies between tasks, thus determining the order in which the tasks must run [26].

Gradle has built-in support for dependency management. It can look for dependencies in several jar repositories and download the libraries into the project [25].

Compared to other build tools like Apache Maven, Gradle is faster [24], thus it was chosen for this project.

2.2.11. Heroku Online Hosting Platform

Heroku is a cloud platform that allows developers to build, deploy and host applications online. Originally, it only supported Ruby, but at the time of writing, it offered support for Java, Node.js, Clojure, Python, PHP, Perl, and Scala [21].

The features which make the Heroku platform appropriate for this project are the integrated PostgreSQL data storage add-on [28] and the possibility to automatically deploy the application on new commits, if the code is hosted on GitHub [27].

2.3. Web Client

In the following subchapters the theoretical aspects of implementing a web application using the Angular 2 Framework are presented.

2.3.1. General Presentation of an Angular 2 Application

Angular 2 is a development framework for building web applications using HTML and a programming language, which is compiled in JavaScript, called TypeScript. Applications created in Angular 2 are separated into services and components, each component managing a HTML template.

The basic building block of an Angular 2 application is the module, which is used to group together related components and services. Every Angular 2 application has a root module, which is used to launch the application. Several other modules can be introduced per feature.

Component classes interact with templates and services, the latter containing the business logic. Components have properties and event handlers which the framework automatically binds to the view declared within the component declaration [1].

Services are another major building block, providing functionality that would not fit within a component, such as retrieving data from a server [1].

In order to specify to the framework which classes to use as components, we use decorators. The "@Component" decorator tells Angular what component one class represents and what HTML template to use when building the view, and other configurations [9].

Another important feature when implementing a web application is the presence of lifecycle hooks [6]. Within a component, a set of methods is provided in order to handle certain events. This way, developers can control the application more thoroughly.

2.3.2. Typescript

Typescript is the preferred language for Angular 2. It is a programming language developed by Microsoft in order to make JavaScript development easier. One of the most relevant features of TypeScript compared to JavaScript is the optional type system [33]. The ability to define types makes it easier for programmers to maintain a large code base and reduces the ambiguity of parameters and return types present in JavaScript. Code written in

TypeScript is transformed to JavaScript code, which is almost identical to the original, missing only the object types.

Since TypeScript is a superset of JavaScript, meaning that it builds upon JavaScript, programmers benefit from all the features JavaScript provides, such as building the project when saving a file, ensuring speed during development.

2.3.3. HTML in Angular

HTML templates are used to render a specific view owned by a component. Additionally, Angular provides developers with a template syntax which is parsed by the framework when building the view. The template syntax introduces support for directives, which are used to transform the HTML document on supplied code expression. Directives such as “ngIf” and “ngFor” are used to layout HTML code based on values from the component class [1].

Besides directives, Angular introduces data binding as one of the central features of the framework. It provides multiple ways of communication between the template and the component class. There are two types of data binding. Event binding allows the application to respond to user input by updated data in the component class. Property binding allows developers to send values from the component into the HTML template [2].

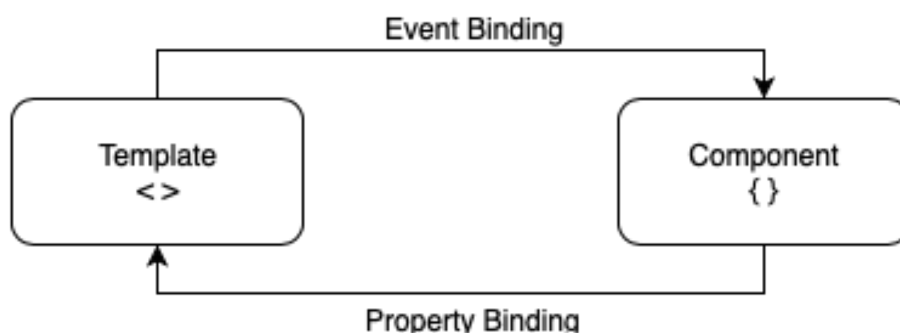

Figure 2.1. Data Binding in Angular

2.3.4. CSS and SCSS

Cascading Style Sheets (CSS) is a language used to describe the presentation of a document written in a markup language like HTML. HTML tags are linked to CSS selectors and the style specified in the CSS file is applied to the corresponding content within the HTML tag.

SCSS is one of the two syntaxes supported by Syntactically Awesome Style Sheets (SASS), which is a CSS extension language written to make CSS more accessible [41].

2.3.5. Angular Routes and Route Guards

The Angular Router provides a complete routing library, allowing the possibility of multiple router outlets, multiple path matching strategies and route gourds [8]. Links on pages can be bound to the router and it will navigate to the appropriate view when the user clicks on the link. “The Router Outlet is a directive from the router library that is used like a component. It acts as a placeholder that marks the spot in the template where the router should display the components for that outlet.” [8]

Normally, a user can navigate anywhere within an Angular application, which is often not the desired behaviour. Hence, the framework provides the developer with Route Guards [8] which have the purpose of resolving a condition or action before navigating to the desired URL. It can be either used to prefetch data for the component or to check if the desired page can be viewed by the authenticated user.

2.3.6. Angular 2 Dependency Manager

The components used by Angular applications are packaged as npm packages, and can be installed using the npm command line interface [7]. All installed packages are identified using a file called “package.json”. It is used by all the projects within the workspace.

When building the project on a new machine, the command “npm install” must be run. This will find the package.json and will download and install all the dependencies required by the application.

2.3.7. Angular Dependency Injection

Having already discussed what dependency injection is, it is worth mentioning that Angular provides its own Dependency Injection Framework [3], allowing developers to increase efficiency and modularity.

Dependency is most often used on services, which must be annotated with @Injectable, which tells Angular that this class can be used as a dependency on another component. Injection is done without any boilerplate code. All the developer has to do is add the service to be injected in the constructor of the component.

2.3.8. HTTP Library

The Angular 2 HTTP library is based on JavaScript's "XMLHttpRequest" [5]. The library is provided as an injectable service and supports all general HTTP requests [4]. The decision of using XMLHttpRequest as a starting point was made as a result of its following capability: it allows for asynchronous HTTP requests [34].

Another important part of the Angular 2 HTTP library is the third party RXJS library, which introduces Observables. An observable can be linked to an event handler, which can later be subscribed to. Normally, components subscribe to results of asynchronous methods called from services. Only after an entity subscribed to the result of a request is the request sent.

Another useful feature provided by the HTTP library is the possibility of intercepting requests and responses. "With interception, you declare interceptors that inspect and transform HTTP requests from your application to the server. The same interceptors may also inspect and transform the server's responses on their way back to the application."[5] Interceptors can be used with multiple purposes, such as logging, caching and token forwarding.

2.3.9. Visual Studio Code

Visual Studio Code is a source code editor developed by Microsoft and available for free. It has built in support for JavaScript, TypeScript and Node.js, with multiple extensions for other languages. It provides integrated debugging and version control, but is missing features compared to an IDE, such as building and running the project, which must be done from the command line.

2.4. iOS Client

In the following subchapters the theoretical aspects of implementing a mobile application for the iOS platform are introduced.

2.4.1. General Presentation of the iOS Operating System

The iOS Operating System originates in June 2007 and has seen great progress during these 12 years. It is programmed mainly in C, C++, Objective C and since Swift appeared, implementation moved to it [39]. The iOS platform is made up of several layers. The first layer is the Core OS, containing the Kernel, File System, etc. The next layer is formed of Core Services, such as SQLite, Networking and Core Location Services. The third layer is the Media Layer, containing Animation Support, Core Audio, Video and Image support. The final layer is Cocoa Touch (or Foundation), it contains Views, Controllers, Touch Handlers and is the only layer developers can work with [43].

2.4.2 Swift Programming Language

The Swift programming language was introduced at Apple’s 2014 Worldwide Developer Conference and was presented as a replacement for the original iOS development language, Objective C. According to the TIOBE index, it surpassed its predecessor in popularity within the first months of 2016 [44].

Swift is a general purpose, multi-paradigm, compiled programming language. “It’s an industrial-quality programming language that’s as expressive and enjoyable as a scripting

language." [10] In late 2015 Swift became open-source under Apache 2.0 license - therefore anyone could contribute to the future of Apple's main development language.

2.4.3. Important Design Patterns in Swift

In software engineering, a design pattern is a general reusable solution to a commonly occurring problem in software design [37]. Therefore, in order to ease development, the usage of design patterns is highly encouraged. By following design patterns, developers are not required to reinvent the wheel and can use tested methods for solving problems.

The Model-View-Controller (MVC) design pattern is one of the most used design patterns in iOS applications. Its main advantages are high cohesion and low coupling, providing high code reusability and contributes to applying the separation of concerns design principle. The pattern "assigns objects in an application one of three roles: model, view, or controller. The pattern defines not only the roles objects play in the application, but also the way objects communicate with each other. Each of the three types of objects is separated from the others by abstract boundaries and communicates with objects of other types across those boundaries. The collection of objects of a certain MVC type in an application is sometimes referred to as a layer—for example, model layer." [13]

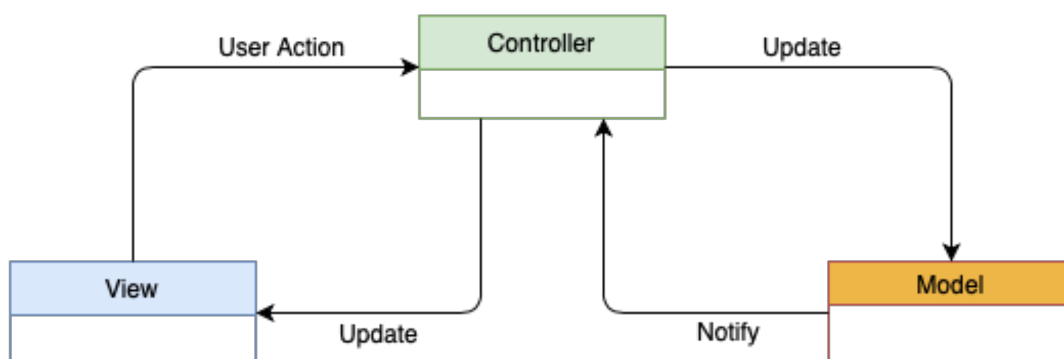

Figure 2.2. Diagram of the MVC Design Pattern

The Delegation pattern is another common pattern in iOS development, heavily used in Apple's frameworks. "Delegation is a simple and powerful pattern in which one object in a program acts on behalf of, or in coordination with, another object. The delegating object keeps a reference to the other object—the delegate—and at the appropriate time sends a message to it. The message informs the delegate of an event that the delegating object is about to handle or has just handled. The delegate may respond to the message by updating the appearance or state of itself or other objects in the application, and in some cases, it can return a value that affects how an impending event is handled." [12]

2.4.4. iOS Application Lifecycle

Every iOS application has five states in its lifecycle: inactive, active, background, suspended and not running [11].

