



Eötvös Loránd University Faculty of
Informatics Department of Media and
Educational Informatics

DocHouse

Supervisor:

Illés Zoltán

Associate Professor - PhD,
habilitation

Author:

Sadi Mamedov

Computer Science
BSc

Budapest, 2020

Contents

1 Introduction	3
1.1 Background	3
1.2 Solution to the problem	4
2 User Documentation	5
2.1 System requirements	5
2.1.1 Software requirements	5
2.2 Installation process	6
2.3 Common Interface Guidelines	7
2.3.1 Home page	7
2.3.2 Login Transition	8
2.3.3 User and Doctor Registration	8
2.3.4 User and Doctor Login	10
2.4 User Interface Guidelines	11
2.4.1 User Navigation Sidebar	11
2.4.2 User Profile	11
2.4.3 Search Doctor	13
2.4.4 View Profile	14
2.4.5 User Chat	15
2.4.6 User Dashboard	16
2.4.7 Payment	18
2.5 Doctor Interface Guidelines	19
2.5.1 Doctor Navigation Sidebar	19
2.5.2 Doctor Profile	19

2.5.3 Doctor Dashboard	21
2.5.4 Doctor Chat	24
3 Developer Documentation	25
3.1 Program Structure	25
3.2 Use Case diagram	28
3.3 Class diagrams	28
3.4 Authorization processes	31
3.4.1 Transition Process	31
3.4.2 Registration Process	31
3.4.3 Login Process	32
3.5 Profile Processes	33
3.5.1 Navigation Process	33
3.5.2 Editing Process	34
3.5.3 Search Process	35
3.5.4 Visit Profile Processes	36
3.5.5 Dashboard Processes	37
3.5.6 Payment Process	37
3.5.7 Chat Process	38
3.6 Testing	39
4 Conclusion	41
4.1 Future work	41
References	42

Chapter 1

Introduction

Over the past decades web development has been reached to advanced improvements, growing its influence as a proper solution and environment to build real world applications. Since the software development period, applications used to require specific software and hardware requirements only to serve features under the local machine environment. After the introduction of web services, several desktop applications have been migrated to the web environment for the reasons that it provides platform independence, device accessibility, and demanding few system requirements.

1.1 Background

Nowadays, information technology plays a significant role in human daily life to such an extent that enables us to manage time efficiently. Moreover, using digital applications we can solve issues related to public services. Traditional Health care services are considered one of the most time consuming, crowded and nerve-racking for both doctors and patients due to long waiting queues, registering patients, scheduling workflow, unavailability of staff members, etc. On the other hand, It's very difficult to find professional doctors based on specific medical fields. Consulting with a doctor is one of the common activities in society, whereas due to aforementioned problems we consider it tedious practice that most of the time leads to ignoring our health issues.

1.2 Solution to the problem

Since web platforms are accessible from various types of devices including laptops, tablets, mobile phones, considering the large audience and necessity of this problem, I decided to build the application over the web surface. By accessing the homepage of this application, patients may benefit from making a medical appointment with qualified doctors, also from receiving answers to their health-related questions through written communication. At the same time, doctors may register to the trusted platform and continue to provide their services to the patients in need. The application intended to serve two sorts of user audience: patients and doctors, who are required to register in order to use services, whereas, guests will be able to collect general details and information about the provided facilities.

Chapter 2

User Documentation

There will be two different portals with distinct interfaces corresponding to the user type.

Through the documentation, patient type users will be referred as ‘users’, while doctor type users will be referred as ‘doctors’. Usage of the program consists of following functionalities:

1. Common interface:

All kinds of users including unauthorized users referred to as ‘guests’ can register, navigate to the login transition page and homepage of the application. After the registration, the type of user is determined.

2. User interface:

Users may sign in, sign out, search doctor, edit user profile, navigate to the doctor’s profile, make an appointment request, delete settled requests, review doctor, make payment, and initiate communication via chat.

3. Doctor interface:

Doctors may sign in, sign out, edit doctor profile, accept a request, reject a request, delete settled requests, join communication via chat.

Some system requirements must be satisfied in order to use the application.

2.1 System requirements

Devices must have network access having bandwidth greater than 50Kb for page size. Also, from hardware minimum 2 GB of RAM, and 64-bit environment 2 GHz dual processor on MS-Windows (7 and later), MacOS (versions 10.2 and later) , latest Linux (64 bit)

distributions, for displaying standard resolution of (1024 x 768) VGA are required.

2.1.1 Software requirements

All modern browsers support react-built web applications. For better performance latest releases of Chrome, Mozilla or Safari recommended. In order to run the client and server side application Node package manager (npm) must be installed in the operating system.

2.2 Installation process

In order to run the application in local environment follow instructions below:

1. Copy the following URL of github public repository of the project from [1].
2. On your local machine, clone the repository to the working directory using `'git clone <https-URL>'` command.
3. Navigate to the cloned repository, and install all dependencies for the server application using the `'npm install'` command.
4. Navigate to the client repository using `'cd doc-house'`. Install all dependencies for the client application using `'npm install'` command.
5. To test the server application run `'npm test'` command.
To test the client application run `'doc-house && npm test'` command.
6. Navigate back to the root repository, and run `'npm start'` which will run both client and server application concurrently.
7. If there have not occurred prior errors until this step, npm successfully will redirect you to 'Dothouse' homepage on your default browser.

2.3 Common Interface Guidelines

In this part of the chapter, a detailed explanation of the common user interface will be presented using figures. The common interface can be accessed by all kinds of users after triggering the client side application.

2.3.1 Home page

After starting the program, the homepage of the website will appear in your default browser. Users may obtain general information about the application and its services. One can review all the details using component scroller. With the help of navigation bar, users may proceed to login transition page (top right corner), and by clicking Menu button (top left corner) to open Sidebar which provides navigation to the other pages (**Figure 2.3.1**).

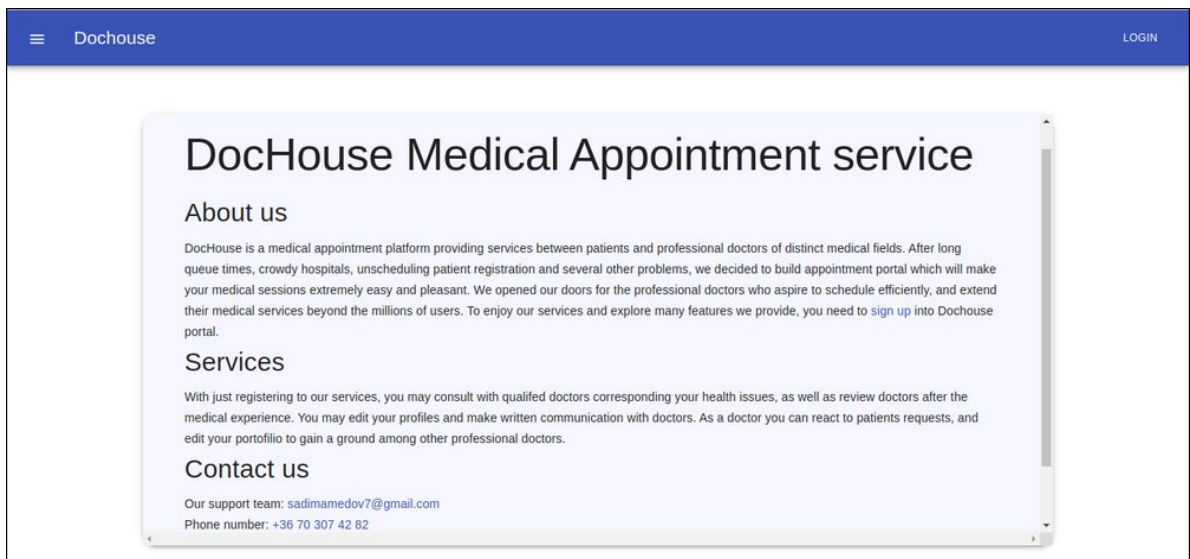


Figure 2.3.1: Home page of the website

Pages, other than those which are not allowed to access from unauthorized users are explorable from the home page.

2.3.2 Login Transition

Transition page provides account options for users and doctors. By clicking on the named buttons, one will be redirected to the dedicated interface in order to register and sign in to the application (**Figure 2.3.2**).

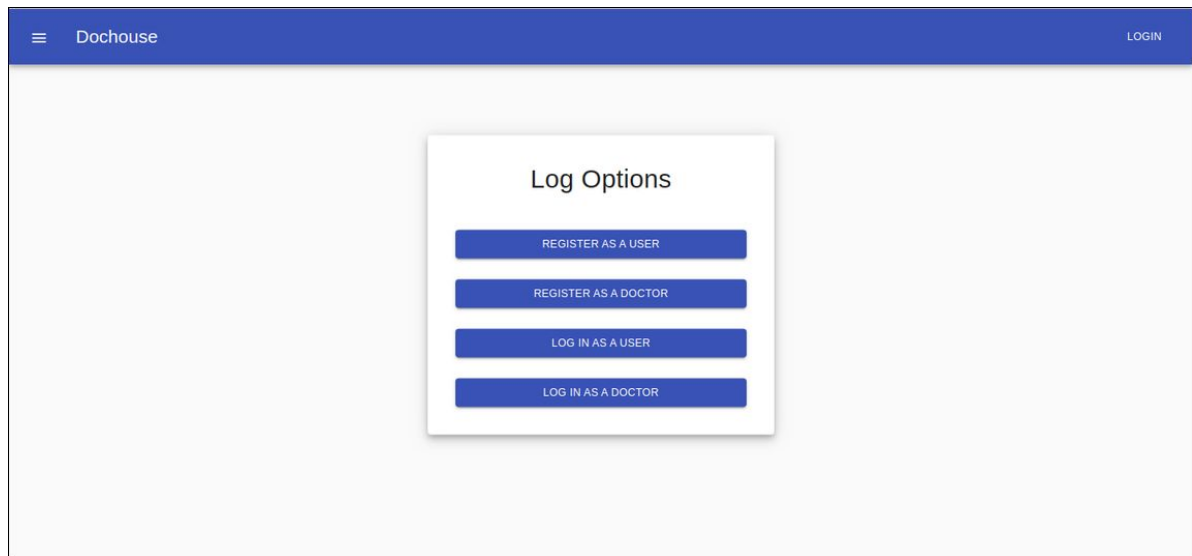
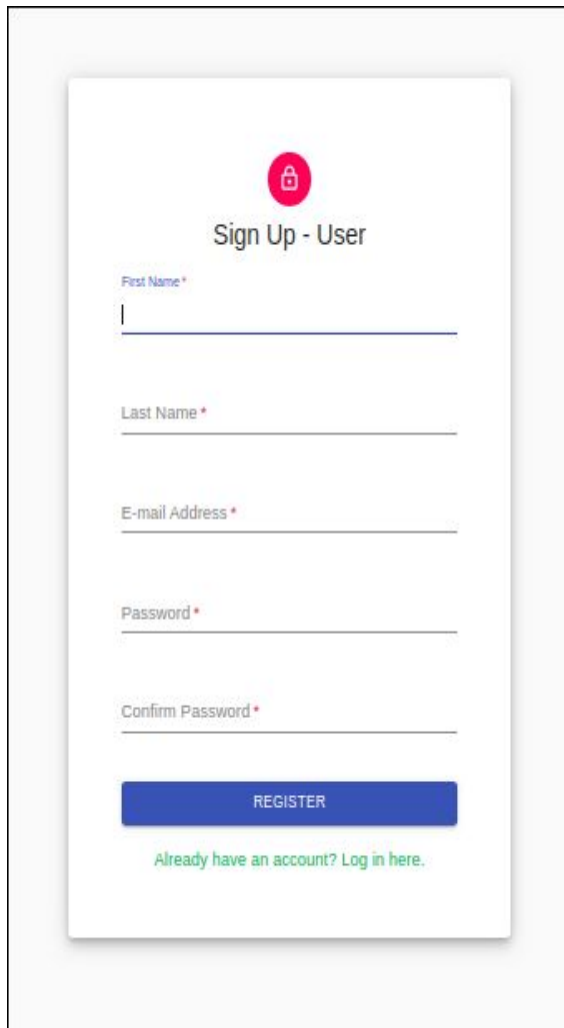


Figure 2.3.2: Login transition page of the website

2.3.3 User and Doctor Registration

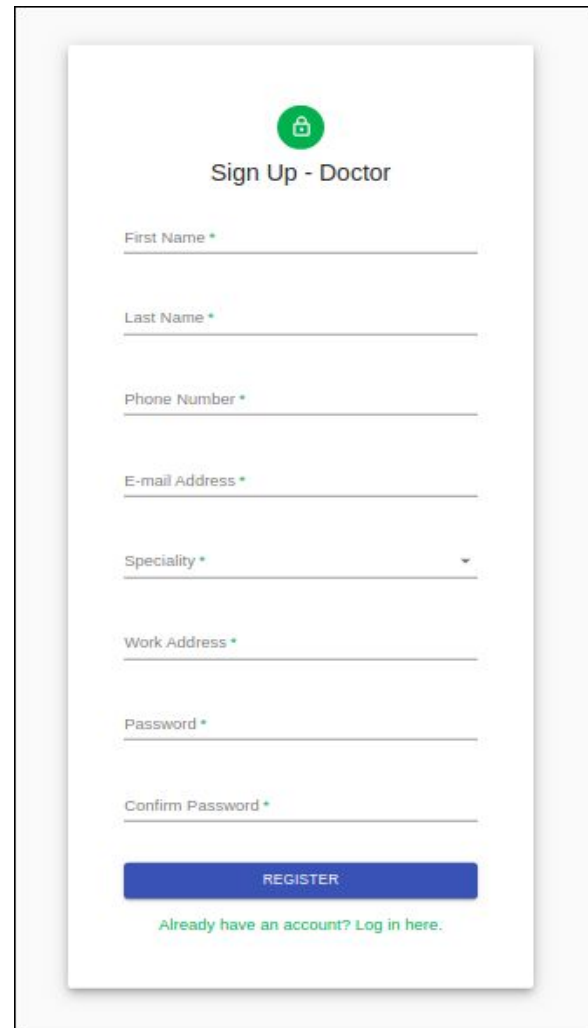
After pressing on *Register as a user* button depicted on (**Figure 2.3.31**), users lead to the registration page. Users must fill out required fields and then submit the registration form. To continue further, users will be tempted to enter their data until all the validation rules are satisfied. In case valid data inserted, within the form snackbar appears illustrating successful registration. As the next step, By following the snackbar link, users may sign in to the application.

