VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES

Hoan Tran

HEALTHCARE ASSISTANCE WEBSITE

Web Application for Searching Nearby Hospitals and Chat Service

School of Technology
2024

VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES
Information Technology

ABSTRACT

Author	Hoan Tran
Title	Healthcare Assistance Website
Year	2024
Language	English
Pages	48 + 6 Appendices
Name of Supervisor	Magnus Sundell

This thesis presents the design and implementation of a health care service support system. The product was created to help patients who need treatment and examination at hospitals in the area search and answer questions through text messaging services. The product was created also for the purpose of applying full stack web development and managing medical appointment schedules.

The web application was built using a modern tech stack, including ReactJS for the front-end interface, ExpressJS and NodeJS for the back-end server, Java Spring Boot for additional back-end functionality, and PostgreSQL for database management. This combination of technologies provides a robust and scalable foundation for the development of a feature-rich healthcare platform.

Throughout the development process, an iterative and agile approach was adopted to continually refine the application's functionality and user experience. Additionally, stringent testing procedures were implemented to ensure the reliability, security, and performance of the Healthcare Assistance Web Application. By leveraging the capabilities of modern web development technologies, this project aims to contribute to the advancement of healthcare services by providing a user-friendly and efficient platform for patient care and administrative management.

Keywords Software engineering, application framework, web development

TABLE OF CONTENTS

ABSTRACT

1 INTRODUCTION 7

2 PURPOSE OF PROJECT 8

2.1 Problems 8

2.2 Objective 9

3 THEORETICAL BACKGROUND 10

3.1 Web Stack 10

3.2 MERN Stack 11

3.3 Java Spring Boot (Framework) 16

3.4 External APIs and Libraries 17

4 APPROACH AND IMPLEMENTATION 18

4.1 Application Workflow 18

4.2 Project Structure 19

4.3 Crafting the User Experience 20

4.3.1 Searching Facilities 22

4.3.2 Reserving an Appointment 24

4.3.3 Chat Support 26

4.4 Data Processing in the Server 29

4.4.1 Authentication 29

4.4.2 Socket Communication 32

4.4.3 Encryption 33

4.4.4 Booking Server (Spring Boot) 35

4.4.5 External APIs 36

4.5 Implementation of Database 38

4.6 Testing 39

4.6.1 User Interface 39

4.6.2 System 40

5 FINAL PRODUCT 41

5.1 Successful Aspects 42

5.2 Challenges Encountered 42
6 CONCLUSIONS 43
REFERENCES 44
APPENDICES 49
Appendix 1 49
Appendix 2 50
Appendix 3 51
Appendix 4 54

LIST OF FIGURES AND TABLES

Figure 1. Web development stack (Adam, 2022) 11
Figure 2. MERN stack development (MERN Stack Web Development, 2021) 12
Figure 3. Key Features of Node.js. (Geeksforgeeks, 2021) 13
Figure 4. How Express.js works (Sankalana, 2021) 14
Figure 5. MongoDB data structure organization (Zoran, 2021) 15
Figure 6. Sample of data saved in documents (Sheldon, 2024) 15
Figure 7. Spring Boot layers (Kamath, 2023) 16
Figure 8. Workflow diagram of application. 20
Figure 9. Components in frontend 21
Figure 10. The "App.jsx" file 21
Figure 11. Google Map viewport and marker with InfoWindow 23
Figure 12. Design of searching facilities service 24
Figure 13. Booking service with useEffect hook 25
Figure 14. Reservation page 25
Figure 15. Show Room action 26
Figure 16. The main chat page 27
Figure 17. Chat Footer with sending message box 27
Figure 18. Chat layout design 28
Figure 19. Create JSON web token 30
Figure 20. Middleware function 30
Figure 21. Structure of JWT (Rishabh, 2022) 31
Figure 22. Hashed password with salt (Tanner, 2022) 32
Figure 23. Socket connection 32
Figure 24. Sending message with socket 33
Figure 25. Symmetric encryption (O'Sullivan, 2023) 34
Figure 26. Applying CryptoJS in component 35
Figure 27. Appointment entities 35
Figure 28. Spring framework annotations (Ramesh, 2023) 36

Figure 29. Send request to Google server 37
Figure 30. Database design 38
Figure 31. Entity relationship 38
Figure 32. MongoDB connection with Node.js server 39

1 INTRODUCTION

During the learning and internship process, mastering the Full Stack is extremely important for a web developer. Full stack development is significant for program designers because it gives flexibility in working on both frontend and backend, driving to an all-encompassing understanding of applications. This mastery improves proficiency and collaboration inside groups, whereas moreover cultivating versatility to unused advances. Additionally, it creates solid problem-solving abilities and offers different career development openings, making it a fundamental expertise set for exploring the energetic scene of the tech industry. This requires proficiency in programming languages, frameworks, tools and data structures. However, connecting the above elements is extremely complicated and information processing between front-end and back-end can easily cause confusion.

In recent years, advancements in technology have revolutionized healthcare delivery, offering innovative solutions to enhance patient care and accessibility. With the increasing demand for convenient and efficient healthcare services, there is a growing need for web-based platforms that provide comprehensive assistance to patients in accessing medical care and information.

Therefore, this thesis project leverages the MERN stack and Java Spring Boot framework for web development to streamline information processing flows. The healthcare web application employs fundamental functionalities such as CRUD (Create, Read, Update, Delete) methods and an authentication system. The primary objective of this thesis is to facilitate timely provision and support for individuals in need of medical assistance. Throughout the development process, iterative stages including conceptualization, coding, research, bug fixing, and testing are rigorously conducted. Additionally, stringent information encryption measures are implemented to safeguard data integrity and prevent unauthorized access.

2 PURPOSE OF PROJECT

2.1 Problems

In Finland, the medicalization theory states that as individuals live longer, there are more people who require care. The proportion of older adults with co-morbid chronic diseases is rising due to epidemiological transition and population aging. Although older adults are more likely to survive their illnesses, they frequently have chronic health issues and require more care. (Tynkkynen et al., 2022)

Elderly individuals face numerous hurdles in accessing healthcare, including physical limitations and limited digital literacy. Complex healthcare systems, transportation issues, and overwhelming online information further complicate their access to care. Managing multiple chronic conditions adds to the challenge, as does communication barriers with healthcare providers. Social isolation, financial constraints, and age-related stigma also contribute to their healthcare struggles. Addressing these issues is crucial for developing tailored healthcare solutions for the elderly. (Wilson et al., 2021)

In October 2020, Finland's Vastaamo psychotherapy center experienced a significant cyberattack, one of the country's largest data breaches. A Hacker accessed the patient database, compromising sensitive personal information, including therapy notes and session records, impacting around 40,000 patients. Some patients received extortion emails threatening to release their therapy notes unless a Bitcoin ransom was paid. The incident caused outrage and distress, highlighting the pressing need for enhanced data protection in healthcare organizations. Vastaamo faced criticism for its delayed response, leading Finnish authorities to launch an investigation to identify the perpetrators and address security flaws. The breach emphasized the critical role of cybersecurity in protecting patient privacy and reinforced calls for stronger data protection regulations in Finland's healthcare sector. (Wikipedia, 2022)

2.2 Objective

The aim of this healthcare assistance web developmen, employing the MERN stack and Java Spring Boot, is to design and implement a comprehensive web application that addresses the diverse needs of users seeking medical care. The key objectives include integrating functionalities such as hospital search utilizing the Google Maps API, real-time chat service, appointment booking system, user account management, and authentication features. By leveraging modern web development technologies, the thesis endeavors to create a user-friendly platform that streamlines the process of accessing healthcare services, facilitating seamless communication with healthcare providers, and simplifying appointment scheduling for users. Additionally, robust user account management and authentication mechanisms will be implemented to ensure data security, user privacy, and personalized user experiences. Through this project, the goal is to contribute to the advancement of healthcare technology and enhance the overall experience of individuals seeking medical assistance online.

Integrating external APIs such as Google Maps and Nodemailer enhances web application functionality and user experience. Google Maps enables interactive maps and location-based services for features like hospital search. Nodemailer automates email communication for notifications and updates. This integration improves efficiency and accessibility in accessing medical care and information.

The Vastaamo breach highlights the vital role of data encryption, especially in healthcare. Encryption ensures confidentiality, regulatory compliance, protection against cyber threats, and enhances trust. Prioritizing encryption is crucial for preserving data integrity and privacy standards in the face of evolving cyber threats.

3 THEORETICAL BACKGROUND

3.1 Web Stack

A web stack refers to a package of software essential for web development, including an operating system, programming language, database software, and web server. This combination, also called a web application stack, forms the foundation for building web-based solutions. (TMDesign, 2023)

The Frontend Application Layer includes the technologies used to build the user interface and client-side functionality of the web application, which is rendered in web browsers. This often involves HTML, CSS, and JavaScript, along with frontend frameworks and libraries such as React.

The Backend Application Layer encompasses the programming languages, frameworks, and libraries used to develop the server-side logic of the web application. Common backend technologies include Node.js, Python (with frameworks like Django or Flask), Ruby on Rails, Java (with frameworks like Spring Boot), and PHP.

The Web Server Layer consists of software responsible for handling incoming HTTP requests from clients (such as web browsers) and serving web pages and other content in response. Popular web servers include Apache, Nginx, and Microsoft IIS.

The Database Layer includes software for managing and storing data used by the web application. This can include relational databases like MySQL, PostgreSQL, or SQL Server, as well as NoSQL databases like MongoDB or Redis.

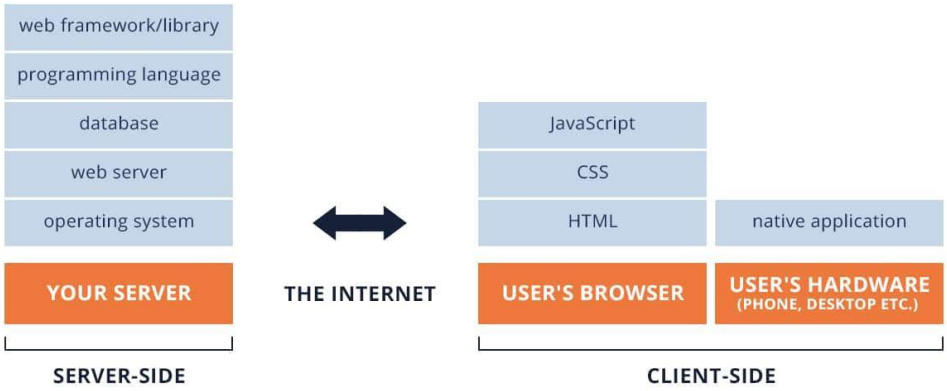


Figure 1. Web development stack (Adam, 2022)

3.2 MERN Stack

MERN is a term used to encapsulate a specific set of JavaScript-based technologies employed in the development of web applications. Conceived with the goal of streamlining the development process, MERN encompasses MongoDB, Express.js, React.js, and Node.js. Together, these components form a cohesive framework that plays a pivotal role in web application development. By leveraging MERN, engineers are equipped with a robust foundation that enhances efficiency and facilitates the creation of dynamic and responsive web applications. (Monika et al., 2021)

Using the MERN stack for a healthcare application with strict data security and privacy needs poses challenges. Vulnerabilities in MongoDB, Express.js, React, and Node.js components can compromise security if not managed carefully. Meeting healthcare regulations like HIPAA or GDPR adds complexity, as does limited control over cloud infrastructure. Despite its advantages, addressing these security challenges requires additional effort, expertise, and diligence. (Dave, 2023)