- Not running - when the application was not started or has been stopped;
- Inactive - when the application is entering the foreground state, but cannot process events.
- Active - when the application enters the foreground state and can process events.
- Background - when the application goes into the background, and if there is executable code, it will execute, and if there is no executable code or the execution is complete, the application will be suspended.
- Suspended - when the backgrounded application goes into a frozen state, unable to execute code. It will terminate if the system runs out of memory.

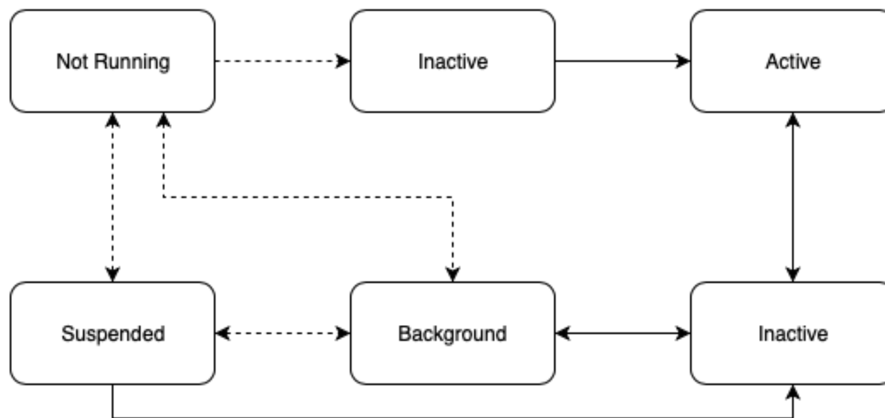

Figure 2.3. iOS Application Lifecycle

In order to respond to changes to the state of the application, handler methods can be implemented. For example, the method applicationDidEnterBackground(_ application: UIApplication) is called every time the application is going to switch to the background state. These handlers are useful for saving local data or stopping running services. The methods are found within the UIApplicationDelegate protocol. A protocol is Swift's version of an interface.

2.4.5. UIViewController Lifecycle

The main building block of an iOS application is the UIViewController. On launch, depending on your option on how to build views (from code, interface editor), the application will require you to provide a root view controller. The view controller is the controller in the MVC pattern. It is responsible with instantiating a view and managing the content that view displays. It is vital for iOS development to understand the UIViewController lifecycle, which is made up of four states: appearing, appeared, disappearing, disappeared [14].

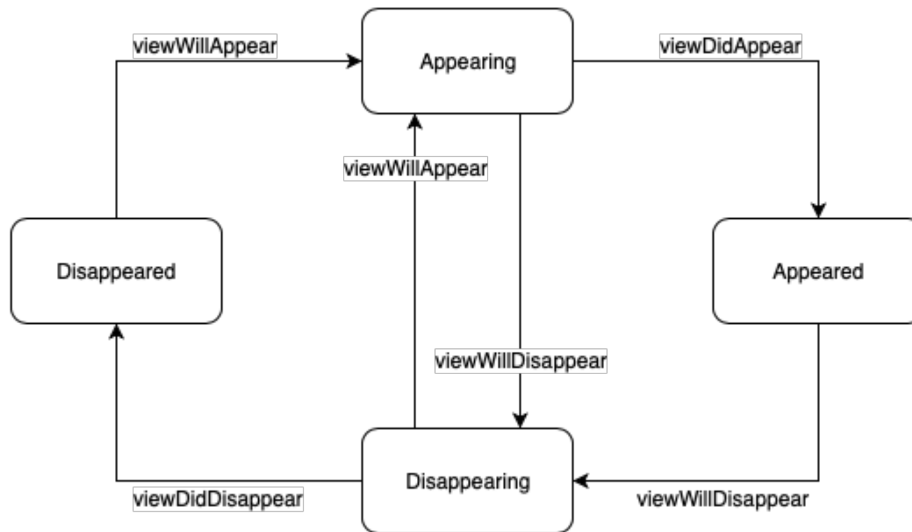

Figure 2.4. iOS UIViewController Lifecycle

When implementing an application, you cannot instantiate a UIViewController, you can only subclass it. When the visibility of its views changes, a view controller automatically calls its own methods, so that subclasses can respond to the change. A method like viewWillAppear(_:) can be used to prepare your views to appear on screen, while viewWillDisappear(_:) can be used to save changes or other state information.

2.4.6. XCode

XCode is the only IDE which provides all the tools required for iOS development, hence it has been used for the development of this project. It provides developers with easy access to the Software Development Kit, an interface builder, multiple debugging tools such as the usual debugger, memory allocation graph, view hierarchy and simulators.

2.4.7. Testing and XCode Server

Testing is very well intertwined in the workflow within XCode. “The Test Navigator makes it incredibly easy to jump to any test in your project, execute an individual test, or execute a group of tests. The Assistant editor has [...] views that automatically track which tests

exercise the code you are presently editing, keeping your tests and code in sync at all times.” [15]

XCode Server is a built-in tool for XCode, starting at XCode 9. It allows for one computer to act as a build server within a local network. Multiple XCode users can access the server from within XCode and run integrations.

Integrations are run using XCode Server Bots, which are highly customisable and can also run tests during integrations. They connect to the remote repository in order to fetch the latest code before building. They can be scheduled to integrate daily or on new commits in the repository.

2.4.8. XCTest and the Page Object Model

XCTest is a testing framework provided by Apple. It encapsulates unit tests, performance tests and UI tests [16]. The written tests are integrated with the XCode project, providing all the functionalities mentioned in the previous chapter.

The Page Object Model (POM) is a design pattern popular in test automation. The design pattern proposes the creation of page objects which serve as an interface for a view of the tested application. It provides methods for interacting with the page and for retrieving page content [32].

POM enhances test maintenance by grouping functionality. If multiple test scripts work with the same view, communicating with the view will not have to be implemented in each test, since the behaviour is encapsulated within the page object class. This also leads to enhanced test maintenance, since changes in the UI layout, for example, switching to a different input type, will only need to be handled within the page object class and not in every test script using that view.

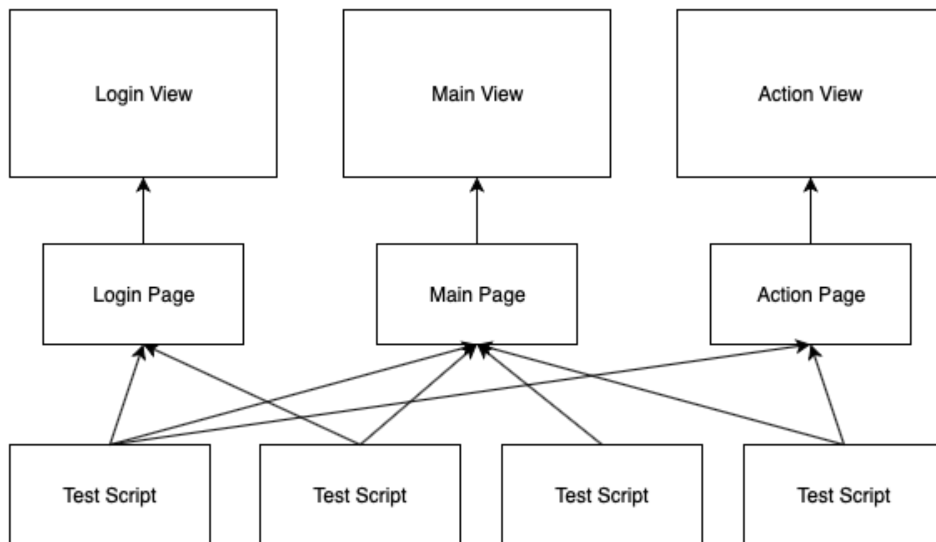

Figure 2.5. Example POM Diagram

2.4.9. Cocoapods Dependency Manager

For Swift and Objective-C projects, the Cocoapods dependency manager is the default choice of developers. At the time of writing this paper, it hosted over 61 thousand libraries and was used in over 3 million applications [20]. As with most dependency managers, usage is simple. Add the name of the library within a file Cocoapods manages, run a command in the terminal and the dependency is downloaded and linked to the application target.

3. Application Implementation

3.1 Server-Side Application

The decisions made when implementing the server-side application were restricted by the functional and non-functional requirements of the project. In the software development process, a set of requirements serves as a guideline, used by programmers when designing and implementing an application.

Functional requirements describe what the application can do. Firstly, since the application serves multiple user types, restrictions needed to be applied to what users can access different resources, so JWT Token filtering was implemented (details in Chapter 3.1.5.). A certain user can only access the endpoints which expose operations for that user type. In order to fully implement this requirement, the application also handles the authorisation process.

The server application provides CRUD operations for a set of data objects: donation bookings, stored blood, blood requests, etc. It exposes an API that allows both the web and mobile applications to retrieve and send data.

3.1.1 Architecture

The application is implemented following a three-layer architecture which respects the separation of concerns principle: the repository layer, the service layer and the controller layer.

The repository layer is responsible for communicating with the database in order to perform CRUD operations. In order to achieve this, a repository class had to be implemented for each object that is stored in the database, hence managed by the Hibernate framework.

The service layer encapsulates the business logic of the application, taking care of data processing and other side-tasks, such as sending notifications to mobile users or sending emails to users on account verification.

The controller layer is where the API operation handlers are implemented. A specific method is fired whenever its API URL is called, it handles the task and then provides the caller with a response. All this is done using HTTP requests, making the application usable from a wide variety of clients.