Similar to users, doctors are also required to register by filling out their personal data. Validation rules will apply until correct standard input is provided, and then a successful registration indicator will pop up. The provided data by user and doctor, will be used in the authentication process to identify log entities (**Figure 2.3.32**).



The 'Sign Up - User' form features a red padlock icon at the top. It includes input fields for First Name, Last Name, E-mail Address, Password, and Confirm Password, each with a red asterisk indicating a required field. A blue 'REGISTER' button is positioned below the fields, followed by a green link that reads 'Already have an account? Log in here.'

Figure 2.3.31: User sign up component



The 'Sign Up - Doctor' form features a green padlock icon at the top. It includes input fields for First Name, Last Name, Phone Number, E-mail Address, Speciality (a dropdown menu), Work Address, Password, and Confirm Password, each with a green asterisk indicating a required field. A blue 'REGISTER' button is positioned below the fields, followed by a green link that reads 'Already have an account? Log in here.'

Figure 2.3.32: Doctor sign up component

2.3.4 User and Doctor Login

By following any of the routes to login pages mentioned in previous parts of chapter, users and doctors can access the corresponding login pages. To continue with the sign in process, the program requires account credentials those of which are acquired in the registration process. Thus, only authorized users and doctors may have account sessions. After entering credentials and pressing *Log In* button, program will redirect to profile page (**Figure 2.3.41**, **Figure 2.3.42**).

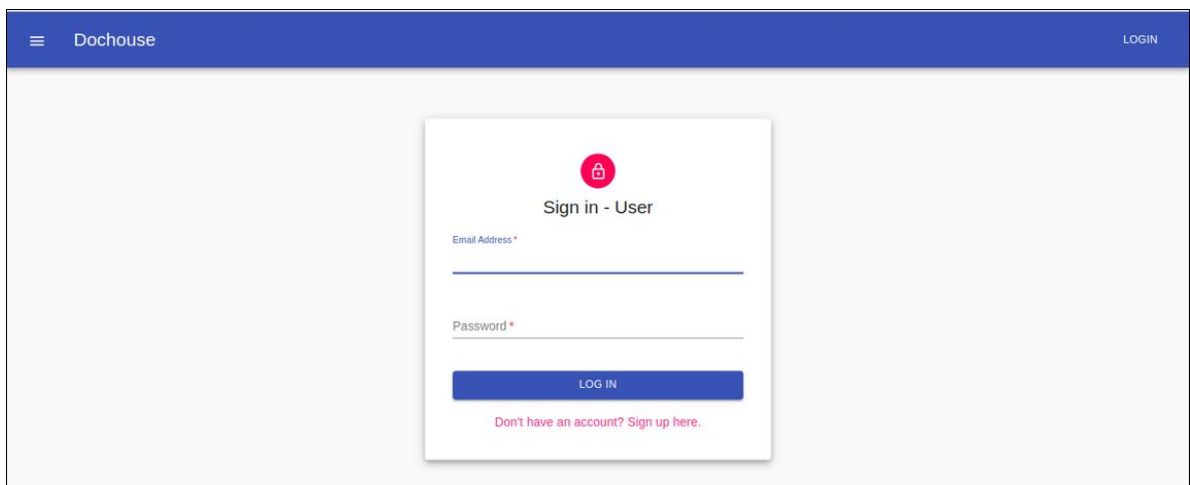
The image shows a web application interface for user login. At the top, there is a blue header bar with a hamburger menu icon on the left, the text "Dochouse" in the center, and a "LOGIN" link on the right. The main content area is light gray and contains a white login card. The card has a red circular icon with a white lock symbol at the top. Below the icon, the text "Sign in - User" is displayed. There are two input fields: "Email Address *" and "Password *", both with red asterisks indicating required fields. Below the input fields is a blue "LOG IN" button. At the bottom of the card, there is a link that says "Don't have an account? Sign up here." in red text.

Figure 2.3.41: User sign in page

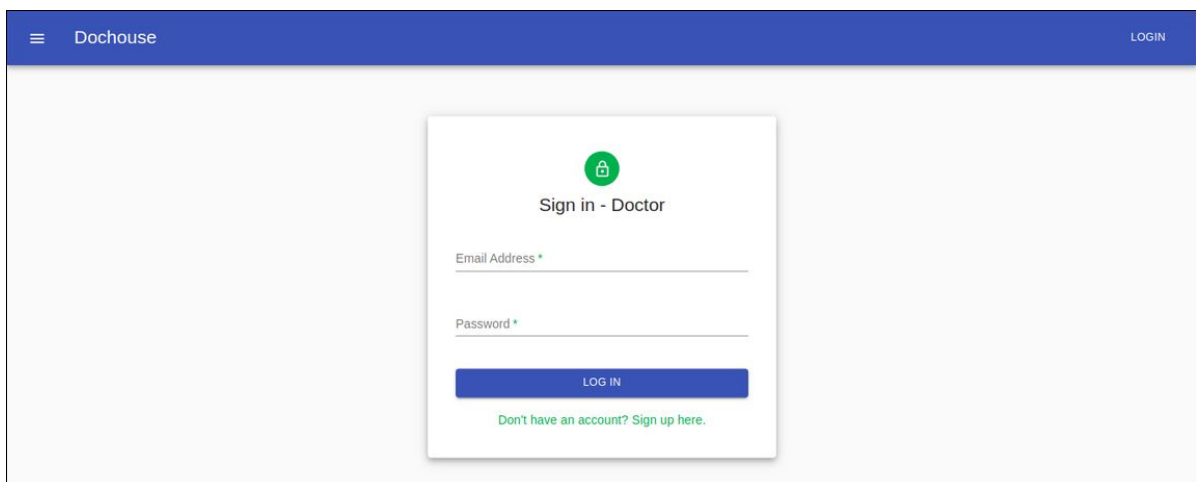
The image shows a web application interface for doctor login. It has the same blue header bar as the user login page, with a hamburger menu icon, "Dochouse" text, and a "LOGIN" link. The main content area is light gray and contains a white login card. The card has a green circular icon with a white lock symbol at the top. Below the icon, the text "Sign in - Doctor" is displayed. There are two input fields: "Email Address *" and "Password *", both with green asterisks indicating required fields. Below the input fields is a blue "LOG IN" button. At the bottom of the card, there is a link that says "Don't have an account? Sign up here." in green text.

Figure 2.3.42: Doctor sign in page

2.4 User Interface Guidelines

In this chapter, user interface components will be introduced using sample figures.

2.4.1 User Navigation Sidebar

User dashboard is the first page after the user successfully logged in. Navigation bar will appear on every page covering the *Logout* button which is responsible to log the user out and terminate the session, and the *Menu* button which provides the sidebar options in the left side of screen (**Figure 2.4.1**). Sidebar gives important navigation facilities to lead to relevant pages, and provides better user experience on small screen devices. More detailed explanation about the dashboard will be presented in the following subchapters.

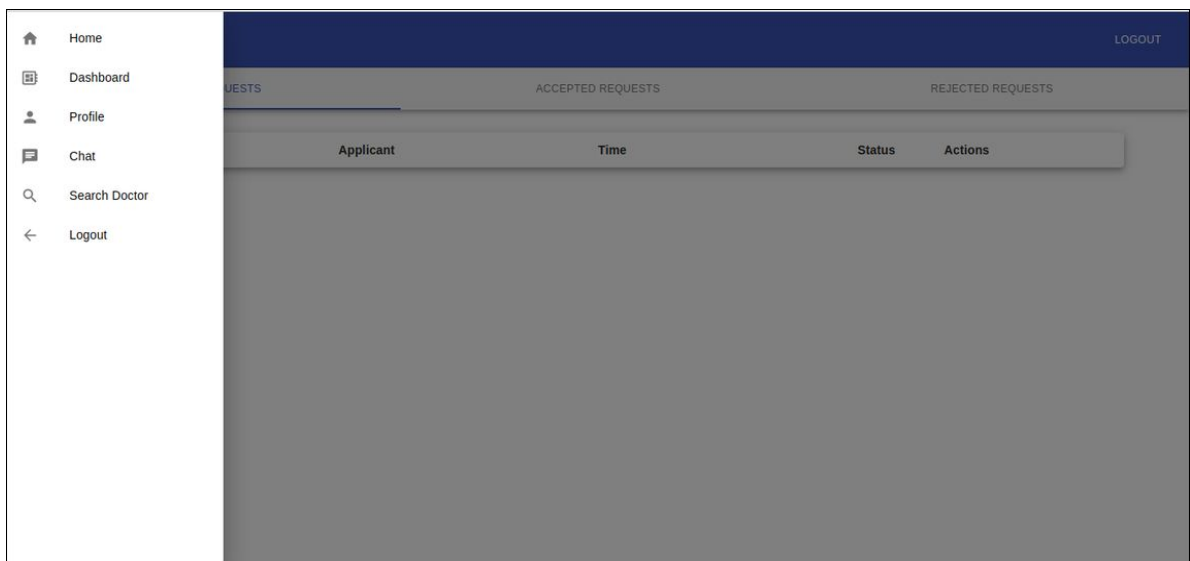


Figure 2.4.1: User dashboard with open sidebar

2.4.2 User Profile

Following sidebar navigation users may reach to the profile page. Profile page resembles a portfolio where users may review their personal information (**Figure 2.4.21**).

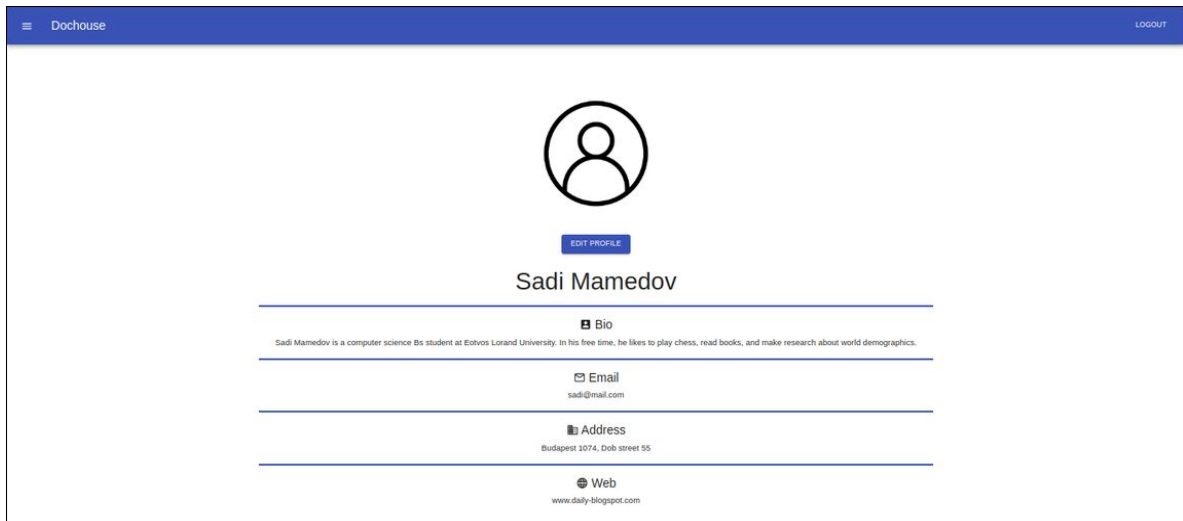


Figure 2.4.21: Profile page of website

Users are given functionality to edit profile details by pressing *Edit Profile* button. After making changes to corresponding fields, one may save all the changes by clicking *Save* button. *Cancel* button will simply close the modal and ignore changes (**Figure 2.4.22**).