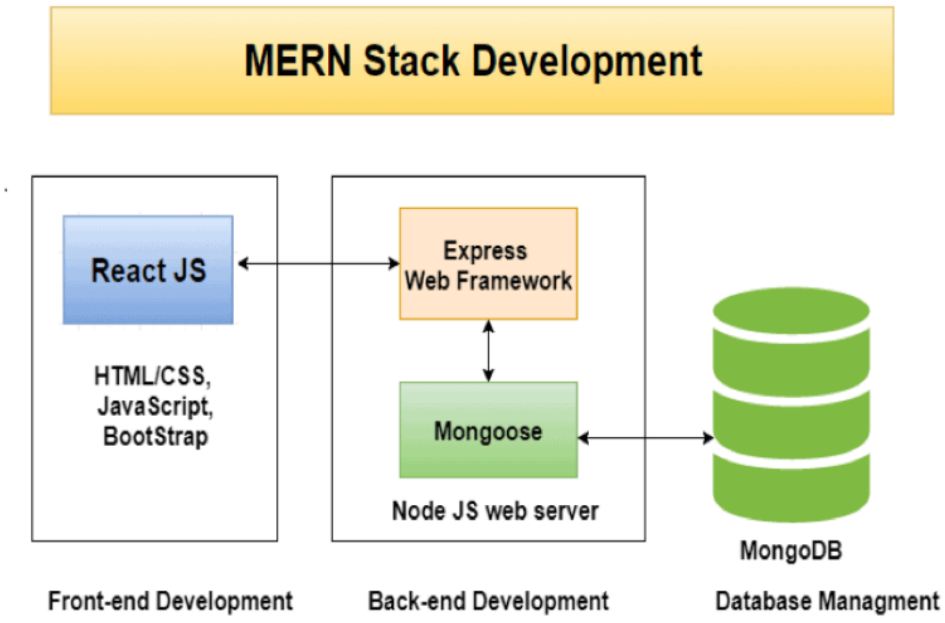


Figure 2. MERN stack development (Monika et al., 2021)

ReactJS, a JavaScript library, is deployed to develop reusable user interface (UI) components, as per the official React documentation at React Website. Described as a tool for constructing modular user interfaces, React facilitates the creation of large and intricate web applications capable of updating data without requiring subsequent page refreshes. It serves as the View (V) in the Model-View-Controller (MVC) architecture, abstracting the Document Object Model (DOM) to provide a simpler, more performant, and robust application development experience. React commonly renders on the server side using NodeJS, and extends its support to native mobile apps through React Native. Implementing unidirectional data flow, React simplifies boilerplate code, making it notably easier than traditional data binding methods.

Node.js, commonly referred to as Node, is an open-source JavaScript runtime environment that enables developers to execute JavaScript code outside of a web browser. Founded by Ryan Dahl in 2009, Node.js is built on Chrome's V8 JavaScript engine and is designed for building scalable and high-performance net-

work applications. Its key features include an asynchronous and event-driven architecture, facilitating non-blocking I/O operations and enabling efficient handling of concurrent connections. With its single-threaded event loop, Node.js can handle multiple requests concurrently without blocking the execution of other code, making it ideal for real-time web applications, APIs, and microservices. Node.js comes with NPM (Node Package Manager), providing access to a vast ecosystem of open-source libraries and modules. Its cross-platform compatibility and support for streaming data processing further enhance its appeal for developers seeking to build fast, scalable, and real-time applications. (Geeksforgeeks, 2021)

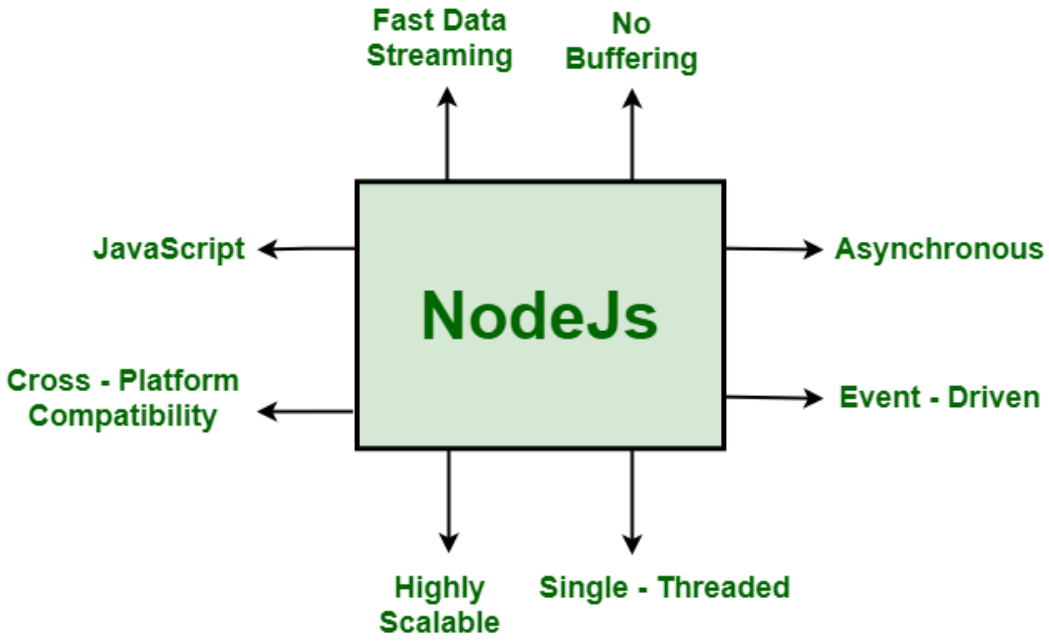


Figure 3. Key Features of Node.js. (Geeksforgeeks, 2021)

Express.js, established in 2010 by Holowaychuk (Wikipedia, 2019), is a minimal and flexible web application framework for Node.js, designed to streamline the development of web applications and APIs. With its robust features and middleware, Express simplifies tasks such as routing, handling HTTP requests, serving static files, and error handling. It offers a simple and intuitive routing system for

defining routes and executing specific logic based on request parameters, along with support for various template engines for dynamic content generation. Additionally, Express integrates seamlessly with Connect middleware, enabling developers to leverage a wide range of third-party middleware modules for additional functionality.

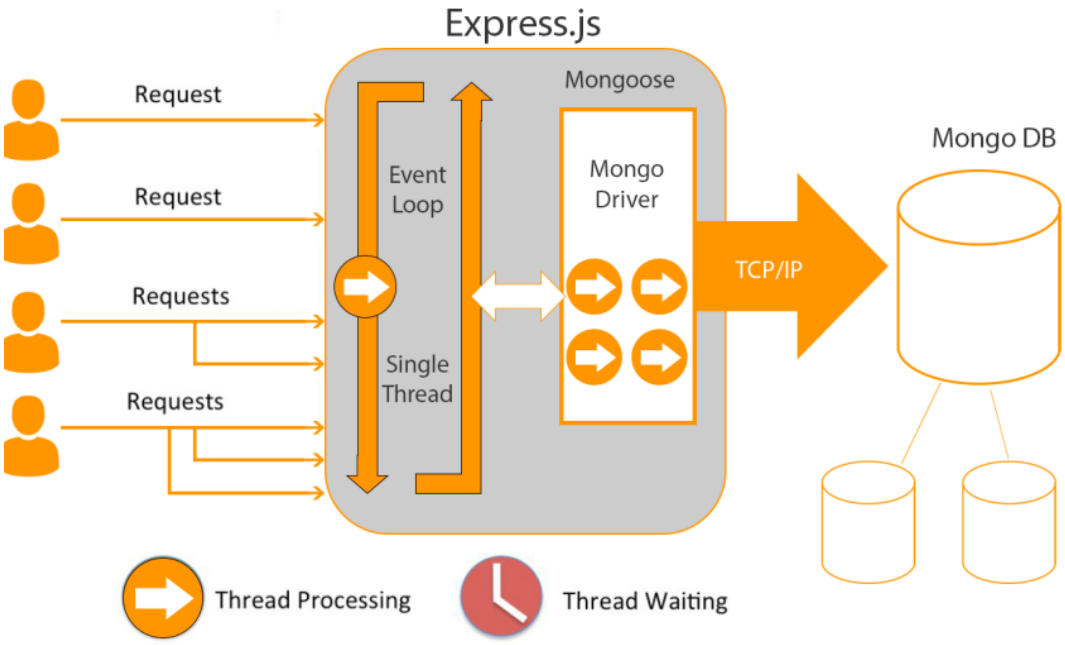


Figure 4. How Express.js works (Sankalana, 2021)

MongoDB is an open-source NoSQL database management system that offers an alternative to traditional relational databases. NoSQL databases, including MongoDB, are particularly advantageous for handling large, distributed datasets. MongoDB excels at managing document-oriented data storage and retrieval tasks. (Simplilearn, 2023)

It serves as a high-performance solution for storing extensive volumes of data efficiently. MongoDB's versatility extends to ad-hoc queries, indexing, load balancing, aggregation, and server-side JavaScript execution, making it a preferred choice for various organizational needs. (Gillis, 2023)

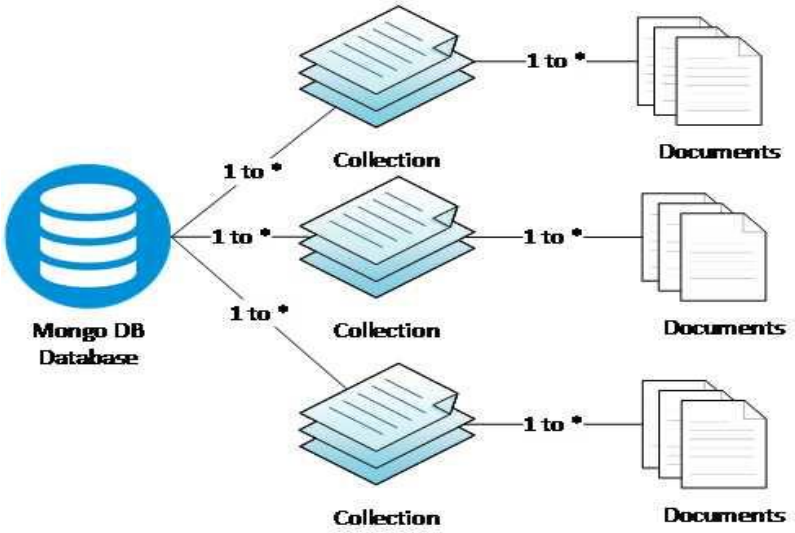


Figure 5. MongoDB data structure organization (Maksimovic, 2021)

Using MongoDB's flexible schema, the application efficiently manages a wide variety of data types and relationships, ensuring that all relevant information is readily accessible and easily modifiable. This structure supports the dynamic needs of the healthcare application, facilitating seamless interactions between users, healthcare providers, and the system. (Sheldon, 2024)

```
{
  _id: 'binder',
  totalPurchased: 305,
  avgPrice: Decimal128('19.867288135593220338983050847457 63')
}
{
  _id: 'notepad',
  totalPurchased: 257,
  avgPrice: Decimal128('21.397804878048780478048780487804878049')
}
{
  _id: 'backpack',
  totalPurchased: 78,
  avgPrice: Decimal128('114.15')
}
```

Figure 6. Sample of data saved in documents (Sheldon, 2024)

3.3 Java Spring Boot (Framework)

Java Spring Boot streamlines Java web application development with tools, libraries, and conventions that minimize repetitive code and simplify configuration. It emphasizes dependency injection, auto-configuration, and compatibility with Spring projects and external libraries, enhancing scalability and resilience. Spring Boot's convention-over-configuration approach automates setup with sensible defaults, reducing manual configuration. Embedded servers, dependency management, and production-ready setups further ease deployment and maintenance. (Thomas, 2023)

