medR: Software System for Managing Medical History and Patient Examination Data

Anita-Bella Réthi
Babeş-Bolyai University
Cluj-Napoca, Romania
rethi.anita@gmail.com

Krisztián-Tamás Antal
Babeş-Bolyai University
Cluj-Napoca, Romania
antal.krisztian@outlook.com

Orsolya Máthé
Codespring
Cluj-Napoca, Romania
mathe.orsolya@codespring.ro

Mátyás Fosztó
Codespring
Cluj-Napoca, Romania
foszto.matyas@codespring.ro

Tamás Koncz
Codepsring
Cluj-Napoca, Romania
koncz.tamas@codespring.ro

Károly Simon
Babeş-Bolyai University
Cluj-Napoca, Romania
ksimon@cs.ubbcluj.ro

***Abstract*—The purpose of the medR project is to lighten the burden on the local healthcare system given by the sub-optimal handling of large amounts of information. The main scope of the project is to provide help during the medical history collection and patient data management phases.**

During the first steps of a patient's examination, critical information may be gathered that determines the entire treatment of the disease. Currently, the Romanian medical system faces cumbersome data recording capabilities, troublesome data retrieval, and a lack of transparency. All of this can lead to the loss of valuable information, wrong decisions, and inadequate patient care, even resulting in financial or physical harm. The medR web application enables extensive data recording possibilities and the generation of statistics, assisting in these tasks by integrating already existing systems and providing easy and simultaneous use for healthcare professionals.

I. Introduction

The main task of the health care system is to deal with the problems of the patients seeking help. In the first steps of this process the health team - the doctor, health assistant, nurse - gathers a large amount of information and then makes critical decisions based on this information. Thus, it is critical for the treatment process that the information coming from the patients and the information gathered during the first examinations is collected correctly and it is managed in an organized and transparent manner. One of the most significant strains on the current Romanian healthcare system is the sub-optimal management of this process [1] [2]. MedR provides a solution for recording, storing, retrieving and processing information obtained from the above-mentioned data collection process.

The path that leads to the correct diagnosis and the selection of the appropriate treatment is: taking the medical history, where data is widely collected (including personal data, past diagnoses, family medical history and other associated health care data), and the patient's current complaints [3]. This data collection process is followed by the physical examination, which is based on viewing, listening, and knocking performed by the healthcare team. Based on the collected information a primary diagnosis is made and the healthcare team may order additional paraclinical examinations. If these tests are ordered based on incorrect information, then an unnecessary financial burden will be placed on the system and the patient, sometimes even exposing the patient to physical harm.

Although there are already software systems that deal with these stages of treatment, these systems aim to provide a comprehensive solution for managing the complete dataset of a patient (see ERA Medical[1], Teamnet Dedalus[2]). Their weakness lies in that they place little emphasis on the first steps of the treatment, collecting and displaying data in a less organized manner, making the retrieval and analysis of said data cumbersome. Based on this observation, the main idea behind the medR project is to create an application that focuses on these first stages of the examination process and works on any platform that can run a modern browser, making it available on desktop computers used in the office, and also on mobile devices used near hospital beds. Using the application, healthcare professionals have the opportunity to build and efficiently manage their patient database by recording patient information and examinations. At the same time, the application helps the interpretation of the data by generating statistics and reports. A wide range of information can be collected about patients: personal data, personal and family medical history, allergies, prescribed medications, special needs, received vaccinations, history of complaints and their localization, strength, quality, frequency, time of onset, different living conditions recognized during the study and data obtained during other physical examinations.

II. Functionalities

Although there are several roles in the healthcare systems, during the development of the medR application the main focus was on the *examiner* role, which represents healthcare professionals such as medical students, nurses, or doctors (residents, specialists, or chief physicians).

[1]ERA Medical: http://www.midasoft.ro/modulul-medical/
[2]Teamnet Group: http://www.teamnet.ro/

A registration is required for using the features of the application. After logging into the system, all of the functionalities can be accessed by the previously registered users.

First of all, the examiner can view a list of his patients. When adding a new patient, an extensive set of information can be introduced into the application: personal information, individual and family medical history, known allergies and medications, special needs, comments, etc. This information will be displayed on the patient's datasheet, supplemented by a list of previous vaccinations and consultations.