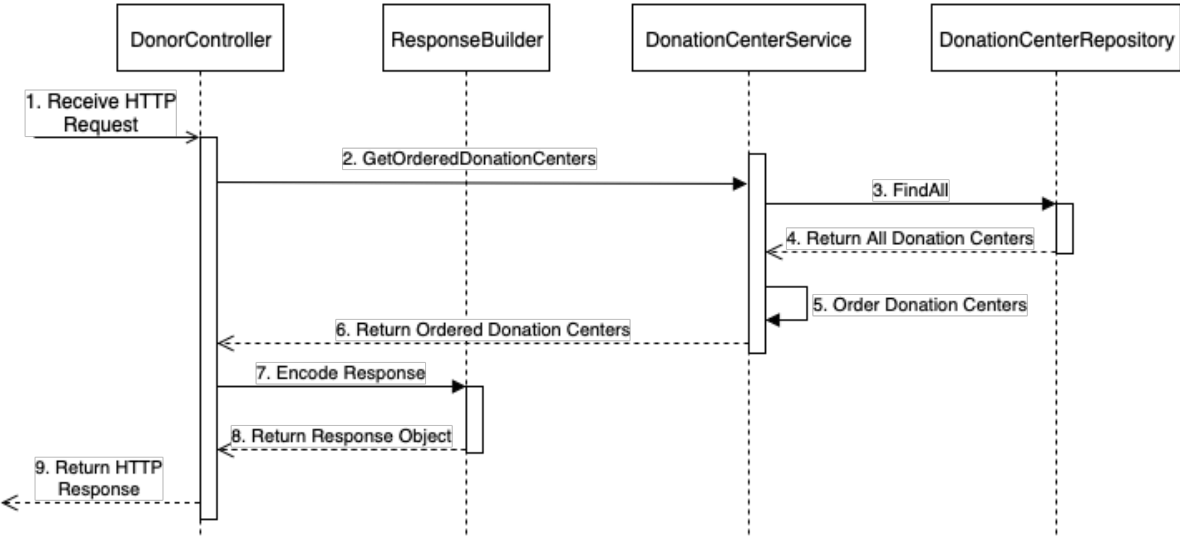

Figure 3.1. Server Operation Sequence Diagram

3.1.2. Application Domain Model

The first task when designing the domain model was putting together the design for the user types. A general ApplicationUser class was created in order to hold the common fields of all users, and this class was extended by each user type. The Doctor class and DonationCenterPersonnel class inherit from the PartnerUser class, which reflects the fact that admins must first approve these types of accounts. Fields have been removed from the following figure for clarity.

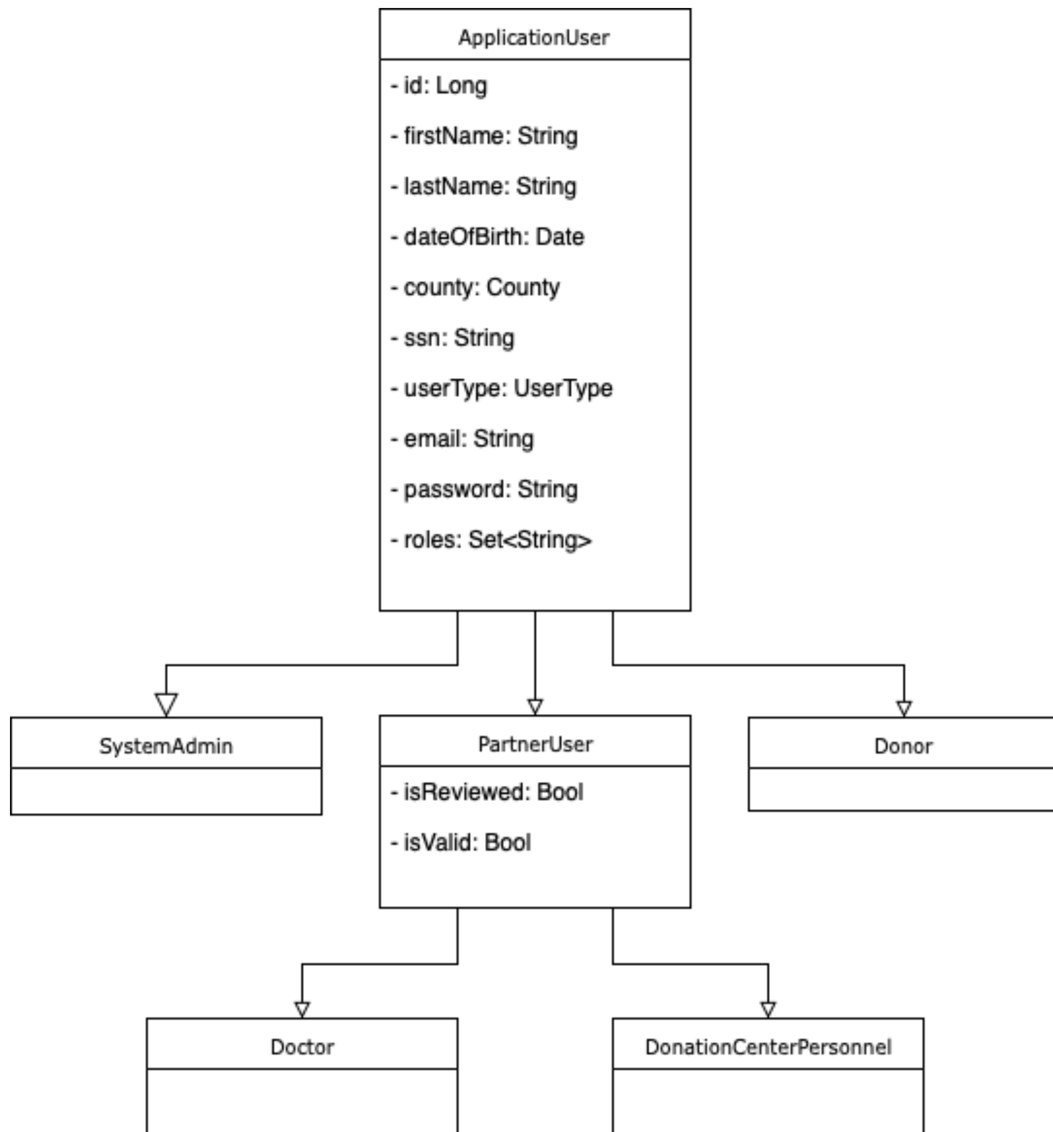

Figure 3.2. UML Class Diagram for Classes Implementing the User Types

Given the features presented earlier, they can be split into two groups: the ones involving donors and DCPs, and the one involving DCPs and doctors. In order to better follow the domain model, the objects created will be discussed with respect to those two parts of the application.

The features revolving around donors and DCPs are related to booking donations and associating test results to those donations after the blood was harvested. In order to implement those functionalities, the following objects and relationships have been created.

Note that all classes that do not inherit from ApplicationUser inherit from a class named BasedEntity, which provides the objects with the Id field. This has been left out of the diagrams for clarity.

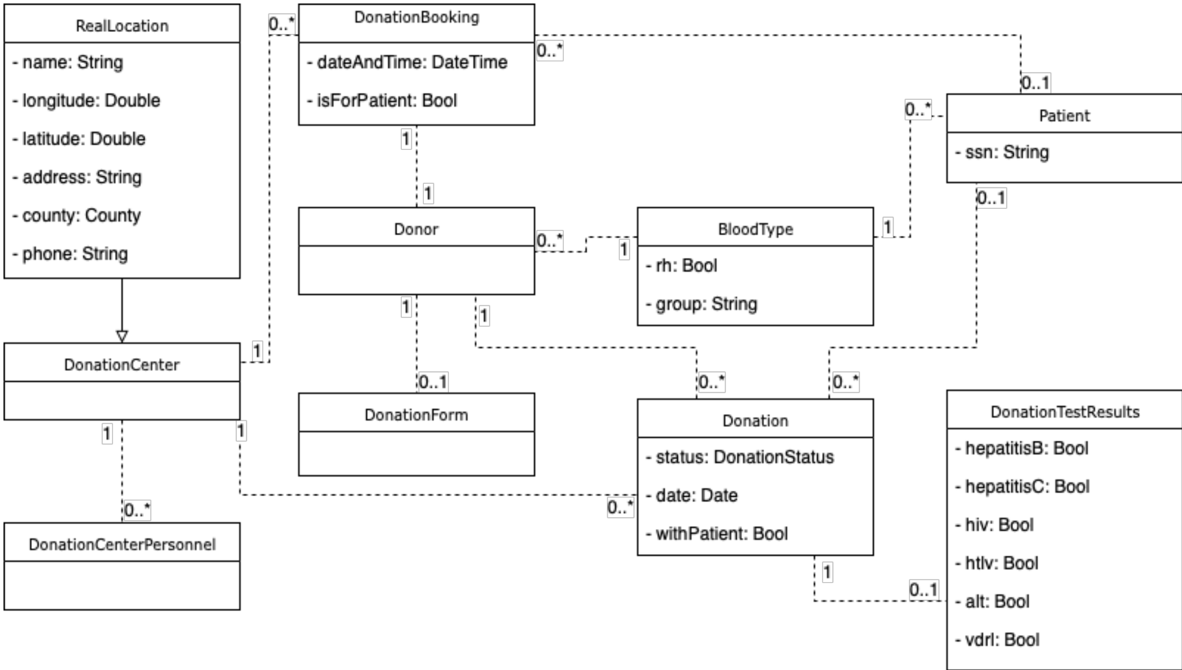

Figure 3.3. UML Class Diagram for Classes Implementing Donor and Donation Center Personnel Related Features

The features involving Doctors and DCPs are the ones regarding blood requests for patients and the process through which the requests get resolved, which was explained in Chapter 1.5. The following figure represents the class diagram of the objects required to implement the aforementioned features.