Java Spring Boot's maintenance advantages stem from its convention-over-configuration approach, ensuring consistent code structure and reducing errors. Its extensive ecosystem supports monitoring, debugging, and testing, enhancing application maintainability. Compatibility with Spring projects and third-party libraries offers flexibility and extensibility for maintenance tasks. Overall, Spring Boot's simplicity, consistency, and compatibility make it ideal for effectively building and maintaining Java web applications. (Guntupally et al., 2018)

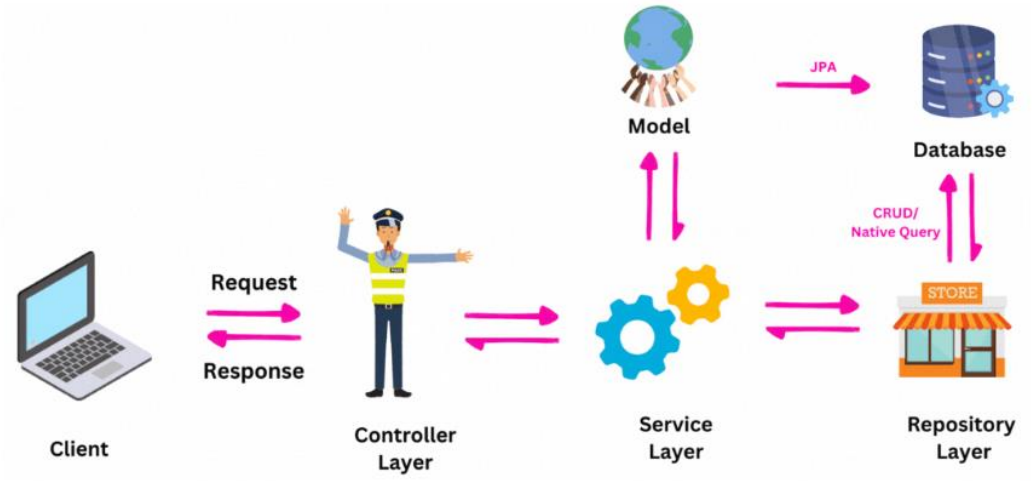


Figure 7. Spring Boot layers (Kamath, 2023)

3.4 External APIs and Libraries

External APIs (Application Programming Interfaces) fundamentally enable communication and interaction between various software systems, services, or platforms. Additionally, APIs often adhere to standard formats such as REST (Representational State Transfer) or SOAP (Simple Object Access Protocol) for data exchange, ensuring interoperability and compatibility across different platforms and technologies. External APIs facilitate seamless communication between software systems, adhering to standard formats like REST or SOAP for data exchange. They offer numerous benefits for application development, reducing time and effort by leveraging pre-existing functionality and promoting scalability and flexibility. APIs foster interoperability by enabling data exchange across platforms, driving efficiency, scalability, flexibility, interoperability, and innovation in software solutions. (Defranchi, 2024)

Libraries simplify development by encapsulating common functionality, promoting code reuse, and reducing complexity. They offer ready-made solutions for tasks like string manipulation or database interaction, saving developers time and effort. Ranging from general-purpose utilities to domain-specific tools, libraries streamline development processes, enhance productivity, and improve code quality. Integrating libraries into projects accelerates development cycles and fosters efficient, maintainable software solutions. (Rachel, 2023)

Integrating third-party libraries introduces dependencies, which can be hard to manage with frequent updates, potentially causing compatibility issues and requiring extra effort for seamless integration. Some libraries, despite being optimized, may add performance overhead, so it's important to assess their impact and consider alternatives if needed. Third-party libraries may also have vulnerabilities, posing security risks. Staying updated, regularly applying updates, and following best practices are essential to mitigate these risks. (Thakur, 2024)

4 APPROACH AND IMPLEMENTATION

The process of the application within a span of nearly two months encapsulated stages, from design and planning to coding, debugging, and testing. The initial phases of design and planning delineated the envisioned final product's appearance, page structure, and component utilization. Concurrently, both frontend and backend development progressed, ensuring seamless interconnection. Engaging with various online forums, notably Stack Overflow and Reddit, proved instrumental in overcoming encountered challenges by providing diverse perspectives and solutions. However, there are areas that need improvement, such as refining the user interface, optimizing performance, and enhancing security measures. Additionally, better documentation and more thorough testing could further improve the application's reliability and usability.

4.1 Application Workflow

In Figure 8 below, the Guest Routes in the healthcare application offer users a straightforward journey starting from the Home Page, where they can explore various features, to the Search Page for finding hospitals based on preferences. Upon deciding on a hospital, users can seamlessly proceed to the Login/Register Page to create an account or log in.

Protected Routes, accessible after user authentication, provide tailored functionalities such as managing appointments, confirming bookings, and direct communication with healthcare professionals. Authenticated users can access their Account Page for personal information, saved messages, and appointment details. They can also edit their profile, view specific appointment details, and save messages for future reference. These features ensure a user-friendly interface and comprehensive healthcare management, enhancing the user experience from initial exploration to personalized interaction and management.

MongoDB serves as the database to efficiently manage user data, appointment information, and messages. The database workflow involves several collections to organize and store the relevant data.

When a user registers or logs into their account, the application verifies the credentials and retrieves user information from the User Collection. After selecting a hospital and scheduling an appointment, the application creates a new document in the Appointment Collection with the relevant details.

When users communicate with healthcare professionals or our stakeholders (hospital agent) through the Chat Page, each message sent and received is stored as a document in the Message Collection.

Users can access their account details, including personal information and appointment history, by querying the User Collection and the relevant Appointment and Message Collections. Then they can easily update the information upon the second.

4.2 Project Structure

The structure of the healthcare application project is designed to ensure organization, scalability, and maintainability of the codebase. It typically follows a modular architecture, separating different components and functionalities into distinct directories and files. There is a brief overview of the project structure is shown in Appendix 2.

The project comprises three primary directories: "client," "booking-server," and "server." The "client" directory corresponds to the front-end aspect of the application and includes subdirectories for libraries, CSS styles, reusable components, and dependencies.

The "booking-server" and "server" directories, as their names imply, serve as the backend components of the application, running on Java Spring Boot and

Node.js, respectively. Their primary function is to handle data processing from the frontend and retrieve information from the MongoDB database. Additionally, they manage intricate operations such as authentication, socket connections, and integration with external APIs.

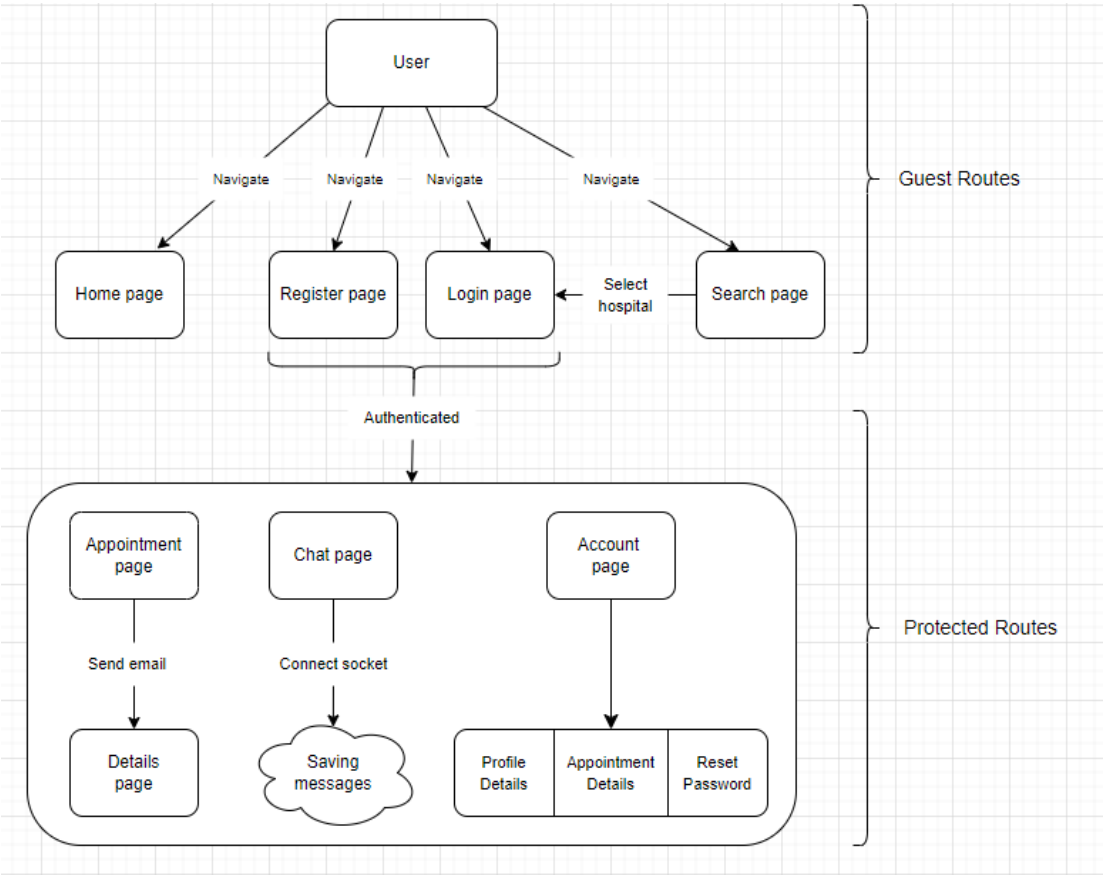


Figure 8. Workflow diagram of application.

4.3 Crafting the User Experience

In the project, the frontend is organized into different components, each serving a specific purpose or representing a distinct feature of the application. These components are structured in a modular manner to promote code reusability and maintainability. The frontend may include components for displaying user profiles, managing appointments, searching for healthcare facilities, sending messages to healthcare professionals, and more, depending on the requirements

of the thesis project. Related components are stored in the "components" folder shown in below:

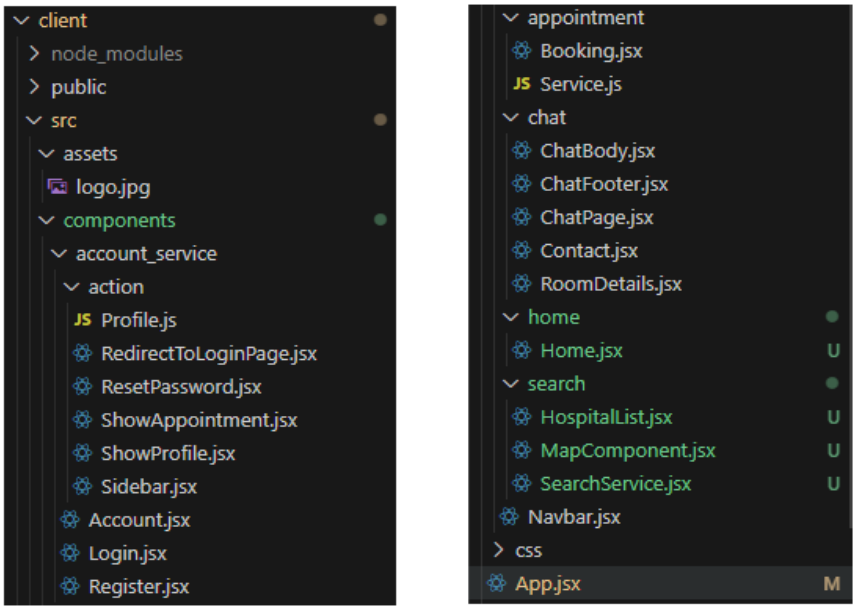


Figure 9. Components in frontend

The App.jsx file is the entry point and central hub of a React application, orchestrating component rendering, managing state, and handling routing. It often includes global configurations like setting up React Router and defining context providers. It intercepts HTTP requests to the backend, appends authentication tokens to headers, and uses middleware to validate these tokens against a database or authentication service.

```
const authorization = async () =>{
  if(token === null) {
    localStorage.setItem('isAuthenticated', false);
    localStorage.setItem('username', null);
  } else {
    try {
      const response = await axios.get('/authenticated', {
        headers: {
          'Authorization': `Bearer ${token}`
        }
      });
      if(!response.status === 200) {
        throw new Error('Failed to fetch data')
      } else {
        localStorage.setItem('isAuthenticated', true)
```

Figure 10. The "App.jsx" file

The ‘useEffect’ hook in React provides numerous benefits, making it a valuable tool for managing side effects in functional components. It allows for synchronization of state and props with external effects, ensuring updates occur when dependencies change. This capability is crucial for scenarios like fetching new data when a prop changes.

