FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor’s thesis

Title:	MediTrack - doctors, examinations and medical reports
Student:	Marília Maurell Assad
Supervisor:	Ing. Zdeněk Rybola, Ph.D.
Study program:	Informatics
Branch / specialization:	Software Engineering 2021
Department:	Department of Software Engineering
Validity:	until the end of summer semester 2024/2025

Instructions

MediTrack is a Java web application serving as a personal health tracker, including functions to track individual medicines, their stock, planning their application and tracking its history.

The goal of the bachelor thesis is the extension of the application with the following functionality:
- keeping record of various user's doctors and other specialists
- planning doctor appointments and other examinations
- keeping reports and results of the appointments and examinations

The thesis realization should follow the best practices of software engineering. The thesis should consist of:
- analysis of the current functionality of the MediTrack application
- analysis of existing solutions to similar domain
- analysis of the new requirements
- design of the solution
- implementation of selected requirements based on an agreement with the thesis supervisor
- appropriate testing of the implemented functionality
- appropriate documentation

Electronically approved by Ing. Michal Valenta, Ph.D. on 15 January 2024 in Prague.

Bachelor's thesis

MEDITRACK – DOCTORS, EXAMINATIONS AND MEDICAL REPORTS

Marília Maurell Assad

Faculty of Information Technology
Department of Software Engineering
Supervisor: Ing. Zdeněk Rybola, Ph.D.
June 27, 2024

Citation of this thesis: Maurell Assad Marília. *MediTrack – doctors, examinations and medical reports.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2024.

Contents

List of Figures

List of code listings

I would like to thank my supervisor Ing. Zdeněk Rybola, Ph.D. for his support and guidance throughout not only this thesis process, but also for the past two years. I thank my friends and colleagues who worked with me on our group project: your enthusiasm and collaborative spirit initiated this project and made the entire process enjoyable and memorable. Special thanks to my family, for their unconditional love and support.

Declaration

In Prague on June 27, 2024

Abstract

The aim of this Bachelor thesis is to expand an existing Java Web application *MediTrack* with new functionalities. The application is a personal health tracker, including functions to track individual medication, their stock, planning their intake and tracking its history. New functionalities consist of user authentication; keeping record of various user's doctors and other specialists; planning appointments or examinations with notifications; and storing reports and results of appointments and examinations with file uploads.

Keywords Meditrack, web application, Java, React, health tracker

Abstrakt

Cílem této bakalářské práce je rozšířit stávající webovou aplikaci *MediTrack* v jazyce Java o nové funkce. Aplikace je osobní zdravotní tracker, který obsahuje funkce pro sledování jednotlivých léků, jejich zásob, plánování jejich užívání a sledování jejich historie. Nové funkce spočívají v autentizaci uživatele, evidenci různých lékařů a dalších specialistů uživatele, plánování schůzek nebo vyšetření s upozorněními a ukládání zpráv a výsledků schůzek a vyšetření s nahráváním souborů.

Klíčová slova Meditrack, webová aplikace, Java, React, sledování zdravotního stavu

Acronyms

API	Application Programming Interface
BLOB	Binary Large Object
CI/CD	Continuous Integration, Continuous Delivery
CLOB	Character Large Object
CRUD	Create, Read, Update and Delete
CSRF	Cross-site Request Forgery
LOB	Large Object
HMAC	Hash-based Message Authentication Code
HTTP	Hypertext Transfer Protocol
IMAP	Internet Message Access Protocol
JDBC	Java Database Connectivity
JPA	Java Persistent API
JSON	JavaScript Object Notation
JWT	JSON Web Token
MVC	Model-View-Controller
NPM	Node Package Management
OAuth	Open Authorization
ORM	Object Relational Mapping
REST	Representational State Transfer
SMTP	Simple Mail Transfer Protocol
SSL	Secure Sockets Layer
TLS	Transport Layer Security
UTC	Coordinated Universal Time
XSS	Cross-site Scripting

Chapter 1
Introduction

In today's fast-paced world, managing personal health effectively has become increasingly challenging. Many individuals, particularly those with chronic conditions, need to keep track of multiple medications, their dosages, and schedules. Furthermore, coordinating appointments with various doctors and maintaining health records adds to the complexity. To address these challenges, a web application was conceived – *MediTrack* – aimed at providing users with a convenient and user-friendly platform to monitor their medication intake, set reminders, and manage their medication history.

1.1 Motivation

Initially, the project was developed with the supervision of Ing. Zdeněk Rybola, PhD, in the subjects of BIE-SP1 and BIE-SP2. The initial work consisted of analyzing the requirements necessary to implement such a project and execute its basic functionalities: setting up a database with medications and packages for each user, creating intake plans with recurring occurrences and registering individual medication usages. The project continues to be developed by a different group of students, in the same subject.

This aim of this thesis is to improve *MediTrack* by implementing new useful functionalities. First feature to be implemented is user authentication, which enables secure access to personal health data: it ensures only authorized individuals can access sensitive health information, and guarantees user-specific data segregation.

Another enhancement is to keep records of various healthcare providers, including doctors and other specialists. Users are able to plan and schedule doctor appointments and other medical examinations, ensuring they never miss an important check-up. The application includes a notification system that sends reminders about upcoming appointments and examinations; these notifications are sent via email, providing timely reminders.

Finally, the new functionalities allow users to store and manage reports and results from their medical appointments and examinations, providing a centralized location for all their health-related information. Users can attach documents, such as lab reports, medical imaging results, and doctor's notes, directly to their records within the application. This feature ensures that all pertinent health information is easily accessible and well-organized, enabling users to have a comprehensive view of their medical history.

1.2 Structure

Chapter 2 provides an analysis of the initial state of the project and what is needed to implement these new functionalities. It includes an overview of the original technologies and architecture, new requirements to be developed and use cases that will result from those requirements. The chapter concludes with a comparison of existing market solutions.

Chapter 3 outlines the necessary changes to the project's domain model and database, new architectural design to be implemented and a study of the design for certain functionalities. Additionally, the chapter brings an analysis of potential technologies that could be employed in the system as part of the proposed solution.

In chapter 4, an overview of implementation details is given, highlighting key aspects and processes involved in realizing the project. It also discusses the testing strategies used to evaluate the application and the project's organization concerning version management, code analysis and documentation.

Finally, chapter 5 brings suggestions for future work and the conclusions of the thesis, summarizing the findings and potential directions for further development.

Chapter 2
Analysis

In this chapter, the application requirements are defined and discussed. An analysis of the current iteration of *MediTrack* is performed; it aims to provide insight into the current state of the application, present its functionalities and examine its limitations and overall condition.

A summary of new features is presented, in relation to what was done before, highlighting possible enhancements and advancements. This step is crucial to outline the trajectory of development, shaping the definition of next steps and future priorities. From this study, a series of use cases is designed, detailing how users will interact with the system to achieve the defined goals.

At last, the chapter ends with the exploration of other market solutions within the same domain. This comparative analysis serves to benchmark *MediTrack* against existing competitors, drawing insights into industry best practices, innovative features, and potential gaps that can be addressed to further enhance the application's competitiveness and user experience.

2.1 Original implementation

Given that this thesis builds upon an existing and functional framework, it is essential to describe the initial setup and how the current work used this software architecture as its foundation. This context will provide a clear understanding of the prior work and the enhancements made in this thesis.

The original implementation allows users to register medications with various administration methods, track inventory of associated packages, and plan medication usage. Its key feature is the ability to create schedules for specific medicines; these intake plans can be customized by frequency, specific times and/or days of consumption, quantity, and the full treatment period length. This process automatically generates future intake events, which users can mark as taken or missed. Both upcoming and completed intakes are displayed in a list format, helping users track their compliance with the medication plan.

To support these functionalities, the domain model of the application plays a crucial role in defining the relationships and behaviors of the different entities involved, such as medications, packages, and intake schedules. A domain model serves as a conceptual representation that incorporates both behavior and data, outlining how the various components of the system interact [1].

Figure 2.1 illustrates how the initial five entities relate to each other: User, the main entity, owns multiple medications, intake plans and usages; Medicine has multiple packages associated to them, facilitating the tracking of stock levels for each medication. Finally, an intake plan can generate multiple usage records, providing a detailed log and monitoring of medication usage.

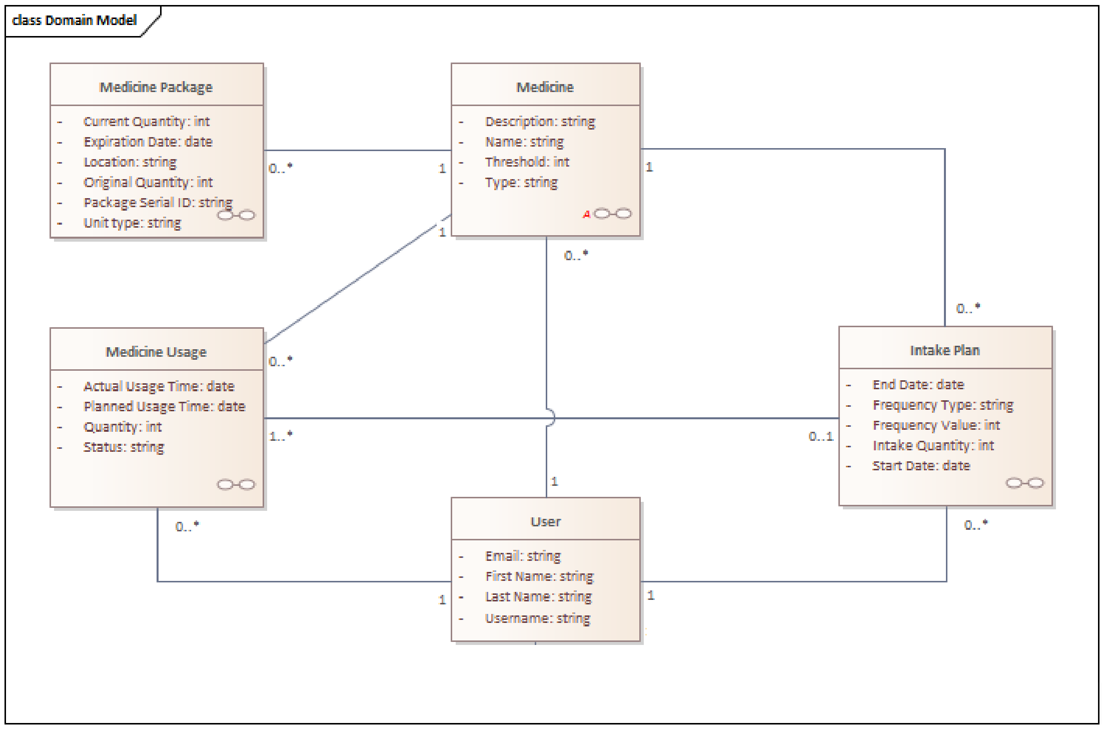


Figure 2.1 Domain model of original implementation

Based on the closeness of those entities, the original work can be divided into three parts. The first part focuses on creating, editing, and deleting remedies alongside with their associated packages. The second part involves generating usage plans and managing intakes, while the last part refers to the user.

The medicine object represents a specific medication, defined by a unique combination of name and type, which is restricted to six options: tablets, capsules, syrups, injections, inhalers or cream. Additionally, it includes a threshold field that specifies the minimum quantity at which the user receives a notification to replenish their inventory.

Packages represent a physical package of a specific medicine in the management system; it contains information about current quantity, expiration date and location. Since they are associated to one medication, they automatically inherit their unit type from it.

Intake plan represents a schedule for taking a specific medicine in the *MediTrack* system. It contains information about the start and end dates, frequency, and dosage related to the medicine intake. The frequency type is limited to a daily, weekly or monthly option, while the frequency value represents how many times in one day the medicine should be taken; the combination of frequency time and value is used to automatically generate recurring individual intakes.

Once the intake plan is defined and its usages are generated, the user has access to their schedule. From this page, it is possible to mark if an existing intake was realized or skipped, with the default behaviour being planned.

Another important feature in usage management is the ability to manually add a new usage outside of the established intake plan schedule, accommodating one-time intakes the user wishes to track. These packages have their stock automatically adjusted upon submission, after a verification ensured sufficient inventory is available.

At last, the final entity to be contemplated in the original work is the user itself. This object represents an individual who is using the *MediTrack* system to track and manage their medicine intake. It contains information about the user's personal details and serves as the central point for connecting the user to their medicines, intake plans, and medicine usage instances.

This entity stands to benefit the most from the upgrades proposed in this work. Firstly, there were no frontend methods available for creating and adding a new user to the database; secondly, there was no user authentication implemented, meaning a third party could read any information in the user's database if they had their username.

2.2 Architecture

2.2.1 System architecture

System architecture, in a software project, refers to the physical organization of software components across various physical nodes, such as servers, computers or databases; it describes how these components interact and communicate within the infrastructure. *MediTrack's* system is a three-tier application with thin client: it is separated into three physical components while relying mostly on the server for processing.

A three-tier architecture is defined by a presentation layer, handled by client devices, an application server that includes the business logic, and a data storage – usually, a database for data persistence. A thin client, then, can be defined as a model where all significant business logic and data processing occur on the server, leaving only presentation logic on the client side – primarily user interface rendering and user input capture. The main advantage of this approach is not requiring software installation on the client device, making deployment and maintenance simpler and more centralized.

MediTrack's client is built using React, a popular JavaScript library for building user interfaces; this layer uses a Node.js server to handle HTTP requests and present static files, together with the Next.js framework. The server is implemented using Java 1.7 with Spring Framework and the database is done in PostgreSQL. Figure 2.2 illustrates how those layers interact.

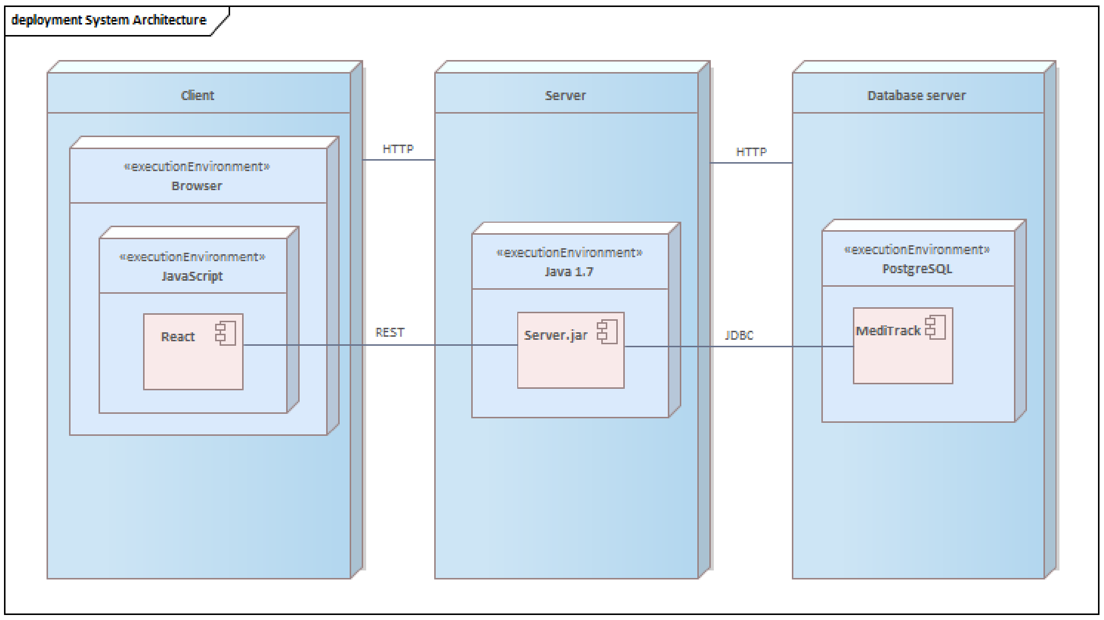

deployment System Architecture
Client
«executionEnvironment» Browser
«executionEnvironment» JavaScript
React
HTTP
REST
Server
«executionEnvironment» Java 1.7
Server.jar
HTTP
JDBC
Database server
«executionEnvironment» PostgreSQL
MediTrack

Figure 2.2 System architecture

2.2.2 Logical architecture

Logical architecture can be defined as the organization of a system's components and their interactions, independent of physical implementation details. It focuses on defining the system's structure and relationships between different modules or subsystems. *MediTrack's* logical architecture is a three-layer application, divided into presentation, business and data layers.