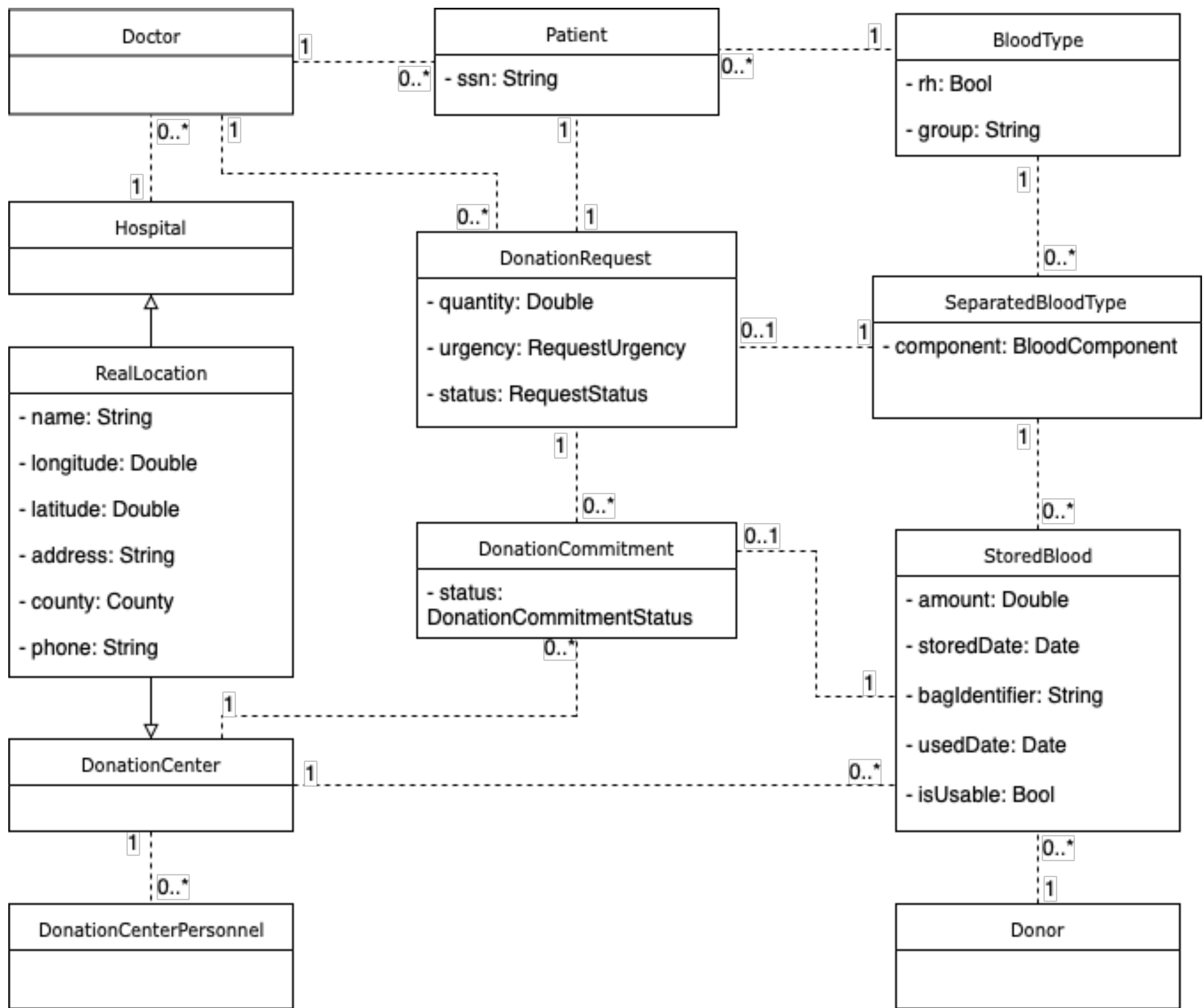

Figure 3.4. UML Class Diagram for Classes Implementing Doctor and Donation Center Personnel Related Features

3.1.3. Proposed Response Standard

As previously mentioned, the application serves clients via HTTP requests. Parameters are added to the body of the request and encoded as a JSON object by the client. The server returns a response and the body of that response is also a JSON object.

Due to the fact that multiple clients are using the API provided by the server application, a way of encoding the response was required in order to ensure that responses do not differ from endpoint to endpoint. Hence, all response objects contain a status flag, which is used by the client to determine if the operation requested finished with success. This will flag

errors that happen during the operation execution on the server, and will not reflect any faults in communication or authorisation. The latter are signalled via HTTP error codes.

In order to build the response in this way, a suite of response builders was implemented, each taking care of one type of response (list, single object, etc.). The appropriate type of builder is called from a general response builder utility class, which uses method overloading to choose what builder to call based on the parameters received from the caller.

In the case of a faulty operation (trying to remove an entity by id, where that id is not in the database), the status flag is set to false, and the rest of the body will be represented by a string which explains the error.

```json
{
    "status": false,
    "exception": "You have already made a request for this patient"
}
```

Figure 3.5. Response Example for an Unsuccessful Operation

In the case of a successful operation, the rest of the body will be represented by a JSON object which will represent the response of that request.

```json
{
    "status": true,
    "object": {
        "newId": 3
    }
}
```

Figure 3.6. Response Example for a Successful Add Operation

```
{
    "status": true,
    "object": {
        "objects": [
            {
                "id": 1,
                "name": "Spital 1",
                "address": "Adresa 1",
                "county": "CLUJ",
                "longitude": 25.3316741,
                "latitude": 43.967557,
                "phone": "0730123123"
            },
            {
                "id": 2,
                "name": "Spital 2",
                "address": "Adresa 2",
                "county": "ARGES",
                "longitude": 23.57949,
                "latitude": 47.65331,
                "phone": "0730123123"
            }
        ],
```

Figure 3.7. Response Example for a Successful Get all Hospitals Operation

This design came as a solution to the necessity of parsing the response in each client. Having a general skeleton as the base of the response body allowed for easier response parsing on the iOS client, which will be detailed later.

3.1.4. Database Creation and Communication

In order to create the database tables, Hibernate has been used to map the managed objects to a relational database. To achieve this result, multiple annotations were used on the domain model.

The "@Id" annotation was used in the class BaseEntity in order to mark the primary key. Each class that needed to be stored in the database was annotated with "@Entity" to signal Hibernate that it is a managed object.

Relationships are annotated with one of the following: "@ManyToOne", "@OneToMany", "@OneToOne" and "@JoinColumn". The annotations also receive parameters, which allowed for fine-tuning the relationship and the fetching strategies between objects. For

example, in order to map the one-to-many relationship between a Doctor and a Patient, the following declarations were required:

```
@OneToMany(mappedBy = "doctor", orphanRemoval = true, cascade = CascadeType.PERSIST, fetch = FetchType.LAZY)
private Set<Patient> patients = new HashSet<>();
```

Figure 3.8. ORM Annotations Example in Doctor.java

```
@ManyToOne
@JoinColumn(name = "fk_doctor")
private Doctor doctor;
```

Figure 3.9. ORM Annotations Example in Patient.java

The Spring Framework was used in order to also handle database communication. By extending the JpaRepository interface provided with the framework, all the CRUD methods required are automatically provided. The underlying implementation uses the EntityManager from JPA together with the Hibernate ORM Framework.

3.1.5. Security

In order to secure the API with JWT and limit which users can access specific endpoints, a set of classes had to be implemented.

Firstly, the WebSecurity class is used to specify which user roles can access the endpoints. In the same class, the JwtConfigurer is added to the security settings. The JwtConfigurer class is responsible with adding the JwtTokenFilter class as a request filter. This translates to the fact that each time a call is made to the application, the method doFilter from the JwtTokenFilter class is called. Finally, the class JwtTokenProvider is used to encode, decode and check tokens for validity.

In practice, on each login, the server application will generate a token, which will be returned to the client. This token must be used as the value for the authorisation header on all subsequent calls to the server as proof of successful authentication. Trying to send a request to the server without a token will result in a HTTP status code unauthorised.

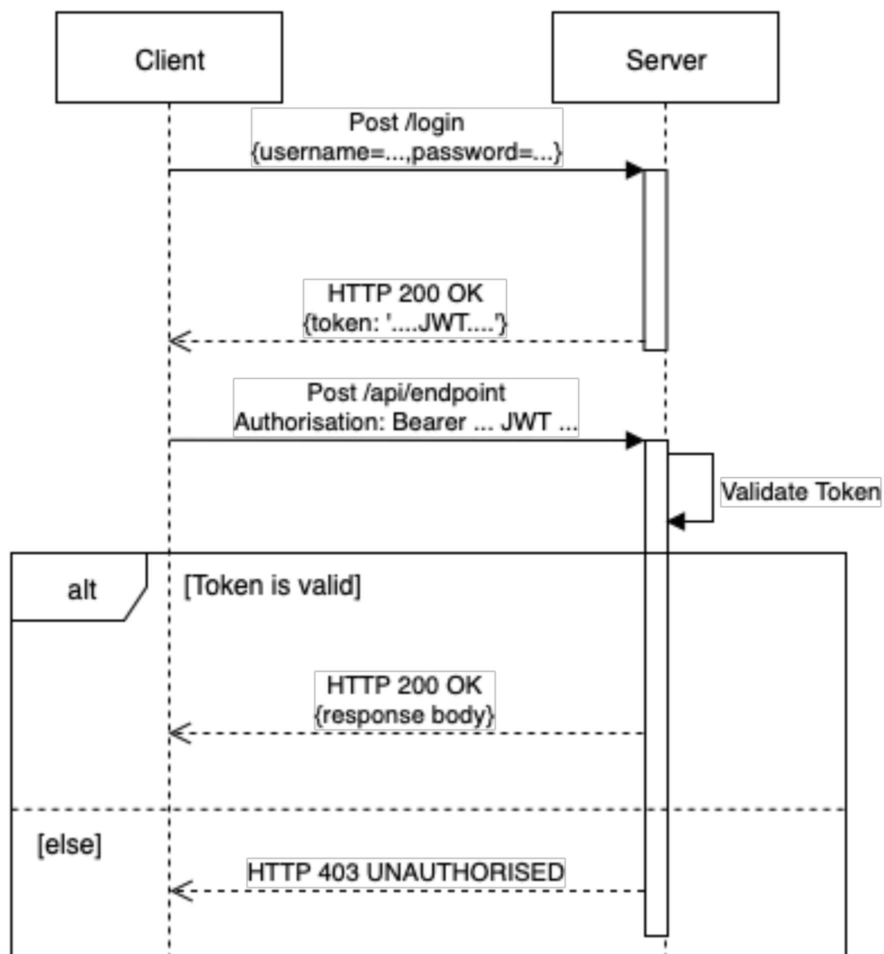

Figure 3.10. JWT Authorisation Flow Diagram

Moreover, to increase security, passwords are stored in the database using a powerful hash algorithm. In the case of a data breach, attackers would not be able to access user accounts, an important feature in an application handling personal data.