When adding a new consultation, the examiner can record the symptoms, their localization, periodicity and intensity, together with accentuating and mitigating causes and possible diagnoses. The examiner can also mention living conditions that might influence the final diagnosis (smoking, alcohol or drug consumption, and other factors). Once a diagnosis or symptom is successfully confirmed, it can be closed, but only after giving the exact reason for the closure. Once finalized, the consultation will become read-only, transferring the diagnoses into the patient's medical history. Each user has the possibility to create different statistics and reports based on the data collected about their patients.

III. Architecture

The medR application is built on a client-server architecture. The server is implemented using the *Spring* framework and provides a *REpresentational State Transfer* (REST) [4] *Application Programming Interface* (API) through which the services can be accessed. The data is stored in a *PostgreSQL* database, and external services are also used as additional data sources. The client is a *Progressive Web Application* (PWA) created with the *Angular* framework. The communication between these two components is handled via REST calls and it is based on the *Data Transfer Object* (DTO) design pattern.

The server communicates with the API provided by the *National Library of Medicine* (NLM) at the *National Institute of Health* (NIH) in the United States [5]. The data provided by the API refers to the international names of symptoms and diagnoses and their *ICD-10* (International Statistical Classification of Diseases and Related Health Problems) codification. This ensures that the names and codifications of the symptoms and diagnoses entered into the system comply with international standards, while also providing a client-side search and auto-complete function.

Figure 1 illustrates the architecture of the application, the mentioned components being detailed in the next sections.

IV. Server

The server is built on a multi-tier architecture. The *persistent* data model is the set of elements that define the structure of the database, while the *domain* model is a specific form of this structure, made up of model classes that represent the most fundamental data in the application and play a role in implementing the business logic. The persistent model is created using the *Java Persistence API (JPA)*, based on annotations placed on the model classes.

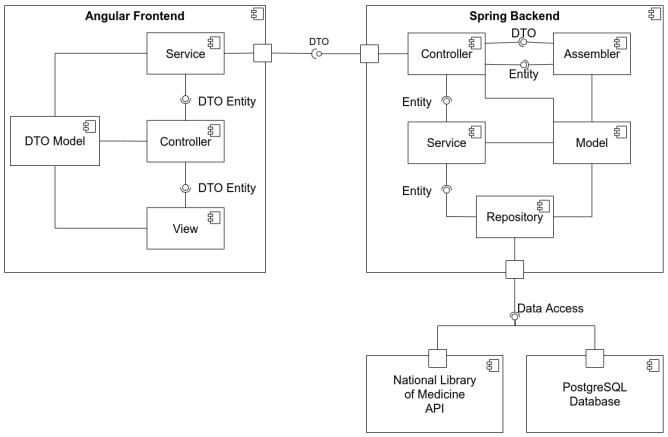

Fig. 1. Architecture of the application.

The multi-layered architecture provides easier code maintenance, it makes it easier to identify and correct errors, and since the communication between the layers is realized through interfaces, each layer can be easily modified or even completely replaced if necessary without influencing other layers.

The bottom layer of the server is the *repository*, the data access layer, which handles the communication with the database, as well as the conversion between persistent and domain models, so the fundamental *CRUD* (Create, Read, Update, Delete) operations and specific queries can be accessed through it.

The *service* layer contains the business logic, implementing the services provided by the system, and publishing these services for the upper layers.

The top layer of the server is the *controller* layer which is responsible for communication. This layer is in charge for receiving REST requests and responding to them. The responses are made up of DTOs, which are specific representations of the models, omitting or adding details to the model objects. These DTOs are created by the *assemblers*, which have the purpose of converting and organizing data from the *service* layer into DTOs.

A. Domain model

The domain data model can be divided into two distinct parts: models that carry information about individuals (patients and examiners) and models that are related to diseases and diagnoses.

In the case of each person, general data is introduced, which includes personal information, address and contact details. If the person belongs to the patient subgroup, additional information is gathered, such as the patient's personal and family medical records, allergies, prescriptions, vaccines, special needs, or other medically relevant social details, including marital status and occupation. If the individual is an examiner, their position in the healthcare system (doctor, nurse or student) and data about the medical unit they work at (hospital or medical station) is collected.