This multi-layered architecture is also known as Model-View-Controller (MVC). It is a widely used software architectural pattern that separates an application into three interconnected components: the Model, the View, and the Controller. The MVC pattern facilitates the separation of concerns, allowing for modular development [2].

The data layer, or model, is responsible for communicating with the data storage, managing the persistence and organization of data within the system. This layer provides the necessary interfaces for accessing and manipulating data.

Business layer, or controller, contains the core functionality of the application, encapsulating the contracts between data manipulation and application behaviour. It acts as an intermediary between the presentation layer and the data layer, processing requests from the user interface and orchestrating data operations.

At last, the presentation layer, or view, focuses on user interaction and interface design, handling the display and interaction with data. By separating concerns into these three layers, software systems can achieve scalability, and maintainability, facilitating easier development, testing, and maintenance of complex applications [3].

A package diagram (figure 2.3) illustrates the layer division in the structure of the project. Model is the data layer on server, while Controller includes both presentation and business layers from backend; View consists of the whole frontend structure.

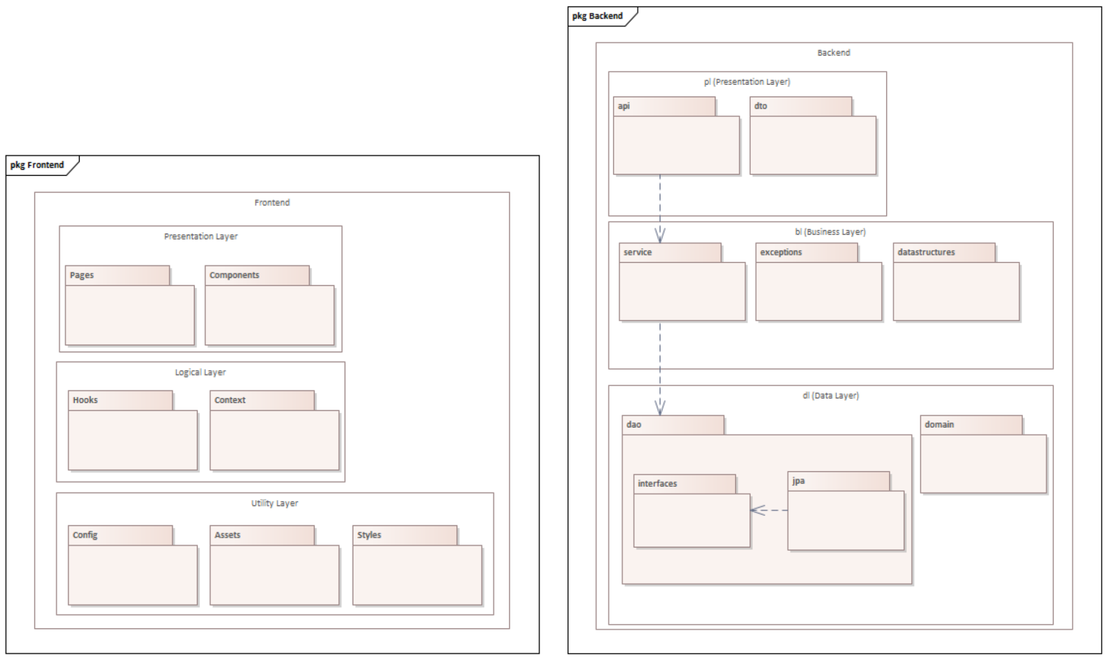


Figure 2.3 Package diagram

To ensure isolation and re-usability in the system, the data layer was abstracted through interfaces; the `dao` package is further divided into two, `interfaces` and `jpa`. The former defines a contract between business and data layers, containing individual interfaces that specify the persistence operations for each entity in the system. Meanwhile, the latter package provides concrete implementations of JPA repositories for each entity – if, in the future, the system must implement a different (or concurrent) type of repository, it can interact directly with the defined interfaces, reducing the amount of rework in the system.

2.3 Technologies

- **Database**

  The chosen relational database management system was PostgreSQL [4], an open-source database. It follows SQL standards, supporting complex queries and diverse data types. Besides having an accessible documentation, this database provides scalability and reliability at a low cost.

  Additionally, PostgreSQL integrates with Java [5] applications through JDBC (Java Database Connectivity) driver, allowing for straightforward database connectivity and operations. It provides an efficient interface for executing SQL queries, managing transactions, and handling database connections. Object-Relational Mapping (ORM) frameworks like Spring Data JPA [6] further simplify the interaction, easily transforming database records into Java objects.

- **Backend**

  The application's backend was built in Java, using Gradle [7] as the build automation tool and Spring Boot framework. In addition to its previously cited database connectivity advantages, Java provides an extensive range of standard libraries that facilitate various stages of development. For example, in the current implementation, Java libraries were used to enhance security features and enable email sending functionality, showcasing its versatility and capability to address diverse project requirements.

  Furthermore, Java's support for RESTful APIs allows communication with clients through HTTP requests, making it easy to build and maintain web services. The combination of Spring Boot and libraries for handling JSON objects guarantees efficient and scalable client-server interactions.

  Finally, it is necessary to highlight two Java features that help current and future students working on this project in keeping the codebase understandable and organized: JavaDoc [8] helps with API documentation, ensuring that developers have easy access to detailed and structured information; testing with JUnit [9] enables thorough unit and integration testing, helping maintain code quality and reliability.

- **Frontend**

  At last, the client side – from where the final user will interact with the whole application – was done using React [10] with Next.js [11] framework. React is an open-source JavaScript library with a component-based architecture, which promotes reusable user interface development. The project is overseen by Npm (Node Package Manager) [12], the package manager responsible for managing dependencies and integrating a vast array of third-party libraries.

- **Docker**

  Docker is an open-source engine that automates the deployment of applications into containers [13]. By running a single application or process in each container, Docker[14] incentives a service-oriented architecture and simplifies distribution.

In this project, three containers were created – database, backend and frontend – using Docker compose, a tool that simplifies the management of multi-container Docker applications by using a single configuration file to define and run multiple services. All containers communicate between each other via HTTP requests.

- **Deployment**

  A CI/CD pipeline follows principles that aim to automate the processes of building, testing, and deploying software applications [15]. Continuous Integration (CI) involves frequently merging code changes from multiple developers into a shared repository, triggering automated builds and tests to detect errors early. Continuous Deployment (CD) focuses on automating the delivery to various environments after a successful integration, releasing reliable software frequently.

  The pipeline was setup with a Gitlab [16] environment as target, consisting of five different stages: build, test, analyze, documentation and publish. The first stage automates the process of creating Docker images for all three layers of the application; the second stage targets backend functionalities, executing automated tests and saving their reports.

  The analyze stage involves automated security testing and a static code analysis, done by a third-party tool, SonarQube [17]; the documentation stage generates a JavaDoc file detailing all backend functionalities. The last stage, publish, is triggered manually and publishes two Docker images to a container registry, one of them with the latest code version.

2.4 New requirements

Having reviewed the original implementation, the next step is to outline additional requirements aimed at enhancing or expanding the project's functionalities. The first essential enhancement is the implementation of user authentication to ensure secure access to sensitive patient data and system functionalities. Another area that is missing is maintaining contacts to individual doctors, as well as tracking past and future appointments and visits. Finally, one extra beneficial addition would be to store all notes, symptoms and exam results in a single location. Next, a list of functional requirements to achieve these goals will be defined.

- **FR1: User authentication and registration**

  Since the goal of this thesis is goal is to release the application for a beta-stage, the application must be secured with authentication – with the simplest method being a check of the user's password and username against the stored credentials in the database. Additionally, the frontend should include a registration form to facilitate the easy addition of new users to the database.

- **NF1: User authorization**

  User authorization is a necessary non-functional requirement that must be added to the project – it ensures users can only access their own data, protecting their privacy. To implement this non-functional requirement, it is crucial to define access control policies specifying who can access what data. As briefly described in section 2.1, the authorization implemented originally required the username as a path parameter, exposing user data to potential third-party interception.

- **FR2: Record of doctors**

  A new entity should be added to the project, representing health care professionals. They should be identified by their name and specialty, and could have other details, such as contact information.

- **FR3: Record of appointments**

  Another enhancement to the project would be to add an appointment entity and scheduling consultations with doctors. This functionality should allow users to register appointments, including details such as doctor's name, location, date and time.

- **FR4: View appointments as calendar**

  The system should incorporate a feature allowing users to view appointments in a calendar format. This calendar view will provide an intuitive and visual representation of scheduled consultations, making it easier for users to manage and track their appointments. This functionality could be repurposed for other existing pages with similar scheduling needs, such as the medicine usage overview.

- **FR5: Send appointment notification**

  Users should be notified about their upcoming consultations, in an effort to reduce missed appointments and improve punctuality. In the context of this thesis, the events contemplated by the notification service primarily include upcoming appointments; however, this functionality can also be extended to broader applications in the future, such as alerting users about expired products, low inventory or reminding them to take medications on time.

- **FR6: Record of reports and related files**

  A report entity would allow users to document descriptions related to their consultations, such as symptoms, feelings, or other relevant observations. Users should also have the capability to attach pertinent files, such as medical records, lab results, or images, to these reports.

  Each report will be directly linked to a specific appointment, providing a comprehensive record of the consultation. This integration ensures that all relevant information is in one place, facilitating better communication between patients and doctors.

2.5 Use cases

After defining the new functional requirements, use cases can provide a clear and structured way to implement these requirements, describing how users interact with the system to achieve specific goals. This section details all use cases identified in section 2.4, outlining the expected interactions between users and the system.

- **User authentication and registration**

  The use case diagram for user management in the *MediTrack's* system consists of three primary use cases: Log in, Create account, and Log out. Figure 2.4 illustrates the mentioned use cases.

  `Log in` use case enables an unauthenticated user to access the system by entering their credentials. User input their valid username and password; upon successful authentication, the user gains access to the system's functionalities and their personal data.

  `Create account` or `Sign up` allows a new user to register and create an account within the system. During this process, the user provides necessary information, such as username, password, and other details. The system validates the provided information and creates a new user profile, if all requirements are met.

  The last use case, `Log out` can only be executed by an authenticated user: after clicking a `Log Out` button, the user terminates their session, preventing unauthorized access to their data. This use case ensures that user sessions are properly managed and that sensitive information is protected when the user is no longer actively using the system.

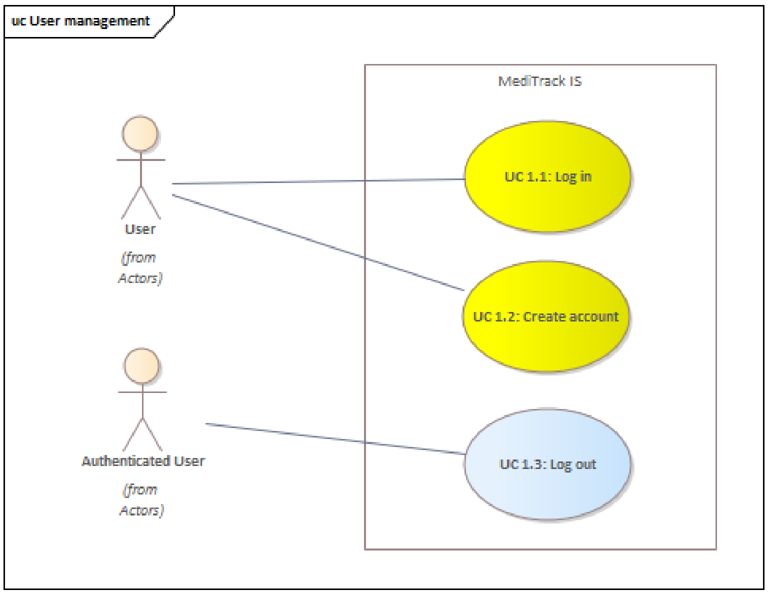


Figure 2.4 Use cases for authentication

- **Doctor management**

  The use case diagram for doctor management presents a set of functionalities that authenticated users can perform to manage their doctor information (see figure 2.5).

  `Add new doctor` allows authenticated users to input and store new doctor information in the system; this involves entering details such as the doctor's name, specialty, email, and phone number.

  `See list of doctors` use case provides authenticated users with a view of all doctors stored in the system; from this page, other use cases can be derived.

  `Edit doctor` enables users to update existing doctor information, changing doctor's details and keeping the database accurate.

  `Delete doctor` allows authenticated users to remove a doctor's information from the system, granted they don't have any associated appointments in the database.

  It is also expected the list view to allow other data manipulation such as filtering, sorting and searching doctors by their attributes – name or specialty, for example.

- **Appointment management**

  The appointment management system has similar use cases functionalities and behavior patterns to those in doctor management system, and is detailed in 2.6.

  `Add new appointment` creates a new appointment by inputting date, title and a registered doctor; `View appointments as list` presents a list of all user's appointments – from this page, it should be possible to edit (`Edit appointment`), delete (`Delete appointment`) or filter/sort/search appointments. Delete is only possible if the appointment has no associated reports.

  At last, the final use case, `View appointment as calendar` allows authenticated users to see appointments in a calendar format and should be independent from the list view.

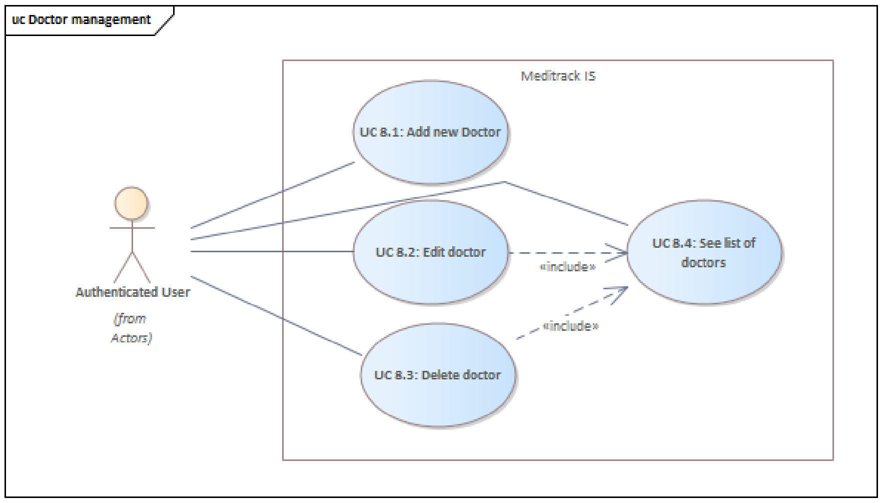


Figure 2.5 Use cases for doctor

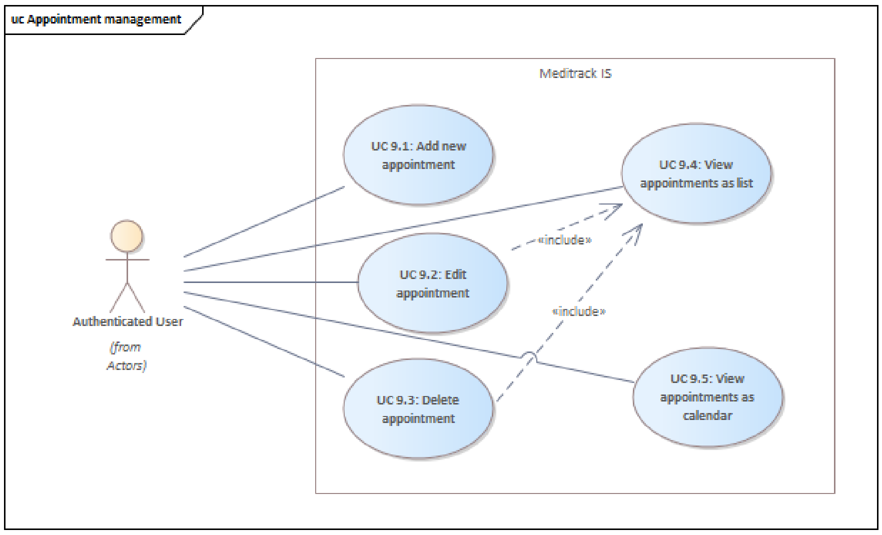

uc Appointment management
Meditrack IS
UC 9.1: Add new appointment
UC 9.2: Edit appointment
UC 9.3: Delete appointment
UC 9.4: View appointments as list
UC 9.5: View appointments as calendar
«include»
«include»
Authenticated User
(from Actors)

Figure 2.6 Use cases for appointment

Report management

Report management has use cases similar to appointment management (figure 2.7). `Add new report` creates a new report with a title, description and associated appointment; `View reports as list` presents a list of all user's reports – and this page allows other use cases such as `Edit`, `Delete` and filter/sort/search features.