3.1.6. Endpoint Mapping with Spring

In order to map endpoints to methods, Spring provides annotations, which can be used on methods within a class. First of all, the class has to be annotated with "@RestController" and "@RequestMapping("endpointAddress")", telling Spring that the class will act as a controller, handling responses from a URL that points to the address specified. Then, on each method that needs to be exposed the annotation "@RequestMapping ("methodAddress")" can be used to map that method to the URL "endpointAddress/methodAddress". The @RequestMapping annotation provides a parameter for method (POST, GET, etc.) or, as an alternative, alias annotations exist: "@PostMapping", etc.

In order to automatically parse the request body, the "@RequestBody" annotation can be used in front of one of the handler method parameters. Java will automatically parse the body, trying to match it to the type provided as parameter.

```
@RestController
@RequestMapping("/doctor")
public class DoctorController {

    @Autowired
    private DoctorServiceInterface doctorService;

    @Autowired
    private PatientServiceInterface patientService;

    @Autowired
    private DonationRequestServiceInterface donationRequestService;

    @Autowired
    private DonationCommitmentServiceInterface donationCommitmentService;

    //================================================================================
    // Managing Patients
    //================================================================================

    @PostMapping("/addPatient")
    private ResponseEntity<Response> addPatient(@RequestBody DoctorPatientCreateDTO patientDTO) {
        try {
            Patient patient = patientService.addPatient(
                    patientDTO.getDoctorId(),
                    patientDTO.getSsn(),
                    patientDTO.getBloodType().getGroupLetter(),
                    patientDTO.getBloodType().getRh());
            return ResponseBuilder.encode(HttpStatus.OK, new OutgoingIdentifierDTO(patient));
        } catch (EntityNotFoundException | EntityAlreadyExistsException | SsnInvalidException e) {
            return ResponseBuilder.encode(HttpStatus.OK, e.getMessage());
        }
    }
```

Figure 3.11. Annotation Examples Within a Java Rest Controller

3.1.7. Class Instantiation with Spring

Maybe the most used Spring feature in this application is dependency injection. It is done by adding a property, whose type is an interface to a class, with the annotation @Autowired. When building the project, Spring will look for classes that implement that interface, and if those classes are managed by Spring (by being annotated as a component), they will be injected as a dependency for that class. For an example, see Figure 3.11.

3.1.8. Using Environment Variables for Local and Distribution Environment

Due to differences between the development environment and the production environment, where the application is hosted online on Heroku, certain runtime environment variables were introduced. These variables were set from Heroku on the deployed application, and from within IntelliJ IDEA when building the server application locally. They are accessed via the System.getenv() method, and they were used for database connection parameters, for passwords that needed to be used within the code (sending emails) or for choosing which certificate to use when sending push notifications to iOS users.

3.2 Web Application

3.2.1. Functionality

The EasyHelp web application provides the functionality required for all user types, but the Donor. Each subsequent layout after login is tailored to represent the features required by each user type. A component has been created for each feature required and a navigation bar has been used in order to allow users to access each feature. Available features have been discussed in Chapter 1.3.

Admin App   Doctor Account Requests   DCP Account Requests   Doctor Accounts   DCP Accounts   Hospitals   Donation Centers   Test Push   Create Mock   Logout

Active Donation Center Personnel Accounts

Name	Email	County	Date of Birth	Donation Center	Donation Center Contact	Actions
Mariana Vitalie	mariana@dcp	ALBA	Thu Oct 27 1994 03:00:00 GMT+0300 (EEST)	CTS Alba	0730123123	Deactivate Account
Mihai Vitalie	mihai@dcp	ALBA	Sat Feb 05 1994 02:00:00 GMT+0200 (EET)	CTS Alba	0730123123	Deactivate Account

Banned Donation Center Personnel Accounts

No banned donation center personnel accounts.

Figure 3.12. Example of the Admin EasyHelp Web Application

3.2.2. Server Communication

Within the EasyHelp web application, server communication is handled through the HTTP library, which was already introduced. A server request is made by calling the adequate method from the HTTPClient library, depending on the type of request we want to send (POST, GET, etc.). The URL, parameters (if any) and headers are added to the request and the request is sent.

Request processing is done within the service, making use of the map functionality, which passes all the results emitted by the source through a transform function. This allowed for response processing at the service level and enabled components to subscribe to a method knowing the type of the result. Services are added to components via dependency injection, similarly to how the HTTPClient is injected within the service.

```
@Injectable({
    providedIn: 'root'
})
export class AuthService {

    constructor(private http: HttpClient) { }

    myheader = new HttpHeaders().set('Content-Type', 'application/json');

    login(email: string, password: string): Observable<LoginResult> {
        return this.http.post(environment.apiUrl + '/users/login',
                              JSON.stringify({email: email, password: password}),
                              {headers: this.myheader})
        .pipe(map((res: any) => {
            if (res.status === true) {
                const profile = new ProfileData();
                profile.id = res.object.user.id;
                profile.token = res.object.token;
                profile.role = res.object.user.userType;
                profile.county = res.object.user.county;
                profile.email = res.object.user.email;
                profile.firstName = res.object.user.firstName;
                profile.lastName = res.object.user.lastName;
                profile.dateOfBirth = res.object.user.dateOfBirth;
                profile.ssn = res.object.user.ssn;
                profile.locationId = res.object.user.locationId;
                this.setUser(profile);
                return new LoginResult(true, profile, null);
            } else {
                return new LoginResult(false, null, res.exception);
            }
        }));
    }
```

Figure 3.13. Example of Injecting and the Usage of HTTPClient in Angular 2

Subscribing to a result creates an anonymous method that will be triggered when result computation is finished within the service, being an asynchronous call. This way, the user interface is never frozen waiting for results, creating a better user experience.

```
this.authenticationService.login(this.f.username.value, this.f.password.value)
.subscribe((res: LoginResult) => {
    this.loading = false;
    if (res.success === false) {
        this.error = res.error;
        return;
    } else {
        this.error = undefined;
        const profileData = res.profileData;

        if (profileData !== undefined) {
            this.redirect(profileData);
        }
    }
});
```

Figure 3.14. Example of Subscribing to an Observable in Angular 2

3.2.3. Managing User Roles and Server Authentication

The first step when using the EasyHelp web application is logging in. After successfully authenticating with the server, on the response, the user data is received, together with the JWT token. The profile data is created, stored in memory and the user is redirected to the home page, containing the features correspondent to the user's role (see Figure 3.13. and Figure 3.14.).

Since admins, doctors and DCPs all use the same web application for accessing their features, a guard has been added to all routes leading to user-specific URLs. This way, when a user attempts to access a certain URL, his role is verified against the role which is allowed to access said URL. If there is a match, then the user can access the page. If not, the user is redirected to login.

This behaviour has been implemented with a class that implements the CanActivate interface, which is then injected as a handler in the route setup. Still in route setup, the user role who is permitted to access that path is added. This way, logged in doctors are not allowed to access the DCP part of the application.

```
{ path: 'doctor', component: DoctorLayoutComponent,
          canActivate: [AuthGuard],
          data: {roles: [UserRole.Doctor]},
          children: [
    { path: '', redirectTo: 'request', pathMatch: 'full' },
    { path: 'request', component: RequestBloodComponent },
    { path: 'patients', component: PatientsComponent },
    { path: 'my-requests', component: MyRequestsComponent }
] },
```

Figure 3.15. Example of Route Mapping and Route Guard Implementation

After a successful login, the JWT token is stored in local memory, together with the user profile data. All that remains is adding the authentication token to every call forwarded by a user. In order to do so, a class implementing the HttpInterceptor interface has been used. Within the implementation, the token is added to the request if it exists; otherwise the request is left untouched.

3.2.4. Creating the HTML Views

When laying out the views managed by each component, extensive use of directives, templates and data interpolation has been used.

When having to display a list of items, the *ngFor directive was used. It allows for iterations over collection of objects. When used to display the collection in a table, adding the *ngFor directive within the table row tag, it will create a row in the table for each element of the collection.

By following this pattern, adding handlers for each table row is easy, since the reference for the item in the collection can be added as a parameter to methods.

```html
<table class="table table-hover"
*ngIf="waitingSendDonationCommitments && waitingSendDonationCommitments.length > 0; else no_proposedcommitments">
    <thead>
        <tr>
            <th scope="col" class="th-id">Urgency</th>
            <th scope="col" class="th-name">Stored Blood Identifier</th>
            <th scope="col" class="th-id">Hospital</th>
            <th scope="col" class="th-name">Hospital Phone</th>
            <th scope="col" class="th-name">Actions</th>
        </tr>
    </thead>
    <tbody>
        <tr *ngFor="let commitment of waitingSendDonationCommitments ; let i = index" data-="blood.id">
            <td class="td-name">{{ commitment.urgency }}</td>
            <td class="td-name">{{ commitment.storedBlood.bagIdentifier }}</td>
            <td class="td-name">{{ commitment.destinationHospital.name }}</td>
            <td class="td-name">{{ commitment.destinationHospital.phone }}</td>
            <td>
                <span class="btn btn-success add-button" (click)="markCommitmentAsDeparted(commitment)">Mark as Shipped</span>
            </td>
        </tr>
    </tbody>
</table>
```

Figure 3.16. Example of Creating a Table with Elements from a Collection in Angular 2

In order to handle the case of an empty collection, templates can be used to substitute HTML content. As seen in Figure 3.16, by using the *ngIf directive, it is checked that the collection we want to iterate is not empty. If it is, the whole table HTML code is replaced with the HTML code contained by the template called no_proposedcommitments. The definition of the template can be found in the figure below.

```html
<ng-template #no_proposedcommitments>
    <h6>You have not forwarded any commitments</h6>
</ng-template>
```

Figure 3.17. Defining an HTML Template in Angular 2

In order to style the views, the Bootstrap library was used. As a consequence, little to no CSS code had to be written in the development of this application. Bootstrap is an open source component library [18]. It provides developers with already implemented style classes for HTML.

3.2.5. Online Hosting and Environment

In order to host the application online, the Heroku platform was chosen. Due to the fact that Heroku does not provide out of the box support for Angular applications, it had to be wrapped in a Node.js application using Express. The Heroku EasyHelp application was setup to redeploy on every commit on the master branch pushed to GitHub.

In order to ease development and not have to change the default server URL variable when building for Heroku, opposed to building locally, two environment files were used: one for local builds and one for Heroku builds. When building for Heroku, after the build is done another script is run, which replaces the local environment with the production environment file, which instead of the local address of the server contains the Heroku server address. An example on the usage of environment variables can be found in Figure 3.13.