However, ‘useEffect’ also has its drawbacks. Managing the dependencies array correctly can be complex, missing dependencies or including unnecessary ones can lead to bugs like infinite loops or stale closures. Performance issues can arise if useEffect is not used properly, such as when effects run too frequently or involve heavy operations, potentially slowing down the application. (Codex, 2023)

4.3.1 Searching Facilities

In the process of searching for healthcare facilities using Axios and react-google-maps library, users input their location or search criteria into the application, triggering Axios to send a request to a backend server or external API. Upon receiving the data, Axios handles the response and passes it to the frontend, where react-google-maps renders the information on a Google Map component.

This includes plotting markers for each healthcare facility and displaying additional details like name, address, phone, opening hours and especially distance. Users can interact with the map, such as zooming or clicking on markers for more details, while dynamic updates ensure that displayed results remain relevant and up to date as users refine their search criteria or move the map viewport.

Figures 11 demonstrates the functionality using the react-google-map library and imported components such as GoogleMap, LoadScript, Marker, and InfoWindow. Initially, the default view of the map centers on Finland's location but dynamically adjusts once the user inputs a specific location, such as Vaasa or Helsinki.

Using ‘react-google-maps/api’ to integrate Google Maps into React applications offers several advantages and disadvantages. On the positive side, the library

provides a user-friendly interface, abstracting away much of the complexity involved in adding and customizing maps, and supports rich features like markers (shown in Figure 11), info windows, and event handling. It is optimized for performance, supports responsive design, and benefits from a large community and extensive documentation, making it easier for developers to implement and customize maps. (Hu, 2012)

```
{hospitals && hospitals.map((hospital, index) => (
    <Marker
        key={index}
        position={hospital.information.geometry.location}
        title="Marker"
        onClick={() => handleMarkerClick(hospital)}
    />
))}

{selectedHospital && (
    <InfoWindow
        key={selectedHospital.information.place_id}
        position={selectedHospital.information.geometry.location}
        onCloseClick={() => setSelectedHospital(null)}
    >
        <div>
            <h3>{selectedHospital.name}</h3>
            <p>{selectedHospital.information.details.formatted_address}</p>
            {selectedHospital.information.distance/1000}km
        </div>
    </InfoWindow>
```

Figure 11. Google Map viewport and marker with InfoWindow

However, there are challenges such as increased complexity for advanced features, dependency on Google's external service which can lead to connectivity issues and is subject to Google's pricing model and rate limits, and potential performance issues when handling many markers or overlays in this project. The learning curve can be steep for newcomers, and there are privacy concerns regarding data sent to Google. Additionally, offline functionality is limited, requiring an internet connection for full functionality. Despite these drawbacks, with careful implementation, 'react-google-maps/api' can be a powerful tool for adding sophisticated mapping capabilities to React applications. (Hu, 2012)

The primary and crucial objective of this project is to enhance user accessibility to nearby medical facilities, facilitating appointment scheduling. The emphasis is on creating a user-friendly interface that is easy to navigate. However, there are areas for improvement, notably the time taken to fetch data from the server,

approximately one second, and the need to reorganize the interface for better visibility of medical facilities.

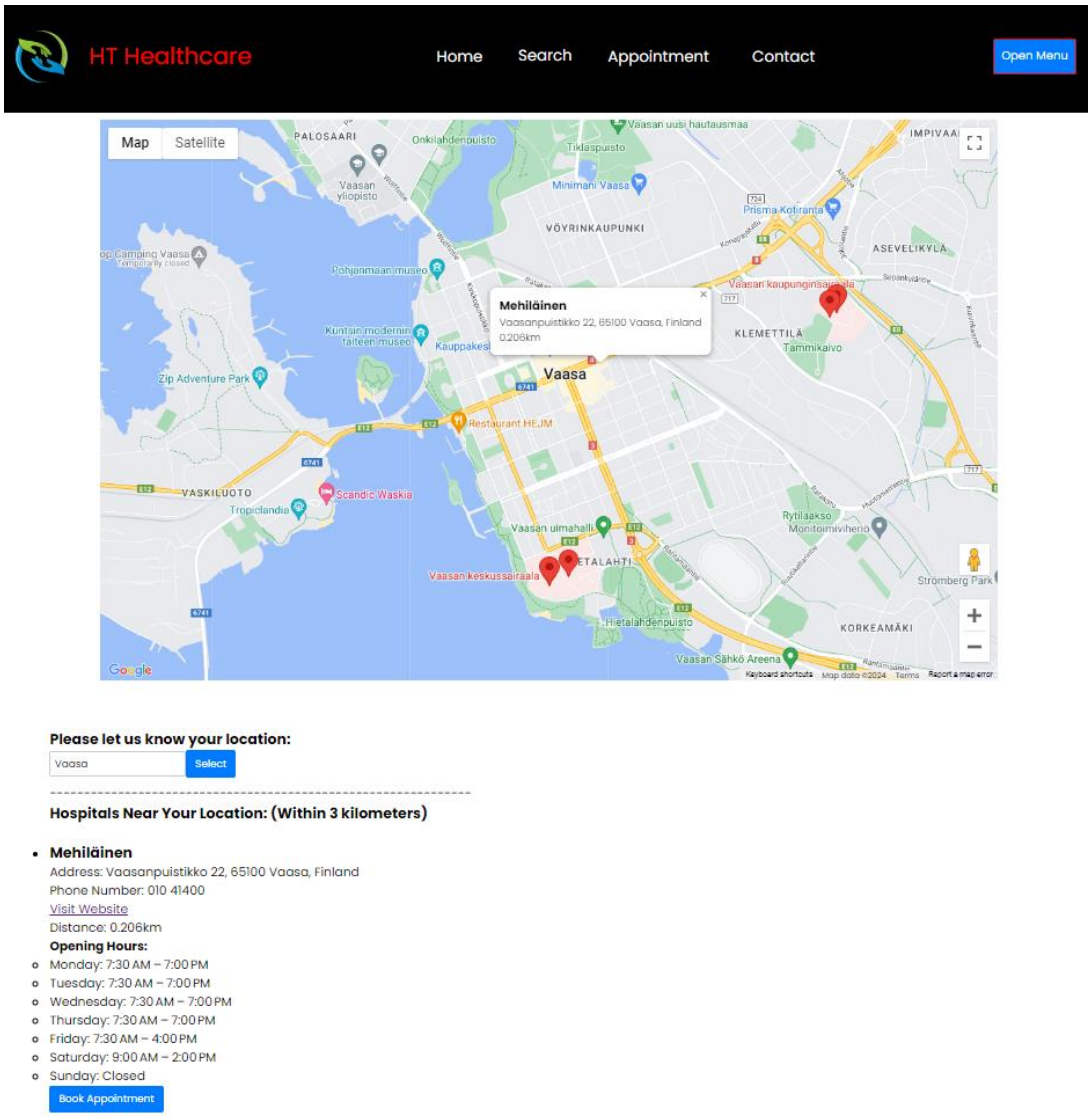


Figure 12. Design of searching facilities service

4.3.2 Reserving an Appointment

The reservation service in the healthcare application streamlines booking after users select a hospital. Users must log in before proceeding, then fill out a booking form with appointment details. Upon successful reservation, an email confirmation is sent, notifying users of their appointment. This workflow ensures a seamless and secure booking experience.

```
19    useEffect(() => {
20        if(selectedHospital) {
21            setHospital({
22                id: selectedHospital.information.place_id,
23                name: selectedHospital.name,
24                address: selectedHospital.information.details.formatted_address,
25                phone: selectedHospital.information.details.formatted_phone_number,
26                website: selectedHospital.information.details.website
27            })
28        }
29    }, [])
```

Figure 13. Booking service with useEffect hook

The 'useEffect' hook is used to set the hospital selected before and to ensure that users already selected the hospital in searching facilities page.

While both Axios and Fetch are popular for HTTP requests in JavaScript, Axios offers advantages like simpler syntax and automatic transformation of response data into JavaScript objects, eliminating the need for manual JSON parsing required by Fetch.

Axios offers built-in methods for data transformations, supports interceptors for modifying requests or responses, and allows setting timeouts for requests, unlike Fetch. It simplifies error handling by automatically rejecting promises for HTTP errors, while Fetch only rejects network errors, requiring extra code for HTTP status checks. Axios also supports request cancellation. (Innocent, 2024)

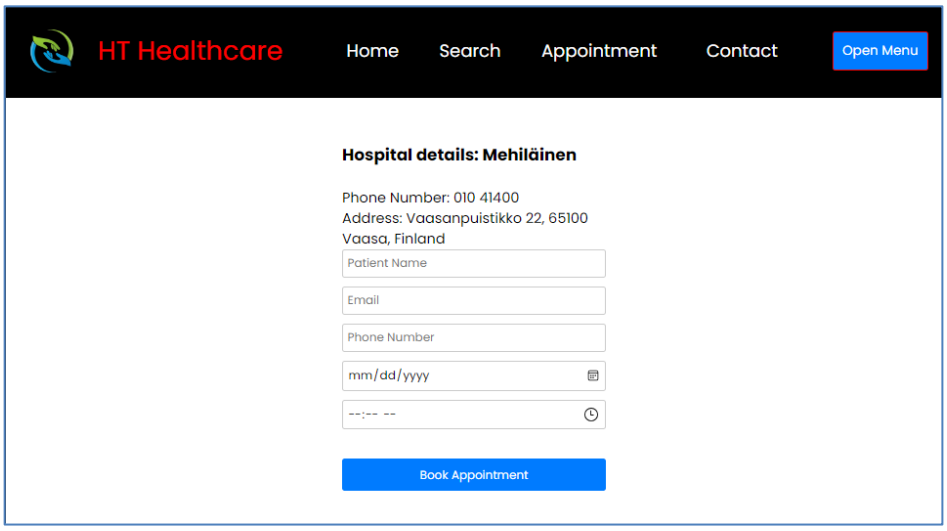


Figure 14. Reservation page

Figure 14 shows the booking page of the application, accessible after choosing a hospital and successfully logging in.

The booking page serves as a pivotal step in the healthcare application, enabling users to schedule appointments conveniently. These outcome pages serve to reassure users of their successful booking and provide any relevant details or instructions. Together, the booking page and outcome pages streamline the appointment scheduling process and ensure a smooth user experience.

4.3.3 Chat Support

The chat service with Socket.io enhances real-time communication in the healthcare application. Users can engage in instant messaging with healthcare professionals by creating or joining chat rooms. Socket.io enables bi-directional communication between client and server, allowing instant and efficient message transmission. It supports features like typing indicators and message delivery receipts, enriching the chat experience.

Similar to the booking service, the chat service also necessitates an initial authentication process through login or account registration. Upon authentication, a connection socket is established when the user triggers the "Show Room" button or inputs their name and symptom title, subsequently pressing the "Create Room" button.