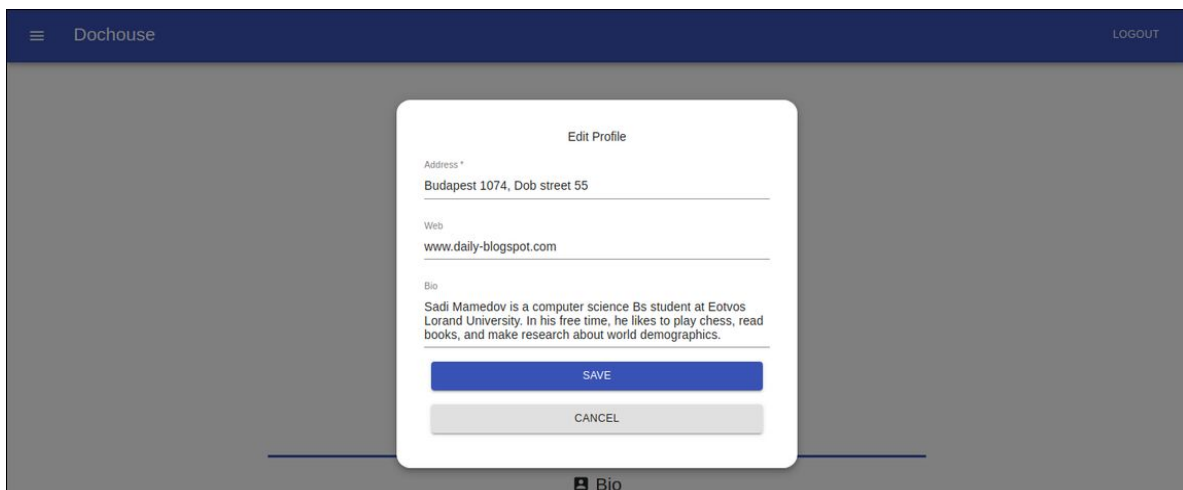
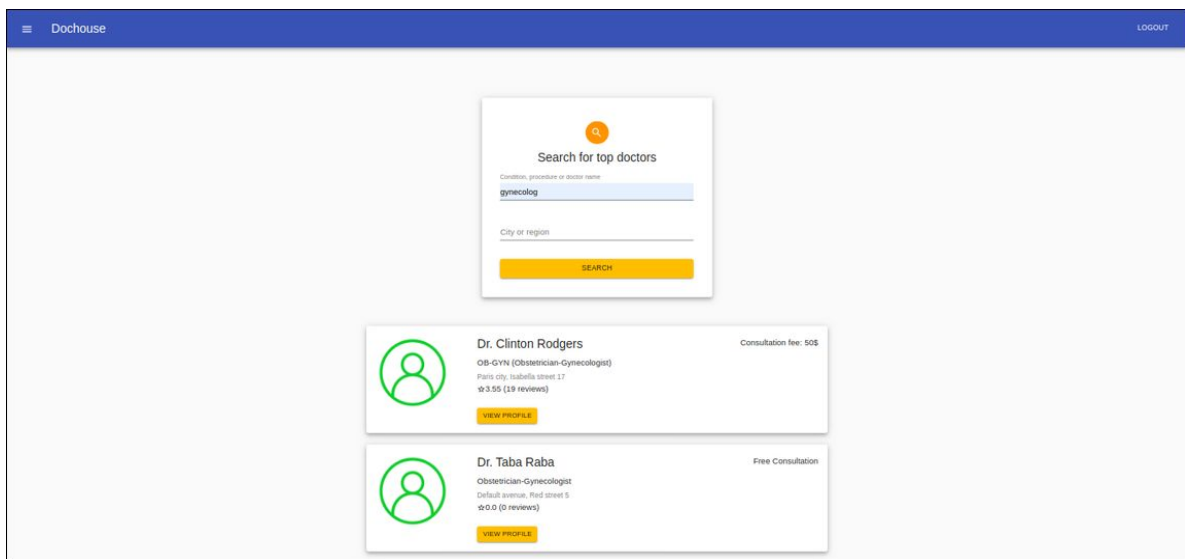


Figure 2.4.22: Edit Profile Modal of profile page

2.4.3 Search Doctor

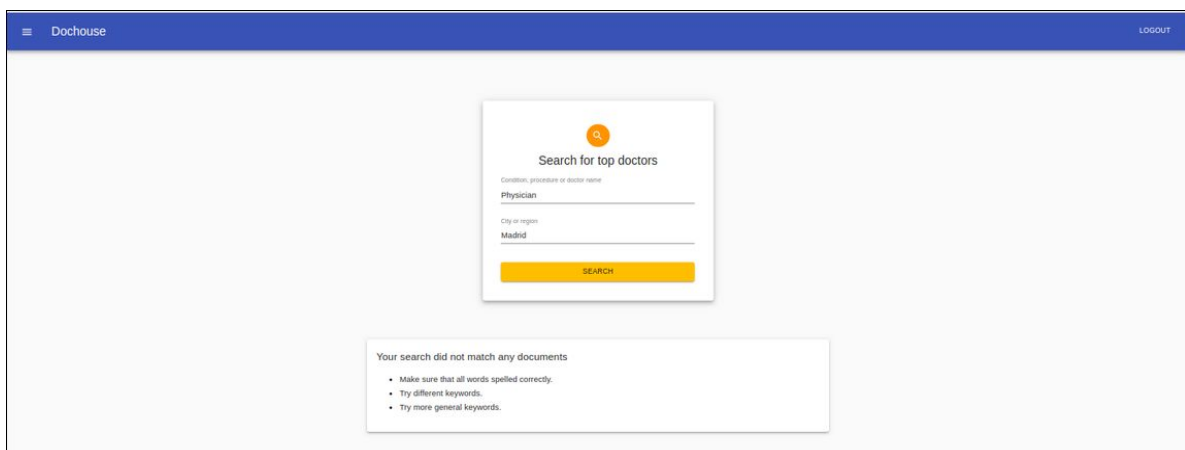
Users may discover doctors who are members of the application network. Doctors may be searched based on *city or region*, and *condition or doctor name* criterias. After clicking on *search* button, a list of doctors will appear representing doctor description including their rating, work address and consultation fee. *View Profile* button will forward the user to the selected doctor's profile (**Figure 2.4.31**).



The screenshot shows the 'Search for top doctors' interface. At the top, there's a search bar with a magnifying glass icon. Below it, the text 'Search for top doctors' is followed by two input fields: 'Condition, procedure or doctor name' (containing 'gynecolog') and 'City or region'. A yellow 'SEARCH' button is at the bottom of the search form. Below the search form, two doctor cards are displayed. The first card is for Dr. Clinton Rodgers, an OB-GYN in Paris city, with a 4.3/5 rating from 19 reviews and a \$50 consultation fee. The second card is for Dr. Taba Raba, an Obstetrician-Gynecologist in Madrid, with a 4.0/5 rating from 0 reviews and a free consultation. Each card includes a green circular profile icon and a yellow 'VIEW PROFILE' button.

Figure 2.4.31: Search Doctor page of website

In case of no information found based on search query, relevant warning will pop up under the search component (**Figure 2.4.32**).



The screenshot shows the 'Search for top doctors' interface with a 'Not found' warning. The search bar has 'Physician' in the 'Condition, procedure or doctor name' field and 'Madrid' in the 'City or region' field. A yellow 'SEARCH' button is at the bottom of the search form. Below the search form, a white box with a yellow border contains the message: 'Your search did not match any documents'. Below this message, there are three bullet points: 'Make sure that all words spelled correctly.', 'Try different keywords.', and 'Try more general keywords.'

Figure 2.4.32: Not found warning of Search Doctor page

2.4.4 View Profile

One of the most necessary part of the application is to view the doctor's profile where users may decide whether to make an appointment request based on the doctor's qualifications. Aside from personal details, review scores conducted by other patients are illustrated. Considering the appointment fee users may submit a request by following *Book* button. By clicking *Review Doctor* button, the user may initiate interactive chat with the doctor. Subject of the appointment, detailed explanation of the health issue and appointment proposal time must be provided by the user (**Figure 2.4.41, Figure 2.4.42**).

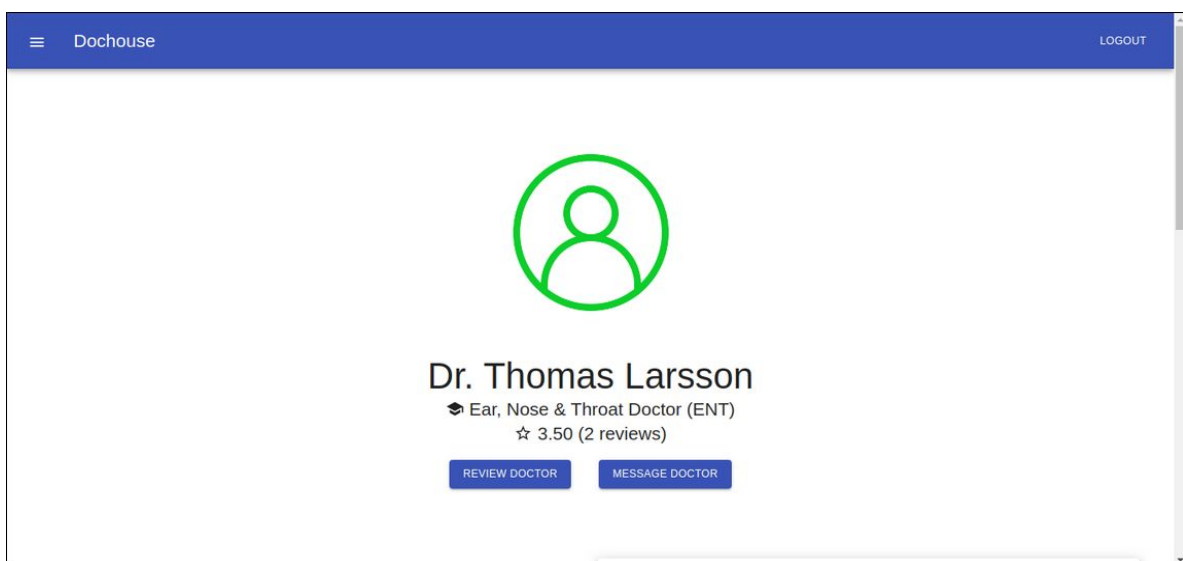


Figure 2.4.41: Upper image of View profile page

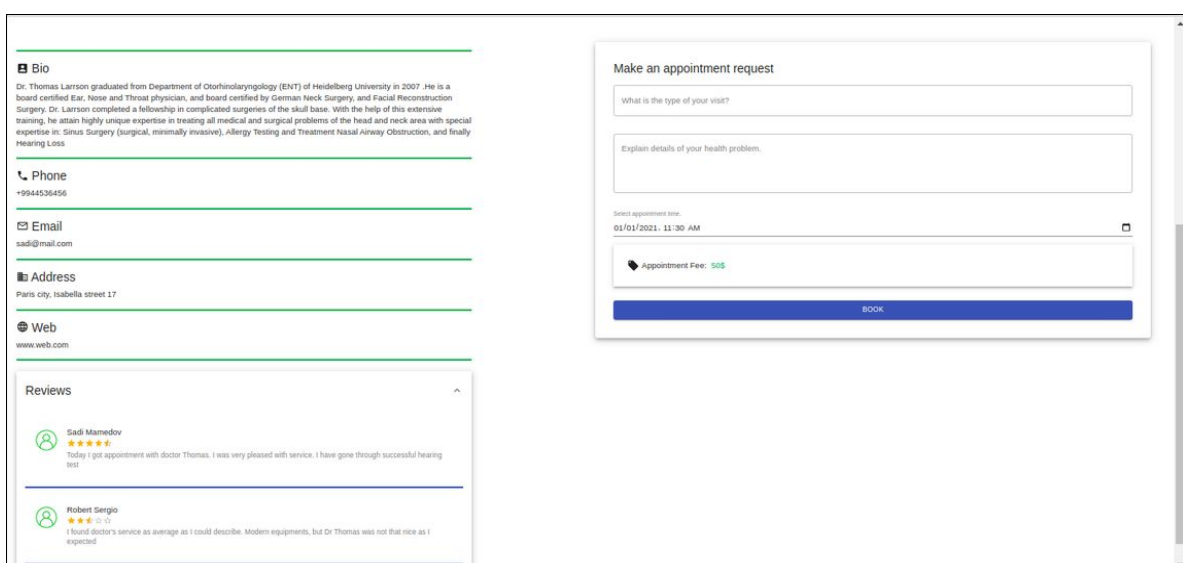


Figure 2.4.42: Lower image of View profile page

Users may share feedback regarding the doctor's service by pressing on *Review Doctor* button. The *Post* button will publish user opinion and an evaluation score represented by stars, while the *Cancel* button will simply ignore the operation and close the review modal (Figure 2.4.43).

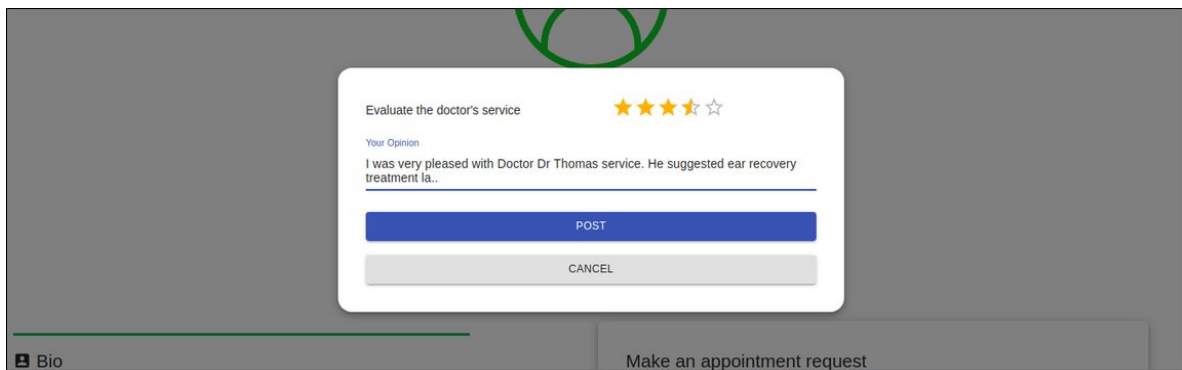


Figure 2.4.43: Review modal of View profile page

2.4.5 User Chat

Users may either initiate interactive chat from doctors profile, or reach directly from sidebar navigation. The chat is implemented on a real-time basis. The left side of the window shows a list of conversations, while the right depicts actual chat messages (Figure 2.4.5).