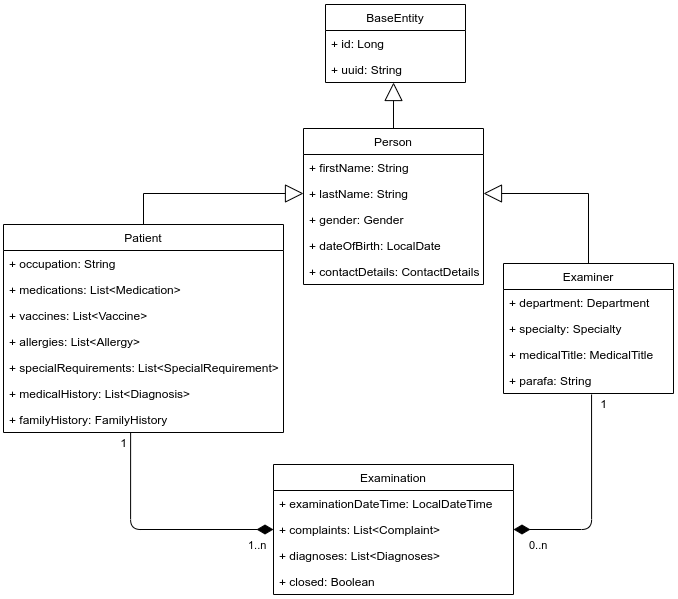

Fig. 2. Part of the Domain model.

The main entity in the disease-related model set is the examination. An examination consists of a set of symptoms, living conditions and associated diagnoses. A complaint includes the name of the symptom, its *ICD-10* code and the characteristics of the complaint, such as localization, severity, timing, frequency and a list of relieving or aggravating factors. Living conditions are elements that may affect a patient's life, health or may be related to specific complaints. This includes, for example, the patient's habits related to tobacco, drug, or alcohol consumption.

B. Data access layer

The communication between the server and the database is realized using an *Object Relational Mapping* (ORM) framework, specifically the *Hibernate* implementation of the *Java Persistence API* (JPA) . Since the *Spring Data JPA* framework is also used as an upper layer built on this, the classes of this layer do not need to be implemented by the developer, only *interfaces* should to be created, extending the *JpaRepository* base interface and using the *@Repository* annotation. Thanks to the *Spring Boot* autoconfiguration mechanism, these *interfaces* are automatically found by the system and *Spring Data JPA* generates the needed implementations.

The *CRUD* (Create, Read, Update, Delete) operations are included in the *JpaRepository*, although there are also ways to create custom queries. The first of these methods is to automatically generate queries based on the names of the *methods* declared in the interfaces. If the query cannot be defined in this way due to its complexity, it is possible to define queries written in *Java Persistence Query Language* (JPQL) using the *@Query* annotation.

Furthermore, *Spring Data JPA* supports query-level paging and sorting through the *Pageable* and *Sort* interfaces. This ensures, on one hand, that such requests do not overload the system and, on the other hand, that the data can be sorted based on specific relevant parameters (e.g. patients by date of birth).

C. Business logic layer

The business logic layer is responsible for processing and organizing data retrieved from the data access layer in order to generate responses for the communication layer. The *Dependency Injection* design pattern is used for managing the dependencies between the components. The *@Service* annotation indicates to the Spring framework that the classes are injectable, so the framework instantiates them upon system startup and provides instances to the classes in which they are required.

The service layer is responsible for generating statistics from patient diagnoses and symptoms, processing data from the *National Library of Medicine* API, and the creation of *Pageable* and *Sort* objects for paging and sorting the data retrieved by certain requests. The core security mechanism of the system is also implemented here, which is described in more details in the following subsections.

D. Communication layer

The communication with the client takes place via the *controller* layer, based on the REST conventions. A separate controller class is assigned to each entity that must be available to the client. The controllers are marked with the *@RestController* annotation and the *@RequestMapping* annotation determines the path to them.