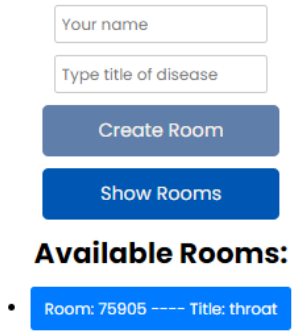


Figure 15. Show Room action

Upon room creation, the server automatically assigns a unique room ID using Math.random(). Subsequently, when another user logs in, they can readily view the created room alongside its ID and title, as illustrated in Figure 15.

```
14      useEffect(() => {
15          socket.on('messageResponse', (data) => setMessages([...messages, data]));
16          console.log(messages)
17      }, [socket, location, messages]);
18      useEffect(() => {
19          socket.on('roomDetails', (data) => {
20              setRoomDetails({
21                  roomID: data.roomID,
22                  selectedName: data.selectedName,
23                  title: data.title
24              })
25          });
26      }, []);
```

Figure 16. The main chat page

The socket connection is now established and ready to facilitate message exchange between users. Upon receiving a response from the server, the client utilizes socket.on() to handle incoming messages, while socket.emit() is employed to transmit information to the server.

```
11  const handleSendMessage = (e) => {
12    e.preventDefault();
13    if (message.trim() && username) {
14      socket.emit('message', {
15        text: message,
16        username: username,
17        roomDetails: roomDetails,
18        id: `${socket.id}${Math.random()}`
19      });
20    }
21    setMessage('');
22  };
```

Figure 17. Chat Footer with sending message box

A message containing room details is sent from the Chat Footer using socket.emit('message', {...}). Here, 'message' serves as the identifier for the transmit-

ted data, which the server will receive via socket.on('message', {...}). Subsequently, the client-side will receive a response through socket.on('messageResponse').

The primary function of the ChatBody component is to render sent messages and identify message senders and composers. This component primarily focuses on rendering received data from the Chat page component without performing extensive information processing. Its role is to display relevant information accordingly.

Initially, it presented a significant challenge, but with each step completed, the process became more manageable, particularly in grasping the operational principles of Socket.IO.

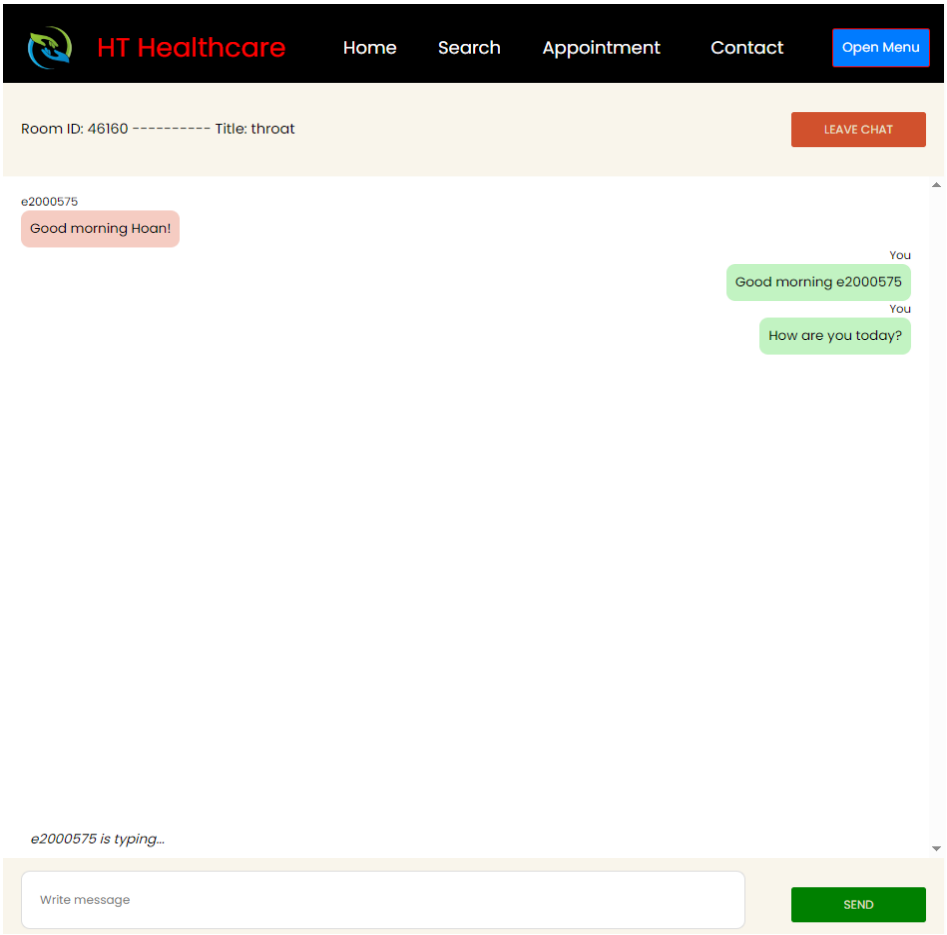


Figure 18. Chat layout design

4.4 Data Processing in the Server

Data processing in the server involves a sequential workflow starting with the reception of requests, which are then routed to the appropriate endpoints based on their path and HTTP method. Subsequently, the server parses the incoming request data to extract relevant information and executes the core business logic associated with the request, often involving database queries, external service interactions, and computations.

Server-side data processing involves robust mechanisms for authentication, encryption, and socket communication to ensure secure, efficient, and real-time handling of data. Authentication verifies user identities and manages access through tokens, encryption protects data both in transit and at rest, and sockets facilitate real-time interactions, enhancing the functionality and security of the healthcare application.

4.4.1 Authentication

When a user attempts to log in or access protected routes, the server processes their credentials to verify their identity. This involves credential verification, where the server compares the provided username and password against stored values, ‘UserDetails’ collection in MongoDB. Upon successful authentication, the server generates a token (JWT - JSON Web Token) which is then sent to the client (Figure 19). This token is used for subsequent requests to validate the user's identity without repeatedly asking for credentials. After generating the token, the application reloads, transmitting the token back to the server for further authentication and session management by validating the token using middleware function (Figure 20) that check the token's validity and ensure it has not expired.

```
const { username, password } = req.body;
try {
  const user = await UserDetail.findOne({ username: username });

  if (!user) {
    return res.status(401).json({ error: 'Invalid username or password' });
  } else {
    bcrypt.compare(password, user.password, (err, result) => {
      if (err || !result) {
        return res.status(401).json({ error: 'Invalid username or password' });
      }

      const token = jwt.sign({ id: user.id, username: user.username }, SECRET_KEY);
      res.json({ token, username });
    });
  }
```

Figure 19. Create JSON web token

```
const jwt = require('jsonwebtoken');
const SECRET_KEY = process.env.SECRET_KEY;
const authenticateToken = (req, res, next) => {
    const token = req.headers['authorization']?.split(' ')[1];
    if (!token) {
        res.sendStatus(401);
    }

    jwt.verify(token, SECRET_KEY, (err, user) => {
        if (err) {
            console.log(err);
            return res.sendStatus(403);
        }
        return res.json(user);
    });
};
```

Figure 20. Middleware function

However, improvements are needed in token storage and optimizing authentication speed and convenience through middleware for protected routes. Enhancing these aspects will ensure a more robust and rigorous authentication process.

JSON web token (JWT) offer stateless authentication, eliminating the need for server-side sessions and making them highly scalable and the structure of JWT is shown in Figure 21. They are compact and can be easily transmitted through

URLs, POST parameters, or HTTP headers. JWTs support cross-domain usage, which is beneficial for distributed systems or microservices.

Security is a major concern, so JWTs must be securely transmitted over HTTPS and stored properly to avoid interception or tampering. Revoking JWTs before their expiration is challenging and often requires additional mechanisms, such as token blacklists. Managing token expiration and refresh tokens adds complexity to the system. (Poddar, 2022)

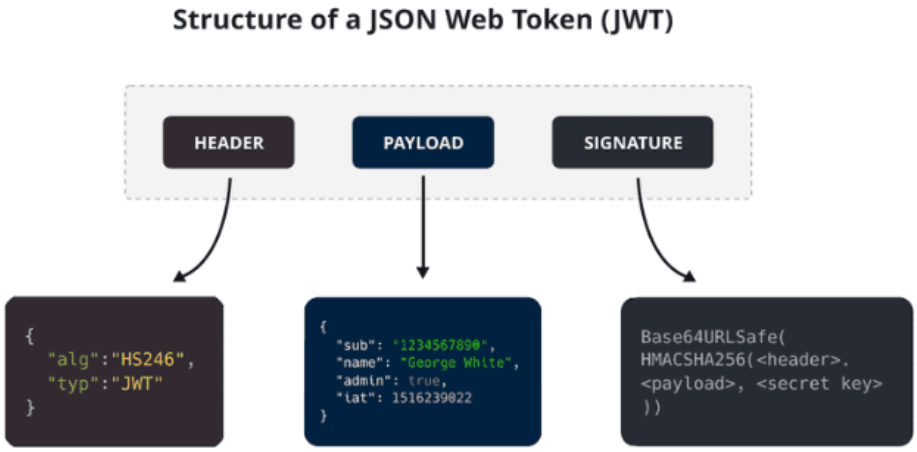


Figure 21. Structure of JWT (Poddar, 2022)

Bcrypt is a slow hash function designed for password hashing, making it resistant to brute-force attacks. It includes a salt to protect against rainbow table attacks and can be configured to increase computational complexity over time, enhancing security as hardware capabilities advance. Bcrypt is widely supported and used, with a strong track record of reliability and security in the industry. (Jones, 2022)

The intentional slowness of Bcrypt can impact performance, especially when handling many authentication requests simultaneously. Its computational complexity demands more CPU resources, which can be a concern for systems with limited processing power. Bcrypt is specifically tailored for password hashing and may not be suitable for other cryptographic applications. (Jones, 2022)

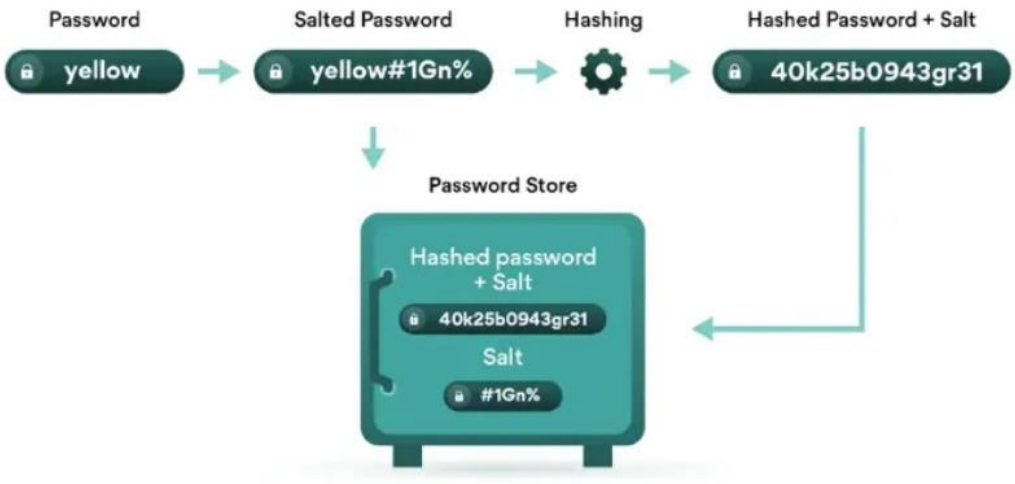


Figure 22. Hashed password with salt (Jones, 2022)

4.4.2 Socket Communication

Chat services are handled using socket communication. When a client connects, the server initializes a socket connection using Socket.IO. This connection remains open, allowing for continuous real-time communication.