The last two use cases, `Add attachment` and `Delete attachment` refer to the file storage functionality; since the attachment is linked to a specific report, these operations should be integrated with the report editing interface. A design decision was made to restrict visibility of files in the user database, allowing users to see files only when associated with a specific report.

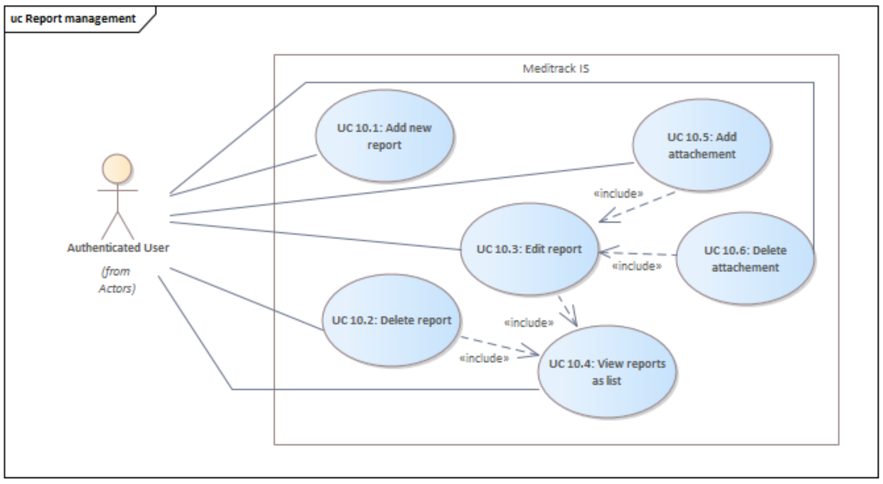


Figure 2.7 Use cases for report

Mail notification

The use case diagram for mail notifications, illustrated in figure 2.8, contains only an automated email sending functionality within the *MediTrack* system. The system is designed to automatically send emails to users – in this work, it is used only for appointment notifications, but it can be reused in other functionalities.

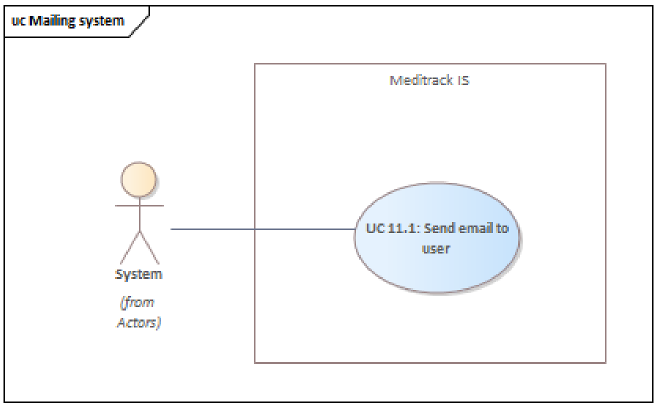


Figure 2.8 Use cases for mailing service

2.6 Existing solutions

In this section, we will compare our project with existing solutions in the Czech Republic market, focusing on their strengths and weaknesses. By examining key features, user experiences, and overall effectiveness, it is possible to identify areas where this application can differentiate itself.

1. **MyTherapy**

   MyTherapy is a popular application designed to support users in managing their medication regimens and health routines. It offers a suite of features, including medication reminders, symptom tracking, and health diary functionalities, which help users follow their prescribed treatments. The app supports a wide range of medications and allows users to log doses, track their progress, and receive push up notifications [18].

   Additionally, *MyTherapy* includes a feature for generating health reports of the metrics measured by the system (medication plan, intakes, measurements and symptom diary). These reports can be shared with doctors, facilitating better communication and informed decision-making.

   The application is free to use with no ads, but accepts a voluntary one-time contribution of CZK 75 or CZK 150. They cooperate with partners such as pharmaceutical companies and healthcare institutes, and vow to not share its user data.

   In comparison to this thesis, *MyTherapy* does not present a list or calendar view of appointments, limiting itself to sending two reminders – one for the day before and another two hours before the appointment. It also lacks a feature to add reports as files, forcing the users to manually input lab values and measurements.

2. **Medisafe**

   Medisafe is a medication management application, whose key features include personalized medication reminders with customizable schedules and dosage instructions. The app offers pill identification, medication interaction alerts, and refill reminders to prevent missed doses and promote safety. *Medisafe* also supports family and caregiver features, allowing users to share their medication schedules and health updates [19].

   The app is free for download and use, but contains a subscription option – CZK 130 monthly or CZK 950 annually – that includes unlimited friends, access to more health measurements and choice of different reminder sounds.

   In comparison to this work, *Medisafe* lacks a built-in calendar management and instead relies on third-party applications like Google Calendar for notifications. It also limits users to adding only textual notes to appointments.

3. **Dosecast**

   DoseCast is a medication reminder application with customizable medication reminders, including options for dosage instructions and timing preferences, ensuring users take their medications on schedule even when traveling between timezones. The app supports a medication list where users can input detailed information about each medicine, including dosage strength and frequency [20].

   It operates on a subscription model, offering monthly payments of CZK 66 or an annual contribution of CZK 625. This subscription includes multi-device synchronization, more types of medication, intake history report and refill alerts.

   The application focus heavily on medication management, not contemplating other functionalities covered by this thesis, such as doctor management, appointment scheduling and detailed reports.

4. **TOM**

   TOM is a daily medication reminder with an integrated medicine list and medication log; the app also records activities and other measurements. It has a pill reminder system as well as graphs and integrated statistics between metrics. The application is completely free and much like *Dosecast*, focuses primarily on medication tracking [21].

The conclusion of this comparison is that, while medication tracking is not a new concept and numerous competitors offer similar functionalities, this project stands out through its integration of medication organization, appointment scheduling and detailed report management.

Some positive aspects from competitors that could be incorporated in the future include a focus on mobile apps with user-friendly interfaces and reminder systems. Another improvement could be the capability to export metrics as a PDF report, providing users with a convenient summary of their health data for easy sharing and reference. *MediTrack* could also expand the range of metrics measured to offer more comprehensive health tracking.

On the other hand, many competitors lack robust features for integrating detailed consultation reports and file attachments. Moreover, while some market leaders offer advanced analytics and insights, they may also come with higher costs or more complex user experiences. By addressing these gaps and building on the strengths observed in other applications, this project aims to deliver a more complete and accessible solution for users.

Chapter 3
Design

This chapter covers the changes necessary to implement the defined new requirements. It will explore the design considerations and strategies required to integrate these new features into the application, including domain model, database, architecture and technologies. Each following section will detail those specific changes and the rationale behind their selection.

3.1 Domain model

To meet the expanded functionalities and new requirements of the project, it was necessary to introduce additional entities into the system. These new entities are: doctors, appointments, reports and files. The user entity also had to be updated to support new functionalities. Figure 3.1 illustrates the changes made in the domain model, with affected entities in a different colour.

The `Doctor` entity includes fields such as name, specialty, phone number, and email contact – with the latter two being optional – and has a many-to-one association with the user entity. `Appointment` will require attributes for date, time, location and note – where the user can questions or symptoms to discuss during the visit –, while being associated to one doctor from the database. Their status field takes into consideration if the given appointment was carried out, missed or is yet to take place.

`Report` entity is linked to one appointment and features a description field for detailed documentation from the patient's perspective, as a result of the appointment or examination. The last new entity, `File`, is responsible for storing any type of documentation related to a specific report; the fields name and type ensure that the byte data is restored to the correct format.

The `User` entity has a one-to-many relationship with doctors, appointments, and reports, facilitating efficient querying of all data related to a single user. Two new fields have been added to this entity: password, used for authentication, and timezone, utilized by the notification service as will be further explained in chapter 4.

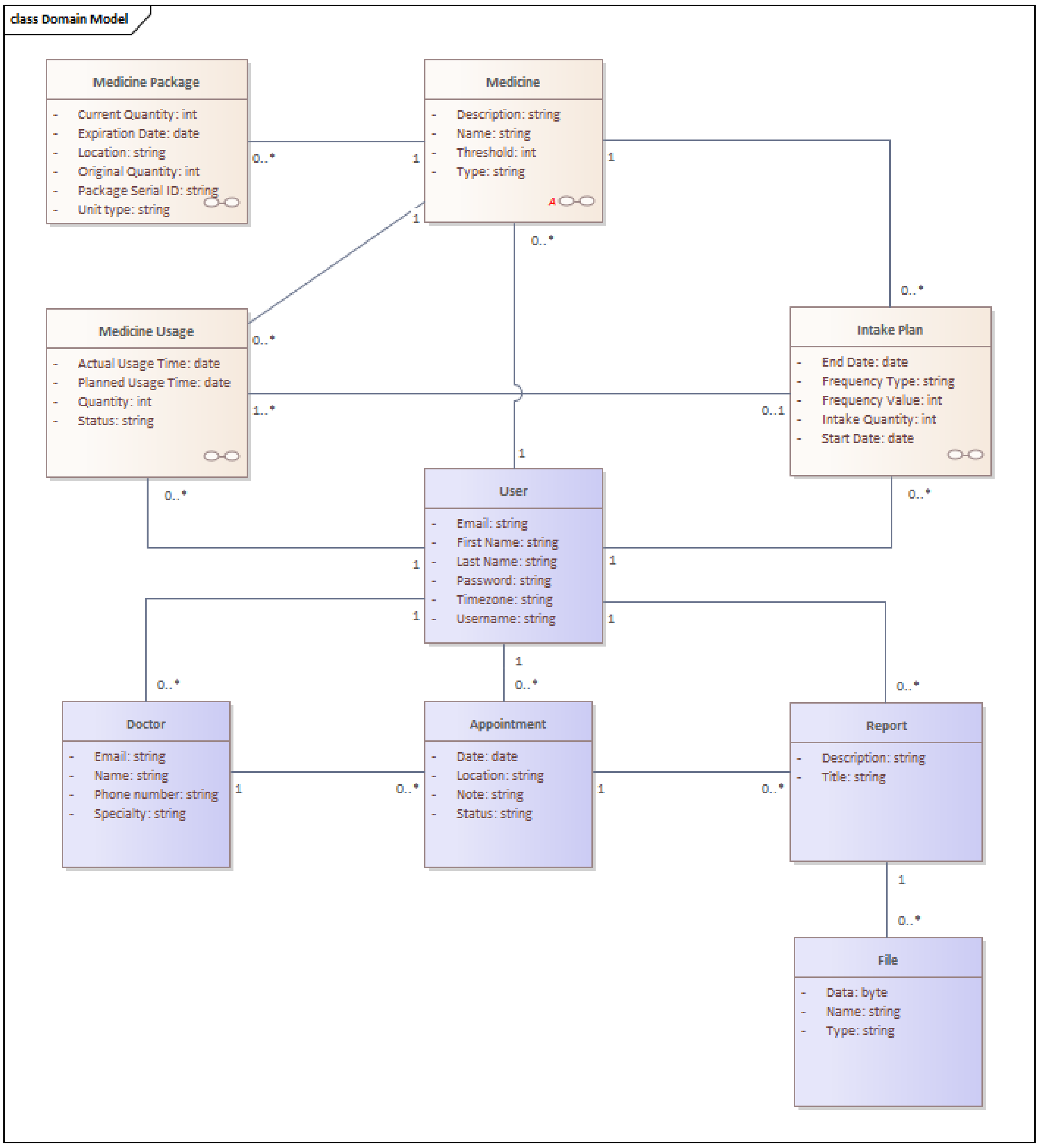


Figure 3.1 Domain model of current implementation

3.2 Technology analysis

In this section, various technological solutions are explored, with the goal of finding the better suited options to address the functional requirements outlined for the *MediTrack* application on section 2.4. This discussion overviews a range of tools and frameworks for implementing user authentication, calendar presentation, email notifications and file management.

3.2.1 User authentication

Authentication and authorization are two fundamental concepts in information security, often used together but serving distinct purposes. Authentication is the process of verifying the identity of a user, ensuring that the entity attempting to gain access is indeed who it claims to be; authorization is the process of determining what an authenticated user is allowed to do, through permissions and access levels [22].

In the context of this work, authorization ensures that users can access only their own data, protecting the privacy of each user's information. Authentication, on the other hand, guarantees that only registered users can access the system, preventing unauthorized access and ensuring that all interactions within the application are performed by verified individuals. Both methods are expected in a secure application, as defined in functional requirement FR1 in section 2.4.

Several approaches to user authentication are commonly used, each with its unique features and benefits. Four common alternatives were considered: Basic Authentication, OAuth, Cookie-session Authentication and Token Based Authentication. Next, a brief description of the original implementation will be given, followed by a discussion over each proposed solution.

3.2.1.1 Original Authentication

In the original implementation, authorization was done using the user identifier — specifically, the username — as a parameter in all REST requests, leading to unsafe data handling. For example, to access all medicines for a user, the server expected a GET request in the format `/medicine/username/{username}`, with this same approach applied to all other CRUD operations. This meant that if a third party obtained a user's username, they could potentially access, read, and modify any information in the user's database.

User authentication was not implemented in the original work. The login page was inadequate for security: it merely checked if the username existed in the database, as users had no password information stored, and redirected to the homepage in case of a match.

3.2.1.2 Basic Authentication

Basic authentication is the simplest form of authentication: the client sends HTTP requests with a header containing the concatenation of the user's username and password in a base64-encoded string, which is then compared with the information stored in the database. Despite its simplicity, Basic Authentication has notable security drawbacks, primarily because the encoding is easily reversible and the credentials can be intercepted if not sent over a secure connection. Therefore, while it is useful for internal applications, it is generally recommended to use more secure authentication methods for production environments [23].

3.2.1.3 OAuth2

On the opposite spectrum, OAuth2 (Open Authorization) is considered one of the most robust authorization frameworks. It delegates user authentication to an external identity provider, which verifies the user's credentials and issues an access token without exposing those credentials to the client application [24]. This approach enhances security by minimizing the exposure of sensitive information. Moreover, OAuth2 supports granular access control through scopes, allowing clients to request limited access to specific resources on behalf of the user.

However, the main disadvantage of this approach – and the reason it was not chosen for implementation in this project – is that it requires either a custom identity provider or the user to have an account with a specific identity provider. This would translate to demanding users have accounts on platforms such as Google, Facebook, Github or Linkedin, which are among the most widely used providers in Java.

3.2.1.4 Cookie Authentication

Cookie-session authentication uses HTTP cookies to authenticate client requests and maintain session information. After a successful login, the server generates a unique session identifier and store it in a cookie, with other relevant information; then the client needs to attach this cookie in all subsequent requests to the server, in order to authenticate the user. Finally, on the logout operation, the server sends back a header that causes the cookie to expire, invalidating future requests with this specific cookie [25].

Though cookie authentication is straightforward to implement, it comes with two major drawbacks: it is susceptible to cross-site scripting (XSS) and cross-site request forgery (CSRF) attacks. On the first type of attack, a malicious script is injected on the user's browser, through inputs that are improperly sanitized or validated [26]. The script can be either stored on the server or executed immediately, potentially stealing data or performing actions on behalf of the users.

The second type of attack, CSRF, is also known as a one-click attack or session riding, as the attack takes advantage of the user's previously authenticated session. A malicious site tricks the user into making a legitimate request and steals the cookie sent for authentication, being able to impersonate them from there on [27].

Both types of attack can be avoided by implementing secure coding practices: output encoding and HTML sanitization for XSS and other techniques such as generating unique tokens for each user session, checking if cookies were sent from same site or using signed double submit cookies with a secret key known only to the server for CSRF. It is also crucial to ensure that any sensitive information stored within cookies is encrypted, to avoid exposing user data that could be used later in a different attack.