In addition to entity-specific controllers, there are controllers for providing authentication, statistics, and data from the *National Library of Medicine*. Each controller class makes a specific data set available through annotated methods corresponding to different request types (GET, PUT, POST, DELETE) and having particular URIs. The methods use *assemblers* to convert the data received from the *service* classes into DTOs, which are then sent back to the client wrapped in *ResponseEntity* objects, supplemented with extra data and HTTP headers as needed.

E. Security

Since one of the primary purposes of the project is to store significant volumes of sensitive information (patients' personal and medical data), security has a great priority during the development. The *Spring Security* framework is used in the implementation, which primarily deals with authentication and authorization, and it is a fairly flexible framework. The *WebSecurityConfigurerAdapter* abstract class is extended in order to create a project specific configuration.

The first functionality related to the security mechanism is the registration. The user's password is immediately encrypted and saved in this format in the database. To encrypt passwords, a *BCryptPasswordEncoder* instance is used, as specified in the *PasswordConfig*.

When a user logs in, the *JwtTokenProvider* generates a *JSON Web Token* [6] containing a certain data needed in order

to identify the user (this is the email address in the case of medR), the time it was created and the expiration time. The token is sent back to the client page in the form of a cookie in the response header, which will later identify the user when they want to access a route after logging in.

Routes matching the */auth/*** pattern are available to anyone, these contain three basic functionalities of the app: registration, login, and logout. If a request is made for a different path, it will be redirected to a *JwtRequestFilter*, which will check if the user handling the request is logged in. The first step is to check whether there is a cookie in the request header that includes a *JWT*. If there is one, the validity of it will be checked and access to the required route is only allowed if there are no problems found. Otherwise, the cookie is deleted, and a message with the 403 status code is returned to the client.

V. Client

The client is an *Angular Single Page Application*, which has the advantage of loading the whole page only once, and then updating only the parts where the content changes. When the user visits the website, it will load all of the required HTML, CSS, and JavaScript files, after which only the data that needs to be displayed will be requested from the server. This makes the application faster and easier to navigate compared to multi-page web applications.

A. PWA and WebApk

During the development of the medR project, it was a high priority to make the application accessible from any platform (desktop, tablet, smartphone). As a result, the client is a progressive web application (PWA), which has the advantage of functioning similarly to native applications without the need to develop a separate version for each operating system. This not only simplifies the planning process but also shortens the overall development time.

Progressive web apps have the advantage of being able to be installed on any device that has a browser. After that, the application icon will appear on the home screen. The application will be started in a separate window rather than a browser tab, much like a native application. This installed version of the application can be used even if there is no internet connection, as the previously cached data can be accessed.

WebAPK is a progression of PWA that began with the premise that a Progressive Web Application could be bundled into an APK (Android application package). As a result, these programs are much more compatible with the Android operating system than before, behaving almost identically to native applications. [7]

B. Components

The components are the so-called building blocks of an Angular application, and the tree structure made up of Angular components is the application itself. These are TypeScript classes, indicated by the @Component decorator.

During the preparation of the project, a strong focus was placed on the separation of commonly used UI elements, incorporating them into their own components. The reason for this was not only to prevent code duplication, but also to make the component with the provided logic reusable, thereby speeding up the development process. The central components of the project were created based on this logic: the buttons, table, patient cards, and so on.

C. Services

To separate the communication with the server-side API, services were created on the front-end. Services are TypeScript classes that use the Dependency Injection (DI) design pattern, and are annotated with the @Injectable() decorator. Services may be injected into any component, simply adding a parameter with the service type to the constructor, and the framework will provide a functional instance. In the medR project, these classes are primarily used to receive and send server-side DTOs that have similar or even identical models written in TypeScript.

D. Routing and Guards

Because Angular is a set of components, and each component has its own view, there has to be a way to navigate between different views, displaying some of them while removing others from the page. The *routing* mechanism is implemented by using the *Angular Router*. A path can be assigned to each component in the *AppRoutingModule*, and these paths interpret a browser URL.

Angular *Route Guard* is a logic or functionality that is executed before loading the selected route or when leaving the route. These can implement the *CanActivate*, *CanLoad* or *CanDeactivate* methods, depending on when and for what purpose are used. In this case, these guards were used to prevent unauthorized (not logged in) users from accessing certain views on the client, so the *CanActivate* and *CanLoad* methods were extended. Route guards can be assigned to any route in the *AppRoutingModule* .