3.3. iOS Application

3.3.1. Functionality

The iOS application is dedicated to donors and provides all the functionality required by donors to interact with the system.

The first screen a user would see in the application is the login/register view. If the user registered through the web application, he can directly login the application and start using it. If the user chooses to register via the application, after registering the onboarding flow will appear, allowing for profile setup.

The main screen of the application allows users to book a donation. The flow includes selecting a donation centre, choosing an available time slot and confirming all the details. Before confirming, the user is also allowed to submit a donation form, in order to not fill it in by hand at the donation centre.

Another functionality implemented is the donation history. A donor can view all the previous donations and their test results. If for one donation the donor failed the control

tests, the section is highlighted to attract attention. The donor can view the donation details on a dedicated page.

The last view the donor can access is the profile details page, where he can see his blood group and edit the donation form, so it will be already filled in when booking a donation.

3.3.2. Server Communication

Server communication has been handled using two third party libraries, Alamofire and SwiftyJSON. They are both provided by Alamofire and work together to help developers handle HTTP communication in Swift.

All server communication is done via the Server.swift singleton. In order to create send a server request, we need the URL and the parameters, encapsulated in the ServerRequest class, a parser for the response, a callback and the HTTP method. All server operations are called from services.

As previously mentioned, the server response encoding has been designed to facilitate client implementation. In the iOS client, all parsers extend the ServerResponseParser class. This class is responsible for detecting if the status of the response is true or false. If true, then it calls the doParse method, which is an abstract method implemented in all other parsers inheriting from ServerResponseParser. If the status is false, then it parses the error.

After response parsing has been done, either the on success callback is called, or the error callback, depending on the status of the request.

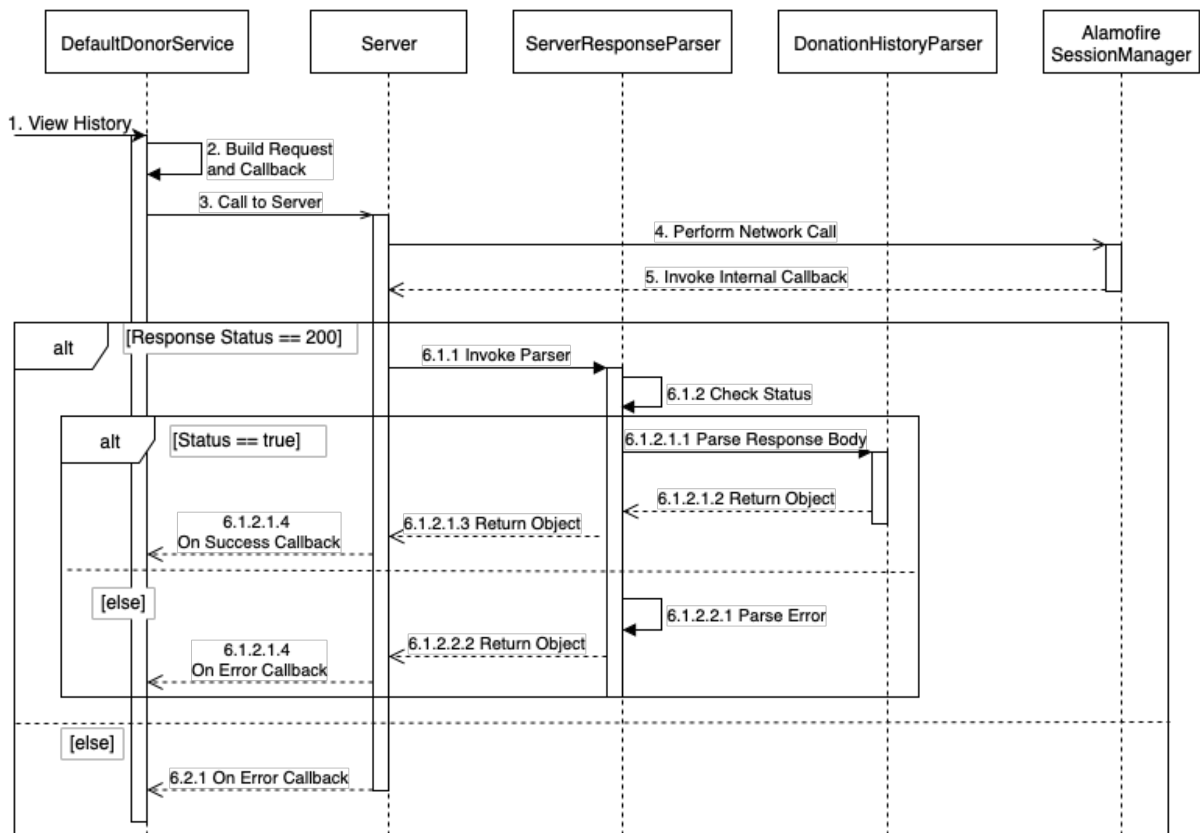

Figure 3.18. Sequence Diagram Detailing EasyHelp iOS Server Architecture

3.3.3. Implemented Services

In order to enhance functionality, several services have been implemented for the application.

First of all, in order for donors to be always up to date with their donation status, push notifications are received once a DCP has entered the test results for the last donation. In order to enable push notifications, the application must first ask for permission from the user. If permission is granted, then a device token is obtained, which is registered with the server. It will later be used to send push notifications to that device.

Another service that has been added to the application in order to enhance usability is the location service. It is used to determine the user's location and fetch the donation centres ordered by distance, so that the closest donation centre is always the first in the list.

3.3.4. Individual Targets for Different Application Versions

To ease development, multiple targets have been created in order to run the application with different settings. This has been achieved by using custom build flags for each target and pre-processor macro in code.

In order to easily switch between the local server and the server application hosted on Heroku, pre-processors check if the "DEVELOPMENT" flag has been set on the target in order to return the correct settings file.

```
fileprivate static let settingsFileName: String = {
    #if DEVELOPMENT
    return "Settings-Dev"
    #else
    return "Settings-Local"
    #endif
}()
```

Figure 3.19. Example of Using Pre-processor Macros in Swift

The contents of the settings file contain the proper URLs for accessing the local or the hosted server application.

The same technique has been used to create a Mock version of the application, which instead of requesting data from the server, gets it from a local mock repository. In addition to not communicating to the server, the mock application is represented by a series of setup controllers for the views. The different possible data values are selected and the views are constructed based on the setup.

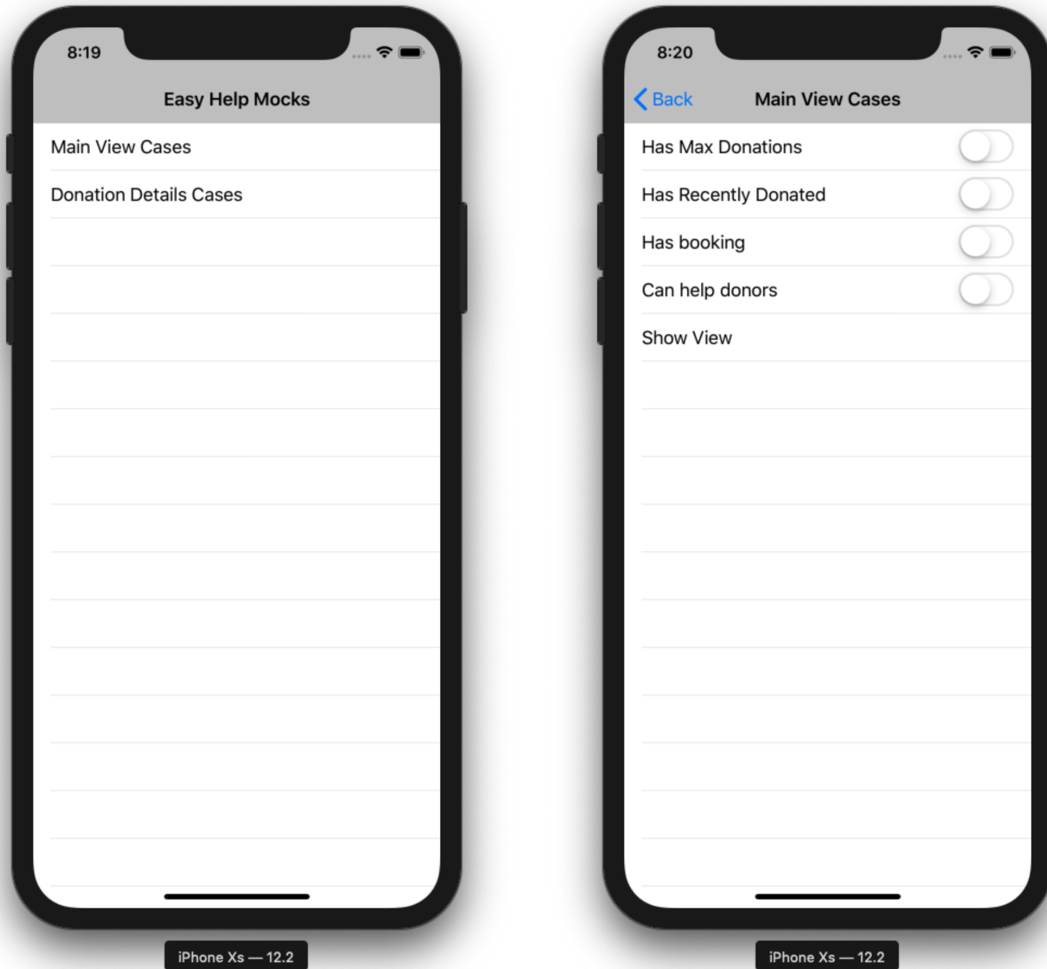

Figure 3.20. EasyHelp Mock Application Screenshots

This version of the application has been used for testing view layouts for multiple possible cases. In this case, the flag that is checked is “MOCK”.

```
class AppServices {
    static var profileService: ProfileService = {
        #if MOCK
        return MockProfileService()
        #else
        return DefaultProfileService()
        #endif
    }()
```

Figure 3.21. Another Example of Using Pre-processor Macros in Swift

3.3.5. Usage of Important Design Patterns

As previously mentioned, the iOS architecture is deeply dependent on two design patterns: the Model View Controller pattern and the Delegation pattern. These two patterns are often used together when subclassing the UIViewController class. The view signals events towards the parent controller via a delegate. The methods handling actions required by the delegate are implemented by the same parent controller. The delegate is implemented as a protocol defined by the view class, which the controller class has to implement.