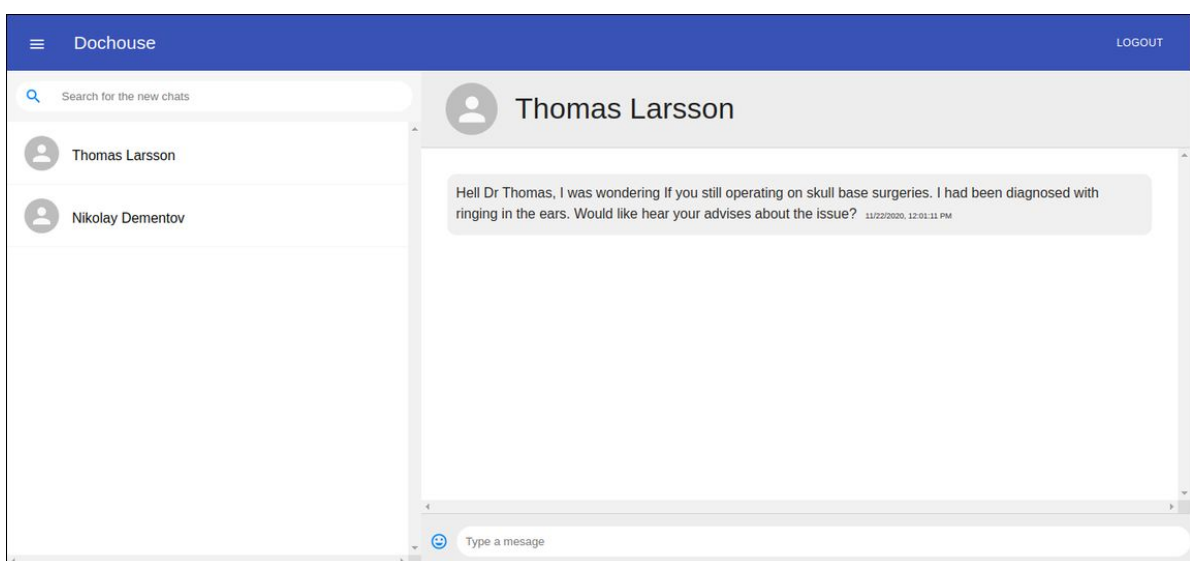
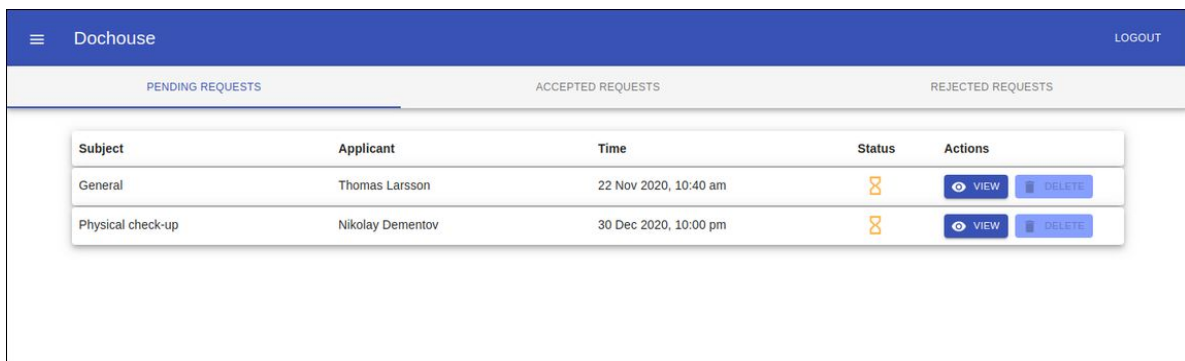


Figure 2.4.5: User mirror of the chat window

2.4.6 User Dashboard

Using sidebar navigation, users may access the dashboard which is the first page that appears after the login process. Users may control all request activities by dashboard tools. The dashboard consists of three categories: *pending requests*, *accepted requests*, and *rejected requests*. The table contains *subject*, *applicant*, *time*, *status* fields regarding request, as well as, *view* and *delete* action tools, which of those are available to users depending on the type of subject.

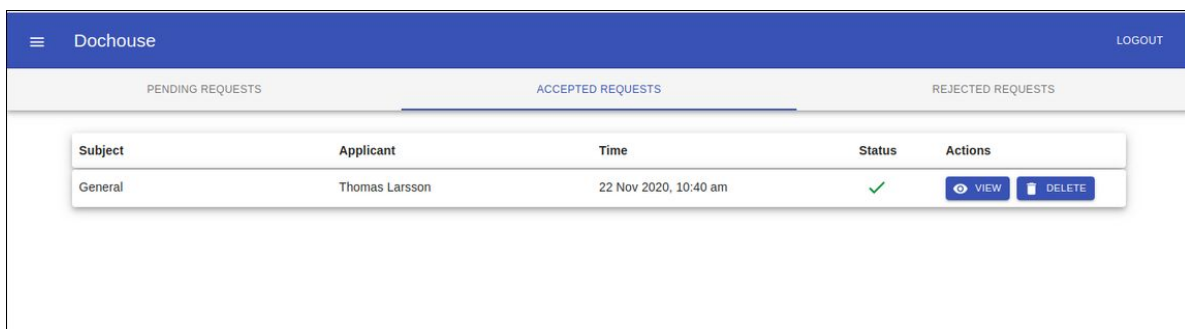
Pending requests represent the appointment requests which have not been resolved by doctors (**Figure 2.4.61**).



Subject	Applicant	Time	Status	Actions
General	Thomas Larsson	22 Nov 2020, 10:40 am		VIEW DELETE
Physical check-up	Nikolay Dementov	30 Dec 2020, 10:00 pm		VIEW DELETE

Figure 2.4.61: Pending requests tab of User dashboard

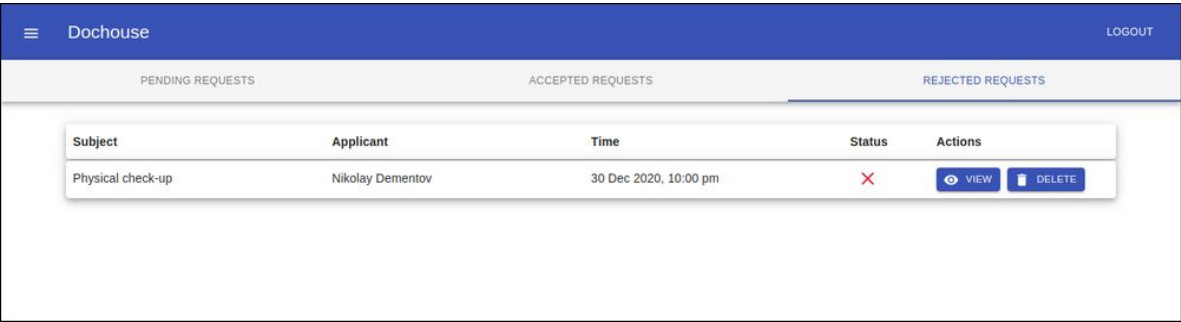
Accepted requests represent the appointment requests which have been accepted by doctors (**Figure 2.4.62**).



Subject	Applicant	Time	Status	Actions
General	Thomas Larsson	22 Nov 2020, 10:40 am		VIEW DELETE

Figure 2.4.62: Accepted requests tab of User dashboard

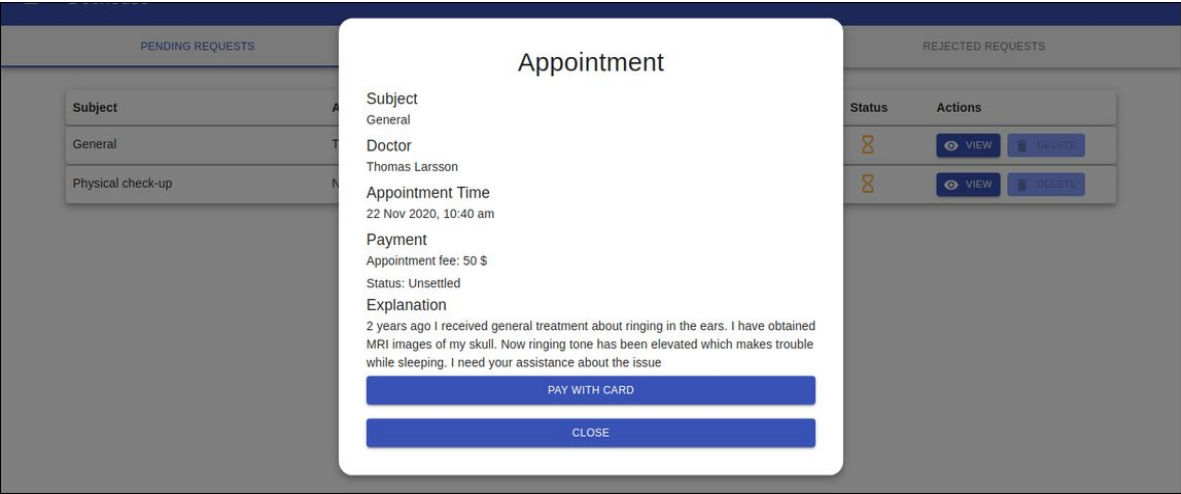
Rejected requests represent the appointment requests which have been refused by doctors (Figure 2.4.63).



Subject	Applicant	Time	Status	Actions
Physical check-up	Nikolay Dementov	30 Dec 2020, 10:00 pm	X	VIEW DELETE

Figure 2.4.63: Rejected requests tab of User dashboard

By applying *view* button users may see a detailed description of the appointment request. If the request has a predetermined appointment fee, users may proceed to the payment gate by following *pay with card* button. Use *close* button in order to close the *appointment* window (Figure 2.4.64).



Appointment

Subject
General

Doctor
Thomas Larsson

Appointment Time
22 Nov 2020, 10:40 am

Payment
Appointment fee: 50 \$
Status: Unsettled

Explanation
2 years ago I received general treatment about ringing in the ears. I have obtained MRI images of my skull. Now ringing tone has been elevated which makes trouble while sleeping. I need your assistance about the issue

[PAY WITH CARD](#)

[CLOSE](#)

Figure 2.4.64: Appointment window of pending request

Except from the pending requests, users are allowed to delete accepted and rejected requests. After the dialog appears, suggested buttons of the dialogue will operate based on their names (Figure 2.4.65).

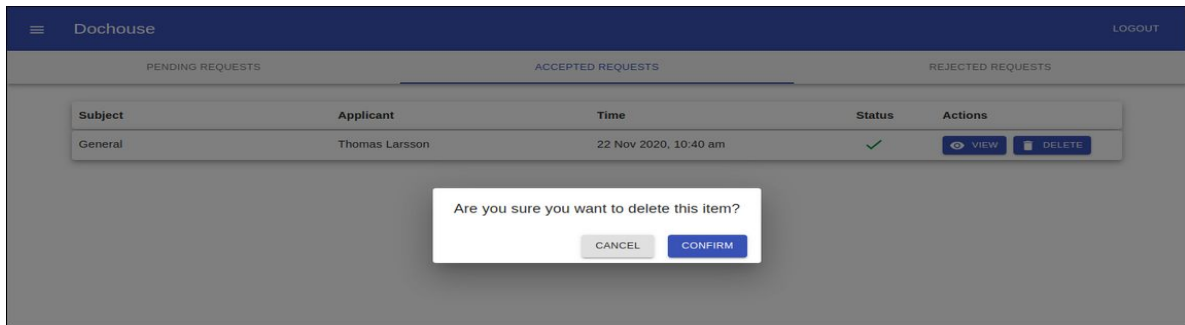


Figure 2.4.65: Delete dialogue of accepted request

2.4.7 Payment

Payment gate is responsible to make card payments according to the corresponding appointment fee. After inserting required personal datas on the first step (**Figure 2.4.71**), the user may proceed to the last step where card details are required to make payment (**Figure 2.4.72**). After the operation is executed, users will be notified about the payment status. On the other hand, doctors will be able to see the payment status of the appointment request.

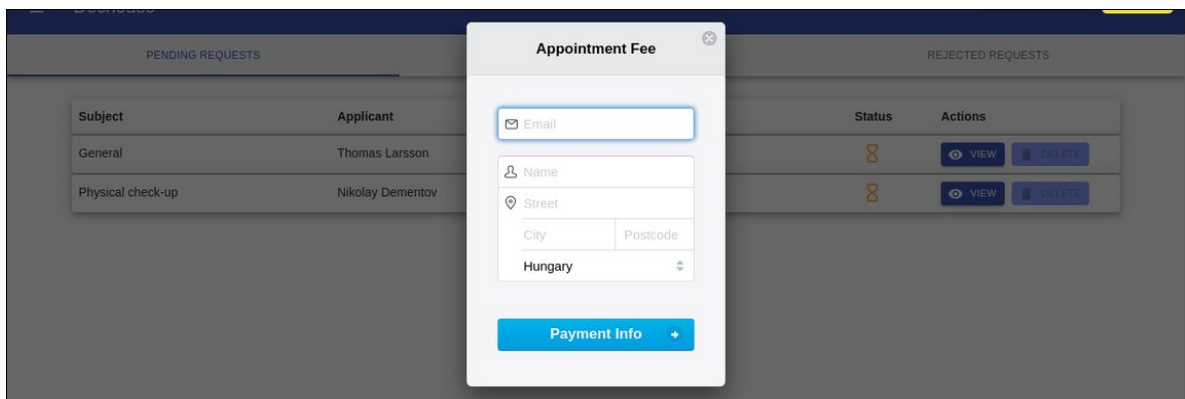


Figure 2.4.71: First step of payment gate

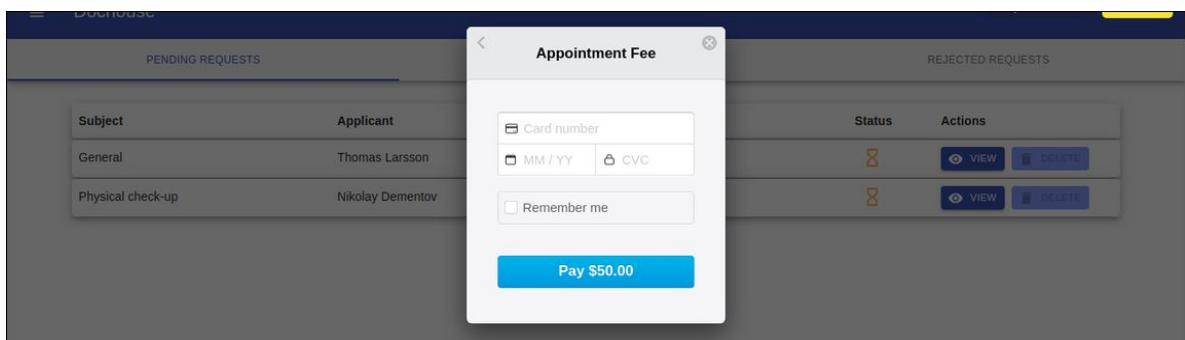


Figure 2.4.72: Last step of payment gate

2.5 Doctor Interface Guidelines

In this chapter, user interface components will be introduced using sample figures.