VI. Technologies and Development Tools

The server is written using the Spring framework [8] and the Spring Boot [9] technology, chosen for their flexibility, cross-platform capabilities, and for their ability to create a runnable application without the need of an external web server. The communication between the PostgreSQL [10] database and the server is handled using Spring Data JPA [11], an abstraction layer over the Hibernate ORM framework, used as JPA implementation. Authentication and authorization are implemented using the Spring Security module [12] and JWT tokens [6].

The client is built using the Angular technology [13], a component-based framework created by Google, which implements the Model-View-Controller and Dependency Injection design patterns. Each Angular component consists of three files: a TypeScript [14] file containing the logic of the component, an HTML-like Angular template containing the

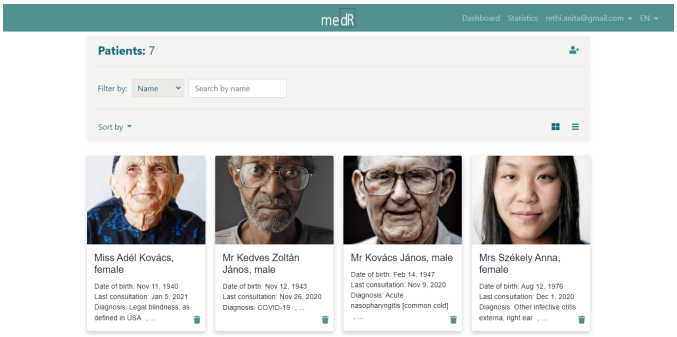

Fig. 3. Dashboard: The main page of the application, where the examiner can see and manage his patients

view, and a CSS or SCSS file containing the used styles. The asynchronous communication with the server is achieved using the Angular HttpClient and RxJS [15] for managing the data stream coming from the backend. The Bootstrap CSS framework [16] and the NG Bootstrap component library [17] is used for styling the application.

Source code management is realized with git [18], using a self-managed GitLab [19] instance as a DevOps platform and remote repository. The backend project uses Gradle as its build and dependency management tool, the frontend project uses Angular CLI and NPM for these tasks. Static code analysis on the frontend is done using ESLint and Prettier, while on the backend CheckStyle, SpotBugs, and PMD are used. For deployment, Docker images are created using GitLab pipelines, which are then used to deploy the application to a Kubernetes [20] cluster. The deployed application is set up using Deployments [21] and Statefulsets [22] for easy horizontal scalability and error recovery.

VII. Using the Application

A registration is required for using the application, during which the examiner must provide his personal information, contact information, medical details (title, role, department), workplace details, together with an email address and a password.

After successfully logging in, the dashboard (see Figure 3) will appear, containing a list of the examiner's previously