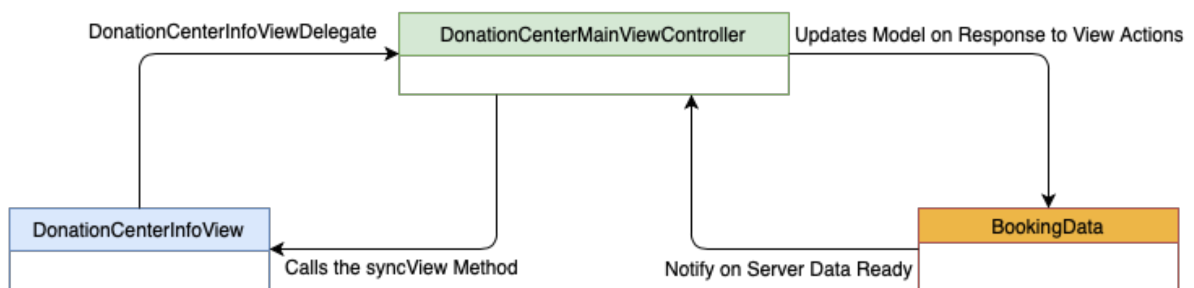

Figure 3.22. Diagram of Concrete Classes Implementing the MVC Pattern

In this figure, the DonationCenterInfoView communicates user actions via a delegate, which uses the DonationCenterInfoViewDelegate protocol, which is implemented within the DonationCenterMainViewController. At the same time, the view is updated by the controller via the syncView method, which is called when all the data has been fetched from the server, from a callback. That callback can be considered the method through which the controller gets notified by the model. The controller updates the model upon user action, within the implementation of the aforementioned delegate methods. Note that the BookingData class does not exist in the implementation, and is used in Figure 3.22. as a representation of the data fields updated by the controller.

3.3.6. The implementation of UI Tests using XCTest

In order to properly test the layout of some views from the iOS client, two test suites have been implemented. In order to ease development of the suites, testing has been done on the Mocks versions of the application, allowing for covering all view layouts without following complicated application logic.

Page objects have been implemented for each individual view component. This does not mean that each screen in the application is mapped to one page object. It might happen, like in the case of the main view in the EasyHelp iOS application, that one screen can hold multiple individual view components.

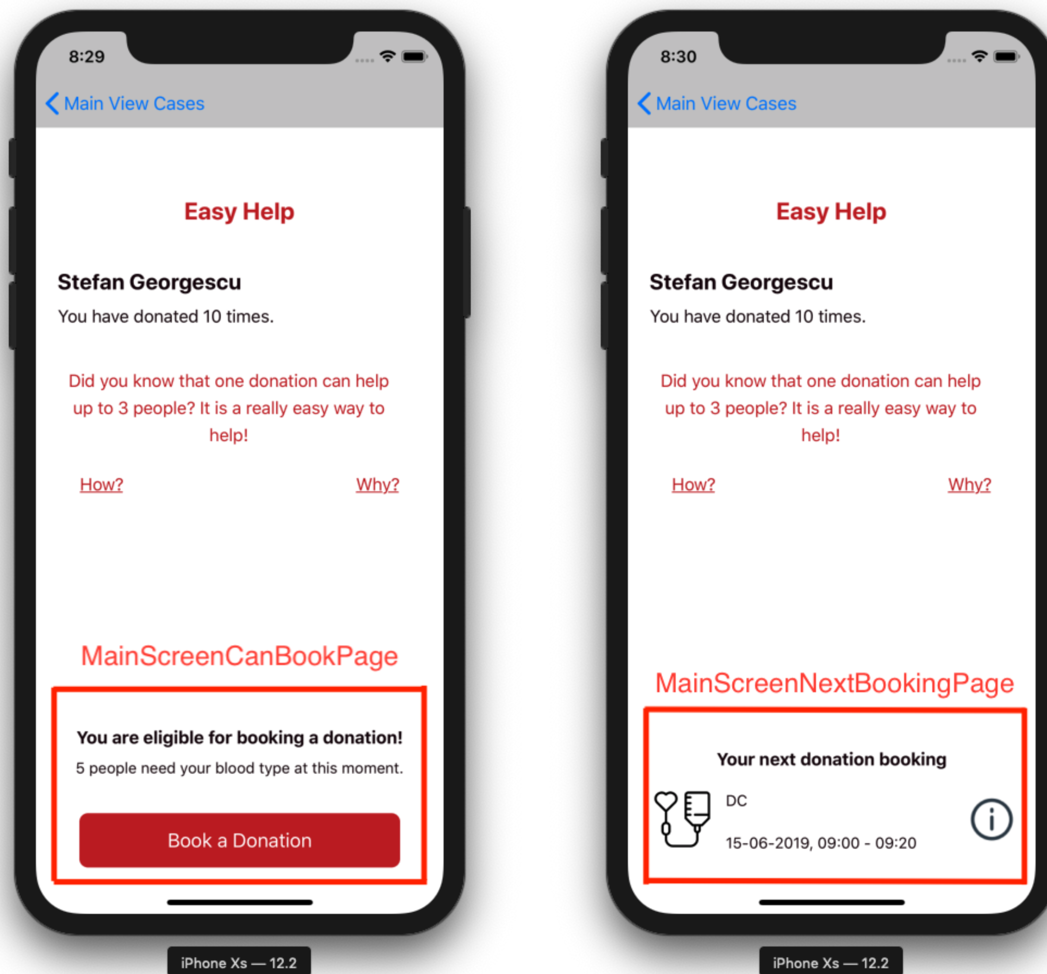

Figure 3.23. Page Object Mapping Example in EasyHelp

Within the source code, page objects, which provide implementation for each of the subviews have been implemented and the main page object inherits these classes in order to be able to interact with all layouts.

Within the XCTest framework, all UI elements are found via an identifier set on the element at creation. Elements are found by navigating the UI hierarchy. XCTest provides an object type called XCUIElement, used to encapsulate any type of UI building block (labels, buttons, scroll views, etc.). Each object of type XCUIElement can either be interacted with or used as a container, with the purpose of finding and interacting with one of its subviews. The root of this hierarchy is given by the XCUIApplication object.

3.3.7. Usage of XCode Server for Continuous Integration

In order to run the tests written for the EasyHelp iOS Client, an XCode Server Bot was created with the task of building and testing the application. Integrations can be run manually or can be scheduled, either as a recurring event or on new commits to the Git repository. The XCode Server has been setup on a secondary machine and it can be accessed on the local network via the XCode IDE.

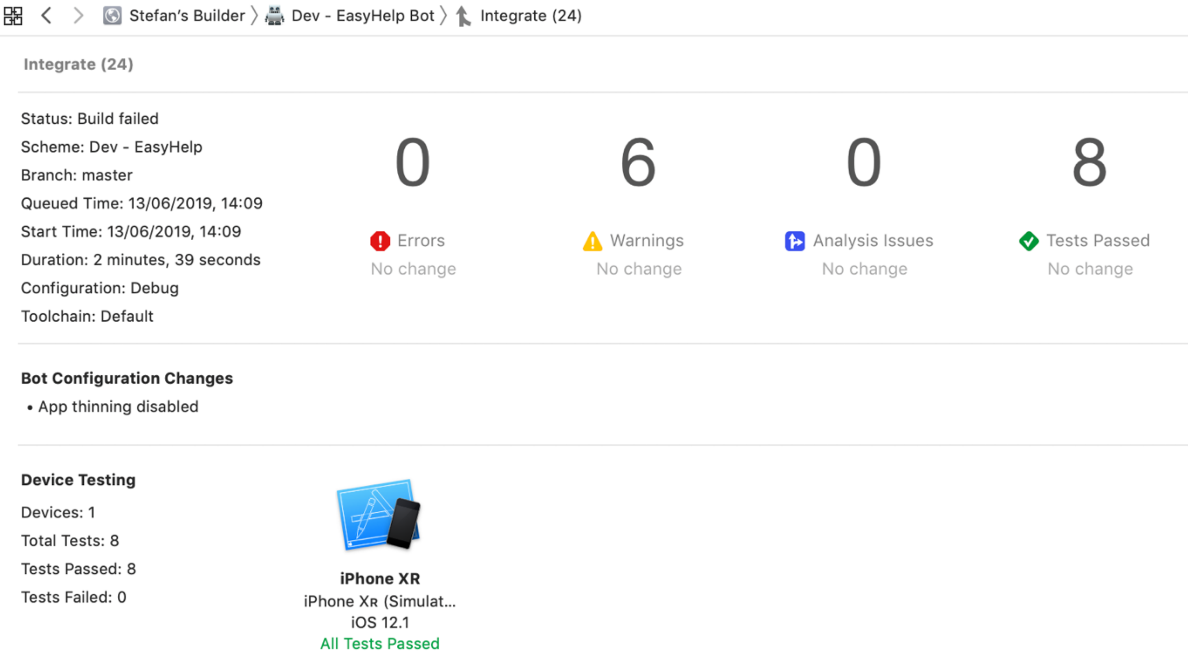

Figure 3.24. XCode Server Bot Integration Results

The setup process consists of giving the bot access to the online repository via HTTPS or SSH, choosing build target and preparing the build conditions. Since our application is using third party libraries installed via a dependency manager, a pre-integration trigger must be run, responsible with installing the required libraries before building.