3.2.1.5 Token Authentication

Finally, the last type of authentication considered for this project was Token Based Authentication, done with JWT (JSON Web Tokens). It involves using a token with all information needed for verification and identification – typically, details about the user and their permissions, token's expiration time and other metadata, all encoded as a Base64 string [28]. Finally, the last step is to create a signature by combining the encoded header and payload with a secret key using either digital signatures or HMAC algorithms (Hash-based Message Authentication Code).

These signed tokens are stored in a local or session storage, on the client side, and sent with each request; this results in a stateless process, since the server does not need to keep a record of each active session but rather know only the secret key and algorithm necessary for encoding/decoding tokens, improving scalability.

However, token-based systems can be more complex to implement, and storing tokens in local storage exposes them to XSS attacks. Token invalidation can be challenging, requiring strategies like token expiration, where a token automatically becomes invalid after a short period of time, used in combination with refresh tokens, where a special token is responsible for requiring new access tokens when the current one expires.

If a token needs to be revoked – for example, after a password change or a logout – it requires additional mechanisms like token blacklists, a list that keeps track of tokens that are no longer valid, even if they have not yet expired [29].

3.2.1.6 Conclusion

Considering the pros and cons of each authentication strategy, token-based authentication was chosen as the most suitable for this project. Tokens are particularly advantageous because they are stateless and self-contained; additionally, they can be used across different platforms and devices, making them ideal for a web application intended to be used in distinct environments.

3.2.2 Calendar

The project could benefit from a visual representation of multiple time-based recurring data sets, as outlined in functional requirement FR4 (section 2.4). To display data in a calendar format, a new component was required on frontend. This calendar functionality was expected to support various views (monthly, weekly, and daily), handle recurring events, allow for easy event creation and editing, and provide a customizable interface to match the application's design.

Given the complexity of implementing a full-featured calendar from scratch, open-source options were considered: calendar functionalities are not an uncommon demand for this type of project. Two popular libraries emerged as strong candidates:"react-big-calendar" available from npm [30], and "FullCalendar" [31].

"react-big-calendar" is a React-specific solution that offers different views, drag-and-drop event management and localization support. It's known for its React-friendly API and good performance with large datasets. However, it has a smaller community compared to "FullCalendar" and less feature updates.

"FullCalendar", on the other hand, is a more established and widely-used library that supports multiple frameworks, including React. It offers set of features such as various views (monthly, weekly, daily, and even timeline), handling of recurring events and advanced timezone support. "FullCalendar" also provides extensive theme options and a larger number of (paid) plugins for additional functionality.

It's worth noting that "FullCalendar" served as the inspiration for "react-big-calendar", suggesting its approach has been validated by other developers and consolidating itself as the choice for this project.

3.2.3 Notification service

Users expect *MediTrack* to notify them about urgent or important events, as described in functional requirement FR5, section 2.4. Given that this project is a web application, the most efficient way to reach our users is through email notifications. In Java environment, there are two recognized libraries for achieving this: JavaMail[32], a standard Java library, and SimpleJavaMail[33], a wrapper built on top of the former library, providing a higher-level API.

The selected framework for this thesis was JavaMail, who sends emails via Simple Mail Transfer Protocol (SMTP) server; retrieving messages would require a different protocol, Internet Message Access Protocol (IMAP), which is not covered in this project. The SMTP server is responsible for processing the email request and actually delivering the email – to do so, it requires a valid email address as an anti-spam measure, to verify the identity of the sender.

To send an email using JavaMail, the application needs to create a Session object, which encapsulates the properties and configurations necessary for establishing a connection with the mail server, such as address and password. Next, a Message object is created and populated with the necessary information, such as the sender, recipients, subject, and message body. Then, a Transport class is used to send the email by connecting to the mail server using the specified SMTP protocol and delivering the Message object.

At last, as a final step, to guarantee a secure connection between the application and the SMTP server, a cryptographic protocol should be established – either TLS (Transport Layer Security) or SSL (Secure Sockets Layer). They encrypt the data exchanged, protecting the email content and any sensitive information it may contain from being intercepted and read by unauthorized parties.

While SSL and TLS serve the same purpose of secure communication, TLS is the successor of SSL and considered more secure, being the recommended protocol to use in modern applications [34]. TLS incorporates improvements and fixes for known vulnerabilities in SSL, offers better encryption algorithms, and provides better protection against various types of attacks, being therefore the choice of protocol for this thesis.

3.2.4 File storage

The *MediTrack* system should allow users to store their examination results as an attachment to reports, as specified in functional requirement FR6 in section 2.4. There are multiple ways to save files in this project, each with its own pros and cons. The primary options are storing files in a local file system, in the cloud, or in the database itself. Each approach has unique benefits and challenges that need to be considered.

3.2.4.1 Local file system

For an application deployed using Docker containers, saving files to a local system involves using volumes [35]. They allow data persistence, ensuring data is not lost when containers are stopped or recreated. A dedicated volume for file storage would need to be created and configured to communicate with the appropriate containers.

To properly retrieve these files from the volume, the database stores a symbolic link that acts as a "pointer" to the file's storage location. This approach supports very large files (greater than 100 MB), but introduces several complications: the database user must have permissions to write outside the database, which is not a default option for PostgreSQL environments; an interface is needed to keep track of externally attached files, and there is a risk of files and the database becoming out of sync, since operations are no longer transactional. Additionally, an extra security layer must be implemented to restrict volume manipulation exclusively to authorized users.

3.2.4.2 Cloud storage

Another viable option is using cloud object storage providers like Amazon Simple Storage Service S3, Azure Blob Storage or IBM Cloud Object Storage. The main advantages of this approach are being able to access files with just an internet connection and being able to scale the service without requiring additional infrastructure [36].

This approach, however, creates a dependency on third-party services, making the project reliant on their availability and policies. Moreover, costs can increase proportionally with usage, potentially making it expensive as data storage and retrieval needs grow.

3.2.4.3 Database storage

The last option is to store the file in the database itself. For this, a manipulation of the file is necessary: transform it into a binary string (`bytea`) [37] or a binary large object (LOB)[38]. The first approach involves encoding the file content into a byte array and storing it directly in a column of type `bytea` in the database table; this means storing and accessing those entries works in a similar way as accessing any other data type. The main drawback with this data type is its size limitation: PostgreSQL can store up to 1 GB of data per entry.

A binary large object, on the other hand, requires special handling due to its size. To manage large objects efficiently, PostgreSQL breaks them into smaller, manageable chunks and stores these chunks in rows within the database. Each chunk is uniquely identified by an Object Identifier (OID), which serves as a reference to that specific part of the large object.

In regards to storage capacity in large objects, PostgreSQL can hold up to 4 TB per entry. It is also possible to seek over entries, which allows search for words inside files. Moreover, LOBs eliminate the need for encoding of data, simplifying the storage and retrieval processes.

The large object can be further split into two classifications: a character large object (CLOB), used to store large text data, or a binary large object (BLOB), for storing binary data like image, audio, or video. Hibernate uses the appropriate large object type based on the Java type of the attribute – strings defaults to CLOB, while array of bytes defaults to BLOB, for example.

3.2.4.4 Conclusion

For this project, the binary large object approach was selected, as it strikes a good balance between high storage capacity and ease of implementation and integration with the existing infrastructure.

3.3 Database

Building upon the foundation established by the domain model (section 3.1) and the subsequent technology discussion (section 3.2), it is now possible to design the necessary changes into the project's database. The domain model provided a conceptual framework of the entities, their attributes and their relationships, while the technology analysis delimited the capabilities and constraints of the model.

The same relational database from the original implementation is being used, with added entities and relations. Figure 3.2 illustrates the changes in the database and how the new entities will relate with one another. For simplification, only the changed entities are represented in the image.

`User` now has two non null-able new fields, password and timezone, responsible for aiding authentication implementation and correctly showcasing the user's timezone regardless of machine setup – both fields are set up as `strings`. `User` is still the central entity of the model, having `One to Many` relations with `Doctor`, `Appointment` and `Report`.

`Doctor` has a unique identifier `doctorId` defined as a `bigserial` – in PostrgreSQL, this defines a large integer that is automatically incremented each time a new row is inserted in the database. The entity includes attributes like name, specialty, email and phone number, all defined as `strings` with the last two being optional. A `Doctor` can have multiple `Appointments` and is related to one single `User`.

`Appointment` is also identified by a `bigserial` unique `appointmentId`; other attributes include date, location, note and status, with the first being a `date` (without timezone information) and the others, `strings`. This entity is connected to both `User` and `Doctor` entities via foreign keys, ensuring it is associated with a specific user and doctor.

`Report`, like the previous entities, is identified by a unique `reportId`. The entity contains title and description, both `strings` but with a greater size for the latter. A `Report` is related to one `User` and one `Appointment` and may have multiple `File`.

The last entity, `File`, is identified by a unique `fileId`, with attributes like name (`string`), type (`string`) and data, defined as a `LOB`. `File` entity is linked to `Report` entity through the `reportId` foreign key, enabling the storage of multiple files relevant to a specific medical report.

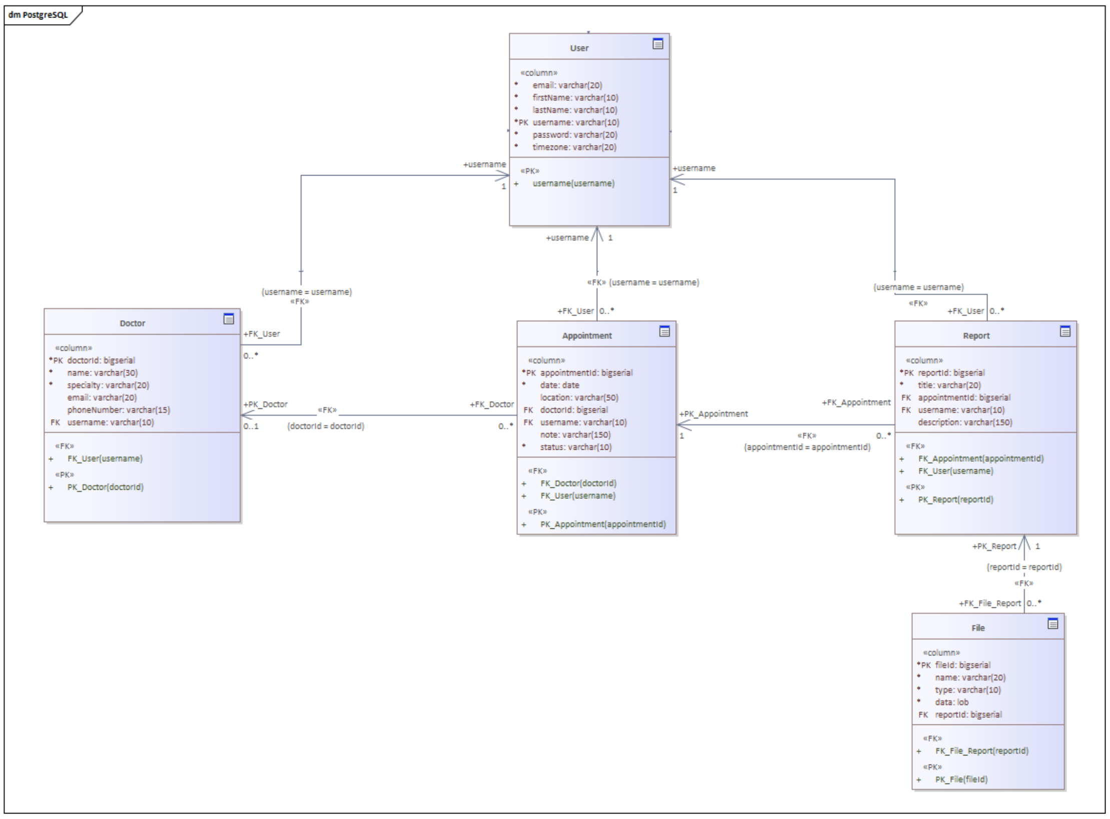


Figure 3.2 Database for new implementation

3.4 Architecture

The physical architecture described in section 2.2 remains the same; the logical architecture, however, will require some enhancements to its package organization. Figure 3.3 displays only the newly introduced classes and packages, for simplicity.

The new `mail` package is responsible for configuring the mail provider; this includes setting up properties such as the SMTP server address, port, and authentication details. These properties are centralized within configuration classes, ensuring they are not scattered throughout the code and enhancing code maintainability.

Similarly, the new `security` package is responsible for configuring the authentication provider. This configuration ensures secure access to the application by setting up properties related to user authentication, authorization mechanisms, and security protocols.

For each new entity added to the database, specific controllers are implemented to create endpoints for CRUD (Create, Read, Update, Delete) operations and other custom functionalities. These controllers handle HTTP requests and responses, ensuring that the application can interact with the database efficiently. `Appointment`, `Doctor` and `Report` controllers implement basic functionalities like creating/editing/deleting an entity, searching one specific entity or searching all entities that satisfy certain conditions.

`File` controller needs to approach the file creation in a slightly different manner, as creating a new file includes uploading it to the database with the approach defined in section 3.2.4. This controller must also incorporate a method to manage file downloads, ensuring that files are retrieved and served in their correct format.

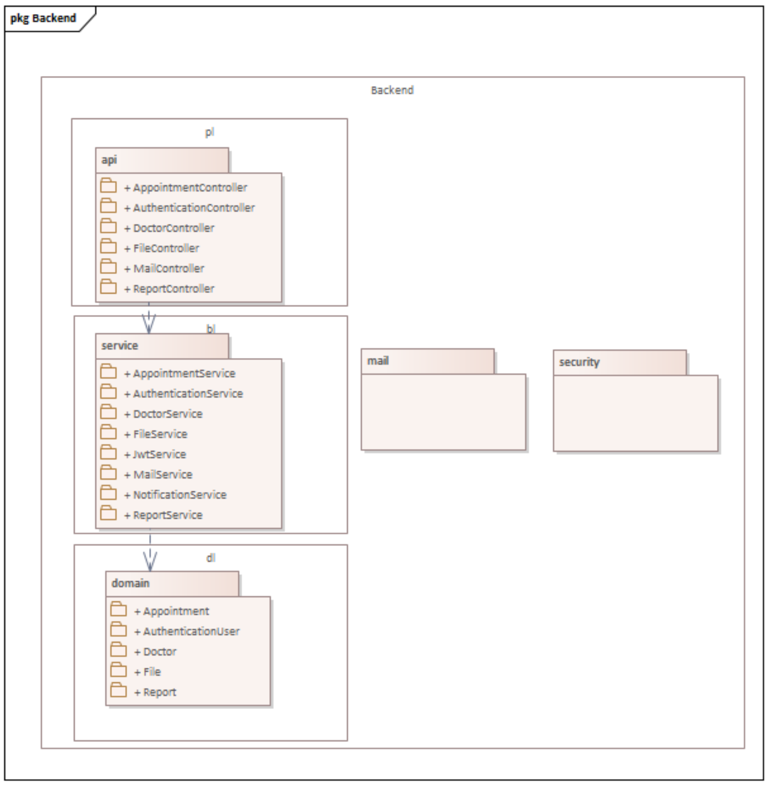


Figure 3.3 Backend package diagram on new implementation

`Mail` controller serves as a specialized interface to handle various email sending scenarios, including both individual and bulk email dispatches. For individual emails, the interface should support personalized content, specific recipient addressing and, in the future, attachment handling. This functionality is appropriate for user-specific notification or account-related messages.

The bulk email interface, on the other hand, should be designed to manage the sending of a template message with personalized content for multiple recipients. This feature is particularly useful for system-wide announcements or large-scale notifications.

At last, the `Authentication` controller is responsible for handling user authentication and authorization processes. It serves as the entry point for authentication-related requests, providing endpoints for login, logout, and potentially other security-related operations. This controller is separated from the `User` controller, a best practice in software design, as it delimits specific responsibilities for each one: `Authentication` deals with username, password and token management, while `User` handles CRUD operations on user profiles, with more details (email, timezone, name, etc).

The service layer acts as an intermediary between the controllers and the repositories; therefore, new classes are needed for the added entities. The `Appointment`, `Doctor`, and `Report` service classes handle the previously defined CRUD operations, while the `File` class implements methods for file upload and download.

`Authentication` service verifies the username and password, and also handles password encryption to protect user data in case of a database compromise. `Jwt` service is responsible for token management, including token generation and extracting its claims such as username and expiry date.