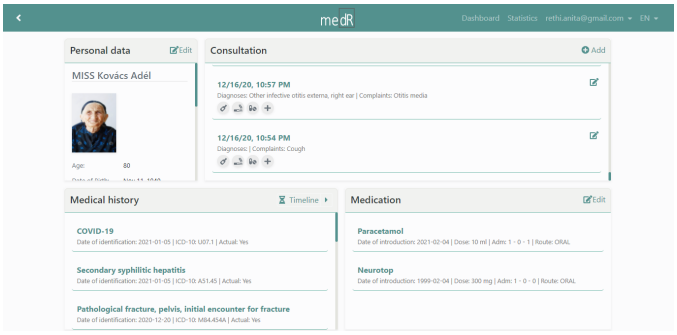

Fig. 4. Part of a patient's datasheet

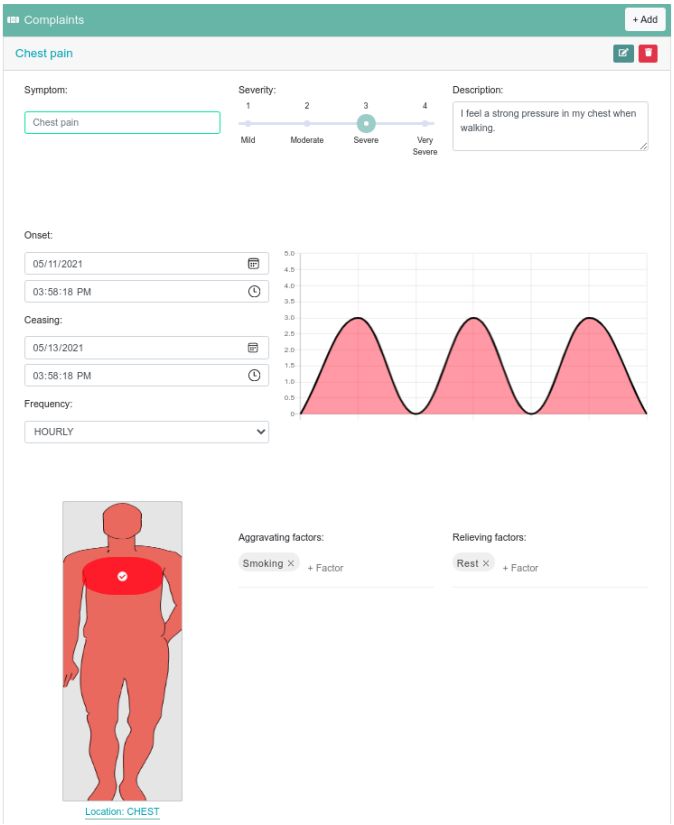

Fig. 5. Complaint editor when editing a consultation.

introduced patients. The patients can be sorted in ascending or descending order by name, age, or the date of their last consultation. To make finding patients easier, the examiner can filter through them by name and diagnosis. The user can also add new patients to his database from this dashboard.

When adding a new patient, the examiner can navigate between six tabs, each of which can manage a different part of the patient's data. Before accessing the other tabs, the user must correctly fill out the form in the Personal Data tab. At this point, the patient is saved to the database, and the examiner can choose to continue filling out the remaining data or fill it out later. This process involves a significant amount of data entry, as in addition to the patients' personal information, it will be collected data about their past diagnoses, family medical history, allergies, medications, received vaccinations, and their special needs.

After completing the patient form, the examiner can access the patient's datasheet (see Figure 4). This page provides an overview of all the information known about the patient, and there is a possibility to edit the existing data or add new data when needed.

The examiner can also see and access the patient's consultations. When creating a consultation, the examiner must first provide the time and date. Following this, the examiner can introduce the patient's complaints, providing the name, severity, and a brief description of the symptom, based on

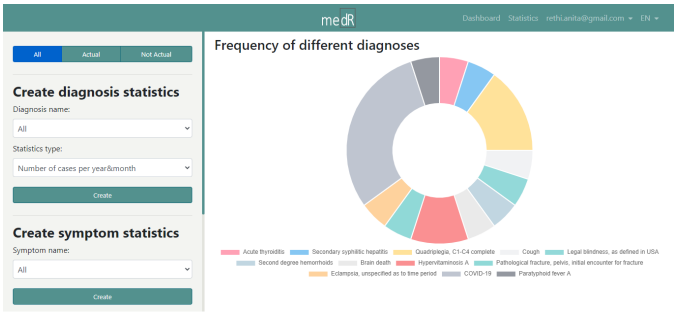

Fig. 6. The statisctics page.

the patient's narration. It is also possible to introduce the onset and ceasing times of the complaint, and the frequency, if it is recurrent. Aggravating and relieving factors may also be mentioned. The last step is to determine the localization of the complaint by selecting the affected body part on the puppet seen in Figure 5. Similarly, all possible diagnoses are recorded, together with the conditions that might influence the final diagnosis. The complaints and diagnoses can be locked, after which these will be no longer editable. When closing a diagnosis, the examiner must mention the reason for closing it, after which the diagnosis will appear in the patient's medical history.

Another important functionality is the Statistics page (see Figure 6), where various statistics can be generated based on the patients of the logged-in user. The examiner can first choose whether the statistics should be based on the data of the patients currently being treated, or on the data of the patients who have already recovered. The examiner can then select one of three statistics types (diagnoses, symptoms, or comparative statistics), select the appropriate options from a series of drop-down lists, and finally, click the "Create" button to display the statistics in the form of an easy-to-read graph.