2.5.1 Doctor Navigation Sidebar

After successful sign in, doctors have access to the account canvas. Navigation sidebar is accessible on all pages from the top left corner, and provides the following options: *Home*, *Dashboard*, *Profile*, *Chat* and *Logout*. Selecting one of the options, doctors may navigate to other pages of canvas (**Figure 2.5.1**).

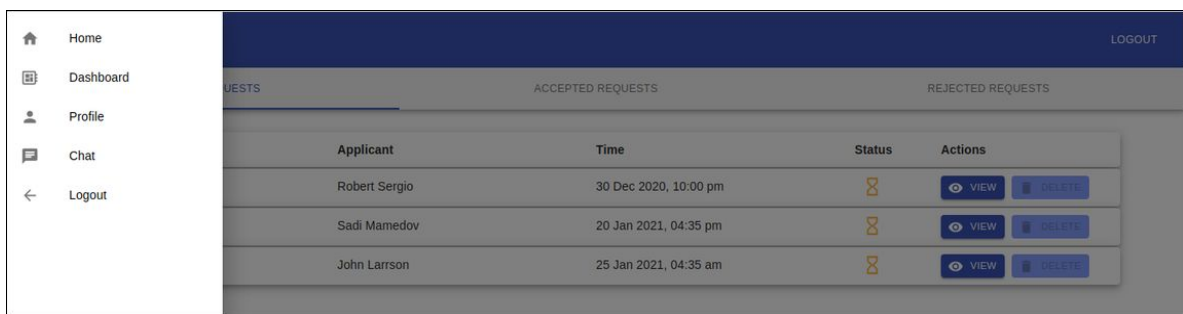


Figure 2.5.1: Doctor dashboard with open sidebar

2.5.2 Doctor Profile

Doctor profile is one of the crucial interfaces that application provides. In this part of the website, important doctor details are illustrated which will be visible to the patients. Contact details such as *phone*, *email*, *speciality*, and *address* are saved after the registration process. Apart from the concrete data, doctors should prepare substantial content for the *Bio* section such as their graduated institutions, work experiences, and some other field qualifications (**Figure 2.5.21** , **Figure 2.5.22**).

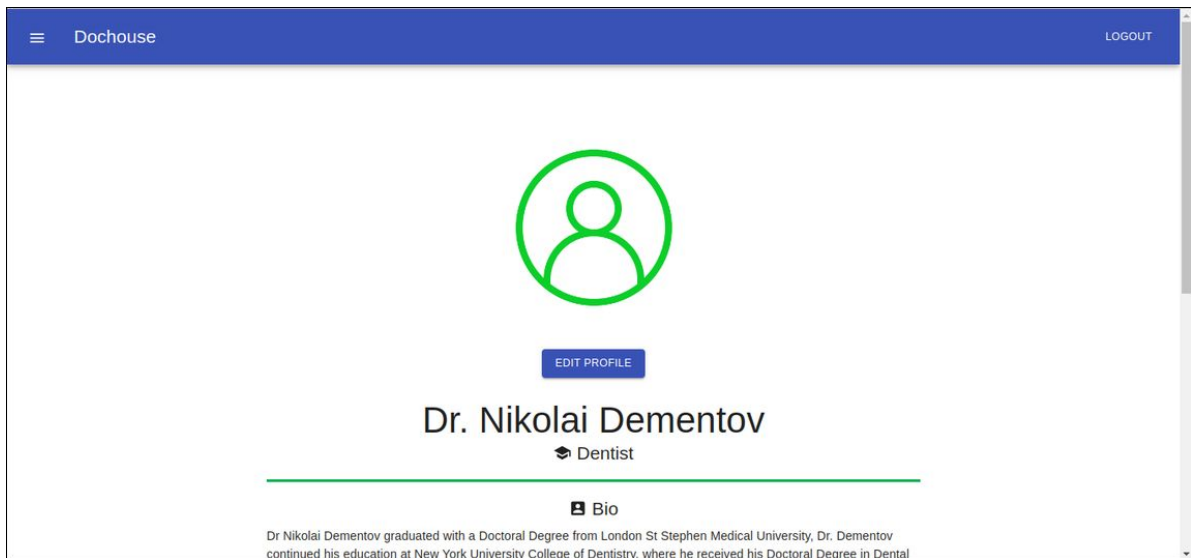


Figure 2.5.21: Upper view of the doctor profile

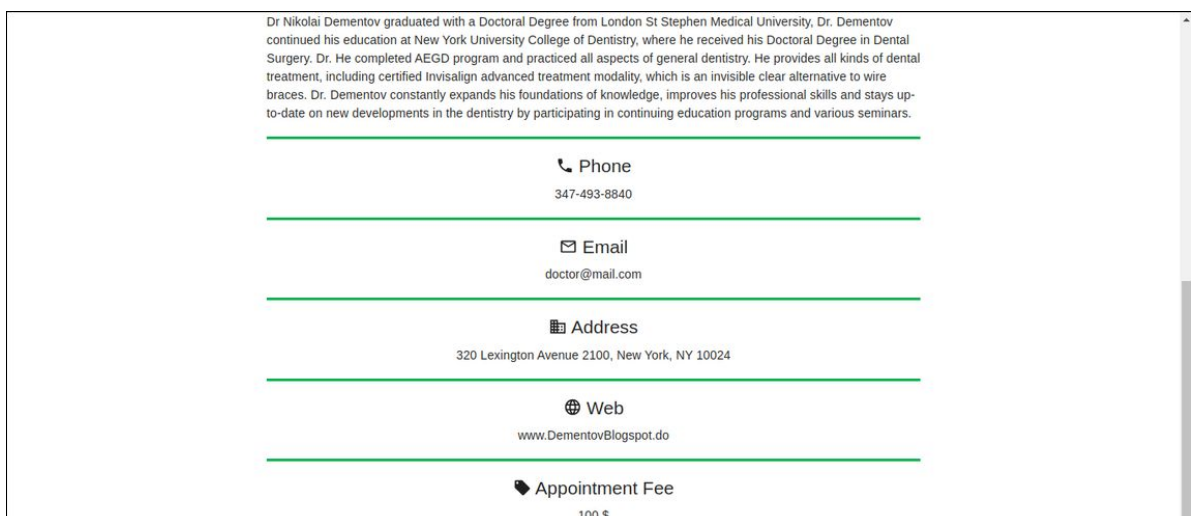


Figure 2.5.22: Lower view of the doctor profile

By pressing *Edit Profile* button, doctors can make some changes on displayed profile details. After making changes to proportional fields, clicking *Save* button will save the recent changes, while pressing *Cancel* button will simply close the window and ignore the changes. (**Figure 2.5.23**).

Figure 2.5.23: Edit profile window of the doctor profile

2.5.3 Doctor Dashboard

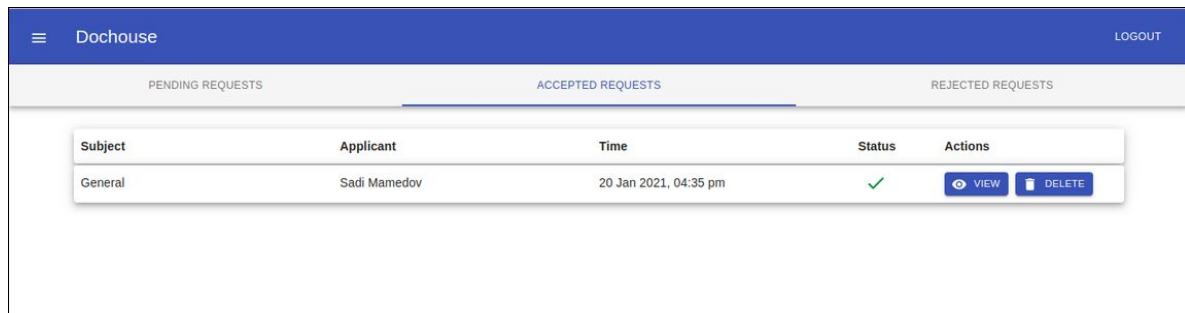
Doctors may manage appointment requests via dashboard tools. The received appointment requests will appear in *pending requests* tab. Depending on the doctor's decision of the pending request, that will be transferred to *either accepted requests* or *rejected requests*. The table contains *subject*, *applicant*, *time*, *status* properties, and *action* toolset.

Pending requests represent the appointment requests that have not been responded by the doctor (**Figure 2.5.31**).

PENDING REQUESTS					ACCEPTED REQUESTS					REJECTED REQUESTS				
Subject	Applicant	Time	Status	Actions										
Treatment	Robert Sergio	30 Dec 2020, 10:00 pm	⌚	VIEW DELETE										
General	Sadi Mamedov	20 Jan 2021, 04:35 pm	⌚	VIEW DELETE										
General	John Larrison	25 Jan 2021, 04:35 am	⌚	VIEW DELETE										

Figure 2.5.31: Pending requests tab of Doctor dashboard

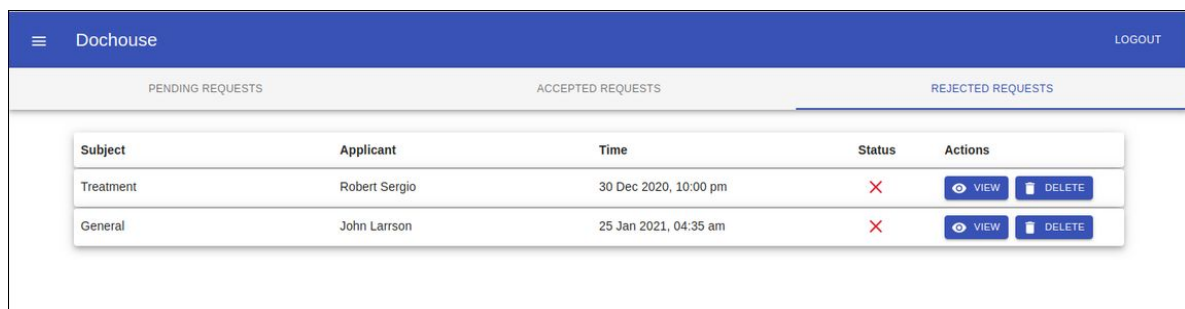
Accepted requests represent the appointment requests that have been accepted by the doctor (Figure 2.5.32).



Subject	Applicant	Time	Status	Actions
General	Sadi Mamedov	20 Jan 2021, 04:35 pm	✓	VIEW DELETE

Figure 2.5.32: Accepted requests tab of Doctor dashboard

Rejected requests represent the appointment requests that have been rejected by the doctor (Figure 2.5.33).



Subject	Applicant	Time	Status	Actions
Treatment	Robert Sergio	30 Dec 2020, 10:00 pm	✗	VIEW DELETE
General	John Larrison	25 Jan 2021, 04:35 am	✗	VIEW DELETE

Figure 2.5.33: Rejected requests tab of Doctor dashboard

In order to see an appointment request, by clicking on *view* button doctors may check detailed description. Moreover, appointment modal depicts *payment status* and *explanation* of a health issue, Considering all the details, doctors may refuse the request by pressing on the red colored *reject* button, also in contrast, they may accept the request following *accept* button. *Close* button will close the *Appointment* modal (Figure 2.5.34).

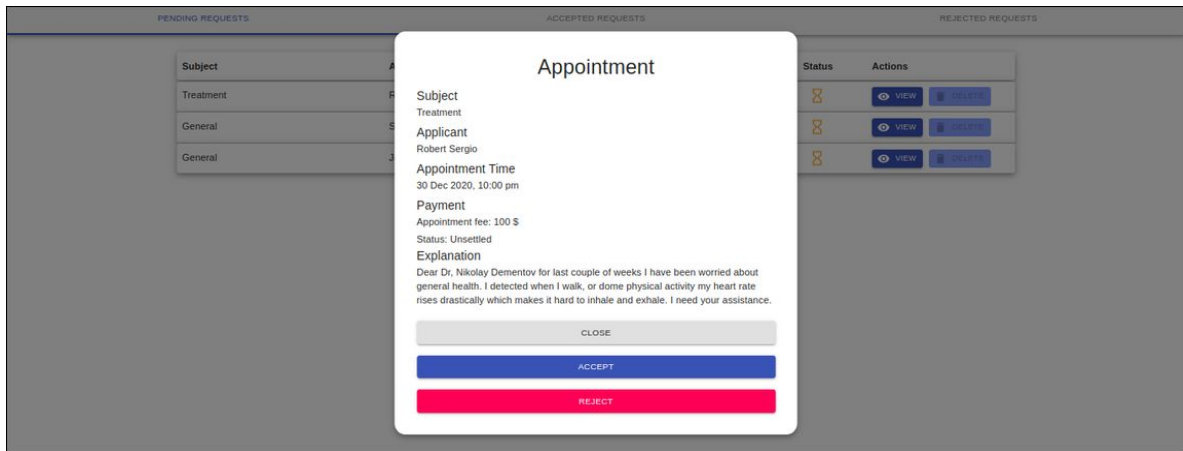


Figure 2.5.34: Appointment window of pending request from doctor canvas

Doctors have the option to delete accepted and rejected requests. After the dialog window appears, choosing *confirm* option will simply delete the selected request from the doctor dashboard (**Figure 2.5.35**).