`Mail` service provides email-sending functionality. `Notification` service, then, utilizes the `Mail` service for specific tasks, such as scheduling emails related to appointments. Equivalent to the pair `User` and `Authentication`, the separation between the `Mail` and the `Notification` is a design choice that follows the principles of modularity and separation of concerns, creating loosely coupled, highly cohesive components that can be easily understood, modified, and reused.

Finally, the data layer must implement the new entities from the domain model – `Appointment`, `Doctor`, `Report` and `File` –, as well as the `AuthenticationUser` entity. This ensures that the database schema supports the new functionalities and entities.

3.5 Authentication design

Based on the analysis of the proposed solution, the most significant change in the original application is the introduction of user authentication. This enhancement will alter how requests are sent and processed between the client and server. To illustrate this change, two sequence diagrams are provided in this section, detailing the execution of the login process and one read operation in the new implementation.

Figure 3.4 illustrates the overall login process discussed in section 3.2.1.5. In this scenario, the user submits their credentials (username and password) to the login endpoint. The server verifies these credentials against the information stored in the database; if the credentials are valid, the server generates a signed token and returns it to the client. The client then stores this token and render the homepage.

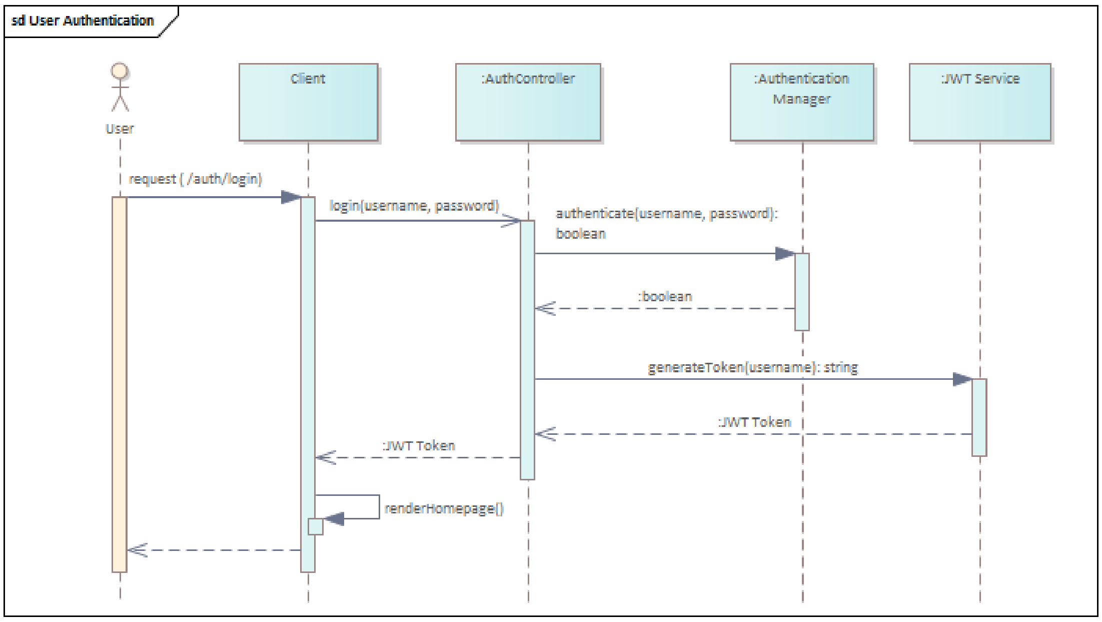


Figure 3.4 Login sequence diagram

Once a user has successfully logged in, every subsequent request must demonstrate that it is from an authenticated user and must also identify the user making the request. Figure 3.5 illustrates a simple read operation in the system, where the user requests information on a specific entity, and how authentication and authorization are applied throughout the process.

First, the user sends a GET request to the endpoint `/doctor/id` to retrieve information about a specific doctor in their database; this request has attached to it the token generated during login. `JwtAuthFilter` – a class from Spring Security framework – intercepts the incoming request before it reaches any controllers and decrypts the token to verify three key aspects: whether the token's signature is valid – ensuring the message has not been tampered with –, if the token's expiry date is still valid, and, in case the two previous checks are successful, the username contained within the token. All this information is then stored in the security context, to be accessed by any class that needs it.

Next, `DoctorController` checks if the context username is authorized to make this request. Specifically, it verifies whether the doctor in question belongs to the user making the request. This authorization check is performed by the service layer, which queries the repository to confirm that the user column of the doctor entity matches the username in the token.

If the authorization check is successful, the service layer retrieves the doctor entity from the repository and returns it to the controller. The controller then prepares the entity and sends it back to the client. Finally, the client renders the correct page with the requested information.

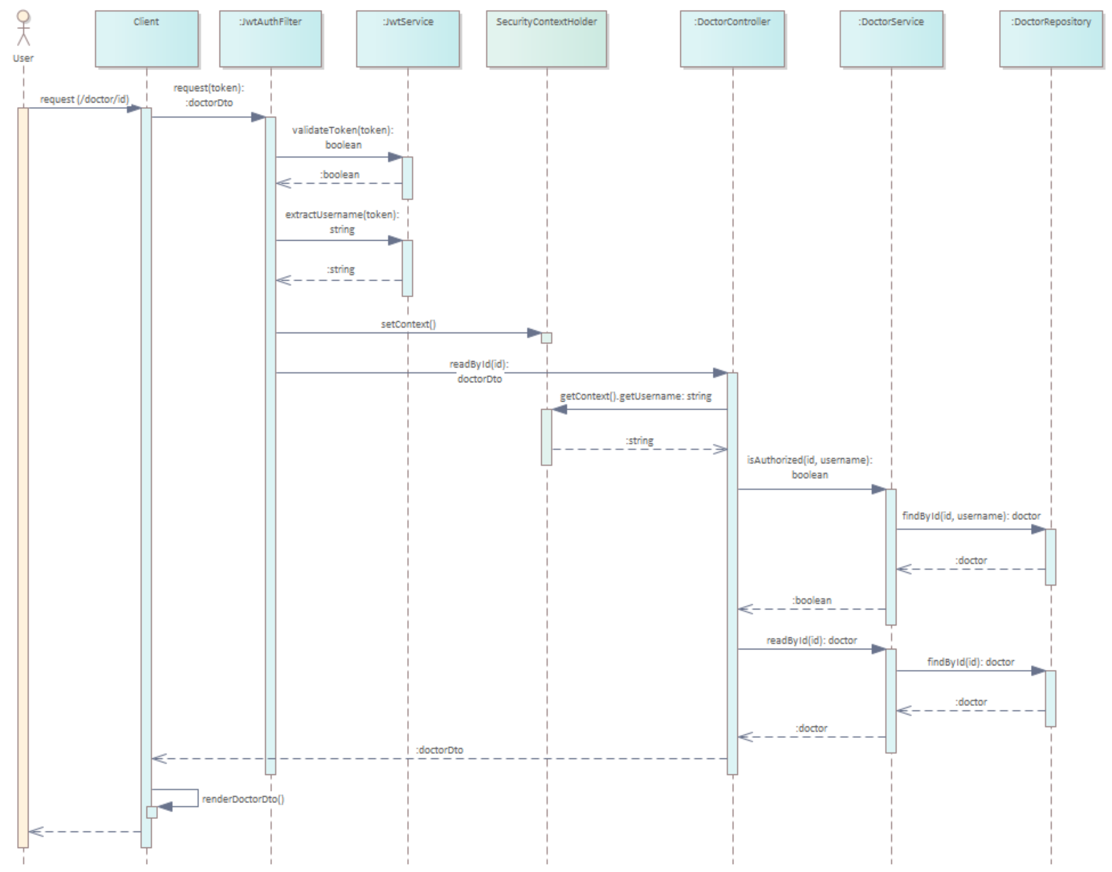


Figure 3.5 Authenticated request sequence diagram

Chapter 4
Implementation

Following the discussion of the original codebase and the necessary changes to meet the new requirements, this chapter presents the implementation phase of the project. It will outline the approach taken to integrate the identified features, refactor existing components and address any technical challenges that appeared during the development cycle.

4.1 Features

4.1.1 Homepage

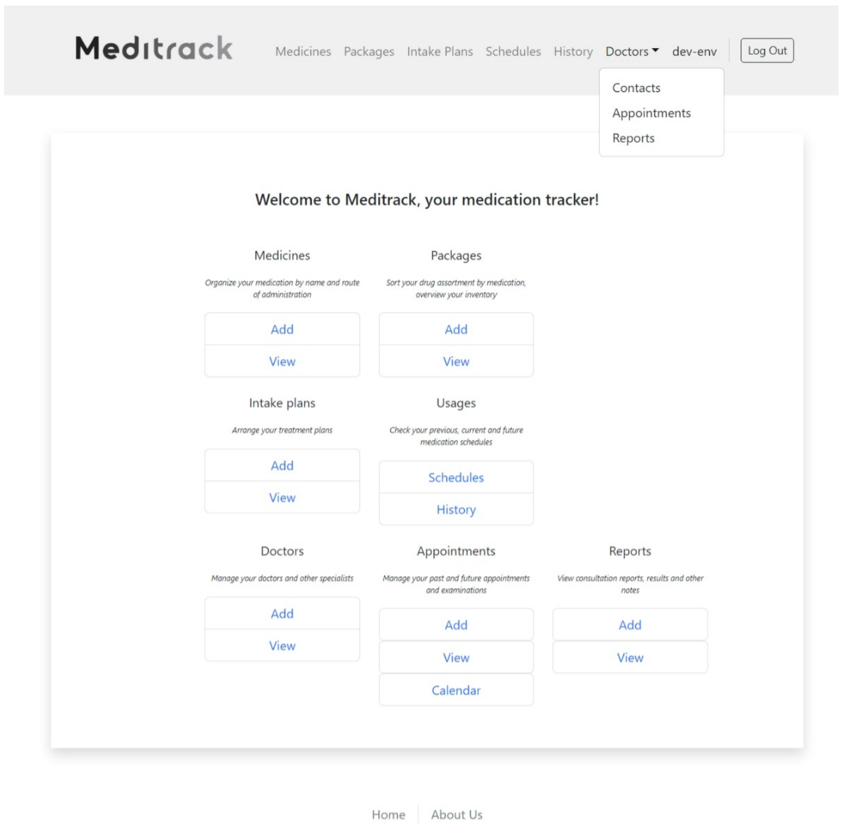
The first step of implementation phase was to update the homepage of the web application to incorporate links to the newly added entities. The new additions integrate with the existing user interface, resulting in a consistent experience, and are represented on figure 4.1. “Add” links redirect to a form to create new entities, “View” links present a list of all entities for the currently authenticated user and “Calendar” presents all appointments in a calendar view. Additionally, a dropdown menu was introduced at the top of the page, providing a centralized navigation system; all links on dropdown menu redirect to “View” pages.

4.1.2 Date manipulation

The original implementation had some technical debt regarding date manipulation. On frontend, date objects were handled as strings when being modified by user input; this method was unsafe due to potential errors in string manipulation, leading to unreliable date handling. To address this, the new implementation employs the DatePicker [39] and TimePicker [40] components from MUI, ensuring that all dates are managed as date objects and correctly validated. Additionally, a global configuration guarantees all dates are presented in the same format, as `day/month/year` with two digits for the first two variables and four digits for the year.

A second issue with dates was coming from backend: all dates were previously defined as `LocalDateTime`. JavaScript’s method to serialize a Date object into a JSON string transforms it into a ISO 8601 format while also converting it into UTC timezone [41]. Java then would re-transform the date into `LocalDateTime`, save it in the database and send it back to the client without any timezone information.

The end result was a miscommunication between the frontend and backend: the client interpreted the received local time as UTC and wrongly adjusted it once more to local time. The solution was to change all `LocalDateTime` to `ZonedDateTime` in the Java code, which carries the timezone information and would no longer transform before saving into the database [42].



Welcome to Meditrack, your medication tracker!

Organize your medication by name and route of administration

Sort your drug assortment by medication, overview your inventory

Arrange your treatment plans

Check your previous, current and future medication schedules

Manage your doctors and other specialists

Manage your past and future appointments and examinations

View consultation reports, results and other notes

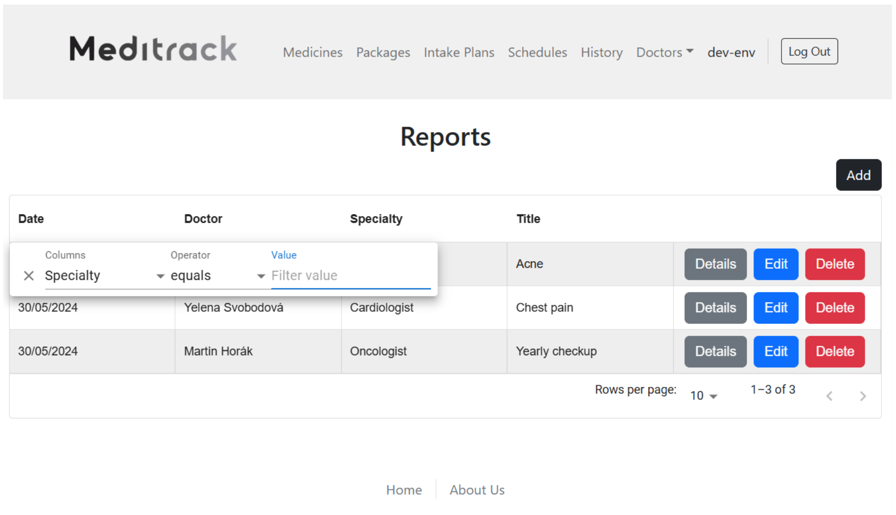
Figure 4.1 Homepage with dropdown menu

4.1.3 Filtering data

The next improvement implemented in the project was using a third party component to sort, filter, search and paginate all pages containing lists of entities. Previously, these functionalities were implemented manually; for example, pagination for packages involved fetching all data at once and then using custom functions to divide it into buckets of a certain size for navigation. Similarly, search functionality relied on simple string matching between the target value and the entity field. This approach resulted in multiple redundant functions that had to be copied and slightly adapted for each page requiring these features.

The same library used for DatePicker, MUI, contains a component for manipulating a set of data into rows and columns, DataGrid [43]. This project uses the MIT license, which comes with basic features like editing, pagination, column grouping, and single-column sorting and filtering. Limitations of the license are: pagination is restricted to pages with a maximum of 100 rows — users can choose to display 10, 20, 50, or 100 items per page — and both sorting and filtering are limited to one column at a time.

Figure 4.2 illustrates one of the pages using the component, with a filter selection enabled. The component was designed to be generic, allowing different pages to customize the table according to their specific needs while maintaining project consistency. The table supports any number of columns that can be filtered and sorted, except for the last column, which contains only buttons. This final column is neither sortable nor filterable and expects predefined behaviors for each button.