CORS, or Cross-Origin Resource Sharing, is a security feature implemented in web browsers that allows or restricts web applications running at one origin (domain) from requesting resources from another origin. By default, web browsers enforce a same-origin policy, which restricts how resources on one origin can interact with resources on another. CORS provides a way to relax this policy by allowing servers to specify which domains are permitted to access their resources. (Maldonado, 2023)

```
const http = require('http').createServer(app);
const io = socketIO(http, {
  cors: {
      origin: "http://localhost:3000"
  }
});

chatComponent(io);
```

Figure 23. Socket connection

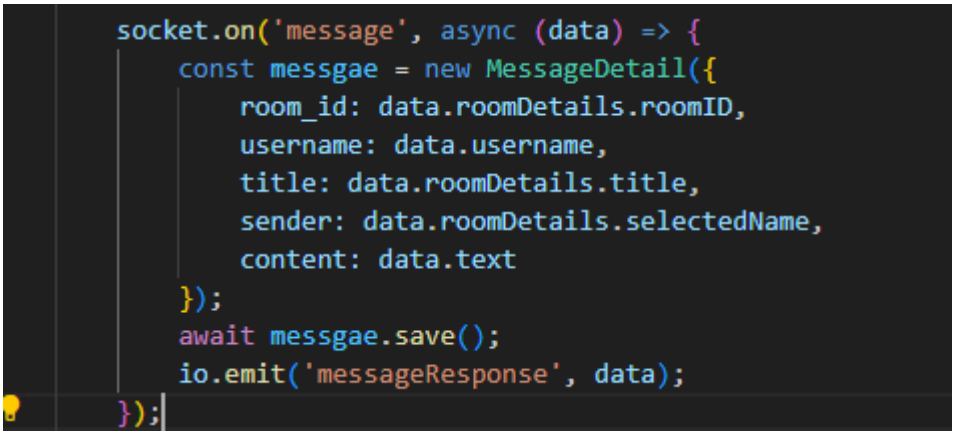


Figure 24. Sending message with socket

The server saves the message in a MongoDB database. This involves creating a new document in 'MessageDetails' collection that contains details such as the room id, username, title, sender's name, and encrypted content (Section 4.4.3).

After successfully saving the message, the server broadcasts it to the intended recipient(s). If the recipient is connected to the socket, the server sends the message directly to them.

Throughout this process, authentication and encryption are crucial. Users are authenticated when establishing socket connections, often using tokens. All data transmitted through the sockets is encrypted to ensure confidentiality and integrity.

4.4.3 Encryption

Encryption is a fundamental component in web development, ensuring the confidentiality and integrity of data transmitted between clients and servers, as well as data stored within applications. It is vital for data security, regulatory compliance, and building user trust. There are various types of the encryption used, including symmetric encryption, which uses a single key for both encryption and decryption, and asymmetric encryption, which employs a pair of keys (public and private) for secure data exchanges. (Clinton, 2023).

Symmetric encryption is a type of cryptographic technique in which the same key is used for both encryption and decryption of data. This key must be kept secret and shared only between the communicating parties. Because the same key is used in both processes, symmetric encryption is generally faster and more efficient compared to asymmetric encryption, making it suitable for encrypting large datasets due to their high performance and speed (Crane, 2020).

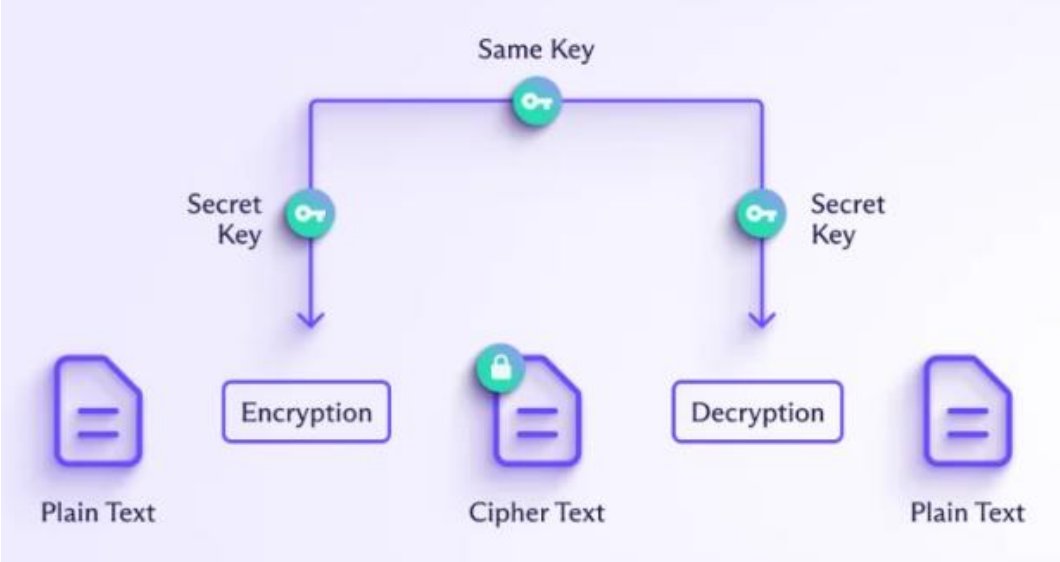


Figure 25. Symmetric encryption (O'Sullivan, 2023)

However, it has notable disadvantages, such as the challenge of securely sharing and managing the secret key, scalability issues in environments with many users, and the risk that if the key is compromised, all encrypted data can be decrypted. Despite these drawbacks, symmetric encryption remains a critical component in protecting sensitive data across various applications.

CryptoJS is a JavaScript library that provides cryptographic functions, including encryption, decryption, hashing, and message authentication codes (MACs). It supports various algorithms such as AES, DES, Triple DES, MD5, SHA-1, SHA-256, HMAC, and PBKDF2. CryptoJS simplifies the implementation of cryptographic operations in web applications by providing a consistent and easy-to-use API. It allows developers to securely encrypt sensitive data, generate hashes for data in-

tegrity verification, and implement secure communication protocols (Yoss, 2020).

In the ‘messageSchema’ (Figure 26), before saving the new message details to MongoDB using the 'save' command, the message content is encrypted with a custom encryption key. This encryption process ensures that the message details are protected, preventing unauthorized disclosure and information leakage.

```
14  messageSchema.pre('save', function(next) {
15      const message = this;
16      const encryptedContent = CryptoJS.AES.encrypt(message.content, encryptKey).toString();
17      message.content = encryptedContent;
18      next();
19  });
```

Figure 26. Applying CryptoJS in component

4.4.4 Booking Server (Spring Boot)

Spring Boot is employed to handle specific microservices within the application, such as appointment scheduling and management. This separation of concerns allows for better scalability and maintenance.

```
@Document(collection = "appointment_details")
@Getter
@Setter
@AllArgsConstructor
@NoArgsConstructor
public class BookingForm {
    @Id
    private String id;
    private String username;
    private String patientName;
    private String email;
    private Hospital hospital;
    private String phone;
    private String date;
    private String time;
    private LocalDateTime createdAt = LocalDateTime.now();
}
```

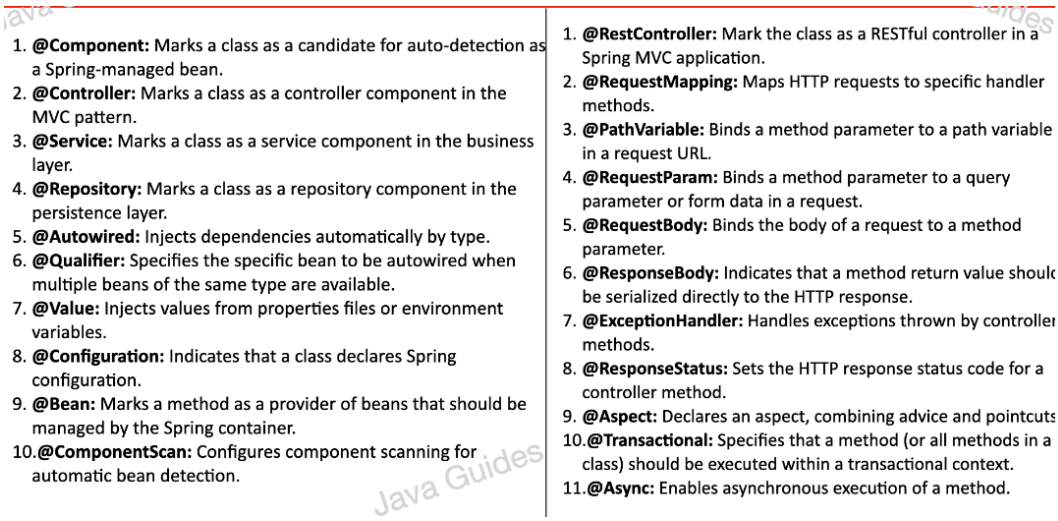
```
@Getter
@Setter
@AllArgsConstructor
@NoArgsConstructor
public class Hospital {

    private String id;
    private String name;
    private String address;
    private String phone;
    private String website;

}
```

Figure 27. Appointment entities

The Spring Boot server processes the business logic related to appointments, including creating, updating, getting appointments and cancellations into MongoDB (Appendix 4).

Using Java Spring Boot offers ease in managing and updating appointment information independently from other Node.js services. The integration between the React frontend and the Spring Boot backend is highly efficient, resulting in minimal latency and ensuring rapid communication.

However, configuring a backend system in Spring Boot is significantly more complex than in Node.js. Dependencies, configuration properties, and components must be meticulously connected using annotations to ensure the system correctly interprets and executes the function of each component. Annotations aid in clearly defining the roles and purposes of classes and functions. They provide a straightforward way to identify the specific responsibilities and behaviors of various components.



1. **@Component:** Marks a class as a candidate for auto-detection as a Spring-managed bean.
2. **@Controller:** Marks a class as a controller component in the MVC pattern.
3. **@Service:** Marks a class as a service component in the business layer.
4. **@Repository:** Marks a class as a repository component in the persistence layer.
5. **@Autowired:** Injects dependencies automatically by type.
6. **@Qualifier:** Specifies the specific bean to be autowired when multiple beans of the same type are available.
7. **@Value:** Injects values from properties files or environment variables.
8. **@Configuration:** Indicates that a class declares Spring configuration.
9. **@Bean:** Marks a method as a provider of beans that should be managed by the Spring container.
10. **@ComponentScan:** Configures component scanning for automatic bean detection.

1. **@RestController:** Mark the class as a RESTful controller in a Spring MVC application.
2. **@RequestMapping:** Maps HTTP requests to specific handler methods.
3. **@PathVariable:** Binds a method parameter to a path variable in a request URL.
4. **@RequestParam:** Binds a method parameter to a query parameter or form data in a request.
5. **@RequestBody:** Binds the body of a request to a method parameter.
6. **@ResponseBody:** Indicates that a method return value should be serialized directly to the HTTP response.
7. **@ExceptionHandler:** Handles exceptions thrown by controller methods.
8. **@ResponseStatus:** Sets the HTTP response status code for a controller method.
9. **@Aspect:** Declares an aspect, combining advice and pointcuts
10. **@Transactional:** Specifies that a method (or all methods in a class) should be executed within a transactional context.
11. **@Async:** Enables asynchronous execution of a method.