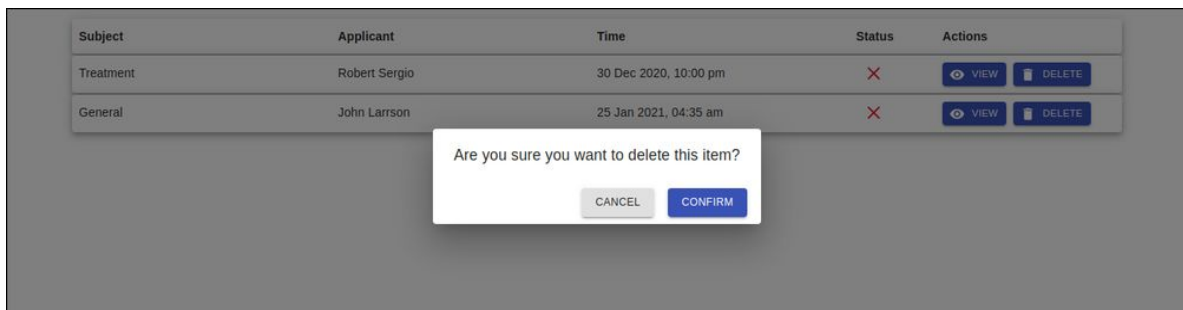


Figure 2.5.35: Delete dialogue of rejected request

2.5.4 Doctor Chat

Doctors may enter interactive chat from sidebar navigation. Left side of the window represents the list of conversations (**Figure 2.5.41**).

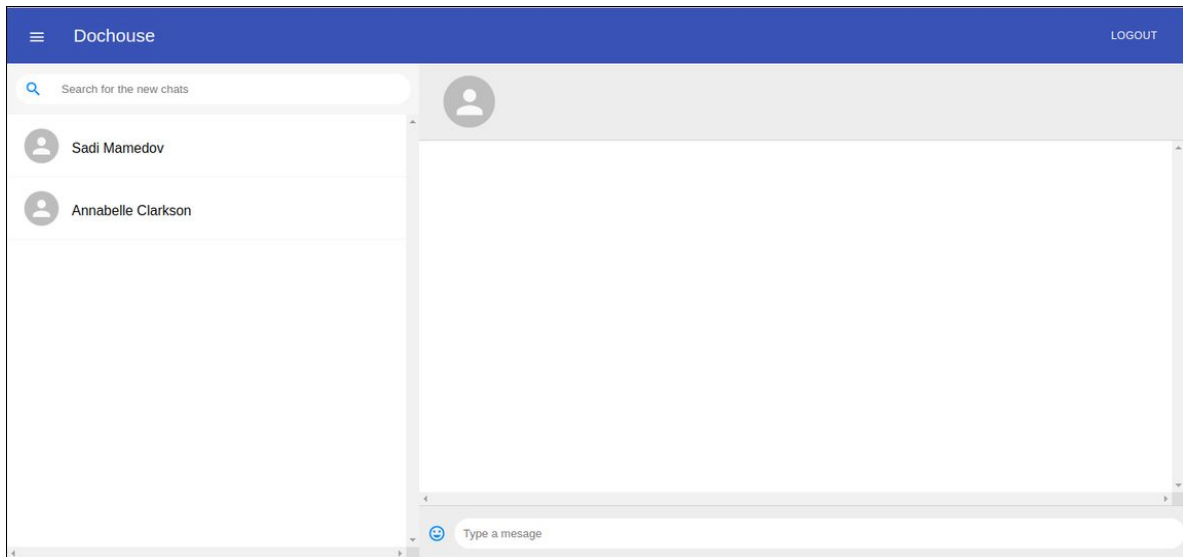


Figure 2.5.41: Doctor mirror of the initial chat window

Pressing on any of the chat conversations, corresponding messages will appear at the right side of the window. Pressing *Enter* button from keyboard, inserted text message will be sent to the receiver. Received messages are indicated within blue layout (**Figure 2.5.42**).

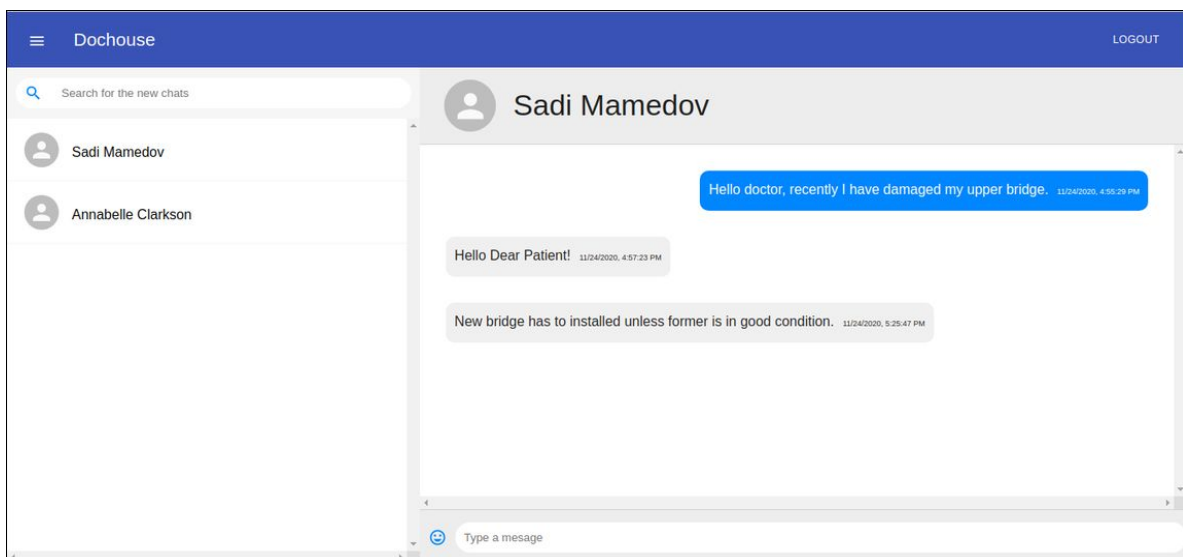


Figure 2.5.42: Doctor mirror of the selected chat window

Chapter 3

Developer Documentation

General directory structure, class and use-case diagrams will be illustrated in this chapter. Additionally, key functionalities of the application, relationship between client application and server application, database management will be analyzed elaborately, as well as testing plan and results will be discussed. For the configuration section of application, instructions in subchapters **2.1 System requirements** and **2.2 Installation Process** should be applied.

3.1 Program Structure

Full Stack application is built according to the MERN Structure. Client side application is developed by React JS framework [2], while server side is developed via Node JS web server [3] and Express JS [4] web framework. And for database management MongoDB database [5] is responsible. General page layout design is built on react library Material-UI [6], and some component structures are inspired from open-source project [7].

React is a declarative, maintainable, component-based JavaScript framework used to build dynamic client-side applications. Using react packages we may connect to the server via sending front-end data through *http requests*. Express JS is a fast, minimalist server-side web framework that runs inside Node JS which is an event-driven, asynchronous JavaScript server runtime environment to build scalable web applications. Using Node JS database drivers or callbacks, documents stored in MongoDB, a cross-platform document-oriented database, may be accessed, updated and deleted. After database operation is completed, a query result in the shape of JSON format is received from a server-side application. Subsequently, received data from the database is forwarded to client side through *http response*. As a result, client-side UI elements are updated simultaneously **Figure (3.1.1)**.

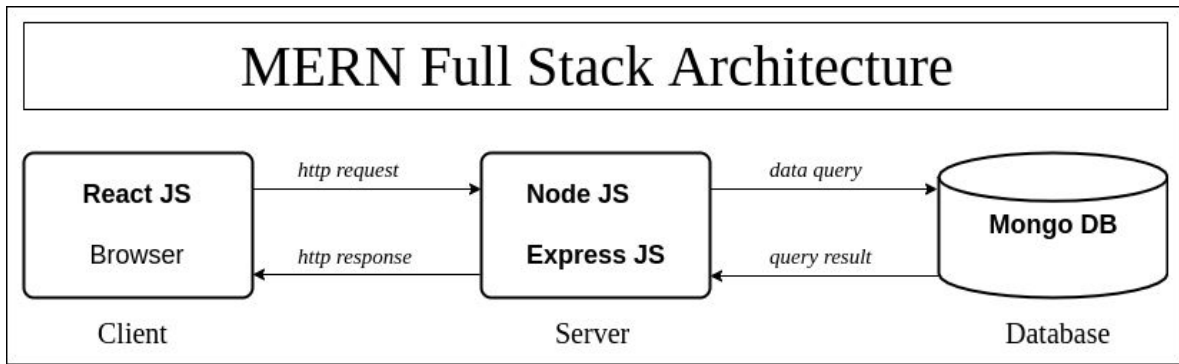


Figure 3.1.1: Full Stack Architecture of the application

Project structure consists of two major subfolders: *backend* (Figure 3.1.2) (server) and *doc-house*(client).The main configuration of the server lies in *server.js* file where initially, node server is launched on any determined port, followed by connection established to MongoDB database. Furthermore, the server app listens to changes on the database to make it real-time for interactive chat. Next, a set of backend REST APIs is set up categorizing these into routers based on database models. We will use those routers to receive and handle http requests coming from the front-end.

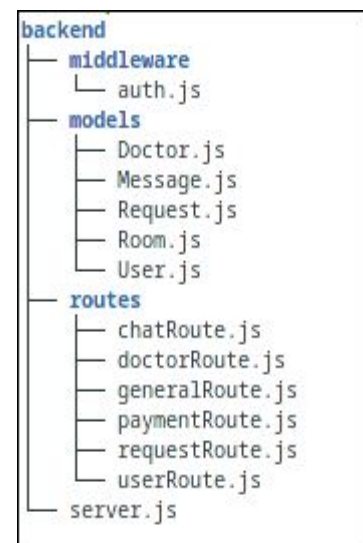


Figure 3.1.2: Tree structure of server application

At the end of the file, the server app continuously listens to the port. Collection of API routes lies under *routes* directory, while database schema models composed under *models*. Most of the user requests will require authentication in *auth.js* which will be applied for data security purposes. Middleware will only accept authenticated user actions and block all the requests made from outside of the application.

Client side application files lie under doc-house directory (**Figure 3.1.3**). Initially, React DOM locating at the top of client web application will render *App.js* component in *index.js* file. Later, it returns the Main component in *Main.js* which wraps all the other components with Router to render them based on url path. Also browser history is created at the root, and passed down to have access from children components. Source folder *src* consists of *assets*, *components*, *context*, *styles*, and *validation* subdirectories. Rendering elements locate under *components*, and designs will apply from JavaScript and CSS files located under *styles* subdirectory. In order to store user details Context API along with *useContext* hook in file *context/userContext.js* will be utilized. Rendering components are stateful class components where rendering is controlled by lifecycle methods, except *chat* components which are stateless, and to control lifecycle methods powerful feature react hooks will be used. Client validation rules under *validation* will apply to form inputs inserted by users and doctors.

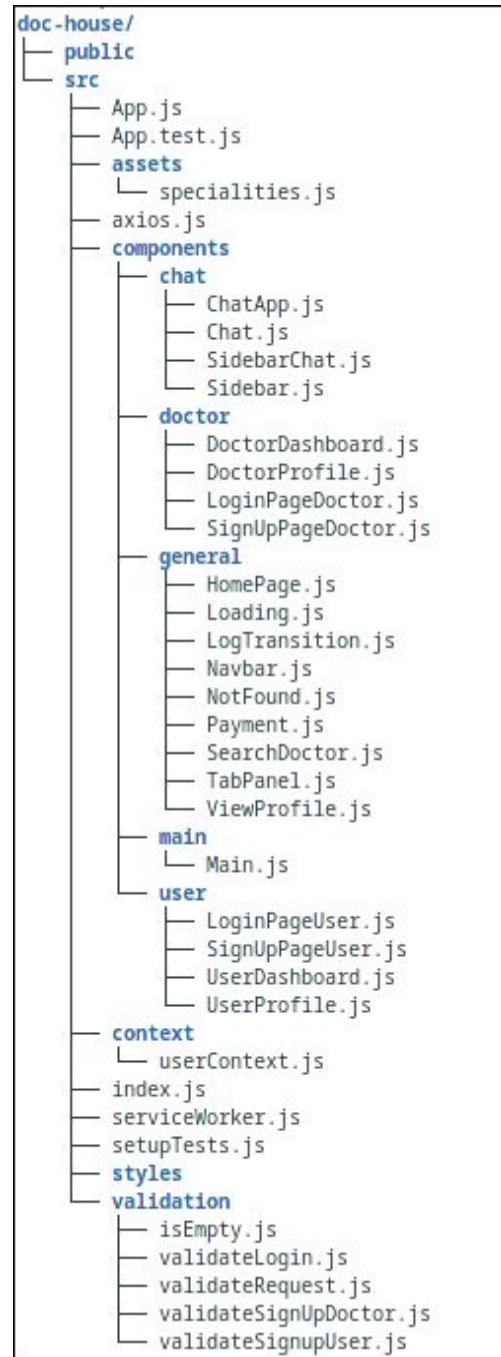


Figure 3.1.3: Tree structure of client application

3.2 Use Case diagram

One of the objectives of developing this application is to accomplish high quality user experience and interface design. Straightforward instructions and overall rudimentary design enable users to navigate through and use functionalities at ease. Both users and doctors are required to register in order to log into the account canvas, and to close the session the both may log out. Functions such as editing profile, using chat, viewing and deleting requests are available to both end-users. Users may search the doctor, make an appointment request and initiate the chat conversation. Moreover, the user may opt to pay the appointment fee, and after the service may even review the doctor. On the other hand, doctors react to the request either accepting or deleting it. Use cases of the sequence are connected with directed arrows where the source case is considered as prerequisite, and the pointed one as dependent case (**Figure 3.2**) .