Meditrack
Medicines Packages Intake Plans Schedules History Doctors dev-env Log Out
Add
Home About Us

Reports

Columns: Specialty | Operator: equals | Value: Filter value

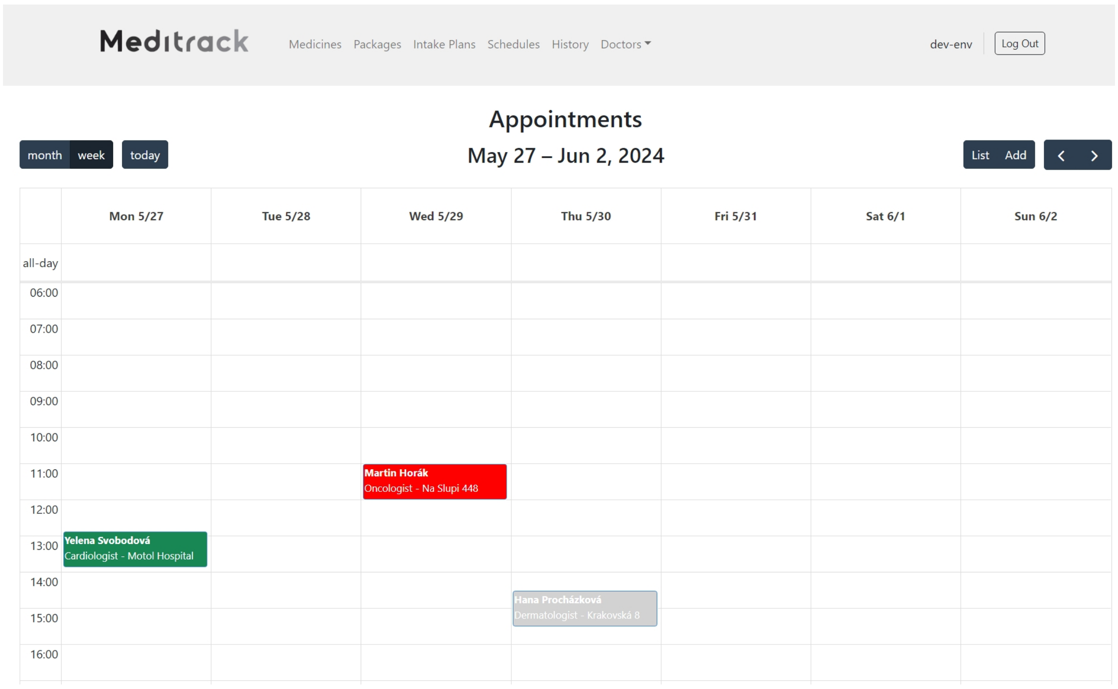
Date	Doctor	Specialty	Title	
			Acne	Details Edit Delete
30/05/2024	Yelena Svobodová	Cardiologist	Chest pain	Details Edit Delete
30/05/2024	Martin Horák	Oncologist	Yearly checkup	Details Edit Delete

Rows per page: 10 1–3 of 3

Figure 4.2 Example of table with filter

4.1.4 Calendar

Similar to the table component, the calendar was built in a generic way to be implemented in different pages. The component was adapted from FullCalendar with minor modifications. Users can switch between weekly and monthly views using the "week" and "month" buttons located at the top left, while the "today" button resets the view to the current date. Additionally, two custom buttons are available at the top right: the "add" button redirects users to the appointment creation form, and the "list" button leads to a table view of all the user's appointments.



Appointments

May 27 – Jun 2, 2024

Martin Horák
Oncologist - Na Slupi 448

Yelena Svobodová
Cardiologist - Motol Hospital

Hana Procházková
Dermatologist - Krakovská 8

Figure 4.3 Calendar with events

Figure 4.3 showcase a weekly view of the Calendar, displaying three different types of events. For each event, the title displays the doctor's name, while the body includes their specialization and the appointment location. Event colors change based on the appointment status: red for skipped, green for completed, and grey for default. This behavior is defined in a global CSS file, implemented by other students in the Team Software Project group.

4.1.5 Authentication

Authentication is the most crucial feature implemented in this thesis, transforming *MediTrack* into a secure platform. As previously described in section 3.2.1, a token-based authentication system was employed. Implementing a JWT authentication system in a Java project involves setting up a series of configurations and classes that work together to secure the application and manage user authentication.

The process begins with configuring security settings through a dedicated class, which extends the features provided by the Spring Security framework. This configuration specifies which endpoints are publicly accessible and which endpoints require authentication. In this work, login and sign-up endpoints are left unauthenticated since the user has not yet had the opportunity to authenticate. Endpoints related to the mailing service are restricted to one authenticated 'admin' user. Currently, this verification is done via username; however, a future improvement would be to incorporate roles into the database, allowing any user with an 'admin' role to access these methods.

The security configuration class also ensures the application does not maintain session state by setting the session management policy to stateless, which is crucial for token-based systems, where each request must be authenticated independently. Code 4.1 illustrates the main method in this class, incorporating the previously described properties.

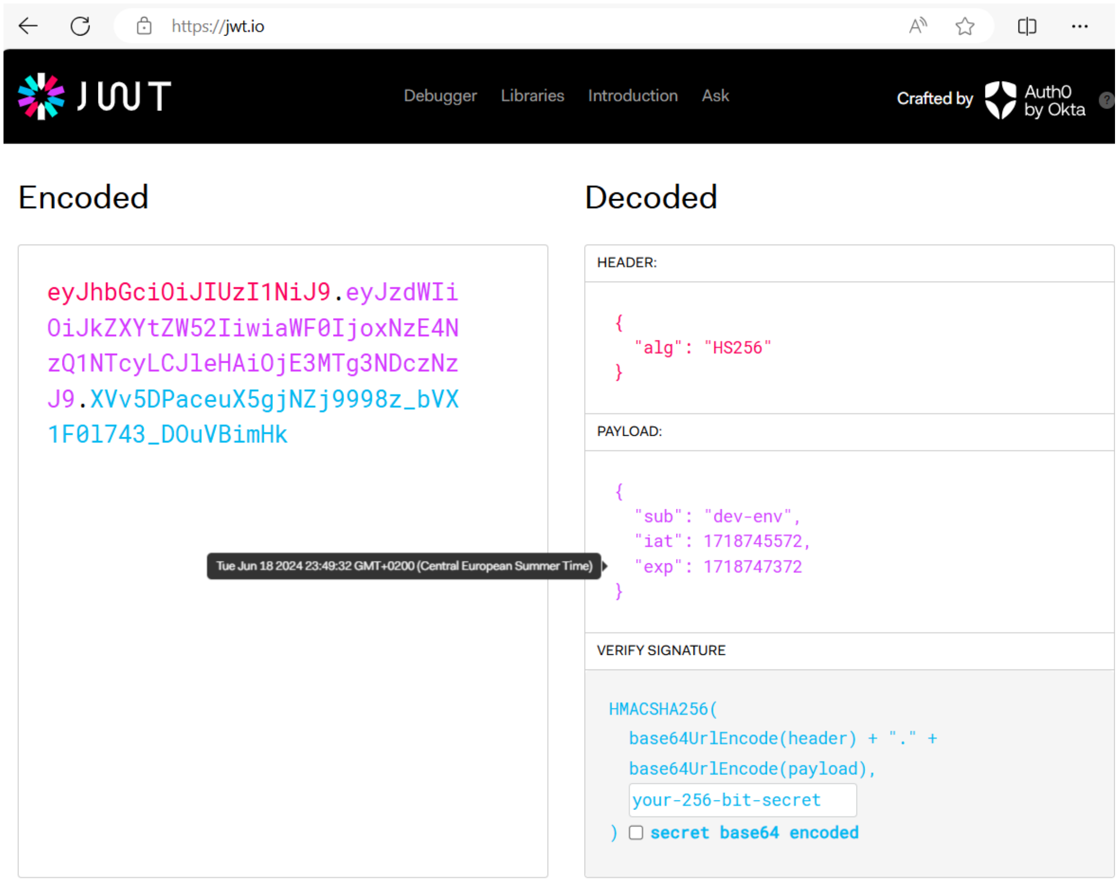
```
@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws
     Exception {
    return http
        .cors().and().csrf().disable()
        .authorizeHttpRequests (auth -> auth
            .requestMatchers("/users/signup", "/auth/**").permitAll()
            .requestMatchers("/css/**", "/js/**", "/images/**").
                permitAll()
            .anyRequest().authenticated())
        .sessionManagement(session -> session.sessionCreationPolicy(
            SessionCreationPolicy.STATELESS))
        .authenticationProvider(authenticationProvider)
        .addFilterBefore(jwtAuthFilter,
            UsernamePasswordAuthenticationFilter.class)
        .build();
    }
```

Code listing 4.1 Security configuration properties

Next, an authentication provider is responsible for processing any incoming authentication request. It performs the authentication logic by validating user credentials against the database. In this step, the password sent from the user is encrypted; the choice for this work was `BCrypt`, an industry standard on Spring Security. This hashing algorithm incorporates the salt into the stored password, protecting it against rainbow table attacks and increasing the necessary computational process for a brute-force attack [44].

The authentication provider communicates with a user details service, who loads user-specific data. This service interacts with the user repository to fetch information based on the provided username during login; it is important to have a separate user service for authentication, rather than using the already implemented user service directly, for a better separation of concerns and integration with Spring Security.

Once the user credentials have been checked and approved, the system generates a JWT token; this string includes claims such as the username and expiration date, and is signed using a HS256 (HMAC with SHA-256) algorithm, to prevent tampering with the message. Figure 4.4 display the decryption of a token generated by *MediTrack* for the `dev-env` user. The tokens are set to expire in thirty minutes and the secret key for signing them is stored in the Docker environment.



```
eyJhbGciOiJIUzI1NiJ9.eyJzdWIi
OiJkZXYtZW52IiwiaWF0IjoxNzE4N
zQ1NTcyLCJleHAiOjE3MTg3NDczNz
J9.XVv5DPaceuX5gjNZj9998z_bVX
1F0l743_DOuVBimHk
```

```
{
  "alg": "HS256"
}
```

```
{
  "sub": "dev-env",
  "iat": 1718745572,
  "exp": 1718747372
}
```

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  your-256-bit-secret
) ☐ secret base64 encoded
```

Figure 4.4 Decoding a JWT Token

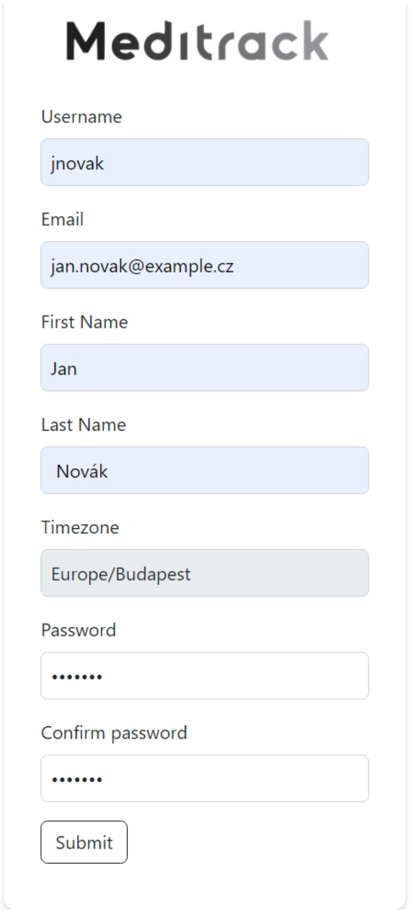
The generated token is then sent back to the client, which saves it in the local storage. For each subsequent request to protected endpoints, the client must include the token in the Authorization header, as exemplified in code 4.2.

Regarding user registration, the sign-up endpoint does not require any authorization in the header and the raw password sent from the client is also encrypted with `BCrypt` before being saved to the database. This ensures the password remains secure even if the database is compromised. However, this approach has a drawback: password recovery is not implemented in the current stage, so if a user forgets their password, they will lose access to their account. This additional functionality is planned for realization in the future.

```
setAuthHeader() {
        const token = localStorage.getItem('token');
        if (token) {
            //Check if token expired
            const tokenData = JSON.parse(atob(token.split('.')[1]));
            const expirationTime = tokenData.exp * 1000;
            if (Date.now() >= expirationTime) {
                localStorage.removeItem('token');
                window.location.href = '/';
            }
            else {
                this.headers['Authorization'] = `Bearer ${token}`;
            }}}
```

Code listing 4.2 Method to add token on authentication header

To create a profile, users provide first name, last name, and email – which is validated for correct formatting, ensuring the presence of an @ symbol followed by text, a dot, and more text. Additionally, two fields are used to confirm the password, making sure they match. Since the username is the unique identifier, the system checks the database to ensure it is not already in use. The timezone, as explained in section 4.1.6, is also set during this process. Figure 4.5 illustrates an example of sign up form.


Meditrack

Username	jnovak
Email	jan.novak@example.cz
First Name	Jan
Last Name	Novák
Timezone	Europe/Budapest
Password	•••••••
Confirm password	•••••••

Figure 4.5 Sign up form

4.1.6 Mailing service

The mailing service was built using JavaMail library, which provides a lot of customization options. First, a mail configuration class (`MailConfig`) was created to set up general properties such as the sender's address, password, SMTP's host and cryptography algorithm – TLS, as described in section 3.2.3.

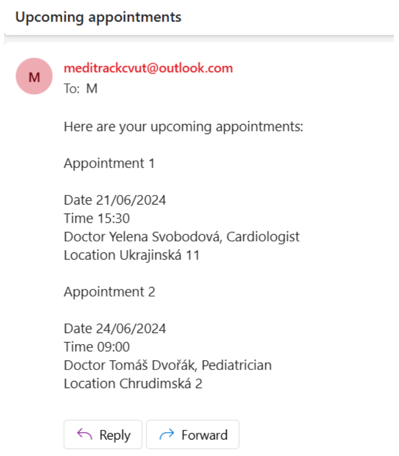
A new email account, `meditrackcvut@outlook.com`, was created for this project; the password is stored as an environment variable in GitLab and injected only at deployment. This approach is safer because it keeps sensitive information out of the source code and repository history, reducing the risk of accidental exposure and unauthorized access. Gitlab provides an additional layer of security by making sure only authorized users and processes can access the environment variables.

Next, a service class (`MailService`) was developed to handle the sending of various types of emails, allowing for reuse with different notification types. In this class, the subject is sanitized by replacing any newline (`\n`) or carriage return (`\r`) characters with an empty string, preventing SMTP header injection. This type of attack involves manipulating email headers by injecting unauthorized code into the message, leading to serious issues such as email spoofing — where the attacker alters the sender's address — and email hijacking, which can redirect or duplicate emails to unintended recipients.

Finally, the email body is interpreted as HTML text, which correctly identifies all special characters in the Czech language, such as `š` or `ř`, ensuring proper display and formatting of the email content.

As discussed on section 4.1.2, all dates stored in the database are saved in a UTC format. This initially resulted in the email service losing the user's local time information; to address this issue, a new field – `timezone` – was added to the User entity to store this local region. This information is set up during user creation (sign up form on figure 4.5) and is automatically gathered from the local system using JavaScript's `Intl.DateTimeFormat().resolvedOptions().timeZone` method [45]. Whenever the email body is constructed with dates, it is possible to convert the UTC dates to the user's local timezone stored in the database.

With the mailing service configured, the final step is to create a notification service, capable of automatically triggering emails whenever specific conditions are met. To alert users of their upcoming appointments, a scheduled task runs daily at 12:01 AM, checking all appointments within the next seven days and grouping them by user. Code 4.3 illustrates how those appointments are processed, while figure 4.6 displays the final email result for a user.


Upcoming appointments

meditrackcvut@outlook.com
To: M

Here are your upcoming appointments:

Appointment 1

Date 21/06/2024
Time 15:30
Doctor Yelena Svobodová, Cardiologist
Location Ukrajinská 11

Appointment 2

Date 24/06/2024
Time 09:00
Doctor Tomáš Dvořák, Pediatrician
Location Chrudimská 2

Figure 4.6 Email notification for appointments

```
@Scheduled(cron = "00␣01␣00␣*␣*␣*")
    public void checkAppointments() {

        ZonedDateTime startDate = ZonedDateTime.now();
        ZonedDateTime endDate = startDate.plusDays(7);

        List<Appointment> appointments = appointmentService
        .readAllBetweenDates(startDate, endDate);

        Map<User, List<Appointment>> appointmentsByUser =
        appointments
        .stream()
        .collect(Collectors.groupingBy(Appointment::getUser));

        for (User user : appointmentsByUser.keySet()) {
            List<Appointment> userAppointments =
            appointmentsByUser.get(user);

            try {
                sendAppointmentEmail(user, userAppointments);
            } catch (MessagingException e) {
                e.printStackTrace();
            }}}
```

Code listing 4.3 Schedule appointment check

Since the scheduled task runs according to the server's local time – which is set to UTC in the current Docker deployment –, users in different time zones might receive their email notifications too late in the day. A possible future solution to this issue is to add a service that periodically converts the user's timezone to UTC midnight, allowing the notification method to be triggered at appropriate times for each user. An endpoint that manually runs `checkAppointments()` was created in MailController class, both for testing purposes and for this potential future implementation.

4.1.7 File upload and download

As previously discussed, file storage in the database was implemented using LOB (Large Object) data types, where the file content is stored as a string of bytes. Each file is limited to 5 MB to prevent excessive memory usage in the database.

To enable file uploads through the report editing page, the Fetch API was used. This API handles the file upload process by sending the file data through a `FormData` object, who encodes the contents of the file – along with any other necessary form fields – as a multipart payload in an HTTP request. This format allows files to be transmitted efficiently over the network, maintaining their integrity and ensuring compatibility with server-side handling of file uploads.

Still on the client side, document selection is done through the operating system's file selection dialog, triggered by the `click()` method on a file input element (see code listing 4.4). An event listener is attached to detect the change event – when the user finally selects one file – which enables the Fetch operation with the first selected file. Figure 4.7 illustrates this upload process on the web page, showcasing the file selection dialog.

For file downloads, a similar approach is adopted. Using the Fetch API, files are fetched from the server's response payload as byte arrays; the filename is transmitted in the response headers via the `Content-Disposition` field, allowing the operating system to accurately reconstruct and save the file with its proper extension. Figure 4.8 shows the download of the file uploaded on figure 4.7.