Figure 28. Spring framework annotations (Fadatare, 2023)

4.4.5 External APIs

The Google Maps API is a set of web services and tools provided by Google that allows developers to integrate interactive maps, geolocation services, and other

geographic data into their applications and websites. This API enables developers to display maps, customize map appearance, add markers and overlays, calculate directions, and perform geolocation-based searches, among other functionalities. It provides a comprehensive and flexible platform for incorporating mapping and location-based services into applications across various industries and use cases.

```
const PlaceURL = 'https://maps.googleapis.com/maps/api/place/nearbysearch/json';
const detailsURL = 'https://maps.googleapis.com/maps/api/place/details/json';

const hospitalRequest = async (APIKEY, location) => {
    let hospitals = [];
    try{
        const response = await axios.get(PlaceURL, {
            params: {
                key: APIKEY,
                location: `${location.lat},${location.lng}`,
                radius: 5000,
                type: 'hospital'
                // keyword: 'operation'
            }
        })
        const data = response.data;
```

Figure 29. Send request to Google server

The ‘geolib’ library provides a method for calculating the distance between two geographical points on the Earth's surface. It utilizes the Haversine formula, which considers the curvature of the Earth, to accurately calculate the distance between two sets of latitude and longitude coordinates. This method considers the Earth as a sphere, making it suitable for short to medium distances.

Nodemailer is a powerful library for Node.js that enables developers to send emails easily from their applications. It provides a simple and intuitive interface for composing and sending email messages programmatically. With Nodemailer, developers can send plain text, HTML, or multipart emails, attach files, and even embed images inline. It supports various email delivery methods, including SMTP, sending mail, and even direct transport, giving flexibility in how emails are sent. (Shotola, 2023)

4.5 Implementation of Database

The steps taken to create the database structure based on the design specifications. This involves setting up databases, collections, defining relationships, and establishing constraints.

Using the services integrated into the healthcare assistance website project, databases are structured with collections to efficiently store essential data and facilitate rapid data retrieval by the servers.

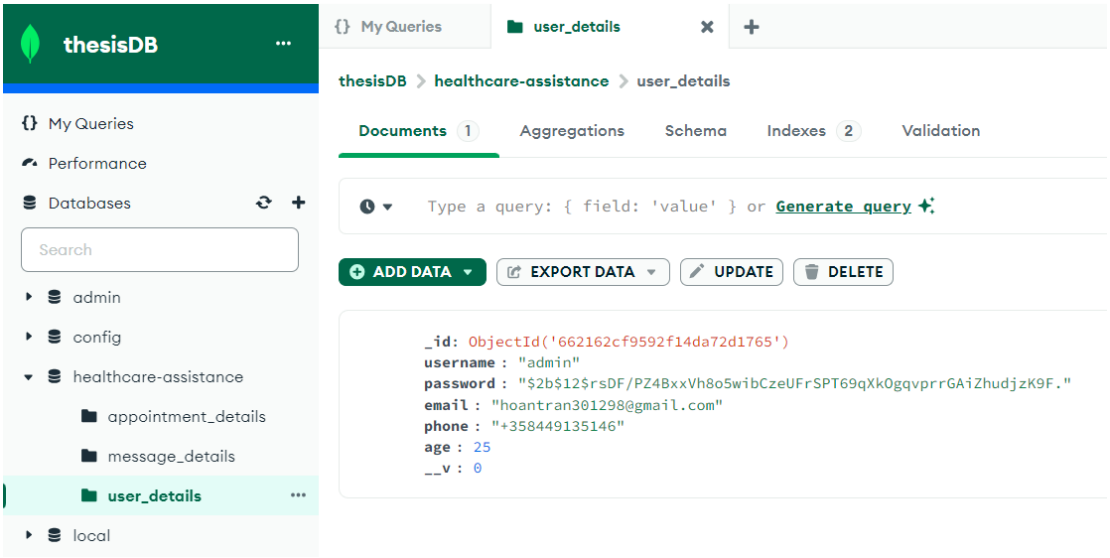


Figure 30. Database design

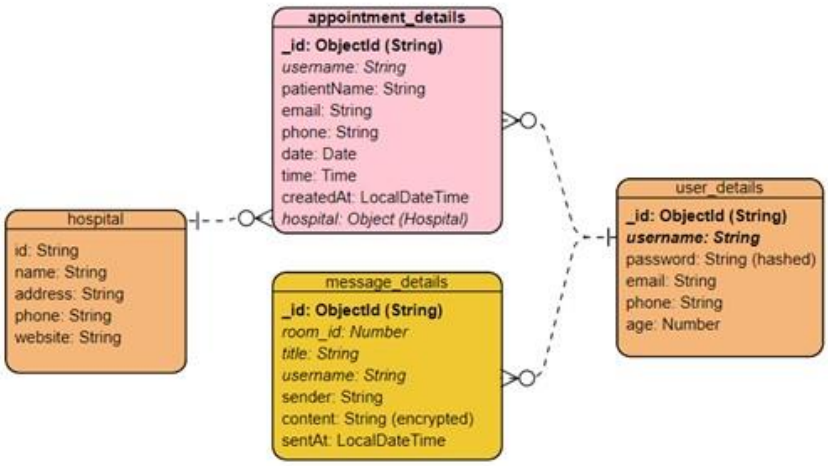


Figure 31. Entity relationship

The process of establishing a connection between a database and an application, enabling the application to interact with and retrieve data from the database. This connection allows the application to send queries, retrieve results, and perform various database operations as needed.

```
const mongoose = require('mongoose');

const connectDatabase = () => {
    mongoose.connect('mongodb://localhost:27017/healthcare-assistance');

    const db = mongoose.connection;
    db.on('error', (error) => {
        console.log(error)
    })

    db.once('open', () => {
        console.log('Database Connection Established!')
    })

}
```

Figure 32. MongoDB connection with Node.js server

4.6 Testing

4.6.1 User Interface

All UI components functioned as expected, including buttons, links, forms, and other interactive elements, ensuring they responded correctly to user inputs. We evaluated the appearance of the UI to ensure adherence to design guidelines and branding requirements, checking for consistency in fonts, colors, layouts, and styles across the application. The UI was tested on various devices and screen sizes, including desktops, laptops, and tablets, to ensure compatibility and responsiveness, identifying and addressing any layout or formatting issues.

We also verified that the UI displayed content correctly in different languages and regions, testing for text truncation, alignment, and encoding issues that could arise with translation.

Overall, all functionalities within the application performed reliably and as intended, with swift loading times for service pages and minimal delays. However, we encountered difficulties with image rendering, which detracted from the application's professionalism and could potentially lead to user disengagement. Addressing these issues will be essential for improving the overall user experience.

4.6.2 System

System testing involves evaluating the entire integrated application to ensure it meets the specified requirements and functions correctly as a whole. This phase includes testing all components and modules, such as user authentication, data processing, external API interactions, and user interface elements, to verify their seamless interaction.

During system testing, we simulate real-world scenarios to check for any discrepancies, performance issues, and security vulnerabilities. This comprehensive testing ensures that the system is robust, reliable, and ready for deployment, identifying and addressing any issues before the application goes live.

When comparing the runtime of Java Spring Boot and Node.js, several key differences emerge. Java Spring Boot applications generally offer high performance and are well-suited for CPU-intensive tasks due to the optimized JVM and multithreading capabilities, though they consume more memory and have longer startup times. This makes them ideal for complex, enterprise-level applications requiring robust performance optimization.

Conversely, Node.js is known for its non-blocking, event-driven architecture, which excels in handling numerous simultaneous connections and I/O-bound tasks with minimal memory usage. Node.js also benefits from faster startup times, making it perfect for microservices and real-time applications that need quick deployment and scalability.

5 FINAL PRODUCT

The Healthcare Assistance Web Application is designed to streamline the process of finding and booking appointments with healthcare facilities. It integrates several key features, including a search functionality powered by Google Maps API, real-time chat capabilities using Socket.io, and secure user authentication with JWT and Bcrypt. The backend is supported by both Java Spring Boot and Node.js, ensuring efficient handling of various tasks such as appointment scheduling and data retrieval from MongoDB.

Initially, the project plan included using the Spring Boot framework for the real-time chat service. However, it was later recognized that Node.js is more suitable for handling a multi-threaded system. This choice facilitates the smoother transmission of information, such as messages and related details, ensuring the server can manage high volumes of requests without performance degradation.

This change enabled the chat service to be completed ahead of schedule while delivering solid performance. Implementing the chat service with Node.js was simpler and reduced the complexity of the server system.

The source code for the application encompasses all the essential components and modules that make up the project. It includes the front-end and back-end code, configuration files, and any third-party libraries or dependencies that are integrated into the application. The comprehensive structure and integration of these components ensure that the application operates smoothly, providing users with efficient and secure access to healthcare services.

The GitHub repository of the application can be found at:

https://github.com/hoan301298/Healthcare-Assistance

5.1 Successful Aspects

The application provides a clean, intuitive interface that guides users through the process of finding healthcare facilities and booking appointments. The use of React ensures a responsive and dynamic user experience.

Implementing Socket.io for real-time chat functionality allows users to communicate instantly with healthcare professionals. This feature enhances user engagement and provides immediate assistance.

The use of JWT and Bcrypt for user authentication ensures that user data is protected. This security measure is crucial for maintaining the confidentiality and integrity of sensitive healthcare information.

By utilizing both Java Spring Boot and Node.js, the application benefits from the strengths of both frameworks. This hybrid approach allows for efficient handling of appointments and other critical tasks.

The integration with Google Maps API provides accurate location-based search results, helping users find nearby healthcare facilities quickly and easily.

5.2 Challenges Encountered

Implementing real-time chat with Socket.io was initially challenging due to the need for continuous communication and event handling. Understanding and effectively using Socket.io required significant effort.

Ensuring robust security measures, such as encryption and secure token storage, was complex. Balancing security with user convenience required careful planning and implementation.

Setting up the backend with Spring Boot was more complicated than anticipated. The configuration of dependencies and linking components through annotations required a deep understanding of the framework.

6 CONCLUSIONS

The completion of this thesis project marks a significant achievement in developing a robust and user-friendly healthcare assistance application. This project has integrated various advanced technologies and methodologies to meet its objectives, providing a comprehensive solution for users seeking healthcare services. The combination of the MERN stack (MongoDB, Express.js, React.js, Node.js) and Spring Boot has proven to be a powerful solution for building scalable and efficient applications. Spring Boot provided a reliable and secure environment for handling critical tasks like appointment scheduling.

The project demonstrated the seamless integration between the React-based front-end and the Spring Boot and Node.js back-end. This integration ensured smooth data flow and user interactions, enhancing the overall user experience. The use of Axios for API calls and react-google-maps/api for map functionalities streamlined front-end operations.

Implementing real-time features using Socket.IO allowed for efficient and continuous communication between users and healthcare professionals. This capability is crucial for functionalities like live chat, which enhances user engagement and support.

Two of the significant challenges were communicating service and ensuring secure token storage. This required a thorough understanding of socket communication and middleware management.

The current user interface lacks professional polish and could benefit from modern design principles to enhance user engagement. Performance optimization is necessary to improve data fetching speed and ensure scalability, with techniques such as data caching and load balancing. Security enhancements, including better token storage and advanced authentication features like two-factor authentication, are crucial.