VIII. Conclusions and Further Development

The current version of the medR project provides a platform that allows healthcare professionals to create their patient database by collecting personal and medical data, to easily manage it, and generate aggregate or comparative statistics based on the information stored about their patients. All phases are aided by a transparent, logically structured user interface. A diagnostic and symptom search function is created with the help of external services, and interactive components are integrated, like the examination editor.

During the planning and development of the project, several further development possibilities were discussed, which will be carried out in a future development cycle. A critical milestone in the lifecycle of the application would be the implementation of collaboration functionalities, which would require the creation of export and import functions. Moduls for supporting educational and research activities could be also included. More medical reports could be generated, which would require the extension and more accurate systematization of the collected data. Despite the extensive data collection, the currently used method provides only a partial coverage for the data amount collected during a medical examination. The extension of the collected data would also help improve the statistics-generation functionality with a module for proving or disproving correlations between symptoms.

Connecting an API that provides drug data is also planned, which could be useful for search functionalities, and it can be used to recognize the relations between the symptoms and some possible side effects of the prescribed medications. Since currently *The European Medicines Agency* does not have a freely available API, using commercial APIs or a proprietary implementation would be required.

In addition, the creation of a web interface for the patients should also be part of the development, where the patients can see and manage the information collected about them. To comply with the data protection and security regulations governing the medical applications, the data has to be made available for the patients, and it must be encrypted, as expected.

References

[1] Mida Soft Business, "Provocările sistemului medical românesc." [Online]. Available: https://www.midasoft.ro/provocarile-sistemului-medical-romanesc/

[2] O. Miron, "O radiografie a sistemului de sănătate din românia," *Revista Polis*, vol. 8, no. 1, 2019. [Online]. Available: https://revistapolis.ro/o-radiografie-a-sistemului-de-sanatate-din-romania/

[3] S. R. Hutchison, D. Hunter, and R. Bomford, *Clinical methods*. Cassell, 1963.

[4] R. T. Fielding, *Architectural styles and the design of network-based software architectures*. University of California, Irvine Irvine, 2000, vol. 7.

[5] National library of medicine clinical table search service. [Online]. Available: https://clinicaltables.nlm.nih.gov/

[6] JWT documentation. Introduction to json web tokens. [Online]. Available: https://jwt.io/introduction

[7] F. S. Correa. Progressive web apps vs. webapks. [Online]. Available: https://www.inovex.de/blog/progressive-web-apps-vs-webapks/

[8] Spring documentation. Spring framework overview. [Online]. Available: https://docs.spring.io/spring-framework/docs/current/reference/html/overview.html

[9] C. Walls, *Spring Boot in action*. Manning Publications, 2016.

[10] PostgreSQL documentation. PostgreSQL: About. [Online]. Available: https://www.postgresql.org/about/

[11] Spring Projects Website. [Online]. Available: https://spring.io/projects/

[12] P. Mularien, *Spring Security 3*. Packt Publishing Birmingham,, England, 2010.

[13] Angular documentation. What is angular? [Online]. Available: https://angular.io/guide/what-is-angular

[14] TypeScript documentation. Typescript for the new programmer. [Online]. Available: https://www.typescriptlang.org/docs/handbook/typescript-from-scratch.html

[15] RxJs documentation. Introduction. [Online]. Available: https://rxjs.dev/guide/overview

[16] Bootstrap documentation. Introduction. [Online]. Available: https://getbootstrap.com/docs/4.3/getting-started/introduction//

[17] NG Bootstrap documentation. Introduction. [Online]. Available: https://ng-bootstrap.github.io/#/getting-started

[18] S. Chacon and B. Straub, *Pro Git*. Apress, 2014.

[19] GitLab documentation. What is gitlab. [Online]. Available: https://about.gitlab.com/what-is-gitlab/

[20] Kubernetes documentation. What is kubernetes? [Online]. Available: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/

[21] ——. Deployments. [Online]. Available: https://kubernetes.io/docs/concepts/workloads/controllers/deployment/

[22] ——. Statefulsets. [Online]. Available: https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/