```
const handleUpload = () => {
file.current.click();
file.current.addEventListener('change', () => {
    uploadFile(reportId, file.current.files[0]);})
}
```

Code listing 4.4 File upload event

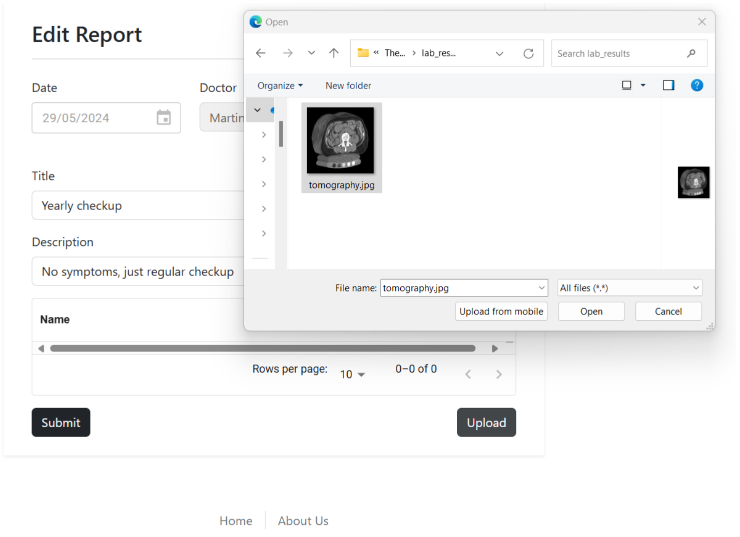


Figure 4.7 File upload dialog from edit page

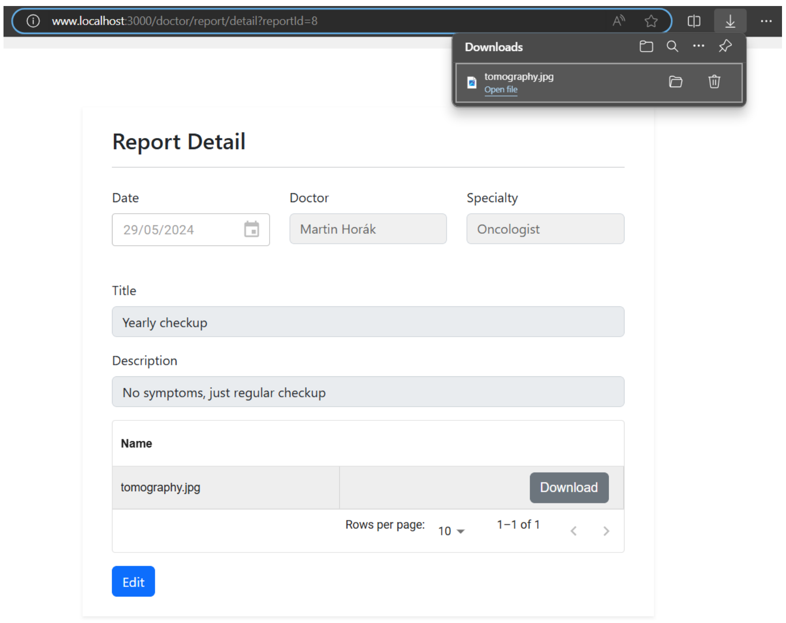


Figure 4.8 File download from detail page

4.2 Testing

Testing the developed software is a crucial phase in the software development lifecycle, ensuring the application's reliability, functionality and scalability. This section focuses on automated and users testing, beginning with a description of the unit and integration tests implemented in the code, followed by a listing of user tests that cover basic functionalities.

4.2.1 Automated tests

Automated tests are a critical tool for early bug detection, code quality improvement and scalability, since they act as documentation of expected behaviour and are particularly effective when combined with continuous integration processes. In this project, all automated tests were integrated into the CI/CD pipeline, ensuring they would run on every commit to the repository; this ensures failing code does not pass this stage, preventing the merging of incorrect code into the main project branch, while also helping developers to quickly identify injected errors.

The technologies used for testing in this work were JUnit and Mockito [46]: the former facilitated the creation and execution of unit tests, while the latter was employed to mock objects, allowing for the isolation of the unit under test.

Before writing tests for the new functionalities, some of previous tests had to be refactored to take into consideration the newly implemented user authentication. For all tests on the presentation layer, it was necessary to add a mocked authenticated user before doing any CRUD operations – see code 4.5 for an example on `MedicinePackageControllerTest`.

```
@Test
@Transactional
@WithMockUser("test-user")
void searchValidPackagesByMedicineId(){
    MedicinePackage mp = medicinePackageRepository.findById(1L).
        orElseThrow();
    Collection<MedicinePackageDto> list = medicinePackageController.
        getUserValidInventory(mp.getMedicine().getId());