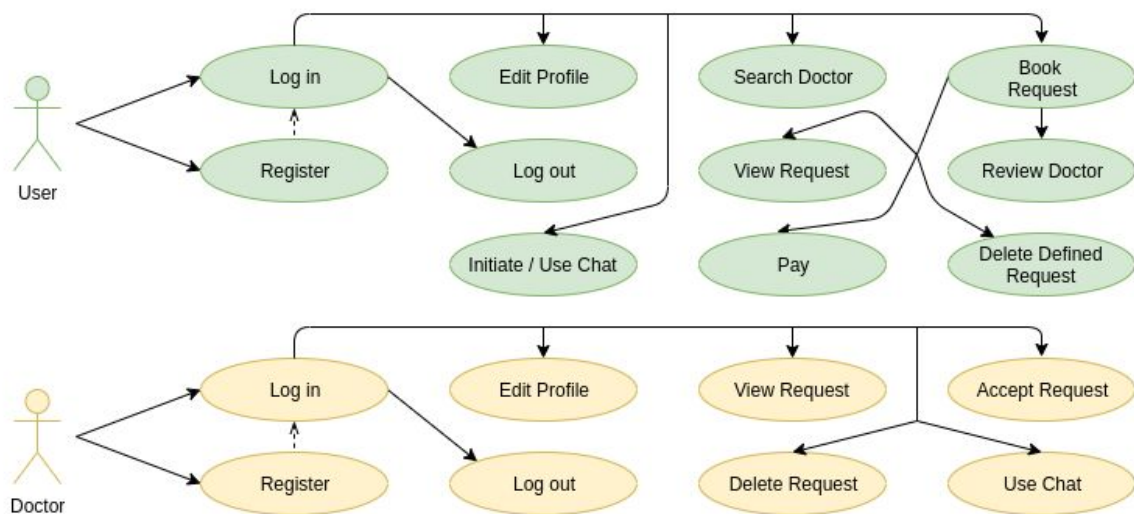


Figure 3.2.1: Use case diagram of application

3.3 Class diagrams

Apart from database schema, Dochouse web application consists of two significant parts: client-side application and server-side application. Client-side class diagram represents the React classes including member methods and relations with other components which all

together constitute the client API built into the web browser (**Figure 3.3.1**) .

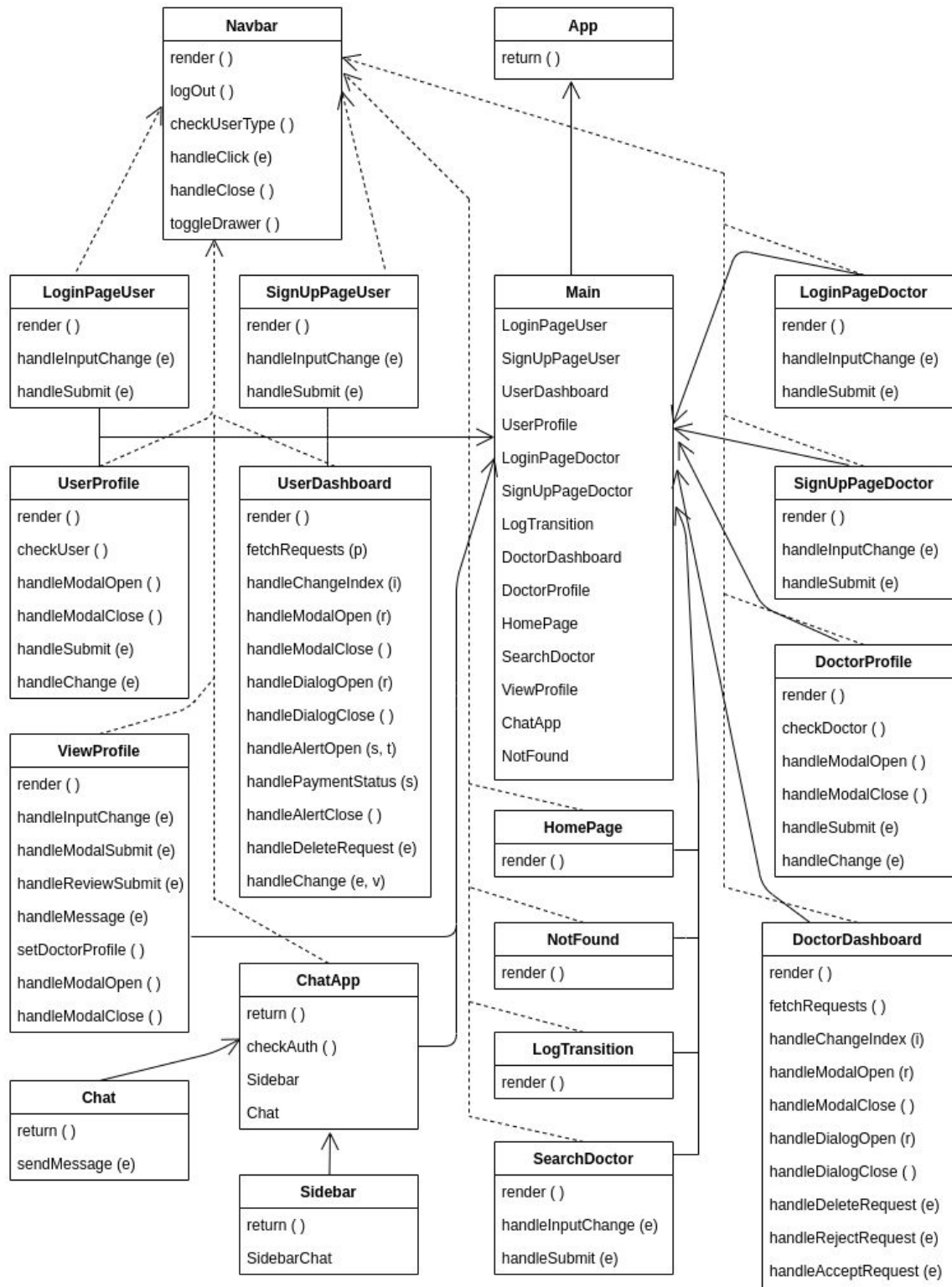


Figure 3.3.1: Class diagram of client application

Server-side class diagram (**Figure 3.3.2**) represents mutual use relationship between server and database, and connection between routes and main server thread. Class diagrams depict determined routes where application's endpoint URLs respond to requests coming from the client-side application. Technically, these routes are http-based RESTful APIs which are mounted on the server app instance. Using Express routers, dedicated URLs can be structured into separate javascript files under different directories. Routes from the figure below contains http request methods like *POST*, *PATCH*, *GET*, *DELETE* and each of them is mounted on following resource locators preserving relative naming conventions : *'/general'*, *'/users'*, *'/doctors'*, *'/requests'*, *'/chat'*, and *'/payment'*.

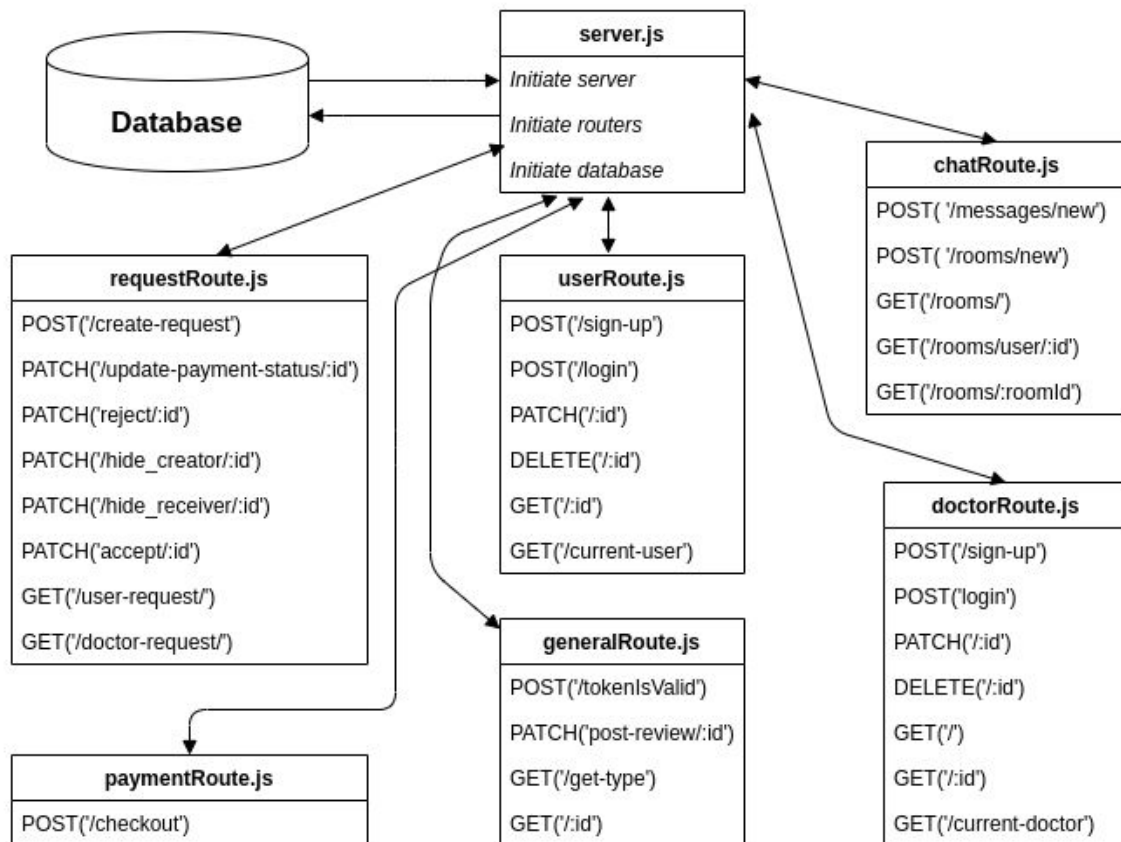


Figure 3.3.2: Class diagram of server application

3.4 Authorization Processes

Authorization is the core segment of Dothouse application. In this chapter development of registration and sign in processes will be elaborated.

3.4.1 Transition Process

As the application serves different types of end users, there must be logical transition between authorization interfaces. For that purpose, the transition page residing inside *LogTransition.js* is constructed as a React component, and renders based on path '/requests' preceded by corresponding client hostname and non-reserved port number which by default is set to *localhost:3000*. Within the paper layout four button elements are located and they redirect to corresponding register and login pages on clicking events. The state of component does not depend on any asynchronous function call which makes component render instantly.

3.4.2 Registration Process

User registration page loads under '/signup-user' path, and covers the registration form with controlled input elements, *Register* submit button, and *Navlink* link element to the login transition page. All the required field elements, validation errors and operation success indicator, a boolean value, are saved to component states to handle dynamic rendering. Event handler functions *handleInputChange()* changes state based on input field, while *handleSubmit()* is responsible for collecting form data and sending it to the server for storing in the database. Before storing form data to the database, it must be legitimate to satisfy client and server validation rules. Client-side validation initially checks data whether being empty, if not further continues applying length requirements and regex expressions. For instance, passwords must be at least 6 characters long, email must match specific format, etc. In case of invalid submission, the submit handler ejects and component state *errors* receive those warning messages which spanned under required fields. Using the *axios* package facility, we send user data via *POST* request to the server under URL closing

with *'/users/signup'*. Initially, the server method handler applies server-side validation to check whether a user with received email registered before, and if not sends a warning response back to the client where the response is assigned to state *errors*. In case of unique email, password is hashed for security purposes and other user data is saved as *Mongodb* schema under corresponding database collection. Afterwards, client application receives success status through http response, and renders snackbar indicating successful registration which leads to the login page.

Similar logic applied to the doctor registration process, except doctors are required to provide extra information such as *Speciality, Work Address and Phone number*. Analogous client and server validation rules will apply including the extra fields. For the specialty field, one of multiple name labels stored in *specialties.js* may be selected. Unlike users, collected doctor data is sent to *'/doctors/signup'* endpoint, where doctor registration is handled. The data of successfully registered doctors is stored under the *doctors* database collection.

3.4.3 Login Process

User Login page renders under *'/login-user'* path, and covers the sign in form with controlled input elements, *Login* submit button, and *Navlink* link element to the login transition page. Two required input elements *email, password*, as well as *errors* object which represent validation errors are saved to component state. This component contains two handler functions: *handleInputChange()* and *handleSubmit()*. First one synchronously updates component state based on target input whereas the latter collects login credentials and submits the form. Client and server validation rules must be satisfied in order to launch a log session. Client-side validation in *validateLogin.js* checks submitted data whether being empty or fulfilling length requirements. Similar to the registration process, warnings will be spanned under corresponding fields in case invalid data provided by the user. Once *errors* object is empty which indicates validity of provided data, login data along *POST* request will be sent to the backend API defined on full path *'http://localhost:5000/users/login'* where the server application runs. At first, the method handler applies server-side validation on whether the received email exists in the database, and if not sends a warning message to client through http response. Subsequently, it checks equality between hashed formats of

entered password and actual user password, and in case of inequality it sends *invalid credentials* warning to client-side browser. We assign a json web token with secret seed and send it back to the client application where the browser will keep the access token until its expiration period in local storage. After a successful login procedure user is redirected to the dashboard. In order to secure data transactions, authorization middleware is implemented as a hidden bridge layer between database and server requests. Therefore, those requests made without an authentic token will be blocked.

Similar approach applied to the doctor sign in process. Analogous client and server validation rules will apply. Different from users, collected doctor data is sent to the *'http://localhost:5000/doctors/login'* endpoint, where the doctor sign in process is handled. After a successful login process, the browser will navigate to the doctor dashboard.

3.5 Profile Processes

In this chapter development of both user and doctor profile functionalities will be discussed.