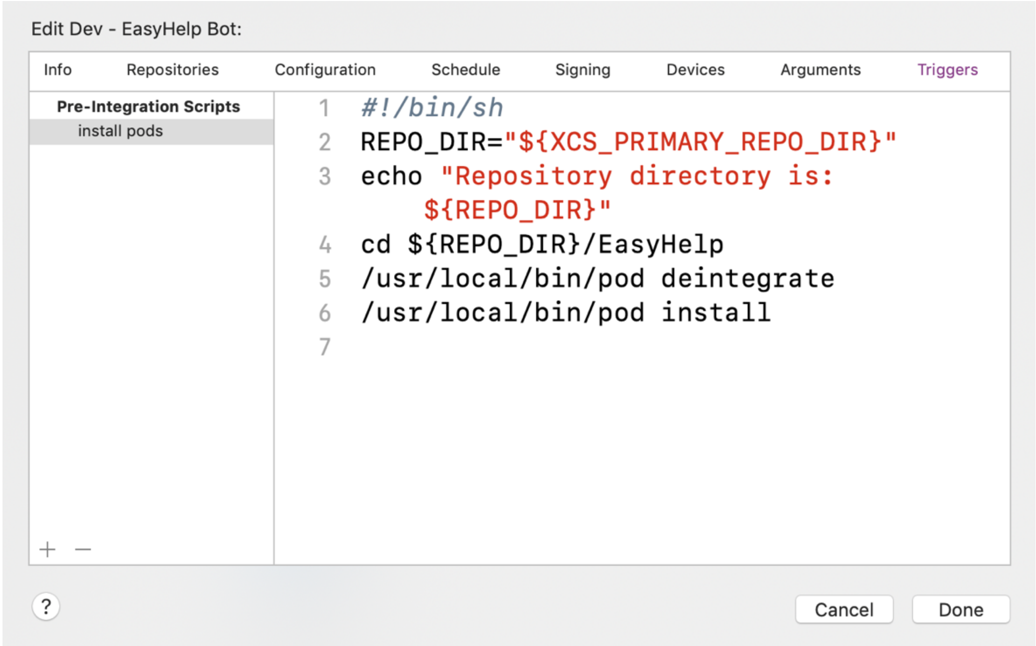

Figure 3.25. XCode Server Bot Pre-Integration Script Setup

4. Conclusion

This paper has presented some of the theoretical aspects and their implementation, required for designing and developing an application suite serving multiple user types with the purpose of managing blood donations throughout Romania.

The months spent developing the software have broadened the author's knowledge about the Java Spring framework, the Angular 2 framework and the iOS development practices, while at the same time, giving more depth into the software development lifecycle, from design to deploy.

The goal of this thesis was to build a full-stack software application and by doing so, learn important tools and practices used in the field. Judging with respect to this objective, the project can be considered a success.

4.1. SWOT Analysis of the Implemented Application

In order to better analyse the EasyHelp software application suite, a Strengths, Weaknesses, Opportunities, Threats (SWAT) analysis has been performed. SWOT analysis is a popular business assessment tool used to understand how business can be driven forward. In this context, SWOT will be used in order to find where the software application needs more work.

Strengths

Compared to other software solutions targeting the same field, EasyHelp managed to fully integrate all the actors involved in the blood donation process. By providing donors with a mobile application, which is extremely accessible, the author believes that incentives to donate can be more easily delivered to the wide population.

Weaknesses

Having had a small development period, the application does have sensitive areas in which edge cases have not been handled in their entirety.

Another process that has been overlooked is the implementation of tests at multiple levels. The application features UI tests for the iOS application, whereas the other applications do not feature any tests.

Opportunities

Having demonstrated that a system managing blood in a country can be designed and implemented in a short time-span, the author believes that there is a lot of potential in the idea behind EasyHelp. Further developing the presented software applications could lead to a solution applicable at a large scale.

Threats

Acknowledging the fact that the EasyHelp application suite handles almost only sensitive data, the biggest threat the system faces is a security breach. Therefore, preparation for such an event is crucial in the case of deployment.

4.2. Further Improvements

Having looked at the strong and weak points of the application, improvements are easy to identify. In the following subchapters we will discuss the most important aspects when considering the hypothetical situation in which this application suite would be used at a national level.

4.2.1. Scalability

The system did not go through stress testing, therefore its capability of handling a large number of users cannot be properly assessed. Moreover, given the project's nature, a high number of users is expected, therefore scalability is one of the areas where improvements need to be implemented.

4.2.2. Security

Security is the area in which the developed application lacks the most, hence the most improvements can be made. End to end encryption and web-based security protocols are two of the possible enhancements.

5. Bibliography

[1] Angular: Architecture Guide, https://angular.io/guide/architecture

[2] Angular: Component Architecture Guide, https://angular.io/guide/architecture-components#data-binding

[3] Angular: Dependency Injection Guide, https://angular.io/guide/dependency-injection

[4] Angular: HTTP Client Documentation, https://angular.io/api/common/http/HttpClient

[5] Angular: HTTP Guide,https://angular.io/guide/http

[6] Angular: Lifecycle Hooks Guide, https://angular.io/guide/lifecycle-hooks

[7] Angular: NPM Packages Guide, https://angular.io/guide/npm-packages

[8] Angular: Router Guide, https://angular.io/guide/route

[9] Angular Documentation: Component, https://angular.io/api/core/Component

[10] Apple: The Swift Programming Language, https://docs.swift.org/swift-book/

[11] Apple Developer Documentation: Application Lifecycle, https://developer.apple.com/documentation/uikit/app_and_scenes/managing_your_app_s_life_cycle

[12] Apple Developer Documentation: Delegation, https://developer.apple.com/library/archive/documentation/General/Conceptual/DevPedia-CocoaCore/Delegation.html

[13] Apple Developer Documentation: Model-View-Controller, https://developer.apple.com/library/archive/documentation/General/Conceptual/DevPedia-CocoaCore/MVC.html

[14] Apple Developer Documentation: UIViewController, https://developer.apple.com/documentation/uikit/uiviewcontroller

[15] Apple Developer Documentation: XCode, https://developer.apple.com/xcode/ide/

[16] Apple Developer Documentation: XCTest,
https://developer.apple.com/documentation/xctest

[17] Auth0: Introduction to JSON Web Tokens, https://jwt.io/introduction/

[18] Bootstrap, https://getbootstrap.com

[19] Chris Hoffman: What is an API?, https://www.howtogeek.com/343877/what-is-an-api/

[20] Cocoapods, https://cocoapods.org

[21] Crunchbase: Heroku Overview,
https://www.crunchbase.com/organization/heroku#section-overview

[22] Git: About the Distributed Features, https://git-scm.com/about/distributed

[23] Github: About, https://github.com/about

[24] Gradle: Gradle vs Maven Comparison, https://gradle.org/maven-vs-gradle/

[25] Gradle Documentation: Dependency Management,
https://docs.gradle.org/current/userguide/introduction_dependency_management.html

[26] Gradle Documentation: Introduction,
https://docs.gradle.org/current/userguide/what_is_gradle.html#what_is_gradle

[27] Heroku: Continuous Integration, https://www.heroku.com/continuous-integration

[28] Heroku: Heroku Postgres, https://www.heroku.com/postgres

[29] Hibernate: Documentation, http://docs.jboss.org/hibernate/orm/5.1/javadocs/

[30] Javatpoint: Java JDBC Tutorial, https://www.javatpoint.com/java-jdbc

[31] Jetbrains: IntelliJ IDEA, https://www.jetbrains.com/idea/

[32] Martin Fowler: PageObject, https://martinfowler.com/bliki/PageObject.html

[33] Microsoft: TypeScript Documentation, Chapter 1.3,
https://github.com/microsoft/TypeScript/blob/master/doc/spec.md#1.3

[34] Mozilla: XMLHttpRequest Documentation, https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest/Synchronous_and_Asynchronous_Requests

[35] Postgres: About, https://www.postgresql.org/about/

[36] Postgres: JDBC Driver, https://jdbc.postgresql.org/about/about.html

[37] WikiBooks: Introduction to Software Engineering, Architecture, Design Patterns, https://en.wikibooks.org/wiki/Introduction_to_Software_Engineering/Architecture/Design_Patterns

[38] Wikipedia: Git, https://en.wikipedia.org/wiki/Git

[39] Wikipedia: iOS, https://en.wikipedia.org/wiki/IOS

[40] Roy Thomas Fielding: Architectural Styles and the Design of Network-based Software Architectures, https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf

[41] SASS: SASS Language Documentation, https://sass-lang.com/documentation/syntax

[42] Spring: Spring Framework Overview, https://docs.spring.io/spring/docs/current/spring-framework-reference/overview.html

[43] Steven F. Daniel: XCode 4 iOS Development Beginner's Guide, Chapter 1, https://subscription.packtpub.com/book/application_development/9781849691307/1/ch01lvl1sec12/layers-of-the-ios-architecture

[44] TIOBE: TIOBE Index, https://www.tiobe.com/tiobe-index/

[45] Ziar Medical: Doar 2% dintre romani donează sânge, https://ziarmedical.ro/2018/11/14/romanii-doneaza-sange/

6. List of Figures

Figure 2.1. Data Binding in Angular **18**

Figure 2.2. Diagram of the MVC Design Pattern **22**

Figure 2.3. iOS Application Lifecycle **24**

Figure 2.4. iOS UIViewController Lifecycle **25**

Figure 2.5. Example POM Diagram **27**

Figure 3.1. Server Operation Sequence Diagram **29**

Figure 3.2. UML Class Diagram for Classes Implementing the User Types **30**

Figure 3.3. UML Class Diagram for Classes Implementing Donor and Donation Center Personnel Related Features **31**

Figure 3.4. UML Class Diagram for Classes Implementing Doctor and Donation Center Personnel Related Features **32**

Figure 3.5. Response Example for an Unsuccessful Operation **33**

Figure 3.6. Response Example for a Successful Add Operation **33**

Figure 3.7. Response Example for a Successful Get all Hospitals Operation **34**

Figure 3.8. ORM Annotations Example in Doctor.java **35**

Figure 3.9. ORM Annotations Example in Patient.java **35**

Figure 3.10. JWT Authorisation Flow Diagram **36**

Figure 3.11. Annotation Examples Within a Java Rest Controller **37**

Figure 3.12. Example of the Admin EasyHelp Web Application **39**

Figure 3.13. Example of Injecting and the Usage of HTTPClient in Angular 2 **40**

Figure 3.14. Example of Subscribing to an Observable in Angular 2 **41**

Figure 3.15. Example of Route Mapping and Route Guard Implementation **42**

Figure 3.16. Example of Creating a Table with Elements from a Collection in Angular 2 **43**

Figure 3.17. Defining an HTML Template in Angular 2 **43**

Figure 3.18. Sequence Diagram Detailing EasyHelp iOS Server Architecture **46**

Figure 3.19. Example of Using Pre-processor Macros in Swift **47**

Figure 3.20. EasyHelp Mock Application Screenshots **48**

Figure 3.21. Another Example of Using Pre-processor Macros in Swift **48**

Figure 3.22. Diagram of Concrete Classes Implementing the MVC Pattern **49**

Figure 3.23. Page Object Mapping Example in EasyHelp **50**

Figure 3.24. XCode Server Bot Integration Results **51**

Figure 3.25. XCode Server Bot Pre-Integration Script Setup **52**