    assertEquals(123l, ((MedicinePackageDto) list.toArray()[0]).
        getPackageSerialId());
    assertEquals(0l, ((MedicinePackageDto)list.toArray()[1]).
        getPackageSerialId());
    }
```

Code listing 4.5 Authenticated user mock on tests

4.2.2 User tests

User tests are an important phase in software engineering where end-users evaluate a software system to ensure it meets business requirements and is ready for deployment. They focus on validating functionality and usability under real-world conditions. Next, the following list presents a set of tests designed to validate the functionalities implemented in this project.

- **Sign up**

  1. On the login screen, click "Sign up" button located at bottom right corner
  2. Fill the required information – all fields are mandatory

3. Click button “Submit”
   If email address is not in the format `text@text.text`, a red warning box at the bottom of the page notifies `Invalid email address` and the form is not submitted
   If password and password check do not match, a red warning box at the bottom of the page notifies `Passwords do not match` and the form is not submitted
4. System checks if username already exists in the database – if it does, a red warning box at the bottom of the page alerts `Username already in use` and the user is not added to the system

- **Incorrect Login**

1. On the login screen, click “Log in” button
2. Write an unregistered username or valid username with incorrect password
3. Red error box at the bottom of the page warns `Incorrect username or password`

- **Add a new doctor**

1. After login, on Homepage, click “Add” on `Doctor` section
2. Fill doctor’s name – if email and/or phone number are left empty, doctor is registered without that information
   If any character is added to email, it checks if it complies to format `text@text.text`
   If any character is added to phone number, it checks if it complies to E.164 international format [47] – an optional `+` character at the beginning, followed by three up to fifteen digits.
3. If any field fails to comply to expected format, they are marked in red and the form is blocked from submission

- **Add report without appointment**

1. After login, on Homepage, click “Add” button on `Report` section
2. Pick a date without any appointments – `Doctor` selection shows message `No Doctor available`
3. Click “Submit” button
4. `Doctor` field is marked in red and form is blocked from submission

- **Edit or delete a `REALIZED` appointment**

1. After login, click “View” link on `Appointment` section
2. Find one appointment with status `REALIZED`
3. Click button “Delete” – a toast error message states `Failed to delete`
4. Alternatively, click button “Edit” and try to submit form on the next page – a toast error message states `Failed to submit form`

- **Upload a file too big**

1. After login, click “View” link on `Report` section
2. Select one report and click “Edit” button
3. Click “Upload” button at bottom right
4. Select a file exceeding the predetermined size limit - for this project, it was set at 5 MB
5. A toast error message states `File exceeds size limit`

- **Download file**

  1. After login, click "View" link on `Report` section
  2. Select one report with attachments and click "Detail" button
  3. Select one file from the list of attachments and click "Download"
  4. Browser prompts to either save the file to a default location or open it directly – depending on file type and browser settings

- **Send one email**

  1. Sign up a user with username `admin`
  2. Login as `admin`
  3. Using any API testing tool (such as Postman [48]), issue a POST request to endpoint `/mail/send` as the user `admin`. This request must include a form-data object with recipient's email address, subject line, and message body.
  4. Recipient receives the defined email coming from address `meditrackcvut@outlook.com`

- **Send appointments notification**

  1. Sign up a user with username `admin`
  2. Login as `admin`
  3. As the user `admin`, use an API testing tool to issue a POST request to endpoint `/mail/schedule/appointment`, with no body
  4. The system automatically verifies upcoming appointments and delivers the correspondent email to all affected users

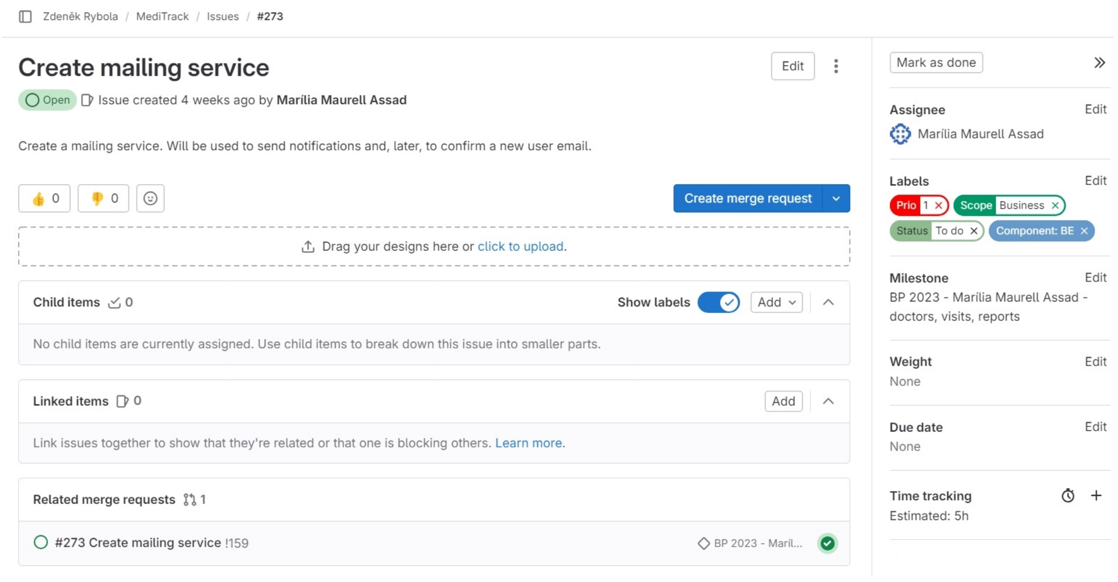
4.3 Project organization

Industry-standard software development practices were implemented during the execution of this project. This section details the development process, such as the use of Gitlab for version control, project management and CI/CD pipeline, as well as application of static code analysis and documentation.

4.3.1 Workflow and version management

Gitlab is a DevOps platform that combines software development with project management tools. In this project, its main functionalities included version control, issue tracking, code review processes and continuous integration/continuous deployment (CI/CD).

For version control, the project adhered to a structured Git workflow. Issues were created to address bugs or implement new features, each labeled and described to clarify the task (see figure 4.9). When a team member took on an issue, they created a new branch named after the ticket number and title. Once the work was completed, a merge request was opened, allowing other team members to review, test, and approve the changes. Finally, the branch was merged into an iteration branch or the master branch at the end of the sprint. This structured approach ensures organized, consistent and efficient collaboration, helping team members manage changes and maintain code quality.



Zdeněk Rybola / MediTrack / Issues / #273

Create mailing service

Open · Issue created 4 weeks ago by Marília Maurell Assad

Create a mailing service. Will be used to send notifications and, later, to confirm a new user email.

Child items 0 — No child items are currently assigned. Use child items to break down this issue into smaller parts.

Linked items 0 — Link issues together to show that they're related or that one is blocking others. Learn more.

Related merge requests 1 — #273 Create mailing service !159 — BP 2023 - Maril...

Assignee: Marília Maurell Assad

Labels: Prio 1, Scope Business, Status To do, Component: BE

Milestone: BP 2023 - Marília Maurell Assad - doctors, visits, reports

Weight: None

Due date: None

Time tracking: Estimated: 5h

Figure 4.9 Issue example

Regarding the CI/CD pipeline, as briefly described in section 2.3, the project follows a five-stage flow of commands that are automatically executed whenever new commits are pushed to the repository. The process starts with building both the backend and frontend components of the project, ensuring the latest code changes are compiled and ready for testing. Once the build is complete, backend tests are executed, to verify that the new code functions correctly and does not introduce any problems, catching issues early in the development cycle.

After the tests, an analysis job is run, using SonarQube and other static analysis tools to assess code quality and security – this step helps identify potential vulnerabilities and code smells. Next, a Javadoc document is generated and saved as an artifact; the final step in the pipeline is a manual publishing job, which creates versioned artifacts ready for deployment. This whole process ensures that all code changes are tested, analyzed, documented, and packaged for release; figure 4.10 illustrates the process in Gitlab.

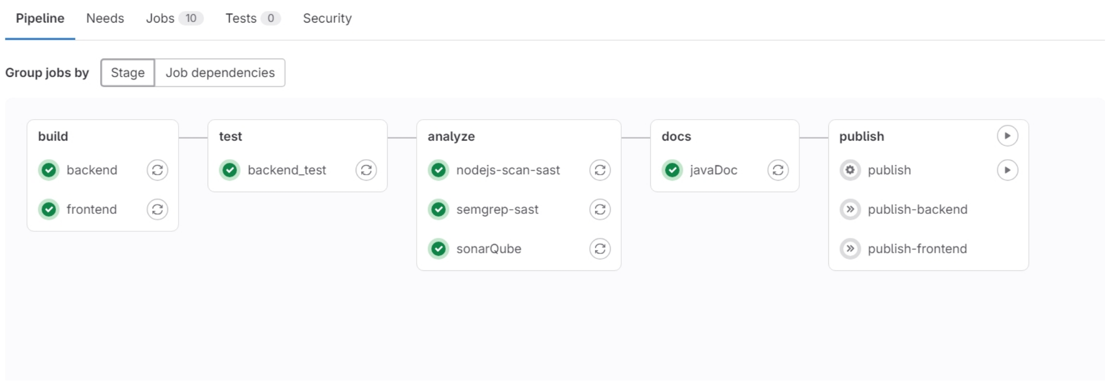


Figure 4.10 CI/CD pipeline

4.3.2 Static code analysis

Static code analysis is an automated process that examines source code without executing it, identifying potential bugs, vulnerabilities, code smells and style violations. By catching these issues before they reach production, static analysis reduces the cost and effort associated with bug fixes and refactoring.

As described in section 2.3, SonarQube was used in this project as a static code analysis platform. SonarQube analysis was integrated into the continuous integration pipeline, ensuring that every commit underwent thorough examination, as detailed in Section 4.3.1. Figures 4.11 and 4.12 present the results of the most recent code analysis conducted on this project, showcasing the quality metrics for the client-side and server-side source code, respectively.

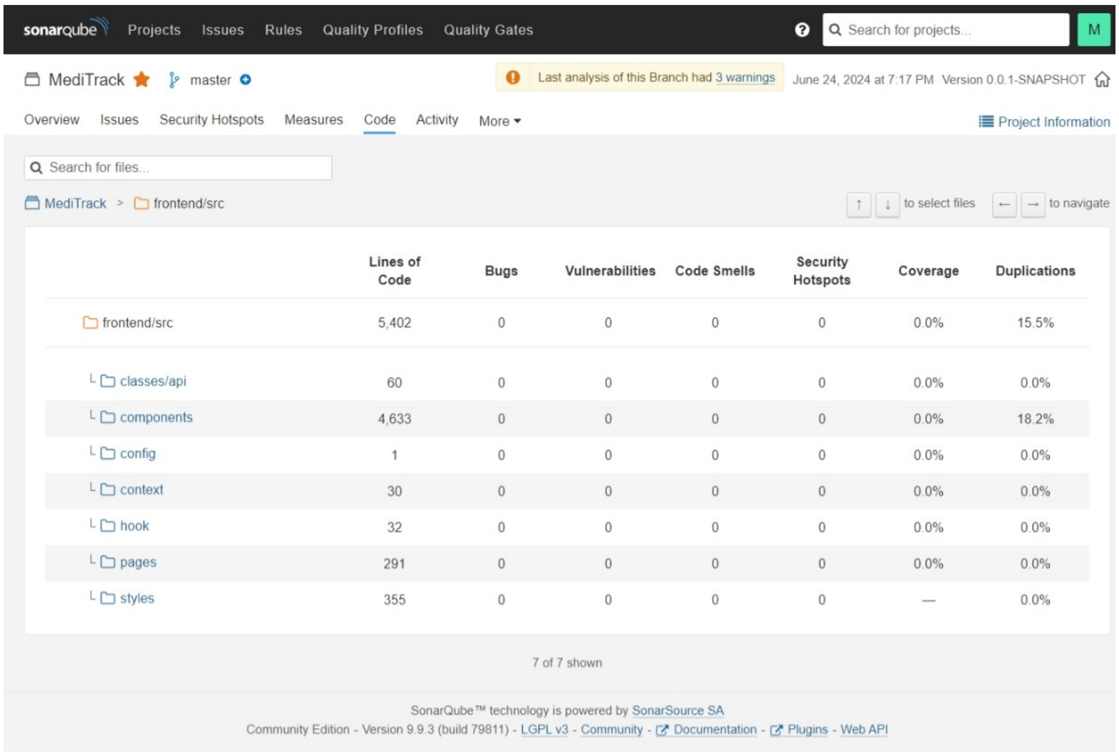
4.3.3 Documentation

Documentation is an important tool for knowledge transfer, maintenance and collaboration; it acts as a guide to help new team members and maintains a collective understanding of the system's functionalities. In this project, three different types of documentation are employed: model documentation, code documentation and a wiki repository.

The model documentation was done using the software Enterprise Architect [49]: the platform generates models of system architecture, business processes and software designs. The tool was used to create UML diagrams (activity, class and sequence diagrams), use case models, model domain and requirements documentation. All diagrams, in conjunction, provide clear visual representation of the system's structure and expected behaviour.

The code documentation is automatically generated through a pipeline job (detailed in Section 4.3.1) using JavaDoc. This tool process comments with a specific syntax to produce a well-structured documentation that is both human-readable and machine-processable, aiding developers to understand the expected inputs, outputs and general behaviour of all methods. By automating this process, the code documentation remains consistently up-to-date with the latest changes in the codebase.

The wiki repository is kept by team members and serves as a centralized knowledge repository, facilitating easy access to crucial information for all team members. It contains more broad information, such as design decision and practical tutorials – including guides on using Postman for API requests or accessing the project's Enterprise Architect model. The combination of those three documentation approaches ensures coverage of both technical details and high-level project knowledge.

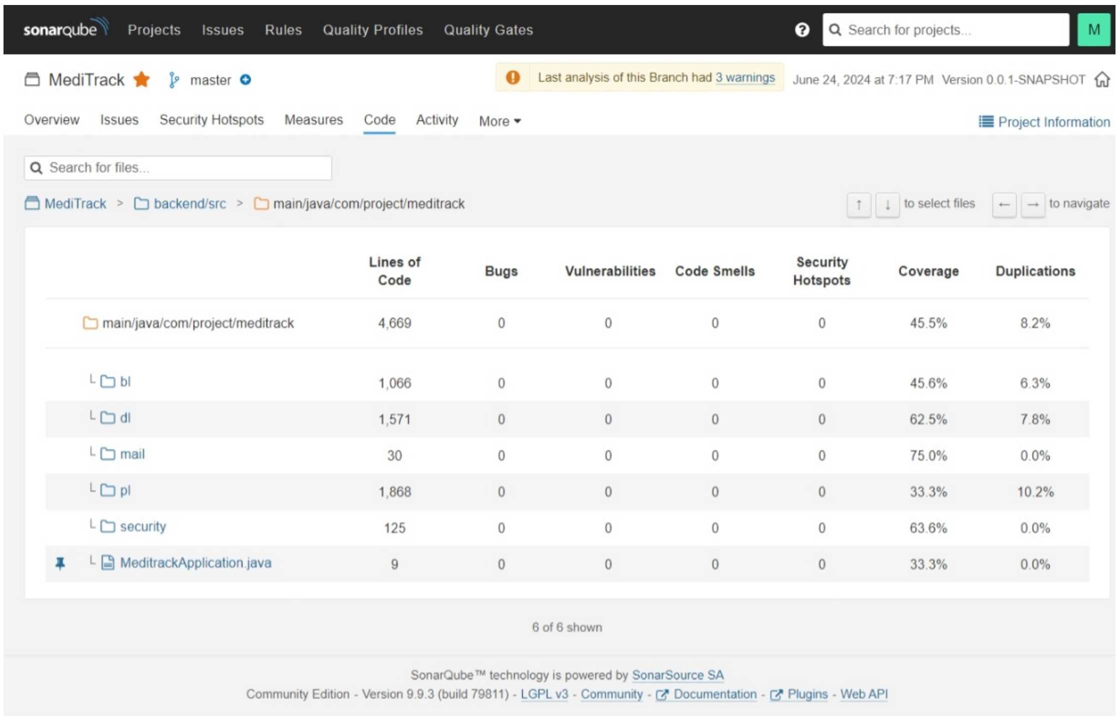


	Lines of Code	Bugs	Vulnerabilities	Code Smells	Security Hotspots	Coverage	Duplications
frontend/src	5,402	0	0	0	0	0.0%	15.5%
classes/api	60	0	0	0	0	0.0%	0.0%
components	4,633	0	0	0	0	0.0%	18.2%
config	1	0	0	0	0	0.0%	0.0%
context	30	0	0	0	0	0.0%	0.0%
hook	32	0	0	0	0	0.0%	0.0%
pages	291	0	0	0	0	0.0%	0.0%
styles	355	0	0	0	0	—	0.0%

7 of 7 shown

SonarQube™ technology is powered by SonarSource SA
Community Edition - Version 9.9.3 (build 79811) - LGPL v3 - Community - Documentation - Plugins - Web API

Figure 4.11 SonarQube analysis for client source code



	Lines of Code	Bugs	Vulnerabilities	Code Smells	Security Hotspots	Coverage	Duplications
main/java/com/project/meditrack	4,669	0	0	0	0	45.5%	8.2%
bl	1,066	0	0	0	0	45.6%	6.3%
dl	1,571	0	0	0	0	62.5%	7.8%
mail	30	0	0	0	0	75.0%	0.0%
pl	1,868	0	0	0	0	33.3%	10.2%
security	125	0	0	0	0	63.6%	0.0%
MeditrackApplication.java	9	0	0	0	0	33.3%	0.0%

6 of 6 shown

SonarQube™ technology is powered by SonarSource SA
Community Edition - Version 9.9.3 (build 79811) - LGPL v3 - Community - Documentation - Plugins - Web API

Figure 4.12 SonarQube analysis for server source code

Chapter 5

Conclusion

The main goal of this thesis was to improve and expand the existing Java Web application, *MediTrack*, in order to get it ready for a beta release. The application now includes user authentication and authorization, ensuring secure access and personal data separation: the user must insert their username and password before being able to access any of their own data.

A registration form was implemented with a final user interface, allowing the addition of new users into the system database through the client application. Additionally, the system now manages three new types of records: doctors, appointments, and reports – with the homepage updated to reflect these changes.

The original implementation was also improved by converting manual date manipulation, filtering, sorting and pagination into global components, capable of being reused. This guarantees consistency throughout the project, with different pages looking and behaving in similar way. The system also features a global calendar view that can be reused by any entity needing to present data over time.

Date information is now stored in UTC timezone in the database, a best practice in the industry. Additionally, users now can receive email notifications – as a first step, it is being used for automatic and scheduled appointment reminders. Finally, the last implemented change is the ability to upload and download files into the system.

5.1 Future work

For future work, more enhancements on user authentication should be a priority, to improve the system's security. The first change could be implementing role-based profiles, separating clients and administrators, resulting in tailored access and functionality based on roles. Next, implement a client feature to enable users to edit their profiles, allowing individuals to update their personal information as needed – including changing password and timezone information. Finally, users should be able to recover their accounts if they forget their passwords by setting a new password through a secure channel.

Email functionalities can be expanded: first, during the sign-up process, a confirmation email should be sent before completing registration, to ensure the validity of user accounts. Secondly, the email service could include attachments – such as reports – to improve data sharing within the application. At last, it should be possible to send other types of notifications, such as warning users of low medicine inventory or expired packages.

Finally, updating the frontend to utilize the timezone set in the database, rather than the local machine's timezone, will ensure consistent time displays even if the user is traveling. Other improvement on the client application includes transforming all relevant information in the view pages into links, so the user can quickly navigate based on attributes – finding all appointments related to one doctor by clicking on its name, for example.

These improvements will continue the work implemented in this thesis, expanding its functionalities and making *MediTrack* more secure and user-friendly.

Bibliography

1. FOWLER, Martin. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, 2003. ISBN 0321127420.
2. GAMMA, Erich. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994. ISBN 0201633612.
3. RICHARDS, Mark. *Software Architecture Patterns, 2nd Edition*. O'Reilly Media, Inc., 2022. ISBN 9781098134273.
4. *PostgreSQL: The World's Most Advanced Open Source Relational Database*. 2024. Available also from: `https://www.postgresql.org/`. Online.
5. *Java*. 2024. Available also from: `https://www.java.com/en/`. Online.
6. *Spring*. 2024. Available also from: `https://spring.io/`. Online.
7. *Gradle Build Tool*. 2024. Available also from: `https://gradle.org/`. Online.
8. *javadoc – Generates HTML pages of API documentation from Java source files*. 2024. Available also from: `https://docs.oracle.com/javase/8/docs/technotes/tools/windows/javadoc.html`. Online.
9. *JUnit 5 User Guide*. 2024. Available also from: `https://junit.org/junit5/docs/current/user-guide/`. Online.
10. *React: The library for web and native user interfaces*. 2024. Available also from: `https://react.dev/`. Online.
11. *Next.js by Vercel – The React Framework*. 2024. Available also from: `https://nextjs.org/`. Online.
12. *npm – Build amazing things*. 2024. Available also from: `https://www.npmjs.com/`. Online.
13. TURNBULL, James. *The Docker Book*. 2014. ISBN 0988820234.
14. *Docker: Accelerated Container Application Development*. 2024. Available also from: `https://www.docker.com/`. Online.
15. JEZ HUMBLE, David Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley Professional, 2010. ISBN 9780321601919.
16. *Gitlab*. 2024. Available also from: `https://about.gitlab.com/platform/`. Online.
17. *SonarQube 10.5 Documentation*. 2024. Available also from: `https://docs.sonarsource.com/sonarqube/latest/`. Online.
18. *My Therapy: Your personal pill reminder and medication tracker app*. 2024. Available also from: `https://www.mytherapyapp.com/`. Online, Accessed: 2024-03-13.

19. *Medisafe: Never miss another dose.* 2024. Available also from: `https://medisafeapp.com/`. Online, Accessed: 2024-05-25.

20. *DoseCast: Pill Reminder App.* 2024. Available also from: `https://www.montunosoftware.com/`. Online, Accessed: 2024-05-25.

21. *TOM: Medication Pill Reminder.* 2024. Available also from: `https://www.tommedications.com/en/`. Online, Accessed: 2024-05-26.

22. *Spring Security.* 2024. Available also from: `https://docs.spring.io/spring-security/reference/features/index.html`. Online, Accessed: 2024-04-20.

23. *Java HttpClient Basic Authentication.* 2024. Available also from: `https://www.baeldung.com/java-httpclient-basic-auth`. Online, Accessed: 2024-04-23.

24. *What is OAuth 2.0.* 2024. Available also from: `https://auth0.com/intro-to-iam/what-is-oauth-2`. Online, Accessed: 2024-05-04.

25. KRISTOL, David M. HTTP Cookies: Standards, Privacy, and Politics. *ACM Transacetions on Internet Technology.* 2001, vol. 1, no. 2. Available from DOI: `10.48550/arXiv.cs/0105018`.

26. *Cross Site Scripting Prevention Cheat Sheet.* 2024. Available also from: `https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html`. Online, Accessed: 2024-06-14.

27. *Cross-Site Request Forgery Prevention Cheat Sheet.* 2024. Available also from: `https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html`. Online, Accessed: 2024-06-13.

28. *What Is Token-Based Authentication?* 2024. Available also from: `https://www.okta.com/identity-101/what-is-token-based-authentication/`. Online, Accessed: 2024-06-14.

29. *How to invalidate a JWT using a blacklist.* 2023. Available also from: `https://dev.to/chukwutosin_/how-to-invalidate-a-jwt-using-a-blacklist-28dl`. Online, Accessed: 2024-06-14.

30. *react-big-calendar.* 2024. Available also from: `https://www.npmjs.com/package/react-big-calendar`. Online, Accessed: 2024-02-05.

31. *FullCalendar: The Most Popular JavaScript Calendar.* 2024. Available also from: `https://fullcalendar.io/`. Online, Accessed: 2024-02-05.

32. *Spring Framework: Email.* 2024. Available also from: `https://docs.spring.io/spring-framework/docs/2.5.x/reference/mail.html`. Online, Accessed:2024-04-02.

33. *Simple Java Mail: Simple API, Complex emails.* 2024. Available also from: `https://www.simplejavamail.org/`. Online, Accessed:2024-05-31.

34. *What's the Difference Between SSL and TLS?* 2024. Available also from: `https://aws.amazon.com/compare/the-difference-between-ssl-and-tls/`. Online, Accessed:2024-06-02.

35. *Docker: Volumes.* 2024. Available also from: `https://docs.docker.com/storage/volumes/`. Online, Accessed: 2024-04-04.

36. *Google Cloud: What is object storage.* 2024. Available also from: `https://cloud.google.com/learn/what-is-object-storage`. Online, Accessed: 2024-04-06.

37. *Binary Files in Database.* 2024. Available also from: `https://wiki.postgresql.org/wiki/BinaryFilesInDB`. Online, Accessed:2024-04-04.

38. *PostgreSQL: Large objects.* 2024. Available also from: `https://www.postgresql.org/docs/current/largeobjects.html`. Online, Accessed:2024-04-04.

39. *MUI X: DatePicker*. 2024. Available also from: `https://mui.com/x/react-date-pickers/date-picker/`. Online, Accessed:2024-02-18.

40. *MUI X : TimePicker*. 2024. Available also from: `https://mui.com/x/react-date-pickers/time-picker/`. Online, Accessed:2024-02-18.

41. *JavaScript: Date.prototype.toJSON()*. 2024. Available also from: `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Date/toJSON`. Online, Accessed:2024-02-22.

42. *Oracle: ZonedDateTime*. 2024. Available also from: `https://docs.oracle.com/javase/8/docs/api/java/time/ZonedDateTime.html`. Online, Accessed:2024-02-22.

43. *MUI X Data Grid*. 2024. Available also from: `https://mui.com/x/react-data-grid/`. Online, Accessed:2024-02-18.

44. NIELS PROVOS, David Mazières. A Future-Adaptable Password Scheme. In: *Proceedings of the FREENIX Track: 1999 USENIX Annual Technical Conference*. 1999. Available also from: `https://www.usenix.org/legacy/events/usenix99/provos/provos.pdf`. Online, Accessed: 2024-06-01.

45. *Intl.DateTimeFormat.prototype.resolvedOptions()*. 2024. Available also from: `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Intl/DateTimeFormat/resolvedOptions`. Online, Accessed:2024-06-12.

46. *Mockito Tasty mocking framework for unit tests in Java*. 2024. Available also from: `https://site.mockito.org/`. Online.

47. *E.164*. 2024. Available also from: `https://en.wikipedia.org/wiki/E.164`. Online.

48. *Postman API platform*. 2024. Available also from: `postman.com`. Online.

49. *UML modeling tools for Business, Software,= and Systems*. 2024. Available also from: `https://sparxsystems.com/`. Online.

Contents of enclosed media

- `source` ... Directory containing the source code
  - `meditrack` ... Implementation source code
  - `javadoc` ... Documentation of source code
- `text` ... The thesis text directory
  - `thesis.pdf` ... Thesis in PDF format