REFERENCES

Adam. (2022). *Why the right technology stack for your web applications is such a crucial strategic decision and a checklist to get it right*. Retrieved from kruschecompany: https://kruschecompany.com/technology-stack-web-applications/

Borozenets. (2022). *What is the best choice for 2022?* Retrieved from fulcrum: https://fulcrum.rocks/blog/vue-vs-react-comparison

Clinton. (2023). *How Website Encryption Works*. Retrieved from freecodecamp: https://www.freecodecamp.org/news/understanding-website-encryption/

Codex. (2023). *What are the advantages of using React Hooks?* Retrieved from reintech: https://reintech.io/blog/tutorial-for-react-developers-what-are-the-advantages-of-using-react-hooks

Crane. (2020). *Symmetric Encryption 101: Definition, How It Works & When It's Used*. Retrieved from thesslstore: https://www.thesslstore.com/blog/symmetric-encryption-101-definition-how-it-works-when-its-used/

Dave. (2023). *Decoding the CTO's Dilemma: MERN vs MEAN Stack Development*. Retrieved from radixweb: https://radixweb.com/blog/mern-stack-vs-mean-stack

Defranchi. (2024). *Public APIs*. Retrieved from blog.axway: https://blog.axway.com/learning-center/apis/basics/different-types-apis#public-apis

Fadatare. (2023). *Spring Boot and Spring Framework Annotations Cheat Sheet*. Retrieved from javaguides: https://www.javaguides.net/2023/07/spring-boot-and-spring-framework.html

Geeksforgeeks. (2021). *What are the key features of Node.js ?* Retrieved from geeksforgeeks: https://www.geeksforgeeks.org/what-are-the-key-features-of-node-js/

Gillis. (2023). *DEFINITION MongoDB*. Retrieved from techtarget: https://www.techtarget.com/searchdatamanagement/definition/MongoDB

Guntupally, Devarakonda, Kehoe. (2018). *Spring Boot based REST API to Improve Data Quality Report Generation for Big Scientific Data: ARM Data Center Example*. Retrieved from ieeexplore: https://ieeexplore.ieee.org/abstract/document/8621924

Hu. (2012). *Online Map Service Using Google Maps API and Other JavaScript Libraries: An Open Source Method*. Retrieved from springer: https://link.springer.com/chapter/10.1007/978-3-642-27485-5_17

Innocent. (2024). *Axios vs Fetch: Which is best for HTTP requests in 2024?* Retrieved from apidog: https://apidog.com/blog/axios-vs-fetch/#:~:text=Automatic%20JSON%20data%20transformation%3A%20Axios,takes%20too%20long%20to%20respond.

Jones. (2022). *SHA-2 and Bcrypt Encryption Algorithms*. Retrieved from medium: https://medium.com/nerd-for-tech/sha-2-and-bcrypt-encryption-algorithms-e0c0599b0da

Kamath. (2023). *Understanding Spring Boot Architecture*. Retrieved from gitconnected: https://levelup.gitconnected.com/understanding-spring-boot-architecture-6083e2631bc6

Maksimovic. (2021). *MongoDB Overview*. Retrieved from syncfusion: https://www.syncfusion.com/succinctly-free-ebooks/mongodb-3-succinctly/mongodb-overview

Maldonado. (2023). *All You Need to Know About CORS & CORS Errors*. Retrieved from telerik: https://www.telerik.com/blogs/all-you-need-to-know-cors-errors#:~:text=CORS%20errors%20happen%20when%20a,by%20the%20server%27s%20CORS%20configuration.

Monika, Manish, Anjali, Charu, Shanu. (2021). *MERN Stack Web Development*. Retrieved from annalsofrscb: http://annalsofrscb.ro/index.php/journal/article/view/7719/5721

Nadaf. (2023). *Quick Node/Express.Js Project Setup Guide.* Retrieved from medium: https://medium.com/@naveednadaf/quick-node-express-js-project-setup-guide-88cd4d9a7af3

O'Sullivan. (2023). *What is encryption?* Retrieved from proton: https://proton.me/blog/what-is-encryption

Poddar. (2022). *What is a JWT? Understanding JSON Web Tokens*. Retrieved from supertokens: https://supertokens.com/blog/what-is-jwt

Rachel. (2023). *What is a Programming Library?* Retrieved from careerfoundry: https://careerfoundry.com/en/blog/web-development/programming-library-guide/

Sankalana. (2021). *Express.js*. Retrieved from medium: https://agasthisankalana.medium.com/express-js-e8df1ba62c69

Sheldon. (2024). *Building MongoDB Aggregations*. Retrieved from red-gate: https://www.red-gate.com/simple-talk/homepage/building-mongodb-aggregations/

Shotola. (2023). *Understanding Nodemailer: A Thorough Guide for Email Sending with Node.js*. Retrieved from medium: https://medium.com/@femishotolaa/understanding-nodemailer-a-thorough-guide-for-email-sending-with-node-js-ebb45417c64d

Simplilearn. (2023). *Why Use MongoDB: What It Is and What Are the Benefits*. Retrieved from simplilearn: https://www.simplilearn.com/tutorials/mongodb-tutorial/what-is-mongodb

Thakur. (2024, Jan). *The Good, Bad, and Ugly of Using Third-Party Libraries*. Retrieved from gitconnected: https://levelup.gitconnected.com/the-good-bad-and-ugly-of-using-third-party-libraries-b0ddb2bf990c

Thomas. (2023). *What are the Essential Technologies for Building a Java-based Web Application?* Retrieved from medium: https://medium.com/@georgethoms852/what-are-the-essential-technologies-for-building-a-java-based-web-application-8940850edb0c

TMDesign. (2023). *Web Development Stacks*. Retrieved from Medium: https://medium.com/theymakedesign/what-is-web-development-stack-a9a01c5ab9e8

Tynkkynen, Pulkki, Leena, Schön, Burström, Keskimäki . (2022). *Health system reforms and the needs of the ageing population—an analysis of recent policy paths and reform trends in Finland and Sweden*. Retrieved from Springer Link: https://link.springer.com/article/10.1007/s10433-022-00699-x#ref-CR55

Wikipedia. (2019). *Express.js*. Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Express.js

Wikipedia. (2022). *Vastaamo data breach*. Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Vastaamo_data_breach

Wilson, H. B.-L. (2021). *Barriers and facilitators to the use of e-health by older adults: a scoping review*. Retrieved from bmcpublichealth:

https://bmcpublichealth.biomedcentral.com/articles/10.1186/s12889-021-11623-w

Yoss. (2020). *Encryption made Simple with CryptoJS*. Retrieved from levelup: https://levelup.gitconnected.com/encryption-made-simple-with-cryptojs-3628f5867d78

APPENDICES

Appendix 1

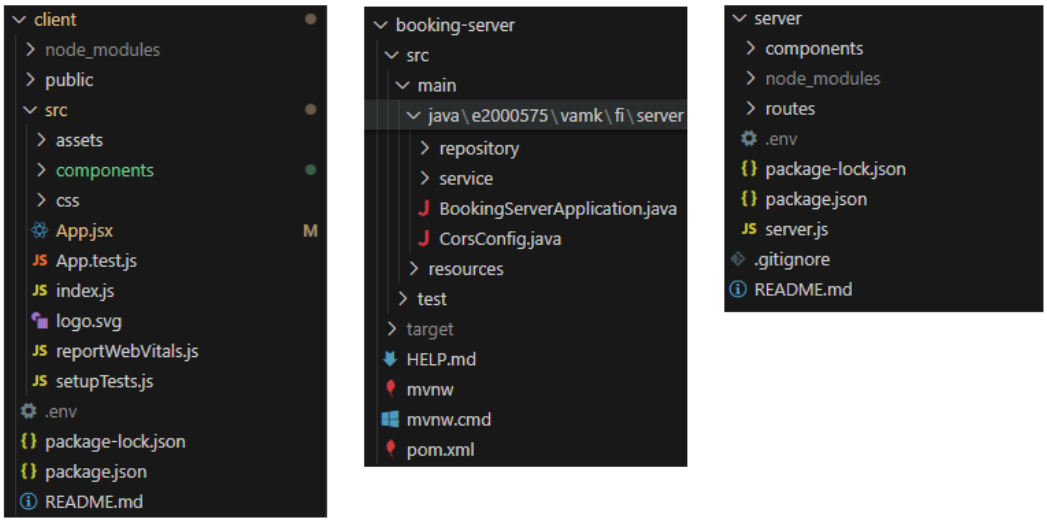
Third-party libraries

Library Name	Purpose	Language	Example Use Cases
Axios	HTTP client for making requests	JavaScript	Fetching data from APIs
@react-google-maps/api	Google Maps integration for React	JavaScript	Displaying interactive maps
Mongoose	Object Data Modeling (ODM) for MongoDB	JavaScript	Interacting with MongoDB databases
Nodemailer	Sending emails programmatically	JavaScript	Sending notification emails
Express	Web application framework for Node.js	JavaScript	Building RESTful APIs
dotenv	Environment variable loader	JavaScript	Managing environment variables
react-router-dom	Routing management for React applications	JavaScript	Easy navigation between pages
geolib	Geospatial calculations	JavaScript	Calculating distances between coordinates
next-ui	UI component library for React	JavaScript	Building consistent, accessible, and responsive UIs

All the libraries were used in project.

Appendix 2

Project structure



The tree of project structure indicates some layers, such as client, booking-service (Spring Boot), server (Node.js).

Appendix 3

Login component

Login component is shown as the figure above with sending login request to server to get a token.

Reset Password component in Account service

```
const ResetPassword = ({username}) => {
    const [oldPassword, setOldPassword] = useState('');
    const [newPassword, setNewPassword] = useState('');
    const [confirmedPassword, setConfirmedPassword] = useState('');
    const [response, setResponse] = useState(null);

    const handleResetPassword = async (e) => {
        e.preventDefault();
        if(newPassword === confirmedPassword) {
            try{
                const res = await axios.post('/account/reset-password', {username, oldPassword, newPassword});
                setResponse(res.data);
            } catch (e) {
                console.log(e);
            }
        } else {
            alert('Your password does not match!');
        }
    }
```

Reset Password component in account service which helps customers to change their password.

Show Profile component in Account service

```
const ShowProfile = ({username}) => {
    const [userDetails, setUserDetails] = useState({});
    const [email, setEmail] = useState(null);
    const [phone, setPhone] = useState(null);
    const [age, setAge] = useState(null);
    const [password, setPassword] = useState(null);
    const [responseData, setResponseData] = useState(null);

    const getUserDetails = async () => {
        try{
            const user = await getProfile(username);
            setUserDetails(user);
        } catch (e) {
            console.log(e);
        }
    }

    useEffect(() => {
        getUserDetails();
    }, []);

    const handleSubmit = async (e) => {
        e.preventDefault();
        const updateUser = {
            username: username,
            password: password,
            email: email,
            phone: phone,
            age: age
        }
        try{
            const update = await updateProfile(updateUser);
            setResponseData(update)
        } catch(e) {
            console.log(e);
        }
    }
```

This component will fetch user's details from server.

Appointment management in Account service

client > src > components > account_service > action > ShowAppointment.jsx > ShowAppointment

```jsx
import axios from "axios";
import { useEffect, useState } from "react";

const ShowAppointment = (username) => {
    const [appointments, setAppointments] = useState([]);

    const getAllAppointment = async () => {
        try{
            const response = await axios.get(`http://localhost:8080/booking-form/${username}`)
            setAppointments(response.data);
        } catch (e) {
            console.error('Error fetching appointments:', e);
        }
    }

    useEffect(() => {
        getAllAppointment();
    }, [username])
```

This component will fetch all reservations upon user request.

Appendix 4

Spring Boot service

```
public List<BookingForm> getAppointmentByUsername (String username) {
    for (BookingForm form : getAllAppointment()) {
        if(form.getUsername().equals(username)) {
            appointments.add(appointment);
        }
    }
    return appointments;
}
```

The function named 'getAppointmentByUsername' is one of the functions in Spring Boot service will fetch all the appointment belonged to user requesting.

Spring Boot controller

```
@GetMapping("/{username}")
public ResponseEntity<?> getAppointmentByUsername(@PathVariable String username){
    List<BookingForm> appointments = bookingService.getAppointmentByUsername(username);
    if(appointments.isEmpty()) {
        return ResponseEntity.notFound().build();
    } else {
        return ResponseEntity.ok(appointments);
    }
}
```

The controller manages the GET method to get all the appointments in controller components.