3.5.1 Navigation Process

Every page of the website contains a Navigation facility which is operated on the *NabBar* component where the left side menu drawer and right side log in/ log out functionalities are located. Navigation elements appear based on type of user and state of logging in and out. Prior to rendering component JSX elements, in *ComponentDidMount()* lifecycle method *checkAuth()* function checks authorization by searching the access token in local storage and assigning it to the component state, followed by asynchronous *checkUserType()* function call which sends a *POST* request to the endpoint *'general/tokenIsValid'*. In the server method handler, the received token is verified via secret character sequence. It's queried to the database, and later validity of token, type of user and user/doctor data are sent to the client. We set timeout after asynchronous calls. Therefore, react JSX elements may be rendered conditionally. When no user has yet signed in, *logout()* method listening on button element terminates active session by removing access token and redirecting to the transition

page. Similarly, left toolbar list elements contain navigation options conditionally, except *Home* element which is consistent member of the drawer menu providing access to the homepage. Navigation bar elements referred by relative icons. *Menu* button is handled upon open/close actions. General bar layout design resides in *styleNavbar.js* file under *styles* directory.

3.5.2 Editing Process

User profile component renders user information by fetching corresponding data from the database. Component first renders spinner object based on loading state until asynchronous fetch method *checkUser()* returns resolved promise which sets *loading* state to false in lifecycle method *ComponentDidMount()*. Initially in the *checkUser()* method body, validity of access token is inspected at URL '*general/tokenIsValid*'. After successful verification response received, another *GET* request is sent to '*/users/current-user*' with authorization token header. If no errors occurred, user data is fetched from the database and sent to the React client, where later *loading* state sets to false and user details are loaded.

User data is set on the registration process, however users have permission to change some information such as *bio*, *address*, and *web*. Open and close state, as well as data parameters of *edit profile* modal are dynamically controlled by handlers where closing *cancel* button, or on backdrop click enabled by default. The *handleSubmit()* asynchronous function assigns recent modal states to *updates* object, and later sends it and current user id along *PATCH* request body and parameters field to the server. In *UserRoute.js* mongodb function *findOneAndUpdate()* is applied on database collection, which is *User* model in this case. Next, we inform the client with successful changes and client implements page reload.

Similar logic applies to doctor profile editing, which has extra data fields in portfolio. Unlike user profile editing, all fields are mandatory to exist in profile, and for instance setting any field empty will yield a warning and prevent editing process. Also to prevent data overflow scrolling option activated for better user experience. *Doctor* model of the database will be updated according to the *PATCH* request sent on endpoint '*/doctors/:id*'.

3.5.3 Search Process

Search process is operated on the controlled search form. The component states *condition_docname* and *city_region* properties are dynamically updated based on form input target with method *handleInputChange()*. On form submit, the handler sends query parameters along with *GET* request to the endpoint */doctors/*. Corresponding method handler in *doctorRoute.js* receive the query parameters, and apply those to get results from the database. According to the code snippet below, we apply regex expressions with case insensitivity option *'(\$options: "\$i")'* on received parameters to match documents of *Doctor* collection. Outermost *\$and* operator gives list of all doctors, if no parameters queried from the client. Inner *\$and* operator translates *true* when both *condition_docname* and *city_region* parameters match, or single of two matches while another being empty. The *\$or* operator requires the first parameter to match any of *first_name*, *last_name*, *speciality* and *bio* properties of *Doctor* schema. Final result of query as a *json* object assigned to the *doctors* which will be sent to the React browser.

```
const doctors = await Doctor.find({$and: [{},
                                          {$and: [{$or: [
{first_name: {$regex: req.query.condition_docname,$options: "$i"}},
{last_name: {$regex: req.query.condition_docname,$options: "$i"}},
{speciality: {$regex: req.query.condition_docname,$options: "$i"}},
{bio: {$regex: req.query.condition_docname,$options: "$i"}} ]},
{address: {$regex: req.query.city_region,$options: "$i"}} ]} ]});
```

After receiving a server response, the state of *doctors* array is set to query result and mapped into *Paper* division fragments which are rendered on screen. On paper list element doctor's information is displayed, including the average review point which is calculated through *average(array)* method. Link button is placed on any fragment which redirects to the corresponding doctor's profile page. Using the *searched* boolean flag and *doctors* array length, an empty result case is determined and the warning fragment with instructions is displayed under the search form.

3.5.4 Visit Profile Processes

Visit profile processes cover functionalities from the significant component *ViewProfile* which is loaded under the path *"/profile/:id"* where the *id* part indicates visited doctor's profile. Component async methods are fetched in loading state. Below the title details and doctor avatar, grid element pattern contains two cells: doctor information including reviews and appointment request form. Controlled form elements are tracked in component state, and once submitted, those are sent to the server within async handler *handleReviewSubmit()* in order to save requests in database likewise other controlled forms that were discussed in previous chapters. Validation rules prevent appointment requests with empty fields being booked. Request form is decorated with material-ui library components including *date picker* object, button elements, icons, etc.

All the reviews may be inspected in a resizable reviews component: material-ui *Accordion*. Review modal is available on the review *doctor* button. Author of the review, five-scale service evaluation score and opinion are patched to server RESTful API. In the method handler, mongodb method *findOneAndUpdate()* updates the *Doctor* document by pushing the user review object into the *reviews* array property. *Try catch* block assures error detecting, successful attempt with status **200 OK**, while failure labeled by status **500 Internal Server Error** transferred to client application. Generally for creation of objects a standard *POST* request method is used, but in this particular example *PATCH* request is necessary as in every submitted review stored in *reviews* property which needs to be updated in the *Doctor* document. Review modal state are regulated with dynamic handlers *handleModalOpen()* and *handleModalClose()*.

Chat conversation is initiated following the *message doctor* button which is handled by *handleMessage()* async method on click event. Initially user and doctor ids, names are collected which of those are core properties to create conversation. Next, the endpoint router handler assigned to *"/chat/rooms/new"* receives the request body and creates a new *Room* schema for conversation. However, using *find()* precondition room duplicates are prevented, therefore, only a single conversation room may exist containing two same members.

3.5.5 Dashboard Processes

Dashboard processes of both end-users covers requests handling. Application bar contains three request tab panels and their properties are defined in *TabPanel(props)*. Active displayed tab is inspected by state *activeTabValue* demonstrating id values of tabs. After active tab selection, asynchronous *fetchRequests(p)* function fetches the requests based on correlated tab value. Result of server response is generated from database collection *requests* value of which can be queried from both the author and the receiver. Once an appointment request is created, that inherits author and receiver details for later reference purposes.

View and *delete* actions are controlled by event handlers. The former redirects to the *Modal* window, while the second belongs to a type of *dialog* process. Deleting process is managed by *handleDeleteRequest()*, and may only be applied on non-pending requests. Basically, we send request details along *PATCH* method which hides visibility of content, however the database admin has privileged access to the request documents. Namely, *creator_visibility* and *receiver_visibility* are the appointment request properties in the database, and those are the condition indicators for end-users.

The set of doctor dashboard functionalities is the superset of the ones in the user dashboard. Two extra methods *accept request* and *reject request* are conducted by async handlers *handleAcceptRequest* and *handleRejectRequest*. Work principle of these functions is to assign a doctor response to the status of corresponding *Request* schema. As a result, requests are classified based on following status values: *pending*, *rejected* and *accepted*.

3.5.6 Payment Process

Payment service implemented by Stripe API [7] that supports debit card payments. *StripeCheckout* component handles payment cooperatively in client and server. Stripe secret token, other purchase details and stripe public key must be passed into *StripeCheckout* component. The collected card and address details with *POST* request are sent to URL

"/payment/checkout". Inside file *paymentRoute.js*, in order to prevent exposing key details we obtain the stripe private secret key that is stored in *.env* file. Charge process is conducted with an idempotency key *uuidv4()* which is a unique value generated by the client and recognized by server on subsequent attempts of the same request. After the transaction procedure we send the status of payment which will be visualised to the user. Later in this procedure, payment status of appointment request is updated and will be visible to doctors. Transaction process is secured by stripe and no database operation is involved. We have proceeded payments in test mode only, however a real authenticated payment system along with legitimate charging protocols is considered for future production releases.

3.5.7 Chat Process

Development of the chat application is implemented based on guides from content [8]. It contains *Sidebar.js* and *Chat.js* stateless components which are constructed on react hooks. Available chat rooms are fetched from the database if the mirror user is part of room members. When one of the rooms clicked, the client redirects to URL *'/chat/rooms/:roomId'* and message contents are visualised in the chat window. Logged in user details are passed to the chat component with *UseContext* API, later those details are used to fetch relevant rooms and messages within. In order to realize interactive message transfers we have utilized websocket. Basic work principle of chat is client socket sends message updates and server socket receives updates and in real-time should respond back to the client.

Socket based framework *Pusher* API will be used as a broker between socket-ends, when triggered it transports the messages. We have to implement real-time synchronization between mongo database collection and frontend elements. Pusher instances are created in *server.js* and *Chat.js* where data transport is realized on the socket channel. In case one of the chat members sends a message, *messages* array property of *rooms* in database collection is updated. Since the connection between server and database was established we watch on *rooms* collection to detect *'update'* operation, and upon detection through the channel the last updated message that is saved is sent back to client. Using *useEffect()* hook on *messages* state of Chat component, after every change made component rerendered.

Messages that are fetched from *http GET* response, handled in backend API `‘..chat/rooms/:roomId’` where corresponding messages are queried based on room id.

3.6 Testing

Testing phase of the application is divided into two parts: client testing and server testing. Software functions and components are tested with Jest [9] and Enzyme [10] JavaScript libraries.

Client testing files reside under the path *doc-house/tests/*. In the client package setup file, namely *package.json*, we add script `"test": "react-scripts test --watchAll --env=jsdom"`. Prompting “npm test” command detects the files that end with “test.js” suffix and runs all test suites sequentially. Unit tests check individual states, procedure, functions of client application. In order to check stability of components after update we use snapshot testing to compare snap codes captured at different moments. Therefore, snaps that are generated in `__snapshots__` directory can be profiled.

```
describe('Analyse React Router V5', () => {
  it('Homepage renders for path "/", () => {
    const wrapper = mount(
      <MemoryRouter initialEntries={[' /' ]}>
        <AppWithRouter/>
      </MemoryRouter>
    );
    expect(wrapper.find(HomePage)).toHaveLength(1);
    expect(wrapper.find(NotFound)).toHaveLength(0);
  });
  it('Invalid path redirects to 404 NotFound page', () => {
    const wrapper = mount(
      <MemoryRouter initialEntries={[' /WrongUrl' ]}>
        <AppWithRouter/>
      </MemoryRouter>
    );
    expect(wrapper.find(HomePage)).toHaveLength(0);
    expect(wrapper.find(NotFound)).toHaveLength(1);
  });
});
```


In the unit test snippet above, this suite tests functionality of routing. The function `it()` is the alias for `test()`, and `expect()` returns boolean value representing test status. Mounted wrapper under the path `"/"` returns *HomePage*, while undefined url returns *NotFound* component.

Server testing is added as `"concurrently \"npm run server-test\" \"jest\""` to the server *package.json* where we initiate the server with **NODE_ENV=test** option. Original database should not be altered for testing operations, so new test database inside mongodb cluster. Both test and original database connection URIs are stored in *.env* file, and by default the node environment is undefined. To check server APIs we introduce unit tests with the *SuperTest* module which has capability to make http request assertions. As an illustration, in *chat.test.js* file, in *beforeAll()* method database connection is established, while *afterAll()* target collection dropped from test database, and closing connection follows. We set *done()* callback methods in each test method to make sure that the case test does not cross to the next one in sequence unless being completed. As asynchronous methods are tested, timeout values are increased to at least 10 seconds per test suite since server operations may reject requests due to server timeout.

```
it("Invalid credentials on Doctor login attempt, logging in Fails.",
  async done => {
    const response = await supertest(base).post('/login').send({
      email: "philidor@hotmail.com",
      password: "somesecretcode2332"
    });
    expect(response.status).toBe(400);
    done();
  });
```

In the test snippet above, we check whether providing invalid password generates an error response with **status 400**. The root of the full URL path is represented by a variable *base* which, in this particular case, holds the value of *'http://localhost:5000/doctors'*.

Chapter 4

Conclusion

Since process automation has been implemented in several industry fields including information technology, introducing digital public services may create potential benefits. Applying software standards and facilities of robust development tools, other industry fields can be migrated to web surfaces. In this thesis, development and usage of a new generation complete stack web application is examined. This application proves to be a real-world instance to the client-server model of distributed systems. By achieving server performance boost and more scalability of resources, web applications will continue to dominate among digital public services and business startup models.

4.1 Future work

There are several new features that application compass can be extended to. Prior to the production step, advanced testing methods, particularly more integration tests should be provided. In order to boost user experience, an instant notification system is considered to be developed later. Eventually, it can be extended by adding a secure system that preserves commerce protocols for processing real payments.

References

- [1] Sadi Mamedov, “*DocHouse*” *github public repository*,
URL: <https://github.com/MoneiBall/DocHouse>, Last accessed on 12/12/2020.
- [2] *React Documentation*,
URL: <https://reactjs.org/docs/getting-started>, Last accessed on 20/11/2020.
- [3] *Node JS Documentation*,
URL: <https://nodejs.org/docs/latest-v13.x/api>, Last accessed on 27/10/2020.
- [4] *Express Documentation*,
URL: <https://expressjs.com/en/5x/api>, Last accessed on 13/10/2020.
- [5] *MongoDB Documentation*,
URL: <https://docs.mongodb.com/manual>, Last accessed on 01/11/2020.
- [6] *Material-UI Documentation*,
URL: <https://material-ui.com>, Last accessed on 05/11/2020.
- [7] Jamie Shi, “*MERN-Social-Network*” *github public repository*,
URL: <https://github.com/jm-shi/MERN-Social-Network>,
Last accessed on 04/10/2020.
- [8] Clever Programmer, *Build a Whatsapp Clone with MERN Stack*
URL: <https://www.youtube.com/watch?v=gzdQDxzW2Tw>,
Last accessed on 05/10/2020.
- [9] *Jest Documentation*,
URL: <https://jestjs.io/docs/en/getting-started>, Last accessed on 05/12/2020.
- [10] *Enzyme Documentation*,
URL: <https://enzymejs.github.io/enzyme>, Last accessed on 05/12/2020.